# Contributions à l'analyse statique de programmes manipulant des tableaux

Mathias Péron

UNIVERSITÉ DE GRENOBLE

**THÈSE**

pour obtenir le grade de
**DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE**
*Spécialité : Informatique*
selon l'arrêté ministériel du 7 août 2006

préparée au laboratoire **Vérimag**
sous la direction de **Nicolas Halbwachs**
dans le cadre de l'**École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

présentée et soutenue publiquement par
Mathias Péron
le 22 septembre 2010

# Contributions à l'analyse statique de programmes manipulant des tableaux

## Composition du Jury

| | |
|---|---|
| Jean-Claude Fernandez<br>*Université de Grenoble* | Président |
| Patrick Cousot<br>*École Normale Supérieure* | Rapporteur |
| Reinhard Wilhelm<br>*Universität des Saarlandes* | Rapporteur |
| Francesco Logozzo<br>*Microsoft Research* | Examinateur |
| Gaël Mulat<br>*The MathWorks* | Examinateur |
| Mihaela Sighireanu<br>*Université Paris Diderot* | Examinateur |
| Nicolas Halbwachs<br>*Centre National de la Recherche Scientifique* | Directeur |

« Alors commençaient les luttes,
les larmes, la persuasion,
l'abstraction en somme. »

– Albert Camus, *La peste*

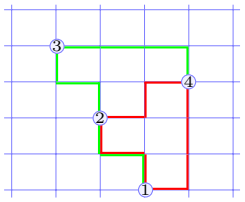
# Table des matières

# Chapitre 1

# Introduction

## 1.1 Contexte et motivations

Le bogue est inhérent à l'informatique telle qu'on la connaît, comme le rappelle simplement G. Berry [Ber08]. Le matériel, rigide (*hardware* en anglais), ne fait qu'exécuter à la lettre la suite d'ordres simples définie par le logiciel, à l'inverse malléable (*software*). G. Berry prend pour exemple le programme ci-contre, en vert, ayant pour propriété de visiter les points numérotés dans l'ordre. Ce programme est défini sur l'ensemble d'ordres : continuer tout droit (T), continuer à droite (D), continuer à gauche (G). Glissons une erreur, et le programme obtenu en rouge a un comportement totalement différent de celui attendu : aucun mécanisme de correction n'intervient comme on l'observe dans les systèmes génétiques par exemple.




vert : TGDTGDDTTD
rouge : TGD*D*GDDTTD

Écrire un programme correct est une tâche complexe, ardue. Ainsi, une erreur minime peut avoir de grandes conséquences. Mais là n'est pas la seule difficulté. Cette tâche nécessite des actions tout au long du développement du programme. On peut distinguer principalement trois activités durant ce développement : la conception, la mise en œuvre et la validation.

Pendant le temps de conception, la spécification du programme, selon laquelle on entendra sa correction, est écrite. Il est difficile de mettre à jour les hypothèses faites sur l'environnement d'exécution du programme et de définir toutes les propriétés souhaitées lors de son exécution. En effet, pour le programme ci-dessus, le dysfonctionnement le plus grave est peut-être de revenir au point de départ, comme le fait le programme rouge. Or cette propriété, « ne pas revenir au point 1 », était peut-être si évidente qu'elle n'a pas été spécifiée.

Pendant le temps de mise en œuvre, le programme est implanté selon sa spécification. Tout programmeur sait qu'il va passer une partie importante de ce temps à vérifier que son développement est exempt d'erreur et, lorsqu'il découvre que ce n'est pas le cas, à corriger ces erreurs. Ces difficultés n'ont pas de raison de faiblir puisque la complexité des logiciels, où le programmeur ajoute, modifie, réutilise des programmes, semble suivre la loi exponentielle de Moore.

Enfin, durant le temps de validation, le programme est comparé à ses spécifications. La plupart du temps ceci est fait en l'exécutant dans de très nombreuses situations. Cependant, les programmes ne sont pas continus : une exécution n'ayant révélé aucun bogue ne dit rien sur le comportement du programme dans une situation très similaire. Or en pratique, seulement une infime partie des comportements d'un programme peut être testée. Ainsi, générer automatiquement un ensemble de situations tel que, toutes exécutions confondues, chaque instruction du programme sera au moins exécutée une fois, est un objectif jugé très difficile [Jac09].

Maints domaines de recherches couvrent ces différentes activités. Le concours de tous est nécessaire pour que soit améliorée la correction des programmes. Aucun domaine ne doit être minimisé au prétexte que les outils qu'il propose sont informels, comme c'est le cas de nombreux outils de génie logiciel. Parmi tous ces domaines de recherche, on trouve celui de la vérification de programmes qui apporte des solutions particulières de validation. Cette thèse se situe dans ce domaine et c'est dans ce contexte très général qu'elle apporte une contribution (Sec. 1.4).

Les techniques de vérification se distinguent des autres techniques de validation par les méthodes formelles sur lesquelles elles reposent. Ces méthodes permettent de donner une sémantique plus ou moins précise à un programme en des termes mathématiques. Dès lors, les techniques de vérifications peuvent, sans jamais exécuter un programme, raisonner sur des ensembles infinis de ses comportements. Elles peuvent ainsi *prouver* qu'un programme vérifie une propriété dans *toutes* les situations d'exécution qu'autorise sa spécification. Cependant, ces techniques se heurtent au théorème de Rice [Ric53] qui nous apprend que toute propriété intéressante sur un programme est indécidable. On distingue alors deux grandes méthodes de vérification :

- les méthodes déductives. Ces méthodes permettent pour un programme donné de produire une démonstration qu'il vérifie les propriétés exigées par sa spécification. L'intervention d'utilisateurs experts pour guider la génération de la preuve est nécessaire. Leur travail est allégé par la génération automatique, à partir de spécifications formelles, d'obligations de preuves intermédiaires. Cependant, quand bien même une grande partie de ces obligations de preuves peut être prouvée automatiquement, ces méthodes restent tout à fait semi-automatiques, et fort coûteuses [1].
- les méthodes algorithmiques. Contrairement aux méthodes précédentes, interactives, elles visent à prouver quasi-automatiquement des propriétés du programme. Certaines de ces méthodes ne s'intéressent donc qu'aux cas décidables, mais leur champ d'application est très limité. Pour le reste, ces méthodes doivent considérer des approximations des comportements du programme pour atteindre leur but. Ces approximations sont conservatives, c'est-à-dire telles que si elles vérifient une propriété il en est de même pour les comportements réels du programme. L'inconvénient étant que l'approximation peut être la cause d'échec de la méthode. Dès lors chaque méthode vise la preuve d'une classe de propriétés particulières, pour lesquelles l'approximation sera minimisée. Cette spécialisation porte aussi, parfois,

1. Le logiciel pilotant les rames de la ligne Météor du métro parisien a en partie été écrit avec une de ces méthodes, la méthode B [Abr96]. Bien qu'il ait fallu 600 hommes-année pour écrire quelques 80 000 lignes de code C, Matra Transport semble avoir été convaincu à l'époque des bénéfices de cette méthode pour le développement d'un tel logiciel critique [BBFM99].

sur une forme particulière des programmes [2].

Ces méthodes algorithmiques recoupent ce que l'on appelle les analyses statiques. Une analyse statique cherche à *découvrir* certaines informations sur un programme, là encore sans l'exécuter. Ces informations sont utilisées par exemple pour transformer le code d'un programme, en analyser les performances, etc. Lorsque c'est pour assurer qu'une propriété est vérifiée par un programme, l'analyse statique est la pièce maîtresse d'une de ces méthodes algorithmiques de vérification que l'on vient de décrire. Ces analyses statiques sont l'objet général de cette thèse.

## 1.2 Sujet de la thèse

Deux sujets ont été abordés durant cette thèse avec, pour chacun, l'objectif de concevoir une nouvelle analyse statique. Ces deux analyses ont des points communs : il s'agit d'analyses statiques de programmes impératifs, correctes, découvrant un invariant de ces programmes. Par correctes, on exprime que les approximations de l'analyse sont conservatives. Par invariant, on entend que l'analyse associe à chaque point de contrôle du programme une propriété vérifiée par l'ensemble des états mémoires qu'atteint le programme en ce point de contrôle.

Les deux sujets de cette thèse découlent de la décomposition d'un même but : découvrir automatiquement des invariants dont les propriétés portent sur le contenu des tableaux, une structure de donnée couramment utilisée. Les éléments de cette structure sont adressés par des entiers. Ainsi, le contenu d'un tableau dépend de propriétés numériques, ce qui fait toute la difficulté de son analyse.

**Non-égalités numériques** D'ailleurs, jusqu'à ces dernières années, la plupart des travaux sur les tableaux se sont seulement intéressés à l'analyse des variables entières du programme utilisées pour accéder aux éléments des tableaux. Par exemple, pour vérifier qu'un programme n'essaye jamais d'accéder à un élément hors des bornes d'un tableau, ce qui aurait pour conséquence une erreur à l'exécution.

Un autre problème évident de l'adressage numérique, autre que le problème de la détection du dépassement des bornes, est le phénomène d'alias. Celui-ci est important pour toute analyse du contenu des tableaux. En effet, supposons qu'une analyse découvre la propriété suivante à un point de contrôle sur le tableau $a$ : la cellule ayant pour indice la valeur de la variable $i$ contient la lettre A. Si, de plus, les variables entières $i$ et $j$ vérifient $i = j$, l'analyse qui découvre ce dernier invariant peut conclure que $a[i]$ et $a[j]$ sont des alias d'une même cellule. Cette analyse peut alors précisément déduire les modifications à apporter sur sa connaissance de $a[i]$ lorsque le programme manipule $a[j]$, par exemple si l'instruction suivante est l'affectation $a[j] := \mathrm{B}$. Si les variables $i$ et $j$ vérifient plutôt que $i \neq j$, *i.e.* que $i$ et $j$ ne sont jamais des alias, c'est tout aussi intéressant pour l'analyse du contenu des tableaux. Si l'analyse le découvre, alors après l'instruction que l'on vient d'évoquer elle peut déduire que $a[i]$ n'a pas été modifiée, qu'elle contient toujours la lettre A au point de contrôle suivant. Lorsque l'analyse numérique n'a pas pu découvrir

2. Un exemple de cette double spécialisation est le logiciel Astrée. Celui-ci utilise plusieurs méthodes algorithmiques pour vérifier automatiquement, et en quelques heures, l'absence d'erreur à l'exécution des programmes de contrôles-commandes des Airbus A340 et A380 (entre 100 000 et 300 000 lignes de code) [BCC+03].

si les variables $i$ et $j$ étaient dans l'une de ces deux situations, d'alias ou de non-alias, par exemple à cause des approximations qu'elle met en œuvre, l'analyse du contenu des tableaux est imprécise. Après l'instruction d'affectation ci-dessus, l'analyse doit répondre pour rester correcte que $a[i]$ contient soit A soit B au point de contrôle suivant.

Il existe de nombreuses analyses numériques qui découvrent des égalités. Plusieurs autres, parfois les mêmes, découvrent des propriétés plus fortes que la non-égalité de deux variables, par exemple que $j \leq i - 1$. D'ailleurs ces propriétés peuvent être bien plus intéressantes pour l'analyse du contenu des tableaux. Cependant, aucune analyse ne s'intéresse spécialement aux propriétés de non-égalité, et donc aucune n'est capable de conserver directement et de raisonner efficacement avec de telles propriétés.

C'est le premier sujet que l'on a abordé : concevoir une analyse visant une classe de propriétés incluant des inégalités et des non-égalités, pour pouvoir conserver toute propriété d'alias et de non-alias, et pouvoir en déduire. Cette analyse ne devant être qu'une composante d'une analyse du contenu des tableaux, que nous voulons efficace, ses capacités de déduction et d'expressivité devront donc être limitées en conséquence.

**Contenu des tableaux** Les analyses du contenu des tableaux sont moins nombreuses dans la littérature. Surtout, dès lors qu'un utilisateur souhaite avoir des résultats intéressants, il n'a à sa disposition que des analyses semi-automatiques.

Prenons l'exemple ci-contre du tri par insertion du contenu du tableau $a$. L'invariant que l'on souhaiterait découvrir en tête de boucle externe (l. **3**) est « le tableau $a$ est trié jusqu'à la cellule $i - 1$ et la valeur de $i$ est comprise entre 2 et $n + 1$ ». Cette propriété peut s'écrire $2 \leq i \leq n + 1 \wedge \forall \ell, 2 \leq \ell < i \Rightarrow a[\ell - 1] \leq a[\ell]$. Comme on pouvait s'y attendre, ces analyses doivent manipuler des propriétés quantifiées non triviales. Or l'expressivité de ce type de propriétés est très riche et les déductions possibles entre de telles propriétés sont nombreuses et complexes. Ainsi, lorsqu'une analyse vise une large classe de propriétés sur le contenu des tableaux, elle doit à un moment de son calcul fixer les propriétés quantifiées vers lesquelles elle dirige son inférence. Pour notre exemple, elle pourrait très bien se diriger vers la propriété équivalente suivante, utilisant deux quantificateurs : $2 \leq i \leq n + 1 \wedge \forall \ell_1, \ell_2, 1 \leq \ell_1 < \ell_2 < i \Rightarrow a[\ell_1] \leq a[\ell_2]$. Ce choix d'une propriété plus riche pourrait l'amener à diverger. Mais un tout autre choix pourrait l'amener à échouer certainement.

```
1  assert n ≥ 1 ;
2  i := 2 ;
3  while i ≤ n do
4  |  x := a[i] ;
5  |  j := i − 1 ;
6  |  while j ≥ 1
   |  and a[j] > x do
7  |  |  a[j + 1] := a[j]
8  |  |  j := j − 1
9  |  a[j + 1] := x ;
10 |  i := i + 1
```

Tri par insertion de $a$

C'est une des raisons principales qui obligent les analyses existantes à demander à l'utilisateur d'annoter le point de contrôle par le prédicat composant la partie droite (de l'implication) de la propriété quantifiée recherchée, pour notre exemple le prédicat $a[\ell - 1] \leq a[\ell]$ ou le prédicat $a[\ell_1] \leq a[\ell_2]$. Comme on le voit ce prédicat guide aussi l'analyse en fixant de fait le nombre de quantificateurs. Cependant, toutes les analyses existantes qui essayent d'analyser le tri par insertion, avec l'aide de ce seul prédicat annoté échouent [GRS05, JM07, GMT08].

Les analyses du contenu des tableaux se heurtent à d'autres difficultés. Il leur faut identifier les parties symboliques des tableaux sur lesquelles une propriété spécifique existe (les parties gauches des propriétés quantifiées ci-dessus). Ceci peut être très difficile à faire

au fil du calcul alors que ces parties symboliques peuvent être complexes et nécessiter de différencier certains cas d'alias. Par exemple, à la sortie de la boucle interne de notre exemple (l. **9**), une analyse précise devrait être capable de découvrir que le contenu de $a[i]$ vis-à-vis de $x$ est différent selon la valeur de j : $\forall \ell, \ell = i = j + 1 \Rightarrow a[\ell] = x$ (la clé x est déjà en bonne place, la boucle interne n'a pas été exécutée) et $\forall \ell, \ell = i > j+1 \Rightarrow a[\ell] < x$.

Dans les faits, les analyses automatiques du contenu de tableaux ne visant pas une classe très spécifique de propriétés n'arrivent à découvrir quasiment que des propriétés unaires. Par unaire, on entend une propriété n'exprimant pas de relation entre des cellules du tableau. Par exemple, des propriétés sur les bornes du contenu d'un tableau numérique : $\forall \ell, 1 \leq \ell \leq n \Rightarrow 5 \leq a[\ell] \leq 10$ [BCC$^+$03, GDD$^+$04, GRS05].

De ce constat découle notre sujet : concevoir une analyse automatique découvrant des propriétés sur le contenu des tableaux unaires comme relationnelles, jusqu'à des propriétés fonctionnelles de programmes dit « simples ». Ces programmes ne manipulent que des tableaux statiques, à une dimension, qu'ils indicent par une constante ou la somme d'une variable et d'une constante, et qu'ils traversent à l'aide de boucles simples (incrémentation/décrémentation), possiblement imbriquées.

Cette classe de programmes est déjà large et certains de ces programmes sont complexes. On peut vérifier que le tri par insertion fait partie de cette classe. C'est aussi le cas de beaucoup d'autres algorithmes d'organisation du contenu : tri à bulle, phase de segmentation du tri rapide, drapeau hollandais (Fig, 7.5, 7.8(b), B.3, B.4). On trouve aussi dans cette classe des algorithmes d'initialisation plus ou moins complexe : initialisation selon la valeur d'un autre tableau, ou de l'indice de la cellule initialisée, ou encore des cellules du même tableau (Fig. 7.3, 7.8(c), B.1). Enfin, on trouve aussi des algorithmes de recherche : recherche de l'élément maximum, de l'indice de la première valeur non nulle, d'un élément particulier d'un tableau dont la borne est gardée par une sentinelle, recherche dichotomique (Fig 7.4, 7.8(d), B.2), etc. On souhaite donc que notre analyse soit polyvalente et suffisamment précise pour découvrir les fonctionnalités d'algorithmes aux objectifs différents.

Enfin, les sujets de cette thèse ne sont pas bornés à la conception d'analyses statiques, mais portent aussi sur la validation expérimentale de ces analyses : ainsi un dernier objectif est la conception d'un prototype d'analyseur statique dont les résultats seront examinés.

## 1.3 Cadre théorique

On choisit d'exprimer nos analyses statiques grâce à la théorie de l'interprétation abstraite [CC77]. Une des raisons est que cette théorie permet de définir des analyses correctes par construction, en respectant un ensemble somme toute restreint d'hypothèses. Une autre raison est que ces hypothèses étant simples et formelles, ce cadre théorique amène naturellement à formuler l'analyse de manière claire et rigoureuse.

Comme nous l'avons dit dans la première section, une analyse statique repose sur une sémantique formelle des programmes à analyser. L'interprétation abstraite exprime toute sémantique comme une équation de point fixe. On détaille comment on s'insère en pratique, et pour notre problème, dans ce cadre unifié.

On peut définir la sémantique de la plupart des programmes par le comportement d'une machine lorsqu'elle les exécute. Cette sémantique se compose alors de l'ensemble des états que va prendre la machine et des transitions que la machine va effectuer entre ces états, transitions induites par les opérations du programme. C'est ce qu'on appelle une sémantique opérationnelle du programme, que l'on décrit par un système de transitions (Déf. 1.1). Pour un programme impératif non procédural, il faut imaginer qu'un état de la machine est un simple couple liant un point de contrôle du programme et un état mémoire.

Définition 1.1 (système de transitions). *Un système de transitions est un triplet* $(S, \tau, S_{init})$ *où* $S$ *est un ensemble d'états, possiblement infini,* $\tau \subset S \times S$ *la relation de transition entre un état et ses successeurs et* $S_{init} \subset S$ *l'ensemble des états initiaux.*

Vu que nous souhaitons découvrir des invariants des programmes, nous pouvons nous contenter de raisonner sur les états accessibles (Déf. 1.2), à partir des états initiaux, de leurs systèmes de transitions.

Définition 1.2 (accessibilité et co-accessibilité). *Soit un système de transition* $(S, \tau, S_{init})$. *Un état* $s' \in S$ *(resp.* $s \in S$*) est dit accessible (resp. co-accessible) depuis un état* $s$ *(resp.* $s'$*) s'il existe un chemin de* $s$ *à* $s'$ *selon* $\tau$.

*Soit* $X \subseteq S$. *On note* $Acc(X)$ *(resp.* $CoAcc(X)$*) l'ensemble des états accessibles (resp. co-accessibles) à partir de l'ensemble des états de* $X$.

L'ensemble des états $Acc(S_{init})$ est une sémantique dite d'atteignabilité du programme. On voit que cette sémantique peut se définir par un point fixe. En notant *succ(X)* l'ensemble des états successeurs des états $X$ dans le système de transition, *i.e.* $\{s' \mid \exists s \in X, (s, s') \in \tau\}$, $Acc(S_{init})$ est le plus petit point fixe de la fonction $post(X) = S_{init} \cup succ(X)$. On note *lfp*(*post*) ce plus petit point fixe (*least fixed point* en anglais).

Une fois dans ce formalisme, la définition d'une approximation conservative de cette sémantique, dite concrète (même si, en effet, elle est déjà une première abstraction de la sémantique opérationnelle), est directe. Il suffit de définir des abstractions, comme l'exige l'interprétation abstraite (Ch. 2), des deux composantes du point fixe : son domaine de calcul et sa fonction. Pour le domaine, il faut définir un *domaine abstrait* $\mathcal{D}^\sharp$, c'est-à-dire une classe $C$ de propriétés munie d'opérateurs ensemblistes pour manipuler ces propriétés. Pour la fonction, la définition d'une fonction abstraite $post^\sharp$ nécessite en pratique de ne donner qu'une abstraction des opérations élémentaires du langage de programmation utilisé. Ainsi, seules quelques opérations abstraites sont à définir.

Le plus petit point fixe de $post^\sharp$ sur $\mathcal{D}^\sharp$ est ce que l'on appelle la sémantique abstraite. Si l'on a respecté les hypothèses qu'impose la théorie sur leur définition vis-à-vis de la sémantique concrète des opérations du programme et du domaine concret, cette sémantique est une abstraction correcte de la sémantique concrète.

Une analyse statique d'un programme pour la classe de propriétés $C$ n'est que le calcul de cette sémantique abstraite. Celui-ci est envisageable dès lors que les éléments de $\mathcal{D}^\sharp$ sont représentables efficacement et ses opérateurs calculables efficacement (ce qui n'est pas le cas par exemple pour le domaine concret $\mathcal{D}$, en l'occurrence l'ensemble des parties de l'ensemble $S$ des états) ; et que les opérations abstraites sont

également calculables efficacement. L'interprétation abstraite explique comment effectuer ce calcul par résolution itérative. Ce calcul est approché car la résolution itérative d'une équation de point fixe sur les domaines utilisés ne converge pas en général. Il est donc demandé un opérateur d'extrapolation sur $\mathcal{D}^\sharp$, qui est la dernière approximation qui devra être définie.

Ainsi, construire une analyse statique dans ce cadre théorique se résume à fournir un domaine abstrait, une abstraction des opérations du langage de programmation utilisé pour écrire les programmes à analyser, et un opérateur d'extrapolation. L'interprétation abstraite assure alors la terminaison et la correction de cette analyse statique.

## 1.4 Contributions

D'un point de vue théorique, nous apportons deux nouvelles analyses statiques automatiques : l'une numérique, ciblant des propriétés de non-égalités (VMCAI'07, [PH07]) ; l'autre symbolique, ciblant des propriétés relationnelles sur le contenu des tableaux (PLDI'08, [HP08]). Alors que la première analyse ne nous a pas donné satisfaction expérimentalement, la seconde a montré sur un ensemble d'exemples hétérogène une précision et des performances bien au-delà de celles des analyses existantes.

Ces expérimentations ont été faites avec un prototype d'analyseur que nous avons développé, baptisé Enkidu. Cette contribution interne au laboratoire a permis notamment l'expérimentation des travaux d'une autre équipe. Ces travaux auxquels nous avons participé portent sur la construction automatique de preuves de correction des invariants découverts par notre seconde analyse (prochainement soumis à IWIL'10, [GPP10]). Enfin, nous avons adapté et implanté notre seconde analyse dans une version en développement de l'analyseur par interprétation abstraite de *Microsoft Research*, Clousot. Ainsi, les retombées pratiques de cette thèse sont principalement pour l'analyse de programmes manipulant des tableaux.

**Une analyse numérique découvrant des non-égalités** Cette première analyse permet de découvrir des propriétés des formes suivantes :

$$(\pm x \leq c) \quad (x - y \leq c) \quad (x \neq 0) \quad (x \neq y)$$

où $x$ et $y$ sont des variables du programme et $c$ une constante. Cette analyse est définie *via* un nouveau domaine abstrait, reprenant l'algorithmique de celui des zones [Dil89], domaine capable de découvrir des inégalités des formes ci-dessus.

*A contrario* du domaine abstrait des zones, la complexité des opérations de notre domaine diffère selon l'ensemble mathématique dans lequel les variables prennent leurs valeurs. Si c'est dans $\mathbb{Z}$, le problème de la satisfaisabilité d'un élément du domaine abstrait (conjonction de contraintes de la forme ci-dessus) est NP-complet [RH80] de par la présence des non-égalités. Ainsi, pour une précision maximale les opérations usuelles du domaine ont une complexité exponentielle dans le nombre de variables. Par contre, si c'est dans $\mathbb{Q}$ ou $\mathbb{R}$ que les variables prennent leur valeurs, la complexité des opérations usuelles est en $\mathcal{O}(n^4)$ (à comparer à la complexité cubique de celles du domaine abstrait des zones). Une version approximée des opérations pour le cas discret est donnée à partir de l'algorithmique du cas dense, avec une complexité identique, $\mathcal{O}(n^4)$.

Côté pratique, nous n'avons trouvé qu'un seul exemple réel, un algorithme d'exclusion mutuelle de deux programmes, pour lequel ce nouveau domaine découvre une non-égalité entre variables. En tous les cas cette analyse n'a pas atteint son objectif, puisqu'elle ne s'est pas avérée utile à notre analyse du contenu des tableaux. Ceci étant, ce domaine s'avère tout de même utile pour conserver des non-égalités triviales dans le code des programmes analysés, par exemple, un programme recherchant simplement un élément dans un tableau (**while** $a[i] \neq \ldots$ **do**).

**Une analyse du contenu des tableaux** Cette seconde analyse permet de découvrir des propriétés quantifiées de la forme suivante, où $\varphi$ et $\psi$ sont respectivement des propriétés des domaines abstraits $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$, paramètres de l'analyse :

$$\forall \ell,\ \varphi(\ell, i, j, n \ldots) \Rightarrow \psi(a[\ell + c], b[\ell + c'], \ldots, x, y, \ldots)$$

où $i$, $j$, $n$ sont des variables numériques entières du programme, $a$, $b$ sont des tableaux du programme, $x$, $y$ sont des variables scalaires du programme, du même type que le contenu des tableaux et $c$, $c'$ sont des constantes. L'analyse effectuant aussi de manière concomitante les analyses de $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$, elle découvre également des propriétés des formes suivantes : $\varphi(i, j, n, \ldots)$ et $\psi(x, y, \ldots)$.

Les tableaux apparaissant dans les propriétés ne pouvant être indicés que *via* une variable quantifiée $\ell$, c'est uniquement des relations *cellule par cellule* qui peuvent être découvertes par l'analyse. Des exemples de telles propriétés relationnelles sont (où $\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{\mathbb{K}}$ est le domaine abstrait des zones) :

- $\forall \ell, 1 \leq \ell \leq n \Rightarrow a[\ell] = b[\ell]$
  invariant à la sortie d'une copie du tableau $b$ dans le tableau $a$ ;
- $\forall \ell, 1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] \leq x$
  invariant de boucle d'une recherche du maximum du tableau $a$, stocké dans $x$ ;
- $\forall \ell, 2 \leq \ell \leq n \wedge j + 1 < \ell < i \Rightarrow x < a[\ell - 1] \leq a[\ell]$
  un des invariants de la boucle interne du tri par insertion.

Un exemple de propriété que l'analyse ne peut pas découvrir :

- $\forall \ell, 1 \leq \ell \leq n \wedge \ell \leq i \Rightarrow a[\ell] \leq a[i]$
  un des invariants de la boucle interne du tri à bulle.

Un élément important au sujet de notre analyse est qu'elle procède en deux temps : une pré-analyse détermine les parties gauches des propriétés quantifiées qui pourront être découvertes en chaque point de contrôle par l'analyse principale. Cette pré-analyse est principalement syntaxique et inspirée de [GRS05]. Cette analyse et l'analyse principale sont définies par deux domaines abstraits.

L'approche que nous avons suivie pour que notre analyse soit automatique transparaît dans cette présentation factuelle. D'une part, nous avons choisi de découvrir des propriétés avec une seule variable quantifiée, éliminant les problèmes évoqués précédemment (Sec. 1.2). Finalement, cette restriction n'empêche pas d'exprimer de nombreuses propriétés intéressantes. D'autre part, nous sommes partis du principe qu'il était possible de découvrir indépendamment les parties symboliques sur lesquelles des propriétés particulières existent. Ainsi, en ajustant *a priori* l'expressivité de l'analyse principale au programme sous-analyse, celle-ci peut être vraiment efficace et se concentrer sur la découverte des propriétés existantes sur les parties symboliques désignées, qui elles sont moins évidentes. Par exemple, le tri par insertion ne compare jamais deux cellules $a[] \leq a[]$,

alors que sa boucle externe sur $i$ laisse supposer un invariant sur la partie symbolique $2 \leq \ell < i$. Bien entendu, le risque de cette approche est de voir l'analyse principale échouer sur un programme dont elle aurait pu découvrir l'invariant si la pré-analyse lui avait donné les parties symboliques adéquates.

En pratique, les expérimentations ont montré que notre analyse était capable de découvrir rapidement et automatiquement les invariants fonctionnels de programmes simples, comme ceux évoqués ci-dessus (copie de tableau, recherche du maximum, etc.), mais aussi de programmes plus complexes, comme le tri par insertion ou la phase de segmentation du tri rapide, qui nous ont apparu représentatifs de manipulations non-triviales du contenu des tableaux. Notre analyse a aussi montré sa capacité à déduire des propriétés numériques à partir des propriétés découvertes sur le contenu des tableaux, en découvrant pour l'exemple ci-contre, d'une traversée de tableau gardée par une sentinelle, que le tableau $a$ est toujours accédé dans ses bornes (*i.e.* $i \leq n$ en tête de boucle). Ces résultats sont excellents vis-à-vis des analyses existantes.

```
1 assert n ≥ 1 ;
2 a[n] := x ;
3 i := 1 ;
4 while a[i] ≠ x do
5 |   i := i + 1
```

Recherche avec sentinelle

Par ailleurs, j'ai implanté avec Francesco Logozzo une version moins précise de cette analyse dans Clousot, où les tableaux ne peuvent apparaître dans les propriétés qu'indicés par $\ell$ (et non par $\ell + c$). Les seules expériences que nous avons eu le temps de mener laissaient penser que les difficultés rencontrées pour que cette analyse passe à l'échelle pouvaient être surmontées. Clousot analysait tout de même la quasi-totalité des 21000 fonctions que compte `mscorlib`[3] en un temps significativement plus élevé, mais pas rédhibitoire. Enfin, ce fut également l'occasion de proposer une analyse des itérateurs, que l'on trouve dans plusieurs langages généralistes et qui se comportent finalement comme la traversée d'un tableau.

**Un banc d'essai pour des analyses statiques** Notre prototype Enkidu repose sur le calculateur de point fixe développé par Bertrand Jeannet [Jea10]. Enkidu est un analyseur très classique, hors de nos deux analyses qu'il fournit. Cependant, il a permis à deux équipes de Verimag d'expérimenter leurs travaux sur le contenu des tableaux suscités par nos propres travaux.

C'est avec cet outil que Valentin Perrelle a expérimenté dans notre équipe une analyse de permutation du contenu des tableaux [PH10] qu'il a défini. Enfin, Manuel Garnacho a instrumenté Enkidu avec notre collaboration, selon une méthode mise au point dans une autre équipe, qui permet aux résultats de notre analyse du contenu des tableaux d'être agrémentés de preuves de correction, vérifiables automatiquement dans Coq. Les expérimentations que nous avions utilisées pour valider notre analyse ont quasiment toutes été reconduites avec succès avec cette version enrichie d'Enkidu.

## 1.5 Plan du mémoire

L'organisation du manuscrit découle de la volonté de ne pas entremêler *strictement* les contributions de la thèse avec l'explication du cadre théorique et des travaux existants.

3. Librairie principale de la bibliothèque logicielle *Microsoft* .Net.

**État de l'art** Dans cette première partie, on trouve trois chapitres. Le premier chapitre est une introduction à l'interprétation abstraite (Ch. 2). Après avoir détaillé les principes de cette théorie, nous décrivons un analyseur d'invariance de programmes impératifs structurés. Générique et correct par construction, c'est cet analyseur qui sera instancié, dans la seconde partie, pour définir nos deux analyses statiques.

Le second chapitre est l'état de l'art correspondant à notre analyse de non-égalités numériques (Ch. 3). Nous présentons l'ensemble des analyses numériques existantes, et les critères qui les différencient. L'analyse des zones est décrite de manière détaillée puisque notre contribution repose sur celle-ci. Enfin, nous avons rassemblé des résultats, importants pour la suite, sur la satisfaisabilité de nombreux systèmes différents mêlant contraintes d'inégalité et contraintes de non-égalité.

Le dernier chapitre est l'état de l'art des analyses statiques visant le contenu des tableaux (Ch. 4). Nous définissons deux grandes classes d'invariants sur le contenu des tableaux, auxquelles on peut rattacher l'expressivité de toutes les analyses existantes. Chaque analyse est passée en revue, et ce, de manière très détaillée pour les analyses les plus automatiques, qui sont celles dont nous nous sommes inspirés mais aussi celles avec lesquelles nous comparons notre travail. Avant de conclure, nous nous intéressons brièvement à des travaux similaires sur des listes chaînées.

**Contributions** Cette seconde partie se compose d'une section préliminaire et de trois chapitres. En Préliminaires, nous présentons un langage impératif simple, mais adéquat pour illustrer nos analyses. Les programmes écrits dans ce langage sont compilés vers le langage d'entrée de l'analyseur générique décrit au Chapitre 2. Ainsi, nous obtenons notamment une sémantique concrète pour ces programmes, que l'on prendra pour référence pour définir formellement les abstractions et les opérations de nos deux analyses.

Les deux premiers chapitres de cette partie présentent, dans l'ordre, l'analyse de non-égalités numériques (Ch. 5) et l'analyse du contenu des tableaux (Ch. 6). Ils sont construits sur le même plan logique : présentation des objectifs ; description de l'expressivité des propriétés pouvant être découvertes et leur représentation ; lien avec la sémantique concrète ; et définition des opérations sémantiques abstraites.

Enfin, le dernier chapitre porte sur l'implantation de nos analyses statiques et les résultats expérimentaux. Une fois notre prototype Enkidu présenté, on discute ses résultats et particulièrement les succès comme les échecs de notre analyse du contenu des tableaux sur plusieurs programmes représentatifs.

S'en suit bien entendu une conclusion (Ch. 8). À sa suite, on trouve une courte Annexe, rassemblant quelques notions classiques, et une collection de programmes manipulant des tableaux, qui est un point de départ à un *benchmark* ambitieux pour les analyses statiques de ces programmes.

# Première partie

# État de l'art

# Chapitre 2

# L'interprétation abstraite

L'interprétation abstraite [CC77] est une théorie générale de l'approximation de sémantiques de programmes. Elle s'intéresse en premier lieu à la construction correcte de ces approximations et à leur calcul approché.

Pour ce faire, elle statue que toute sorte de sémantique – opérationnelle, dénotationnelle, axiomatique, etc – peut être exprimée comme un point fixe de fonctions monotones sur des structures partiellement ordonnées. Puis elle s'appuie sur des résultats fondamentaux de mathématiques discrètes, notamment les théorèmes de points fixes de Tarski et Kleene et les connexions de Galois pour formaliser qu'une sémantique est une approximation correcte d'une autre (Sec. 2.1.1). Enfin, elle apporte des techniques pour calculer une approximation de ces points fixes lorsque ceux-ci sont ceux de fonctions calculables et sont exprimés sur des structures représentables en machine (Sec. 2.1.2). Ainsi, on définit également l'interprétation abstraite comme une théorie de résolution approchée d'équations de points fixes.

Une fois ces principes expliqués, on rappelle comment les utiliser en pratique pour construire un analyseur statique, correct, générique et automatique, d'invariance de programmes impératifs (Sec. 2.2).

## 2.1 Principes

Le principe étant de plonger toute sémantique dans le cadre unique de points fixes de fonctions monotones sur des structures partiellement ordonnées il faut d'abord s'assurer de leurs existences. C'est un résultat qu'énonce, entre autres, le théorème de point fixe de Tarski [Tar55].

THÉORÈME 2.1 (points fixes de Tarski). *Soit* $(\mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ *un treillis complet et* $f : \mathcal{D} \to \mathcal{D}$ *une fonction monotone. L'ensemble des points fixes de* $f$ *est un treillis complet et, pour* $x \in \mathcal{D}$ *:*

$$lfp_x f = \sqcap\{y \in \mathcal{D} \mid x \sqsubseteq y \sqsupseteq f(y)\} \qquad gfp_x f = \sqcup\{y \in \mathcal{D} \mid x \sqsupseteq y \sqsubseteq f(y)\}.$$

Un treillis complet ne pouvant être vide, le théorème établit bien l'existence d'au moins un point fixe de $f$, de surcroît d'un plus petit point fixe. Ce dernier est défini ici comme la borne inférieure de tous les post-points fixes de $f$, une définition non constructive. Lorsque nous nous intéresserons au calcul pratique de ces points fixes, nous utiliserons plutôt des caractérisations de ceux-ci par des séquences d'itérations.

### 2.1.1 Abstraction correcte de sémantiques

Différents formalismes, exigeant des hypothèses plus ou moins fortes, ont été définis pour assurer la correction des approximations d'une sémantique. On appelle domaine concret et domaine abstrait les structures partiellement ordonnées sur lesquelles sont définies la sémantique concrète et celle qui l'approxime, la sémantique abstraite. De même, on parle de fonction concrète et de fonction abstraite, pour les fonctions des points fixes définissant la sémantique concrète et la sémantique abstraite.

Ces formalismes nécessitent que soit définit pour tout élément du domaine abstrait quels éléments du domaine concret ils approximent correctement. Cette relation d'approximation correcte $R$ doit respecter les ordres partiels des deux domaines : si l'on a $x^\sharp$ qui approxime $x^\flat$, ce que l'on note $R(x^\flat, x^\sharp)$, alors $x^\sharp$ approxime également tout élément plus petit que $x^\flat$, *i.e.* $R(y^\flat, x^\sharp)$ pour tout $y^\flat \sqsubseteq^\flat x^\flat$ ; de même $R(x^\flat, y^\sharp)$ pour tout $y^\sharp \sqsupseteq^\sharp x^\sharp$.

Enfin ces formalismes expriment diverses conditions, dont des conditions sur la fonction abstraite, pour assurer la correction de la sémantique abstraite.

#### 2.1.1.1 Abstraction sous connexion de Galois

Un premier formalisme est celui exigeant l'existence d'une connexion de Galois, propriété dont nous rapportons ici la définition donnée dans l'article fondateur [CC77]. Il s'agit du cas où l'on sait faire correspondre à tout élément du domaine abstrait le plus grand élément du domaine concret qu'il approxime correctement, et dualement que l'on sait faire correspondre à tout élément du domaine concret le plus petit élément du domaine abstrait qui l'approxime correctement.

Définition 2.1 (connexion de Galois). *Soit deux ensembles partiellement ordonnés $(\mathcal{D}^\flat, \sqsubseteq^\flat)$ et $(\mathcal{D}^\sharp, \sqsubseteq^\sharp)$. Le couple de fonctions $(\alpha, \gamma)$, où $\alpha : \mathcal{D}^\flat \to \mathcal{D}^\sharp$ et $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$, forme une connexion de Galois entre $\mathcal{D}^\flat$ et $\mathcal{D}^\sharp$ si :*

$$\forall x^\flat \in \mathcal{D}^\flat, \forall x^\sharp \in \mathcal{D}^\sharp, \quad \alpha(x^\flat) \sqsubseteq^\sharp x^\sharp \Leftrightarrow x^\flat \sqsubseteq^\flat \gamma(x^\sharp).$$

*On appelle $\alpha$ la fonction d'abstraction et $\gamma$ celle de concrétisation.*

De cette propriété découle, entre autres, la monotonicité des fonctions $\alpha$ et $\gamma$, l'extensivité de $\gamma \circ \alpha$, c'est-à-dire $x^\flat \sqsubseteq^\flat (\gamma \circ \alpha)(x^\flat)$ pour tout $x^\flat \in \mathcal{D}^\flat$, qui rend compte de la perte d'information induite par l'approximation et la rétractation de $\alpha \circ \gamma$, c'est-à-dire $(\alpha \circ \gamma)(x^\sharp) \sqsubseteq^\sharp x^\sharp$ pour tout $x^\sharp \in \mathcal{D}^\sharp$. Surtout l'existence d'une connexion de Galois est une hypothèse forte qui permet d'établir la notion de meilleure fonction d'abstraction de $f^\flat$ dans $\mathcal{D}^\sharp$. Aussi appelée abstraction standard, elle est définie comme $f^\sharp_{std} = \alpha \circ f^\flat \circ \gamma$.

Remarque. *Une conséquence de l'existence de la meilleure fonction d'abstraction est qu'elle permet de comparer formellement la précision de domaines abstraits liés par une connexion de Galois à un même domaine concret. On obtient alors un ordre partiel sur les abstractions par connexion de Galois de ce domaine [CC79b].*

**Condition de correction de la fonction abstraite** Dans ce cadre, toute fonction abstraite $f^\sharp$ est considérée comme une abstraction correcte de $f^\flat$ si et seulement si $f^\sharp \sqsupseteq^\sharp f^\sharp_{std}$ ; ou encore si et seulement si $f^\sharp$ est une approximation supérieure de $f^\flat$ : la concrétisation du résultat d'un calcul dans le domaine abstrait ($f^\sharp$) sur-approxime celui du calcul effectué, après concrétisation, dans le domaine concret ($f^\flat$).

Définition 2.2 (approximation supérieure). *Soit* $(\mathcal{D}^\flat, \sqsubseteq^\flat, \sqcup^\flat, \bot^\flat)$ *et* $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \bot^\sharp)$ *deux ordres partiels complets,* $f^\flat : \mathcal{D}^\flat \to \mathcal{D}^\flat$ *et* $f^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *deux fonctions monotones, et* $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$ *une fonction monotone de concrétisation. Alors,* $f^\sharp$ *est une approximation supérieure de* $f^\flat$*, si :*

$$\forall x^\sharp \in \mathcal{D}^\sharp, \quad (\gamma \circ f^\sharp)(x^\sharp) \sqsupseteq^\flat (f^\flat \circ \gamma)(x^\sharp).$$

*En cas d'égalité, on dira que* $f^\sharp$ *est une approximation exacte de* $f^\flat$*.*

**Transfert de point fixe** Une fois cette condition de correction remplie par $f^\sharp$, le résultat fondamental suivant assure que la sémantique abstraite est bien une approximation correcte de la sémantique concrète.

Théorème 2.2 (transfert des points fixes de Tarski). *Soit* $(\mathcal{D}^\flat, \sqsubseteq^\flat, \sqcup^\flat, \bot^\flat)$ *et* $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \bot^\sharp)$ *deux ordres partiels complets liés par une connexion de Galois* $(\alpha, \gamma)$ *et* $f^\flat : \mathcal{D}^\flat \to \mathcal{D}^\flat$ *et* $f^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *deux fonctions monotones telles que* $f^\sharp$ *est une approximation supérieure de* $f^\flat$*. Alors, pour* $x^\sharp \in \mathcal{D}^\sharp$*, si* $\gamma(x^\sharp)$ *est un pré-point fixe de* $f^\flat$*, on a :*

$$lfp_{\gamma(x^\sharp)} f^\flat \sqsubseteq^\flat \gamma(lfp_{x^\sharp} f^\sharp).$$

Le calcul de la sémantique concrète peut alors être transféré à celui de la sémantique abstraite. On notera que même en choisissant $f^\sharp_{std}$ comme fonction abstraite, la concrétisation de son plus petit point fixe peut être moins précise que le plus petit point fixe de la fonction concrète.

#### 2.1.1.2 Abstraction sous fonction de concrétisation

Parfois il n'est pas facile de donner une connexion de Galois. Par conséquent on souhaite travailler dans un formalisme où seule l'une des deux fonctions, d'abstraction ou de concrétisation, est définie [CC92a]. Nous nous intéresserons ici au cas, plus usité en pratique, où la seule exigence est l'existence d'une fonction de concrétisation $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$ monotone.

Ne sachant pas exprimer quelle est la meilleure abstraction d'un élément concret, on perd la notion de meilleure fonction abstraite, ainsi que la possibilité de comparer la précision de deux sémantiques abstraites. Cependant, cela n'empêche pas d'utiliser la notion d'approximation supérieure (Déf. 2.2) comme condition de correction pour la fonction abstraite $f^\sharp$.

Avec ce formalisme la théorie ne peut pas donner un résultat d'approximation de la sémantique concrète par la sémantique abstraite. Cela dit la théorie va permettre d'assurer par une technique particulière, l'approximation dynamique de point fixe, que le calcul de la sémantique abstraite a pour résultat une approximation correcte de la sémantique concrète.

### 2.1.2 Calcul du point fixe

En pratique, pour calculer le plus petit point fixe d'une fonction monotone et continue, on utilise sa caractérisation due à Kleene [Kle52]. Le théorème de point fixe suivant

[CC79a] établit, comme le Théorème 2.1, l'existence de points fixes mais les caractérise par les itérations de Kleene [1].

Théorème 2.3 (points fixes de Kleene). *Soit $(\mathcal{D}, \sqsubseteq, \sqcup, \bot)$ un ordre partiel complet et $f : \mathcal{D} \to \mathcal{D}$ une fonction monotone et continue. Alors, pour $x \in \mathcal{D}$ pré-point fixe de $f$, la séquence infinie d'itérations de $f$ à partir de $x \in \mathcal{D}$ est stationnaire et :*

$$lfp_x f = \sqcup_{n \geq 0}\ f^n(x).$$

On semble donc détenir une méthode pour mener à bien le calcul de ces points fixes, pour peu que leurs fonctions composantes soient calculables et que les éléments du domaine $\mathcal{D}$ soient représentables en machine. Mais il reste que dans le cas général on ne sait pas calculer la limite d'une itération infinie.

#### 2.1.2.1 Calcul exact du point fixe

Dans le cas où le domaine sur lequel est défini le point fixe est de profondeur finie, par conséquence du Théorème 2.3 la séquence d'itération converge en un temps fini. Ce résultat s'applique évidemment aux domaines finis, comme par exemple le domaine des signes [CC76]. Parmi les domaines infinis vérifiant la condition précédente on peut citer le domaine des constantes [Kil73].

Quelques rares travaux ont présenté, sous certaines hypothèses, des algorithmes qui calculent exactement les plus petits points fixes de fonctions définies sur des domaines de profondeur infinie. C'est le cas pour les fonctions affines définies sur le domaine des intervalles [CC76], pour lesquelles [SW04] introduit une séquence d'itérations particulière – qui, dans les faits, reprend l'opérateur d'élargissement (*cf.* Déf. 2.3) standard de ce domaine, convergeant en un temps cubique.

#### 2.1.2.2 Approximation dynamique du point fixe

En général, pour les domaines de profondeurs infinie on ne peut garantir la convergence de la séquence d'itérations. Une solution générale consiste à extrapoler la limite de la séquence. On utilise pour cela un opérateur d'élargissement [CC76] qui, appliqué itérativement, donne une limite à une suite infinie au vu des ses premiers éléments.

Définition 2.3 (opérateur d'élargissement). *Soit $(\mathcal{D}, \sqsubseteq, \sqcup, \bot)$ un ordre partiel complet. Un opérateur d'élargissement est une fonction $\nabla : (\mathcal{D} \times \mathcal{D}) \to \mathcal{D}$ satisfaisant :*

1. $\forall x, y \in \mathcal{D}, \quad x \sqsubseteq x \nabla y \ \wedge \ y \sqsubseteq x \nabla y$ ;
2. *pour toute suite $(x_n)_{n \geq 0}$ dans $\mathcal{D}$, la suite croissante définie par*

$$y_0 = x_0, \quad y_{n+1} = y_n \nabla x_{n+1}$$

*converge en un nombre fini d'itérations.*

1. Le théorème cité ne fait pas l'hypothèse que la fonction $f$ est continue. La caractérisation du point fixe est alors généralisée par une séquence transfinie.

L'extrapolation consiste alors à utiliser l'opérateur d'élargissement entre chaque itéré de la séquence d'itération, assurant ainsi sa convergence. Le théorème suivant (Thm. 2.4) assure que le résultat obtenu est un post-point fixe de la fonction et donc une sur-approximation correcte de son plus petit point fixe.

Ce qui est particulièrement intéressant, c'est que cette approximation dynamique de point fixe s'associe bien aux abstractions correctes de sémantiques vues précédemment, qu'elles soient basées sur une connexion de Galois ou sur une fonction de concrétisation. En effet en effectuant une approximation dynamique du point fixe de la fonction abstraite on obtient une sur-approximation correcte de la sémantique concrète, c'est le second résultat énoncé par le théorème suivant [CC92b].

Théorème 2.4 (approximation de point fixe par élargissement). *Soit* $(\mathcal{D}^\flat, \sqsubseteq^\flat, \sqcup^\flat, \sqcap^\flat, \bot^\flat, \top^\flat)$ *et* $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp, \top^\sharp)$ *deux treillis complets,* $f^\flat : \mathcal{D}^\flat \to \mathcal{D}^\flat$ *et* $f^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *deux fonctions monotones telles que* $f^\sharp$ *est une approximation supérieure de* $f^\flat$*,* $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$ *une fonction monotone de concrétisation et* $\nabla^\sharp : (\mathcal{D}^\sharp \times \mathcal{D}^\sharp) \to \mathcal{D}^\sharp$ *un opérateur d'élargissement. Soit, pour* $x^\sharp \in \mathcal{D}^\sharp$*, la suite* $(y_n^\sharp)_{n\geq 0}$ *définie par :*

$$y_0^\sharp = x^\sharp, \quad y_{n+1}^\sharp = y_n^\sharp \nabla^\sharp f^\sharp(y_n^\sharp).$$

*Alors, si* $\gamma(x^\sharp)$ *est un pré-point fixe de* $f^\flat$*, la suite précédemment définie converge en un nombre fini d'itérations vers* $y_{dyn}^\sharp$*, un post-point fixe de* $f^\sharp$*. De plus,*

$$lfp_{\gamma(x^\sharp)} f^\flat \sqsubseteq^\flat \gamma(y_{dyn}^\sharp).$$

**Amélioration de l'approximation dynamique** Si $y_{dyn}^\sharp$ n'est pas un point fixe de $f^\sharp$, il est possible d'améliorer cette approximation en itérant de nouveau la fonction $f^\sharp$, sans élargissement. Tous les éléments de la suite, appelée séquence descendante, définie par :

$$z_0^\sharp = y_{dyn}^\sharp, \quad z_{n+1}^\sharp = f^\sharp(z_n^\sharp)$$

sont des approximations correctes de *lfp* $f^\flat$. Aucun miracle, la suite peut être celle des éléments d'une chaîne décroissante infinie, dont on ne saura pas calculer la limite. Il a donc été proposé un opérateur de rétrécissement [CC76] qui, à l'instar de l'opérateur d'élargissement, permet d'extrapoler la limite de la séquence descendante.

La technique d'approximation dynamique est puissante. Dans [CC92b], il est démontré qu'elle est plus générale que l'utilisation d'une connexion de Galois avec des domaines de profondeur finie dans lesquels on effectue un calcul exact du point fixe. Informellement, si l'on veut pour une large classe de programmes obtenir des résultats similaires, le domaine doit en général contenir un ensemble infini de valeurs (et l'on ne peut à la seule lecture du programme décider du sous-ensemble fini suffisant pour obtenir des résultats similaires). Donc considérer la conception d'un domaine de profondeur fini nécessitera d'abstraire ce domaine infini, sur lequel les résultats seront *in fine* d'une précision moindre.

Enfin, rien n'interdit d'utiliser l'approximation dynamique sur des domaines de profondeur finie – particulièrement ceux étant très larges, pour converger plus rapidement vers le plus petit point fixe que ne le fait la séquence d'itération de Kleene, en contrepartie d'une perte de précision.

## 2.2 Application à l'analyse statique

L'interprétation abstraite permet de construire des analyses statiques correctes en considérant des sémantiques, effectivement calculables, approximant la sémantique d'exécution des programmes dont on souhaite analyser le comportement.

Ce schéma de construction nécessite d'exprimer d'abord la sémantique concrète comme un point fixe. Afin d'obtenir une analyse statique effective, on va s'intéresser à la décomposition des fonctions sémantiques, qui mène à considérer un système d'équations de points fixes. Une technique est alors donnée pour approximer dynamiquement dans un domaine abstrait le résultat de ce système (Sec. 2.2.1).

En particulier, on donne pour les programmes impératifs structurés la construction systématique d'une sémantique concrète d'atteignabilité, décomposée selon les états de contrôle du programme. Dans ce cadre on définit l'ensemble des opérateurs et paramètres nécessaires au calcul du système d'équations abstraites correspondant (Sec. 2.2.2). Nous donnons un exemple de domaine abstrait muni de ces opérateurs et l'utilisons pour analyser un programme très simple (Sec. 2.2.3).

### 2.2.1 Partitionnement statique de sémantique et résolution

En général le domaine $\mathcal{D}^\flat$ de la sémantique concrète se décompose naturellement selon un ensemble fini $L$ : $\mathcal{D}^\flat = \bigcup_{\ell \in L} \mathcal{D}^\flat_\ell$. On peut alors souhaiter considérer des propriétés locales, *i.e.* liées à chaque élément $\ell$ et étendre cette décomposition à la fonction sémantique. Il est correct de remplacer l'équation de point fixe $x^\flat = f^\flat(x^\flat)$ par le système d'équations de points fixes suivant, où $n = |L|$ :

$$\bigwedge_{\ell \in L} x^\flat_\ell = f^\flat_\ell(x^\flat_1, \ldots, x^\flat_n)$$

et où $x^\flat_\ell \in \mathcal{D}^\flat_\ell$ est définit par $x^\flat_\ell = x^\flat \cap \mathcal{D}^\flat_\ell$ et $f^\flat_\ell : \prod_{i=1}^n \mathcal{D}^\flat_i \to \mathcal{D}^\flat_\ell$ est définie par $f^\flat(x^\flat_1 \sqcup^\flat \ldots \sqcup^\flat x^\flat_n) \cap \mathcal{D}^\flat_\ell$.

Pour calculer la solution d'un tel système d'équations l'idée est d'appliquer les séquences d'itérations vues précédemment (Sec. 2.1.2) sur chaque équation, en appliquant une fois le pas d'itération de l'une, une fois celui d'une autre, et ce de manière équitable. C'est la technique d'itération chaotique [CC77], où dans le cas général les pas d'itérations de plusieurs séquences d'itérations peuvent être effectués en parallèle :

Définition 2.4 (itération chaotique). *Soit un système d'équation de point fixe $\bigwedge_{\ell \in L} x_\ell = f_\ell(x_1, \ldots, x_{|L|})$. Une stratégie d'itération chaotique est une suite $(L_n)_{n \geq 0}$ de parties de $L$ où tout $\ell \in L$ apparaît infiniment souvent, i.e. $\forall \ell \in L$ l'ensemble $\{n \mid \ell \in L_n\}$ est non borné.*

*La résolution du système selon la stratégie d'itération $(L_n)_{n \geq 0}$ consiste à calculer en parallèle à chaque étape $n$ les nouvelles valeurs de $x_\ell$ pour tout $\ell \in L_n$. On désigne par l'ensemble $W \subseteq L$ les équations de points fixes résolues par approximation dynamique (opérateur d'élargissement $\nabla$). On a :*

$$x^{n+1}_\ell = \begin{cases} x^n_\ell & \text{si } \ell \notin L_n \\ x^n_\ell \nabla f_\ell(x^n_1, \ldots, x^n_{|L|}) & \text{sinon, si } \ell \in W \\ x^n_\ell \sqcup f_\ell(x^n_1, \ldots, x^n_{|L|}) & \text{sinon} \end{cases} .$$

La technique de résolution par itération chaotique peut-être vraiment efficace en tirant partie du fait que les fonctions $f_\ell$, déjà plus simples que la fonction $f$ grâce à la décomposition, ne dépendent pas en général de toutes les valeurs locales $x_1, \ldots, x_n$. On représente cette relation de dépendance par un graphe orienté $\mathcal{G}$, ayant pour nœuds l'ensemble $L$, et un arc de $\ell_2$ à $\ell_1$ si $f_{\ell_1}$ dépend de $x_{\ell_2}$.

Dans la pratique on choisit une stratégie d'itération où un seul calcul est effectué par étape. Par exemple, pour le graphe $\ell_1 \to \ell_2 \to \ell_3$ la stratégie $(\{\ell_1\}\{\ell_2\}\{\ell_3\})^*$ optimise le nombre de calculs nécessaire à la résolution du système, si $x_{\ell_1}$ croît : sa valeur est propagée à $x_{\ell_3}$ en deux étapes et deux calculs, alors que la stratégie parallèle $(\{\ell_1, \ell_2, \ell_3\})^*$ nécessiterait deux étapes, six calculs et plus de mémoire. On voit aussi que faire converger les valeurs appartenant à une composante fortement connexe de $\mathcal{G}$ avant de considérer les valeurs en aval de celle-ci peut éviter de nombreux calculs.

Dans le cas où un opérateur d'élargissement doit être utilisé pour assurer la convergence de l'itération chaotique, on profite du partitionnement de l'équation de point fixe, où l'opérateur est déjà plus facile à concevoir, pour l'appliquer de manière sélective et ainsi améliorer la précision de la solution. Pour garantir la convergence de l'itération chaotique il faut que toute récursion mutuelle du système d'équations soit coupée par une application de l'opérateur d'élargissement. En d'autres termes pour chaque cycle de $\mathcal{G}$, un de ses nœuds doit appartenir à $W$.

Il reste que le choix d'un ensemble $W$ minimal respectant cette condition est un problème NP-complet.

Enfin, si l'on considère une abstraction correcte du système d'équations concrètes, c'est-à-dire où les fonctions $f_\ell^\sharp$ sont des approximations supérieures de $f_\ell^\flat$, le théorème suivant (Thm. 2.5) nous assure que sa résolution par itération chaotique avec élargissement atteint une approximation correcte du plus petit point fixe du système d'équations concrètes. Ce théorème étend ainsi celui d'approximation dynamique (Thm. 2.4) :

Théorème 2.5 (approximation de système de points fixes par itération chaotique). *Soit un système d'équations de points fixes* $\bigwedge_{\ell \in L} x_\ell^\flat = f_\ell^\flat(x_1^\flat, \ldots, x_{|L|}^\flat)$ *défini sur un domaine concret* $\mathcal{D}^\flat$ *et pour tout* $\ell \in L$ *une approximation supérieure* $f_\ell^\sharp$ *de* $f_\ell^\flat$ *vis-à-vis de* $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$ *une fonction monotone de concrétisation. Alors la résolution par itération chaotique du système d'équations abstraites suivant, avec pour valeurs initiales* $x_l^{\sharp 0} \in \mathcal{D}^\sharp$ *telles que* $(\gamma(x_1^{\sharp 0}), \ldots, \gamma(x_{|L|}^{\sharp 0}))$ *est un pré-point fixe de* $f_1^\flat \times \ldots \times f_{|L|}^\flat$ *:*

$$\bigwedge_{\ell \in L} x_\ell^\sharp = f_\ell^\sharp(x_1^\sharp, \ldots, x_n^\sharp),$$

*selon une stratégie et un ensemble de points d'élargissements* $W$ *valides, converge en un temps fini vers une sur-approximation du plus petit point fixe du système d'équations concrète, plus grand que* $(\gamma(x_1^{\sharp 0}), \ldots, \gamma(x_{|L|}^{\sharp 0}))$.

On notera qu'il est possible d'améliorer le résultat obtenu par l'approximation dynamique du système d'équations de points fixes en effectuant une séquence descendante $(x_\ell^n \sqcap f_\ell(x_1^n, \ldots, x_{|L|}^n))$ soit quelques pas, soit en utilisant un opérateur de rétrécissement sur les équations désignées par $W$ pour en assurer la convergence.

Remarque. *Si le domaine abstrait se décompose naturellement, on peut souhaiter ajuster la granularité de cette décomposition : le choix de l'ensemble $L$ influe fortement sur la qualité de l'approximation. Ceci s'illustre dans le cadre d'une abstraction basée sur une connexion de Galois où on voit que la composition des meilleures abstractions $f^\sharp_{1std}$ et $f^\sharp_{2std}$ de $f^\sharp_1$ et $f^\sharp_2$ n'est que rarement la meilleure abstraction de $f^\sharp_2 \circ f^\sharp_1$.*

### 2.2.2 Une analyse d'invariance de programmes impératifs structurés

Dans cette thèse nous sommes intéressés par des analyses statiques permettant de découvrir des invariants de programmes impératifs structurés. Nous présentons une construction générale de telles analyses pour des programmes non procéduraux : cette construction est classique et rappellera celle de [CC76] ou celle de [Cou99] plus complète.

Pour des analyses de programmes impératifs structurés et ayant trait à d'autre type de propriétés, on peut noter les travaux sur les propriétés de vivacité [BCC+07], notamment la terminaison [CS01, Cou05], ou encore ceux sur les propriétés non-fonctionnelles [CL05].

#### 2.2.2.1 Modèle des programmes et sémantiques

On prend pour modèle des programmes impératifs structurés des automates modifiant l'état de la mémoire. On note $Q$ l'ensemble des états mémoires possibles. Les places [2] de l'automate sont appelées des points de contrôle, dont l'ensemble est noté $K$ et ses transitions sont étiquetées par des commandes, conjonctions d'une condition (ou garde) et d'une action. Informellement, la première indique si la transition peut-être prise vis-à-vis de l'état mémoire, c'est une fonction $g : Q \to \mathbb{B}$, la seconde donne le nouvel état mémoire si l'automate suit la transition, c'est une fonction $a : Q \to Q$.

Définition 2.5 (automate interprété). *Un automate interprété sur $Q$ est un triplet $(K, \delta, k_0)$ où $K$ est un ensemble fini de places, $k_0 \in K$ est la place initiale et $\delta \subset K \times C \times K$ est la relation de transition. Une transition est étiquetée par une commande qui est formée d'une garde et d'une action, notée $(g, a) \in C = (Q \to \mathbb{B}) \times (Q \to Q)$.*

*Soit $k, k' \in K$ et $q \in Q$ l'interprétation courante. Le contrôle de l'automate interprété passe de $k$ à $k'$ s'il existe une transition $(k, (g, a), k') \in \delta$ telle que $g(q)$ est vrai ; l'interprétation courante devient $a(q)$.*

L'état d'un automate interprété est donné par un certain état mémoire $q \in Q$ à un certain point de contrôle $k \in K$. On donne aisément une sémantique opérationnelle à ces automates en terme de système de transition (Déf. 1.1). Considérer le système définit par l'ensemble d'état $S = (K \times Q)$, l'ensemble d'état initiaux $S_{init} = \{(k_0, q) \mid q \in Q\}$ et la fonction de transition $\tau$ définie par : $((k, q), (k', q')) \in \tau$ si $\exists (k, (g, a), k') \in \delta$ telle que $g(q)$ est vrai et $q' = a(q)$.

**Sémantique d'atteignabilité** Ce que l'on souhaite c'est disposer d'une sémantique d'atteignabilité des automates interprétés puisque l'on s'intéresse à leurs invariants – s'intéresser aux propriétés de vivacité par exemple aurait nécessité de définir une sémantique de traces, plus fine [Cou02]. Là encore nous savons déjà définir (Déf. 1.2, p. 6)

2. On préfère utiliser ici le terme de place à celui plus correct, d'état, afin d'éviter quelques confusions.

les états accessibles depuis $S_{init}$ par l'automate, notés $Acc(S_{init}) \in \mathcal{D}^\Phi = \mathcal{P}(S) = \mathcal{P}(K \times Q)$, comme le plus petit point fixe de l'équation :

$$x^\Phi = f^\Phi(x^\Phi)$$

où $f^\Phi(x^\Phi) = S_{init} \sqcup^\Phi \{s' \in S \mid \exists s \in x^\Phi, (s, s') \in \tau\}$ est la fonction *post* définie en introduction (Sec. 1.3).

**Sémantique collectrice** On voit que l'ensemble des points de contrôle est un candidat naturel pour décomposer le domaine concret $\mathcal{D}^\Phi = \bigcup_{k \in K} \mathcal{D}^\Phi_k$ où $\mathcal{D}^\Phi_k = \{k\} \times \mathcal{P}(Q)$. Partitionné ainsi cette sémantique concrète exprime l'ensemble des états atteignables en chaque point de contrôle, *i.e.* $x^\Phi_k = \{(k, q) \in Acc(S_{init})\}$ ou, sans perte d'information, l'ensemble des états mémoire atteignables en chaque point de contrôle, *i.e.* $x^\flat_k = \{q \mid (k, q) \in Acc(S_{init})\} \in \mathcal{D}^\flat = \mathcal{P}(Q)$.

Entendu de cette manière, on peut formuler directement cette sémantique d'invariance comme suit. Les états mémoires atteignables en un point de contrôle $k'$ sont l'union de ceux atteints aux points de contrôle ayant une transition vers $k'$ dans l'automate interprété, filtrés par la garde et modifiés par l'action de cette transition :

$$x^\flat_{k_0} = Q_{init} \wedge \bigwedge_{k' \in (K \setminus k_0)} x^\flat_{k'} = \bigsqcup^\flat_{(k,(g,a),k') \in \delta} \{\!|a|\!\}^\flat \circ \{\!|g|\!\}^\flat(x^\flat_k) \tag{2.1}$$

où $Q_{init} = \{q \mid (k_0, q) \in S_{init}\}$, $\{\!|a|\!\}^\flat(x^\flat_k) = \{q' \in Q \mid \exists q \in x^\flat_k,\ q' = a(q)\}$ et $\{\!|g|\!\}^\flat(x^\flat_k) = \{q \in Q \mid q \in x^\flat_k,\ g(q) \text{ est vrai}\}$.

On notera que les fonctions sémantiques $\{\!|\cdot|\!\}^\flat : \mathcal{D}^\flat \to \mathcal{D}^\flat$, appelées aussi fonctions de transfert, sont bien monotones.

On démontre comment cette sémantique concrète, appelée aussi collectrice, se dérive de celle non partitionnée et de plus bas niveau.

*Démonstration.* La sémantique résultante du partitionnement selon $\mathcal{D}^\Phi_k$ est par définition :

$$\bigwedge_{k' \in K} x^\Phi_{k'} = f^\Phi(\bigsqcup^\Phi_{k \in K} x^\Phi_k) \sqcap \mathcal{D}^\Phi_{k'}.$$

Les états atteignables en $k'$ dépendent de ceux ayant une transition vers celui-ci dans l'automate interprété. On isole le cas du point de contrôle initial.

$$x^\Phi_{k_0} = f^\Phi(\emptyset) \wedge \bigwedge_{k' \in (K \setminus k_0)} x^\Phi_{k'} = f^\Phi(\bigsqcup^\Phi_{(k,(g,a),k') \in \delta} x^\Phi_k).$$

Par définition de $f^\Phi$ on obtient :

$$x^\Phi_{k_0} = S_{init} \wedge \bigwedge_{k' \in (K \setminus k_0)} x^\Phi_{k'} = \{s' \in S \mid \exists s \in \bigsqcup^\Phi_{(k,(g,a),k') \in \delta} x^\Phi_k, (s, s') \in \tau\}.$$

L'opérateur d'union est exact. On explicite les états :

$$x^\Phi_{k_0} = S_{init} \wedge \bigwedge_{k' \in (K \setminus k_0)} x^\Phi_{k'} = \bigsqcup^\Phi_{(k,(g,a),k') \in \delta} \{(k', q') \in S \mid \exists (k, q) \in x^\Phi_k, ((k, q), (k', q')) \in \tau\}.$$

Par définition de la transition $\tau$ :

$$x^{\Phi}_{k_0} = S_{init} \wedge \bigwedge_{k' \in (K \setminus k_0)} x^{\Phi}_{k'} = \bigsqcup^{\Phi}_{(k,(g,a),k') \in \delta} \{(k', q') \in S \mid \exists (k, q) \in x^{\Phi}_k, q' = a(q) \text{ et } g(q) \text{ est vrai}\}.$$

L'état de contrôle atteint est une information redondante que l'on peut oublier. On obtient la sémantique collectrice précédemment définie sur le domaine concret $\mathcal{D}^\flat$. □

#### 2.2.2.2 Interpréteur abstrait

En sachant donner systématiquement à tout automate interprété une sémantique concrète partitionnée sur le domaine $\mathcal{D}^\flat$, nous pouvons définir un analyseur statique générique. Cet interpréteur abstrait prend en paramètre un *domaine abstrait sémantique*, notion regroupant un domaine abstrait $\mathcal{D}^\sharp$ et un ensemble d'opérateurs permettant, selon le Théorème 2.5 (p. 19), de construire une abstraction correcte de la sémantique concrète et de la résoudre effectivement par itération chaotique. L'interpréteur abstrait ne fait alors qu'appliquer cette itération chaotique jusqu'à convergence.

Pour construire la sémantique abstraite il suffit d'un treillis complet, d'une fonction d'abstraction et d'approximations supérieures pour chaque fonctions

$$f^\flat_{k'} = \bigsqcup^{\flat}_{(k,(g,a),k') \in \delta} \{\!|a|\!\}^\flat \circ \{\!|g|\!\}^\flat(x^\flat_k).$$

Cette dernière condition se réduit à détenir une approximation supérieure $\{\!|\cdot|\!\}^\sharp$ de chaque fonction de transfert : la fonction

$$f^\sharp_{k'} = \bigsqcup^{\sharp}_{(k,(g,a),k') \in \delta} \{\!|a|\!\}^\sharp \circ \{\!|g|\!\}^\sharp(x^\sharp_k)$$

est alors une borne supérieure de $f^\flat_{k'}$ – la preuve, aisée, repose sur la monotonicité de la fonction de concrétisation. Si l'on suppose que l'état mémoire initial de l'automate interprété est toujours indéfini – cas auquel on peut toujours se ramener, la sémantique abstraite s'écrit :

$$x^\sharp_{k_0} = \top^\sharp \wedge \bigwedge_{k' \in (K \setminus k_0)} x^\sharp_{k'} = \bigsqcup^{\sharp}_{(k,(g,a),k') \in \delta} \{\!|a|\!\}^\sharp \circ \{\!|g|\!\}^\sharp(x^\sharp_k). \tag{2.2}$$

Pour pouvoir résoudre par itération chaotique cette sémantique abstraite il faut disposer d'un opérateur d'élargissement lorsque le domaine abstrait est de profondeur infinie. D'autre part il est nécessaire que les éléments du treillis abstraits soient représentables en machine et que chaque opérateur soit effectivement calculable (notamment, parce qu'il n'apparaît pas explicitement dans les formules, l'ordre partiel qui est utilisé pour repérer la terminaison de l'itération : $x^{\sharp n+1}_\ell \sqsubseteq^\sharp x^{\sharp n}_\ell$).

On peut désormais définir la notion de domaine abstrait sémantique propre à l'analyse d'invariance.

Définition 2.6 (domaine abstrait sémantique). *Un domaine abstrait sémantique pour $\mathcal{D}^\flat = \mathcal{P}(Q)$ est la donnée :*

1. *d'un treillis complet* $(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \bot^\sharp, \top^\sharp)$ *dont les éléments sont représentables en machine ;*
2. *d'une fonction de concrétisation* $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$ *monotone telle que* $\gamma(\bot^\sharp) = \bot^\flat = \emptyset$ *;*
3. *d'un algorithme effectif pour calculer l'approximation supérieure de chaque fonction de transfert* $\{\!|\cdot|\!\}^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ *;*
4. *d'un algorithme effectif pour décider la relation d'ordre partiel* $\sqsubseteq^\sharp$ *et calculer la borne supérieure* $\sqcup^\sharp$ *;*
5. *d'un algorithme d'élargissement effectif* $\nabla^\sharp : (\mathcal{D}^\sharp \times \mathcal{D}^\sharp) \to \mathcal{D}^\sharp$ *si* $\mathcal{D}^\sharp$ *est de profondeur infinie.*

On peut bien entendu disposer d'une connexion de Galois entre $\mathcal{D}^\flat$ et $\mathcal{D}^\sharp$. On cherchera alors à prendre les meilleurs abstractions comme opérateurs abstraits quand celles-ci sont calculables efficacement. D'ailleurs l'opérateur de borne inférieure abstrait $\sqcap^\sharp$ est souvent une abstraction exacte de $\sqcap^\flat$.

Dans tous les cas, la conception d'un domaine abstrait sémantique relève de nombreux compromis afin que les analyses statiques l'utilisant puissent avoir un coût raisonnable et être cependant suffisamment précises pour découvrir une classe d'invariants d'intérêt.

Afin qu'un domaine sémantique abstrait puisse analyser correctement des programmes ayant des gardes ou actions dont il ne définit pas les fonctions de transfert abstraites, des opérateurs abstraits d'« oubli » peuvent lui être ajoutés. Par exemple, quel que soit le domaine abstrait choisi, les fonctions de transfert $\{\!|c|\!\}^\sharp x^\sharp = x^\sharp$ et $\{\!|a|\!\}^\sharp x^\sharp = \top^\sharp$ sont correctes pour toute condition $c$ et action $a$. Les abstractions d'oubli que l'on peut définir sont souvent moins brutales. Par exemple les éléments du domaine abstrait sont généralement organisés selon les variables du programme : ainsi on peut définir une fonction d'oubli pour les affectations que l'on ne sait traiter qui n'oubliera que l'abstraction associée à la variable affectée.

Enfin on notera que le domaine abstrait sémantique peut se contenter de définir uniquement les fonctions de transfert des conditions de bases : les fonctions de transfert de conjonction et de disjonction de ces conditions peuvent être définies comme suit :

$$\{\!|c_1 \textbf{ and } c_2|\!\}^\sharp(x^\sharp) = \{\!|c_1|\!\}^\sharp(x^\sharp) \sqcap^\sharp \{\!|c_2|\!\}^\sharp(x^\sharp)$$
$$\{\!|c_1 \textbf{ or } c_2|\!\}^\sharp(x^\sharp) = \{\!|c_1|\!\}^\sharp(x^\sharp) \sqcup^\sharp \{\!|c_2|\!\}^\sharp(x^\sharp).$$

**Paramètres de l'itération chaotique** Pour calculer la sémantique abstraite, l'interpréteur abstrait doit définir une stratégie d'itérations chaotiques valide et des points d'élargissement assurant sa convergence.

Pour ces derniers on utilise le fait que le graphe de dépendance $\mathcal{G}$ des équations de points fixes est réductible, car tel est le graphe de contrôle d'un programme impératif structuré. L'ensemble des têtes des arrêtes arrières de $\mathcal{G}$, ou autrement dit des points de contrôle en tête de boucle, est suffisant pour couper tout cycle de $\mathcal{G}$ [Cou81]. Ce choix n'est bien entendu pas optimal : considérer l'automate interprété de la Figure 2.1 où $k_1$ peut-être retiré de l'ensemble de points d'élargissement $\{k_1, k_2, k_4\}$ obtenu par cette heuristique.

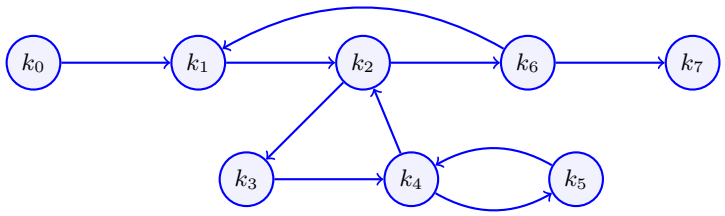



Figure 2.1 – Un automate interprété avec trois boucles imbriquées

En pratique, pour repérer ces têtes de boucles dans le graphe $\mathcal{G}$ on utilise l'algorithme des sous-composantes fortement connexes proposé dans [Bou93] – qui fonctionne également sur les graphes non réductibles que l'on rencontre par exemple pour l'analyse interprocédurale. L'algorithme est basé sur une application itérative de l'algorithme de Tarjan déterminant les composantes fortement connexes d'un graphe [Tar72] : à chaque itération on ôte du graphe la tête de chaque composante fortement connexe non triviale découverte et on l'ajoute à l'ensemble des points d'élargissement. L'algorithme termine lorsque le graphe ne comporte plus que des composantes fortement connexes triviales. Sa complexité sur les graphes de programmes structurés est $\mathcal{O}(k.n)$ où $k$ est le nombre maximum de boucles imbriquées dans le programme et $n$ le nombre d'instructions dans ces boucles.

Quant à la définition de la stratégie d'itération, on utilise le fait que l'algorithme de Tarjan donne un ordre topologique des composantes fortement connexes qu'il calcule. Ainsi, l'algorithme des sous-composantes fortement connexes peut fournir un tri topologique faible (*i.e.* qui ne prend pas en compte les arrêtes arrières) du graphe $\mathcal{G}$ où les sommets de sous-composantes fortement connexes différentes ne s'entrelacent pas, ou autrement dit, où les sommets sont regroupés par sous-composantes fortement connexes (s.c.f.c.). Pour l'exemple de la Figure 2.1, si on utilise un parenthèsage pour expliciter ce regroupement, l'ordre topologique faible obtenu est $k_0, (k_1, (k_2, k_3, (k_4, k_5)), k_6), k_7$.

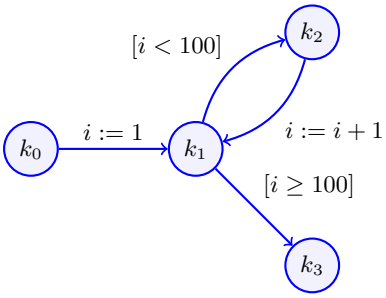
[Bou93] propose de définir à partir de l'ordre topologique faible renvoyé par l'algorithme des s.c.f.c. une stratégie d'itération non parallèle, dite récursive, qui consiste à stabiliser d'abord les équations des sous-composantes fortement connexes les plus profondes. Si l'on note $(\ldots)^*$ l'opérateur d'« itération jusqu'à stabilisation », on obtient pour notre exemple la stratégie d'itération récursive $\{k_0\}, (\{k_1\}, (\{k_2\}, \{k_3\}, (\{k_4\}, \{k_5\})^*)^*, \{k_6\})^*, \{k_7\}$. La stabilisation d'une composante se détecte par la stabilisation de sa tête, qui est un point d'élargissement par définition. En pratique ces stratégies récursives sont efficaces : on se reportera à [Bou93] pour une étude de leur complexité.

Il ne reste à l'interpréteur abstrait qu'à donner les valeurs initiales de l'itération chaotique. Tout point de contrôle non initial est donné non atteignable – $\gamma(\bot^\sharp)$ est toujours un pré-point fixe de $f_k^\flat$ :

$$x_{k_0}^{\sharp 0} = \top^\sharp, \quad \forall k \in (K \setminus k_0), x_k^{\sharp 0} = \bot^\sharp.$$

### 2.2.3 Illustration sur un programme simple

Nous illustrons l'utilisation de l'interpréteur abstrait que nous venons de définir par l'analyse de l'automate interprété de la Figure 2.2(a). Il est issu d'une classe de programmes très particulière où seule une variable $i$, prenant ses valeurs dans l'ensemble mathématique parfait $\mathbb{Z}$, est utilisée : on prend donc pour cette classe le domaine concret $\mathcal{D}^\flat = \mathcal{P}(\mathbb{Z})$. La sémantique concrète de ce programme définie selon l'Équation 2.1 (p. 21) sur ce domaine est donnée à la Figure 2.2(b).




(a) Automate interprété incrémentant la variable $i$ de 1 à 100

$$\begin{cases} x^\flat_{k_0} = \mathbb{Z} \\ x^\flat_{k_1} = \{\!|i := 1|\!\}^\flat(x^\flat_{k_0}) \sqcup^\flat \{\!|i := i+1|\!\}^\flat(x^\flat_{k_2}) \\ x^\flat_{k_2} = \{\!|i < 100|\!\}^\flat(x^\flat_{k_1}) \\ x^\flat_{k_3} = \{\!|i \geq 100|\!\}^\flat(x^\flat_{k_1}) \end{cases}$$

(b) Sémantique collectrice sur $\mathcal{D}^\flat = \mathcal{P}(\mathbb{Z})$

Figure 2.2 – Un automate interprété et sa sémantique concrète

#### 2.2.3.1 Le domaine sémantique des intervalles

Nous allons utiliser pour domaine sémantique celui des intervalles [CC76]. Ses éléments sont soit l'ensemble vide soit un intervalle dont les bornes peuvent être infinies :

$$\mathcal{D}^{int} = \bot^{int} \cup \{[\ell, u] \mid \ell \in (\mathbb{Z} \cup \{-\infty\}), u \in (\mathbb{Z} \cup \{+\infty\}), \ell \leq u\}.$$

L'ordre partiel $\sqsubseteq^{int}$ est l'inclusion sur les intervalles. Une partie de $\mathbb{Z}$ est abstraite par le plus petit intervalle contenant ses éléments : $\alpha^{int}(x^\flat) = [\min x^\flat, \max x^\flat]$ et $\alpha(\emptyset) = \bot^{int}$. La concrétisation d'un intervalle est la partie constituée des éléments qu'il représente, $\gamma^{int}([\ell, u]) = \{z \in \mathbb{Z} \mid \ell \leq z \leq u\}$ et $\gamma^{int}(\bot^{int}) = \emptyset$. Ces fonctions forment une connexion de Galois.

La borne inférieure $\sqcap^{int}$ définie comme l'intersection des intervalles est exacte, *a contrario* de la borne supérieure définie comme le plus petit intervalle contenant les deux intervalles que sont ses opérandes :

$$x^\sharp_1 \sqcup^{int} x^\sharp_2 = \begin{cases} [\min\{\ell_1, \ell_2\}, \max\{u_1, u_2\}] & \text{si } x^\sharp_1 = [\ell_1, u_1] \text{ et } x^\sharp_2 = [\ell_2, u_2] \\ x^\sharp_1 & \text{si } x^\sharp_2 = \bot^{int} \\ x^\sharp_2 & \text{si } x^\sharp_1 = \bot^{int} \end{cases}.$$

Nous ne donnons que l'abstraction des fonctions de transfert dont nous avons besoin pour analyser notre programme. Pour les conditions sur $i$ l'opérateur de borne inférieure est utilisé :

$$\{\!|i \geq z|\!\}^\sharp(x^\sharp) = x^\sharp \sqcap^{int} [z, +\infty] \text{ et } \{\!|i < z|\!\}^\sharp(x^\sharp) = x^\sharp \sqcap^{int} [-\infty, z-1].$$

Pour les affectations inversibles une translation des bornes de l'intervalle est effectué :

$$\{\!|i := i + z|\!\}^\sharp(x^\sharp) = \begin{cases} [\ell + z, u + z] & \text{si } x^\sharp = [\ell, u] \\ \bot^{int} & \text{sinon} \end{cases},$$

$$\{\!|i := z|\!\}^\sharp(x^\sharp) = \begin{cases} [z, z] & \text{si } x^\sharp \neq \bot^{int} \\ \bot^{int} & \text{sinon} \end{cases}.$$

Enfin un opérateur d'élargissement est donné, le domaine $\mathcal{D}^{int}$ étant de profondeur infinie. Il consiste à mettre à l'infini toute borne s'éloignant : la hauteur des suites élargies est donc de quatre. On a $\bot^{int} \nabla^{int} x^\sharp = x^\sharp \nabla^{int} \bot^{int} = x^\sharp$ et :

$$[\ell_1, u_1] \nabla^{int} [\ell_2, u_2] = [\ell, u] \text{ où } \ell = \begin{cases} -\infty & \text{si } \ell_2 < \ell_1 \\ \ell_1 & \text{sinon} \end{cases} \text{ et } u = \begin{cases} +\infty & \text{si } u_2 > u_1 \\ u_1 & \text{sinon} \end{cases}.$$

**Extension non-relationnelle** Ce domaine peut être utilisé simplement pour définir un domaine sémantique abstrait permettant d'analyser des programmes manipulant plusieurs variables entières. On définit ce domaine par $\mathcal{D}^\sharp = V \to \mathcal{D}^{int}$, où les valeurs atteignables de chaque variable sont abstraites par un intervalle : si l'on représente un état mémoire de ces programmes par une fonction des variables dans $\mathbb{Z}$, *i.e.* $Q = V \to \mathbb{Z}$, on a :

$$\gamma(x^\sharp) = \{q \in (V \to \mathbb{Z}) \mid \forall i \in V, q(i) \in \gamma^{int}(x^\sharp(i))\}.$$

Les opérateurs de $\mathcal{D}^\sharp$ sont simplement défini par *lifting* de ceux de $\mathcal{D}^{int}$. La définition des fonctions de transfert nécessite un peu plus de travail, *cf.* [Min04] par exemple.

#### 2.2.3.2 Résolution

Nous avons donc pour sémantique abstraite définie sur $\mathcal{D}^{int}$, d'après l'Équation 2.2 (p. 22), le système d'équations de points fixes suivant, où l'on a explicité quelques fonctions de transfert :

$$\begin{cases} x^\sharp_{k_0} = [-\infty, +\infty] \\ x^\sharp_{k_1} = [1, 1] \sqcup^{int} \{\!|i := i + 1|\!\}^\sharp(x^\sharp_{k_2}) \\ x^\sharp_{k_2} = x^\sharp_{k_1} \sqcap^{int} [-\infty, 99] \\ x^\sharp_{k_3} = x^\sharp_{k_2} \sqcap^{int} [100, +\infty] \end{cases}.$$

L'interpréteur abstrait choisit pour stratégie d'itération chaotique $\{k_0\}, (\{k_1\}, \{k_2\})^*, \{k_3\}$ et pour ensemble de points d'élargissement le singleton $\{k_1\}$.

La Figure 2.3 donne les itérés obtenus lors de la résolution statique et dynamique du système d'équation abstrait. Dans le cas sans élargissement (Fig. 2.3(a)), la composante fortement connexe converge au 100$^{\text{e}}$ itéré, et la concrétisation du plus petit point fixe obtenu est le plus petit point fixe du système concret. Dans le cas avec élargissement (Fig. 2.3(b)), l'itération croissante converge dès le 2$^{\text{e}}$ itéré et le post-point fixe alors atteint a pour solution concrète l'ensemble $\{100, 101, 102, \ldots\}$ pour le point de contrôle terminal, une approximation grossière. Cependant l'application d'un seul pas de l'itération descendante améliore significativement ce résultat, en atteignant le plus petit point fixe.

On remarque que l'opérateur d'élargissement a un rôle essentiel : il a en charge de deviner les invariants inductifs du programme. Influant donc fortement sur la précision de l'analyse sa conception doit être faite avec soin.

| | *0* | *1* | *2* | ... | *100* ↻ | *101* |
|---|---|---|---|---|---|---|
| $x^\sharp_{k_0} =$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ |
| $x^\sharp_{k_1} =$ | $\bot^{int}$ | $[1, 1]$ | $[1, 2]$ | ... | $[1, 100]$ | $[1, 100]$ |
| $x^\sharp_{k_2} =$ | $\bot^{int}$ | $[1, 1]$ | $[1, 2]$ | ... | $[1, 99]$ | $[1, 99]$ |
| $x^\sharp_{k_3} =$ | $\bot^{int}$ | $\bot^{int}$ | $\bot^{int}$ | $\bot^{int}$ | $\bot^{int}$ | $[100, 100]$ |

(a) Sans élargissement

| | ← itération croissante → | | | | ← itération décroissante → |
|---|---|---|---|---|---|
| | *0* | *1* | *2* ↻ | *3* | *1* |
| $x^\sharp_{k_0} =$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ | $[-\infty, +\infty]$ |
| $x^\sharp_{k_1} =$ | $\bot^{int}$ | $[1, 1]$ | $[1, +\infty]$ | $[1, +\infty]$ | $[1, 100]$ |
| $x^\sharp_{k_2} =$ | $\bot^{int}$ | $[1, 1]$ | $[1, 99]$ | $[1, 99]$ | $[1, 99]$ |
| $x^\sharp_{k_3} =$ | $\bot^{int}$ | $\bot^{int}$ | $\bot^{int}$ | $[100, +\infty]$ | $[100, 100]$ |

(b) Avec élargissement et application d'une séquence descendante

Figure 2.3 – Résultats des résolutions du système abstrait

Enfin, on notera que cet exemple ne met pas en lumière l'abstraction induite par le domaine des intervalles. Il suffit pour cela de considérer l'instruction $i := i + 10$ en place de l'instruction d'incrémentation. Que ce soit statiquement ou dynamiquement, l'analyse donne pour ensemble de valeurs atteignables au point de contrôle final l'ensemble $\{100, 101, \ldots, 109\}$ alors que le résultat exact est $\{101\}$. C'est l'abstraction de l'invariant de boucle, l'intervalle [1,99] qui est trop imprécise : il faut utiliser un domaine abstrait capable d'exprimer de plus des relations d'égalité modulo $i = 1[10]$, afin d'obtenir ce dernier résultat. On remarque que c'est l'invariant de boucle qui détermine l'expressivité nécessaire au domaine abstrait pour analyser précisément le programme.

## 2.3 Conclusion

Avant de conclure on rappelle un aspect important de la théorie de l'interprétation abstraite, qui est de permettre de combiner aisément des analyses. Nos travaux seront entièrement basés sur cette capacité. On donne quelques exemples classiques de combinaisons, que l'on évoque dans la suite de la thèse.

**Combinaison de domaines abstraits** Soit $\mathcal{D}^1$ et $\mathcal{D}^2$ deux domaines abstraits d'un même domaine concret $\mathcal{D}^\flat$. On note leurs fonctions de concrétisation $\gamma^1$ et $\gamma^2$ respectivement.

- Une combinaison simple est le produit cartésien des deux domaines abstraits, $(\mathcal{D}^1 \times \mathcal{D}^2)$. En définissant ses opérations ensemblistes et sémantiques élément par élément, ce produit permet d'appliquer simplement les deux analyses correspondantes en parallèle. La fonction de concrétisation de ce domaine est définie par $\gamma^{1\times 2}((x^1, x^2)) = \gamma^1(x^1) \sqcap^\flat \gamma^2(x^2)$.

- La précision de cette première combinaison peut-être améliorée si les deux valeurs abstraites s'échangent leurs informations après chaque opération sémantique par exemple. Cet échange se définit par une fonction $\sigma$, telle que si $(y^1, y^2) = \sigma((x^1, x^2))$ alors $y^1 \sqsubseteq^1 x^1$, $y^2 \sqsubseteq^2 x^2$ et $(y^1, y^2)$ a même concrétisation que $(x^1, x^2)$. On appelle $\sigma$ une réduction et le produit correspondant un produit réduit [CC79b].
  Dans l'exemple pris dans le dernier paragraphe de notre toute dernière section (Sec. 2.2.3.2) si on avait utilisé le produit $(\mathcal{D}^{int} \times \mathcal{D}^{mod})$, où $\mathcal{D}^{mod}$ est le domaine abstrait des congruences simples [Gra89] (Fig. 3.2(c), p. 31) on aurait trouvé avec le produit cartésien la valeur abstraite $([100, 109], 1[10])$ au point de contrôle final. Il est facile de définir une fonction $\sigma$, ne réduisant que l'intervalle du domaine, telle que le produit réduit aurait découvert le résultat exact $([101, 101], 1[10])$.
- Un autre type de combinaison est la complétion disjonctive [CC92a]. Elle répond au besoin d'éviter la perte d'information des opérations d'union que l'on trouve après chaque conditionnelles, ou chaque itération de boucle. Une valeur abstraite $x^\sharp$ est un ensemble, dont la taille varie, de valeurs abstraites $(x^1_1, \ldots, x^1_n)$ d'un même domaine $\mathcal{D}^1$ et qui permet de représenter exactement des disjonctions. Sa concrétisation est donc $\gamma^\sharp(x^\sharp) = \bigvee_i \gamma^1(x_i)$. Ce domaine abstrait est en fait théorique et en construire une instance est complexe, la disjonction amenant les opérations à diverger.
- Une solution pratique au même besoin est le partitionnement de traces [MR05]. L'idée consiste à limiter le nombre de disjonctions et donc *in fine* de reporter les opérations d'unions à plus tard, ce qui permet toujours un gain de précision. L'analyse considère à un point de contrôle une partition couvrante de l'ensemble des traces du programme, choisie statiquement ou dynamiquement, et associe à chaque élément de cette partition une valeur abstraite de $\mathcal{D}^1$. Par exemple à la fin du programme : **if** $x < 0$ **then** $y :=$ $-1$ **else** $y :=$ $1$, elle peut choisir la partition la plus fine qui considère les deux traces d'exécutions ouvertes par la conditionnelle. Avec $\mathcal{D}^1 = \mathcal{D}^{int}$ on a $(x \in [-\infty, -1] \wedge y = -1) \vee (x \in [0, +\infty] \wedge y = 1)$. Si l'instruction suivante effectue une division par $y$, l'absence d'erreur à l'exécution est assurée. *A contrario* de la complétion disjonctive rien empêche après cette instruction de passer sur la partition confondant les deux traces ci-dessus. L'ensemble atteignable n'est alors représenté que par le résultat de l'union $(y \in [-1, 1])$. En plus de partitionner selon le flux de contrôle, il est aussi possible de partitionner selon les valeurs des variables. Par exemple pour une boucle $i :=$ $1$; **while** $(i \leq 10)$ **do** une pré-analyse peut considérer qu'il est intéressant d'analyser à part les deux premières itérations de la boucle des suivantes, et considérer alors la partition de traces $\{i = 1, i = 2, i \in [3, 10]\}$.

L'interprétation abstraite donne un cadre théorique solide pour formuler des abstractions décidables et correctes de systèmes infinis. Elle explique comment faire pour que leur calcul soit efficace et terminant. Bien entendu d'autres théories existent pour construire des analyses statiques. Les systèmes de types [Pie02] permettent des analyses efficaces et compositionnelles, mais découvrent des propriétés faibles. *A contrario*, le *model checking* [Sif82, CGP99] est capable de vérifier des propriétés très complexes, par exemple de vivacité, mais exige une abstraction spécifique du programme analysé.

Finalement, l'interprétation abstraite est à nos yeux le cadre théorique le plus adapté pour découvrir automatiquement les propriétés relativement complexes qui nous intéressent (Sec. 1.2), et répondre adroitement à l'indécidabilité du problème correspondant.

# Chapitre 3

# Analyses numériques et non-égalités

En introduction (Sec. 1.2), nous avons dit que l'un des sujets de cette thèse était la conception d'une analyse numérique capable de découvrir des invariants de non-égalités.

On peut imaginer qu'une telle analyse considère des contraintes de non-égalité très générales comme les non-égalités linéaires, par exemple $x_1 - 2x_2 + x_3 \neq 5$. Quelle que soit l'expressivité des contraintes de non-égalité qu'elle manipule, l'intérêt d'une telle analyse augmenterait si elle considérait également d'autres types de contraintes : égalités, inégalités ou encore égalités modulo. En effet, des contraintes de non-égalité pourraient être déduites des contraintes de l'autre type et inversement. Par exemple $x_1 = x_2 + 2 \wedge x_3 = x_4 + 2 \wedge x_1 \neq x_3 \Rightarrow x_2 \neq x_4$. Si aucun domaine abstrait ne permet d'exprimer directement des contraintes de non-égalité, il en existe de nombreux basés sur des contraintes d'égalités ou d'inégalités, modulo ou non. Ainsi, une idée consiste à choisir l'un d'entre eux, d'augmenter son expressivité et de modifier ses algorithmes, afin qu'il puisse également exprimer et raisonner sur des contraintes de non-égalité. On peut alors espérer réutiliser tout le travail effectué sur la représentation et les opérations sémantiques de ce domaine « hôte ». On commence donc par rappeler dans ce chapitre les différents domaines abstraits numériques existants (Sec. 3.1). Ceci permettra également de situer, dans le paysage ainsi décrit, le nouveau domaine abstrait que l'on propose (Ch. 5). On détaille justement le domaine qui a été choisi comme hôte pour le définir, qui est celui des zones (Sec. 3.1.1).

Enfin, on s'intéresse aux résultats d'autres communautés de recherche, par exemple du *model checking* ou de la programmation par contraintes, qui étudient des ensembles de contraintes dont certaines sont des contraintes de non-égalité (Sec. 3.2). On trouvera principalement des résultats sur la satisfaisabilité de ces ensembles de contraintes.

## 3.1 Les domaines abstraits existants

Dans la suite on considère un ensemble de variables scalaires $V = \{x_1 \dots x_n\}$ prenant leurs valeurs dans $\mathbb{K} = \mathbb{Z}, \mathbb{Q}$ ou $\mathbb{R}$. On notera $c$ une constante de $\mathbb{K}$.

Les domaines abstraits historiques sont les domaines des intervalles, des signes [CC76] et des polyèdres [CH78]. Depuis de nouveaux domaines abstraits apparaissent

régulièrement pour l'analyse de variables numériques. On les présente d'un point de vue global : À la Figure 3.1 les domaines abstraits sont classés à la rencontre de plusieurs critères. Le critère principal est celui de leur pouvoir d'expression : les flèches ordonnent les domaines abstraits selon l'inclusion de leur pouvoir d'expression. On voit que les domaines abstraits historiques occupent les deux extrêmes de cette échelle d'expressivité, et par conséquent sont aussi extrême en terme de coût des opérations.

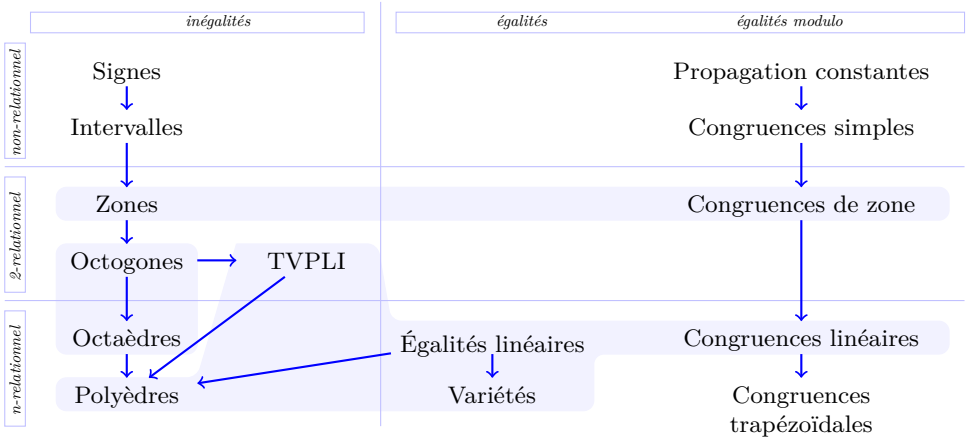
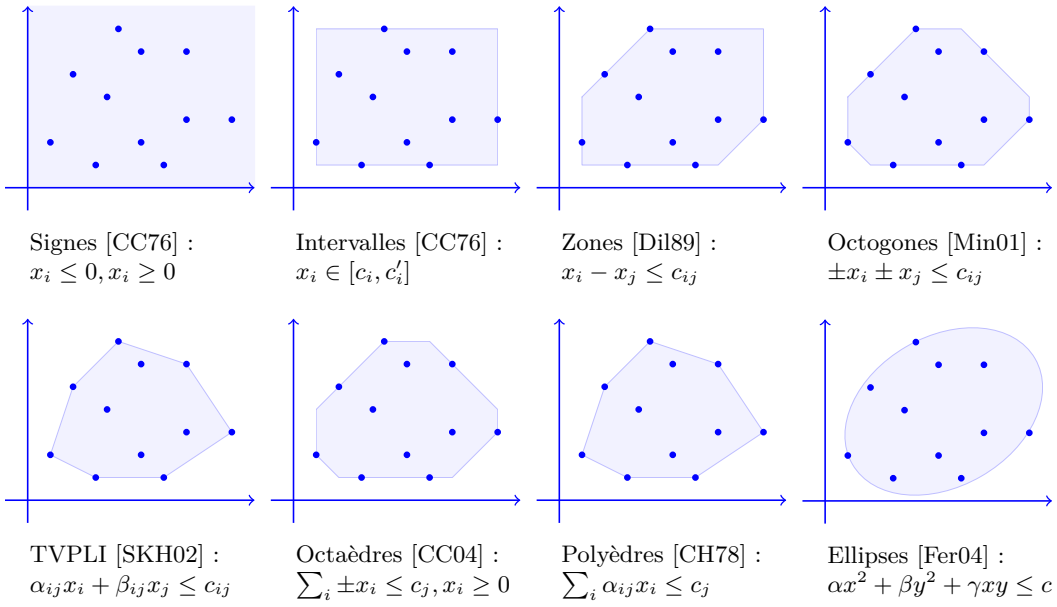



Figure 3.1 – Pouvoir expressif des domaines abstraits standards, selon le type, le pouvoir relationnel et les coefficients des contraintes qu'ils utilisent

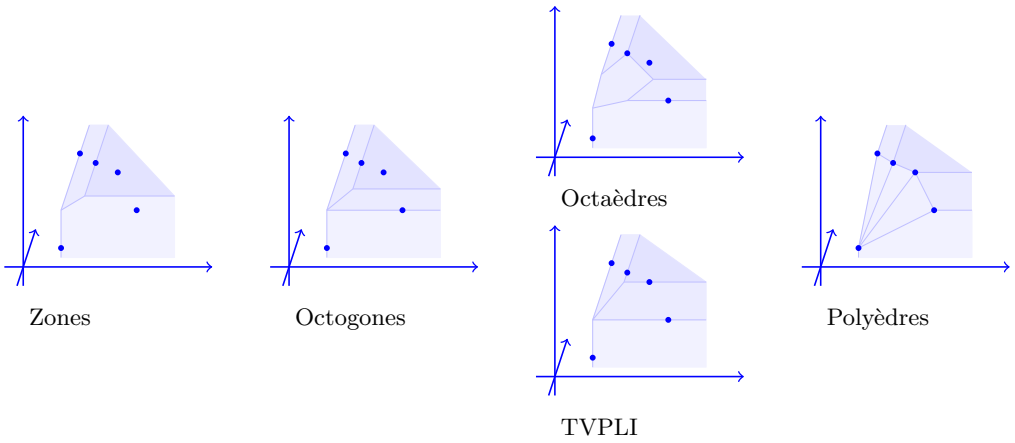
L'avènement des autres domaines correspond, soit à l'exploration de nouveaux types de contraintes (les congruences, en colonne de droite, ou encore les polynômes), soit justement à des compromis entre coût et expressivité, qui peuvent porter sur trois autres critères :

- le type de contraintes (les colonnes de la Figure 3.1) : soit des contraintes d'inégalité, soit des contraintes d'égalité seulement ;
- le pouvoir relationnel (les lignes) : celui-ci s'entend selon le nombre de variables pouvant être mises en relation dans une contrainte : une, deux ou un nombre quelconque. Le cas où l'on a deux variables a été consacré dans [Min02], sous le nom de domaines faiblement relationnels. Dans ces travaux, les domaines des zones, des octogones et des congruences de zones sont plongés dans un même formalisme ;
- les coefficients des expressions (les espaces colorés chevauchant lignes et colonnes) : c'est justement ce qui différencie les trois domaines faiblement relationnels utilisant des contraintes d'inégalités, que sont le domaine des zones, le domaine des octogones et le domaine TVPLI. Si on note $\alpha_1 x_1 + \alpha_2 x_2 \leq c$ une contrainte d'un de ces domaines, pour les zones on a $\alpha_1 \in \{0, 1\}, \alpha_2 \in \{0, -1\}$, pour les octogones $\alpha_1, \alpha_2 \in \{-1, 0, 1\}$ et $\alpha_1, \alpha_2 \in \mathbb{K}$ pour TVPLI.

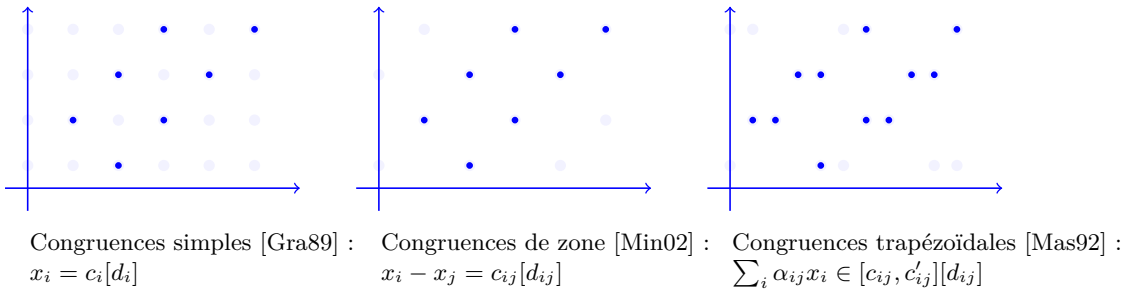
Pour rendre plus concret toutes ces différences, on donne à la Figure 3.2 pour la plupart de ces domaines, la forme exacte des contraintes qu'ils utilisent et l'abstraction qu'ils font d'un ensemble de points de $\mathbb{R}^2$, ou $\mathbb{R}^3$. Il ne manque dans cette figure que les constantes [Kil73], contraintes de la forme $x_i = c_i$, les égalités linéaires [Kar76], contraintes de la forme $\sum_i \alpha_{ij} x_i = c_j$, et les congruences linéaires [Gra91], contraintes de la forme $\sum_i \alpha_{ij} x_i = c_j [d_j]$.




(a) Illustration d'abstractions convexes d'un ensemble de points dans deux dimensions




(b) Illustration des abstractions à l'angle d'un ensemble de points dans trois dimensions




(c) Illustration d'abstractions non-convexes d'un ensemble de points dans deux dimensions

Figure 3.2 – Les différentes abstractions faites par certains domaines abstraits numériques

Enfin, il reste à évoquer la complexité de ces domaines abstraits, dont a bien compris qu'elle était corrélée à leur expressivité. Les domaines non-relationnels présentés ont des opérations linéaires dans le nombre de variables ; les domaines faiblement relationnels et les égalités linéaires, cubiques dans le nombre de variables ; le domaine TVPLI a une complexité théoriquement non-bornée, en $\mathcal{O}(n^5 \log^2 n)$ si on autorise au plus $n$ inégalités entre paires de variables ; les domaines relationnels manipulant des inégalités ont des opérations exponentielles dans le nombre de variables.

En pratique, les analyses utilisant les domaines non-relationnels peuvent passer à l'échelle. Pour les autres, lorsque l'analyse cherche à relier dans un même élément abstrait des milliers de variables, c'est peine perdue. Dans la pratique, pour assurer le passage à l'échelle avec ces domaines, on éclate, par des heuristiques simples, l'ensemble des variables en des petits paquets de variables dont on pense qu'elles sont en relation, et on analyse le programme en utilisant un élément abstrait pour chacun de ces paquets. On pourra se reporter à [Min04] pour une présentation complète de cette technique avec des octogones. Cependant, pour le domaine des polyèdres, et même si dans la pratique son comportement est plutôt polynomial, l'utilisation de paquets ne permet pas toujours de passer à l'échelle. Reste alors un autre critère à faire jouer, celui du pouvoir d'un domaine abstrait à déduire des contraintes implicites. D'où plusieurs propositions, tel que le domaine des *templates* [SSM05], qui ne sont que des polyèdres dont sont fixés, à l'avance et par des heuristiques, les coefficients des inégalités linéaires pouvant apparaître. Ou encore cela peut être le domaine des sous-polyèdres [LL09], qui a le même pouvoir d'expression que les polyèdres, mais qui utilise des opérateurs, notamment celui d'union, moins coûteux et moins précis. On fait alors usage d'heuristiques pour retrouver des contraintes que le domaine des polyèdres aurait trouvées grâce à ses procédures de déduction plus puissantes.

**Conclusion** Ce tour d'horizon effectué, on a pu voir qu'aucun domaine abstrait existant ne serait comparable avec un domaine capable d'exprimer directement des contraintes de non-égalité. Ce qui ne veut pas dire que l'on ne peut pas déduire, de certains éléments de ces domaines existants, des contraintes de non-égalité. Celles que peuvent impliquer les domaines convexes (Fig. 3.2(a)) ne sont pas utiles (par exemple $x_1 - x_2 \leq -5 \Rightarrow x_1 \neq x_2$), mais celles impliquées par les domaines de congruences (Fig. 3.2(c)) peuvent l'être beaucoup plus dans le cadre d'un produit réduit avec un domaine convexe. Par exemple, si l'on sait que $x_1 \in [0, 2]$, exprimer que $x_1 \neq 1$ peut aussi s'exprimer par $x_1 = 0[2]$.

Ces remarques font penser à la complétion disjonctive (Sec. 2.3). L'utilisation de cette construction avec des zones est illustrée à la Figure 3.3. Là encore, on voit que la disjonction de ces deux zones implique une non-égalité, $x - y \neq -1$. Cette disjonction est d'ailleurs beaucoup plus précise que la conjonction de cette non-égalité et l'union convexe de ces deux zones. On touche ici le côté impraticable des complétions disjonctives, qui sont davantage une construction théorique [1]. Lorsque l'on a be-

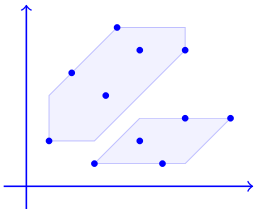


Figure 3.3 – Abstraction d'un ensemble de point dans deux dimensions par complétion disjonctive de zones

1. Bien sûr l'opérateur d'élargissement pourrait introduire une approximation, mais il serait très

soin d'autant de précision, on leur préfère les techniques de partitionnement de traces (Sec. 2.3) qui permettent de moduler la précision de l'analyse.

Nous retrouverons, à la section suivante, des structures de données permettant d'exprimer des disjonctions similaires mais aussi d'exprimer directement des contraintes de non-égalité (Sec. 3.2.1). Cependant il n'a pas été défini, sur ces structures, toutes les opérations nécessaires pour parler de domaine abstrait.

Auparavant on décrit le domaine abstrait des zones, dont on réutilisera notamment les représentations des éléments et les algorithmes dans le Chapitre 5.

### 3.1.1 Le domaine abstrait des zones

Les zones ont été introduites pour la vérification des automates temporisés. Le contrôle de ces automates est contraint, au niveau des états, par des invariants sur des horloges de la forme $x \prec c$, où $\prec \in \{<, \leq, =, \geq, >\}$ et $c$ est une constante. Il est aussi contraint au niveau des transitions, par des conditions sur les horloges de la même forme.

Lorsque l'on souhaite effectuer une analyse d'accessibilité sur ces automates [Dil89] on est amené à considérer dans quelle intervalle se trouve une horloge et la différence de deux horloges, *i.e.* des contraintes de la forme $(x_1 - x_2 \prec c)$. On appelle les contraintes de cette forme des contraintes de zones (Déf. 3.1).

Définition 3.1 (contrainte de zone). *Une contrainte de zone est soit une contrainte de potentiel $x_i - x_j \leq c$ où $x_i, x_j \in V$ et $c \in \mathbb{K}$, soit une contrainte d'intervalle $x_i \in [c_1, c_2]$, où $c_1, c_2 \in \mathbb{K}$, que l'on exprimera comme une contrainte de potentiel à l'aide d'une variable toujours égale à zéro $x_0 = 0$ : $x_i - x_0 \leq c_2$ et $x_0 - x_i \leq -c_1$.*

Au-delà de l'analyse des automates temporisés, on rencontre fréquemment des contraintes de zones dans les programmes impératifs. Par exemple sur le morceau de code suivant $x_1 \mathbf{:=} 1; \mathbf{while}\ (x_1 \leq x_2)\ \mathbf{do}\ x_1 \mathbf{:=} x_1 + 1$, l'invariant en tête de cette boucle est $1 \leq x_1 \leq x_2{+}1$. C'est-à-dire, sous forme de contraintes de zones, $x_0{-}x_1 \leq -1 \wedge x_1{-}x_2 \leq 1$.

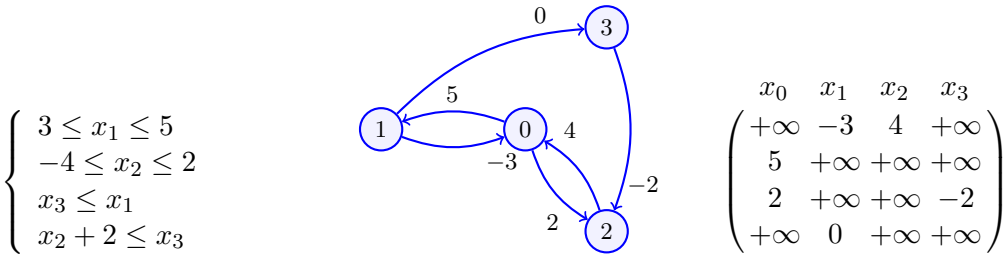
Un ensemble des points de $\mathbb{K}^n$ satisfaisant un ensemble de contraintes de zones est appelé une zone. Pour pouvoir raisonner sur cet ensemble de contraintes, on le représente sous forme de graphe, dit graphe de potentiels (Déf. 3.2). Simplement, ce graphe orienté a pour sommets les variables et utilise une arrête valuée pour représenter chaque borne que $C$ donne sur la différence de potentiel entre deux variables.

Définition 3.2 (graphe de potentiels). *Soit $C$ un ensemble de contraintes de zones, et $L = \{0, \ldots, |V|\}$ un ensemble de labels pour identifier les variables $\{x_0\} \cup V$. On note $\mathcal{G}(C) = (L, E, w)$ le graphe orienté valué défini par :*

$$\begin{array}{ll} E \subseteq L \times L & E = \{(i,j) \mid (x_i - x_j \leq c) \in C\} \\ w \in E \to \mathbb{K} & \forall e = (i,j) \in E, w(e) = \inf\{c \in \mathbb{K} \mid (x_i - x_j \leq c) \in C\}. \end{array}$$

On donne à la Figure 3.4(a) un exemple d'ensemble de contraintes de zones et à la Figure 3.4(b) le graphe de potentiels correspondant.

coûteux : par exemple pour les polyèdres, [Jea00] remarque qu'un algorithme simple sera quadratique dans le nombre de polyèdres et renverra possiblement un nombre quadratique de polyèdres.




(a) Un ensemble $C$ de contraintes de zone définies sur $V = \{x_1, x_2, x_3\}$

(b) Le graphe de potentiels $\mathcal{G}(C)$ associé à $C$

(c) Une DBM m représentant $C$

Figure 3.4 – Les deux représentations d'une conjonction de contraintes de zone

Grâce au graphe de potentiels, on se rend compte que les ensembles de contraintes de zones n'ayant pas de solutions dans $\mathbb{K}^n$ sont ceux contenant un cycle de poids strictement négatif dans leur graphe de potentiels. On trouve ce résultat sur la satisfaisabilité des conjonctions de contraintes de zones (Thm 3.1) dans [Pra77] par exemple.

Théorème 3.1 (Satisfaisabilité d'une conjonction de contraintes de zones). *Soit $C$ un ensemble de contraintes de zones, alors :*

$$C \text{ est inconsistant } \Leftrightarrow \mathcal{G}(C) \text{ contient un cycle de poids strictement négatif.}$$

Cette présentation faite des conjonctions de contraintes de zones, on se lance dans la définition proprement dite du domaine abstrait des zones, dont les éléments, ces conjonctions de contraintes, sont en pratique représentées par des matrices.

#### 3.1.1.1 Implantation par matrices de bornes (DBM)

Soit $\overline{\mathbb{K}} = \mathbb{K} \cup \{+\infty\}$.

Définition 3.3 (matrice de bornes). *Soit $C$ un ensemble de contraintes de zones sur $V$. On définit la matrice de bornes représentant $C$ comme la matrice d'adjacence de $\mathcal{G}(C)$.*

*Cette matrice est donc carrée, de taille $((n+1) \times (n+1))$, et ses éléments prennent leurs valeurs dans $\overline{\mathbb{K}}$.*

Ainsi, le nom de matrices de bornes [Dil89] (on est plus précis en anglais : *Difference Bound Matrices*) réfère simplement au fait que la matrice stocke à la ligne $i$ et la colonne $j$ la plus petite borne, s'il en est dans $C$, sur la différence $x_i - x_j$. La Figure 3.4(c) donne la matrice de bornes représentant l'ensemble de contraintes donné à la Figure 3.4(a).

Le domaine des zones $\mathcal{D}^{zone}$ a pour éléments l'ensemble des DBM possibles. On lui ajoute l'élément $\bot^{zone}$, un représentant canonique de la zone vide. On a $\mathcal{D}^{zone} = \bot^{zone} \cup \{m \mid m \in \text{DBM}(\mathbb{K})\}$.

On définit formellement la zone que représente un élément de $\mathcal{D}^{zone}$. Pour cela, on considère le domaine concret $\mathcal{D}^\flat$, qui pourrait être celui utilisé pour décrire la sémantique d'un programme manipulant les variables de $V$, par $\mathcal{D}^\flat = \mathcal{P}(Env)$, où $Env = V \to \mathbb{K}$.

On a alors pour fonction de concrétisation :

$$\begin{array}{lcl} \gamma^{zone}(\bot^{zone}) & = & \emptyset \\ \gamma^{zone}(m) & = & \{(x_1,\ldots,x_n) \to (c_1,\ldots,c_n) \in Env \mid \forall i,j \in [0,n] \quad c_i - c_j \leq m_{ij} \\ & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{où } c_0 = 0\}. \end{array}$$

Il est aisé de donner une fonction d'abstraction qui forme avec $\gamma^{zone}$ une connexion de Galois entre $\mathcal{D}^{zone}$ et $\mathcal{D}^\flat$ (*cf.* Sec. 5.3.2 pour une idée de cette fonction et de la nature exacte de cette connexion).

L'important est de remarquer qu'une même zone, qu'elle soit la zone vide ou non, peut être représentée par différents éléments de $\mathcal{D}^{zone}$ Sur notre exemple (Fig. 3.4(c)), si la matrice de bornes exprimait en plus la contrainte $x_3 \leq 5$, elle représenterait la même zone vu que cette matrice exprime déjà que $x_3 \leq x_1$ et $x_1 \leq 5$. Ainsi, on s'intéresse à la représentation canonique d'une zone par une matrice de borne, et ce, pour la précision des opération sémantiques. Prenons pour exemple l'approximation de l'union de la zone représentée par la DBM $m$ de la Figure 3.4(c) et celle représentée par la DBM $m'$ exprimant seulement que $x_3 \leq 5$ (*i.e.* $m'_{30} = 5$ et $m'_{ij} = +\infty$ sinon). Si on utilise $m$ et $m'$ telles quelles, pour trouver que la zone $\{c \in \mathbb{K}^3 \mid x_3 \leq 5\}$ est une sur-approximation correcte, il va falloir se rendre compte que $m$ représente une zone où $x_3 \leq 5$ également. L'opération de normalisation de $m$ explicitera cette contrainte.

**Normalisation** Il s'agit donc du pouvoir de déduction du domaine des zones, pouvoir dont on disait précédemment qu'il était un des critères que l'on pouvait faire jouer pour moduler le compromis entre coût et précision d'un domaine (Sec. 3.1). Ici on va choisir la déduction maximale, c'est-à-dire à partir d'une DBM $m$, trouver la DBM $\overline{m}$ qui explicite *toute* contrainte de zone implicite dans $m$. Une telle DBM, qui représente la même zone que $m$, est en fait unique. Chacune de ses bornes sur $x_i - x_j$ se définit comme le plus court chemin dans le graphe de potentiels que représente $m$, de $j$ à $i$. Attention, les plus courts chemins d'un graphe ne sont correctement définis que s'il n'existe pas de cycle de poids strictement négatif dans ce graphe. Dans ce cas la forme normale de $m$ est $\bot^{zone}$.

Définition 3.4 (forme normale). *Soit $m$ une DBM et $\mathcal{G}$ son graphe de potentiels. On note un chemin simple du sommet $i$ au sommet $j$ dans $\mathcal{G}$ par $\langle i,\ldots,j\rangle$. Alors la forme normale de $m$, notée $\overline{m}$, est :*

- *$\overline{m} = \bot^{zone}$ si $\mathcal{G}$ contient un cycle de poids strictement négatif ;*
- $\begin{cases} \overline{m}_{ii} = 0 \\ \overline{m}_{ij} = \min_{\langle i=i_1,\ldots,i_\ell=j\rangle} \sum_{k=1}^{\ell-1} m_{i_k i_{k+1}} \text{ si } i \neq j \end{cases}$ *sinon.*

*On a $\gamma^{zone}(m) = \gamma^{zone}(\overline{m})$ et $\overline{\overline{m}} = \overline{m}$.*

Dans la pratique on utilise l'algorithme de Floyd-Warshall pour calculer la fermeture des plus courts chemins (Fig. 3.5). La complexité de cet algorithme est clairement cubique dans le nombre de variables.

Outre son implantation triviale, il a l'avantage de résoudre aussi le cas où $m$ représente un ensemble de contraintes inconsistant. En effet, après son appel, tester la présence de cycles strictement négatifs revient à vérifier si sur la diagonale de la matrice il existe une valeur strictement négative. Ainsi, l'algorithme de normalisation consiste à appeler l'algorithme de Floyd-Warshall, qui retourne une DBM $m'$, et s'il existe $i \in [0,n]$ tel que $m'_{ii} < 0$ à renvoyer $\bot^{zone}$ sinon à renvoyer $m'$.

```
for k ← 0 to n do
    for i ← 0 to n do
        for j ← 0 to n do
            m_ij ← min(m_ij, m_ik + m_kj)
```

Figure 3.5 – L'algorithme de Floyd Warshall calculant la fermeture des plus courts chemins d'un graphe orienté représenté par la matrice $m$

Une propriété intéressante des DBM en forme normale est celle de la saturation de leurs bornes (Thm. 3.2). Cette propriété dit simplement que chaque contrainte $x_i - x_j \leq c_{ij}$ que représente $\overline{m}$ est une face de la zone $\gamma^{zone}(\overline{m})$. Ou, autrement dit, qu'aucune borne de $m$ ne peut-être diminuée sans modifier la zone que représente $m$.

Théorème 3.2 (saturation des DBM closes). *Soit $m$ une DBM en forme normale. Alors pour tout $i, j \in [0, n]$ :*

- $m_{ij} < +\infty \Rightarrow \exists \rho \in \gamma^{zone}(m)$ *telle que* $\rho(x_i) - \rho(x_j) = m_{ij}$ *;*
- $m_{ij} = +\infty \Rightarrow \forall M < +\infty, \exists \rho \in \gamma^{zone}(m)$ *telle que* $\rho(x_i) - \rho(x_j) \geq M$*.*

Il ne nous reste plus qu'à définir les opérateurs du domaine abstrait des zones. Pour ce qui est des opérateurs sémantiques, on se limite exclusivement à ceux auxquels on fait référence par la suite. Pour le reste, comme pour ce qui vient d'être dit, on pourra se référer à la présentation exhaustive de ce domaine, accompagnée de toutes les preuves, dans [Min04].

**Opérateurs ensemblistes** Si on veut savoir précisément si une zone représentée par une matrice $m$ est incluse dans celle représentée par une matrice $m'$, vu ce qui a été dit précédemment, il suffit de comparer les bornes de la forme normale de $m$ et les bornes de $m'$ : si ces dernières sont toutes plus grandes que les premières alors l'inclusion est garantie.

Pour l'union de deux zones représentées par $m$ et $m'$, l'opération consiste simplement à prendre point par point la borne la plus faible, non pas de $m$ et $m'$, mais de leurs formes normales afin de ne perdre aucune contrainte implicite. L'opérateur d'intersection lui, qui prend point par point la borne la plus forte, n'a pas besoin de faire appel à la normalisation de ses opérandes pour être précis.

Définition 3.5 (opérateurs ensemblistes). *On définit les opérateurs ensemblistes du domaine des zones comme suit : pour tout $m, m' \in \mathcal{D}^{zone}$,*

- $m \sqsubseteq^{zone} m' \overset{déf}{\Leftrightarrow} (\overline{m} = \bot^{zone})$ *ou* $(\forall i, j \in [1, n],\ \overline{m}_{ij} \leq m'_{ij})$ *;*
- $(m \sqcup^{zone} m')_{ij} \overset{déf}{=} \max(\overline{m}_{ij}, \overline{m'}_{ij})$ *;*
- $(m \sqcap^{zone} m')_{ij} \overset{déf}{=} \min(m_{ij}, m'_{ij})$*.*

On notera que l'on a une propriété plus forte que la monotonicité de $\gamma^{zone}$ : en effet $m \sqsubseteq^{zone} m' \Leftrightarrow \gamma^{zone}(m) \sqsubseteq^{\flat} \gamma^{zone}(m')$, ce qui se comprend de par la propriété de saturation. Par ailleurs $\sqcup^{zone}$ est la meilleure abstraction de $\sqcup^{\flat}$ et $\sqcap^{zone}$ l'abstraction exacte de $\sqcap^{\flat}$ (*cf.* Sec. 5.6 pour une idée des démonstrations).

**Opérateurs sémantiques** Pour les affectations, on donne simplement le cas d'affectations où l'expression affectée est de la forme $x_k + c$ ou $c$. D'ailleurs toute affectation avec une expression d'une forme plus complexe ne serait pas abstraite exactement, puisque son résultat ne pourrait pas s'exprimer sous la forme d'une contrainte de zone. L'affectation inversible n'est donc rien d'autre que l'incrémentation d'une variable $x_k$, et il suffit donc de translater d'autant les bornes connues sur les différences entre $x_k$ et toute autre variable :

$$(\{\!|x_k := x_k + c|\!\}^{zone}(m))_{ij} = \begin{cases} m_{ij} - c & \text{si } i = k \wedge j \neq k \\ m_{ij} + c & \text{si } i \neq k \wedge j = k \\ m_{ij} & \textit{sinon} \end{cases} .$$

On notera que si $m$ était en forme normale avant l'opération, elle l'est également après.

L'affectation non-inversible $x_k := x_\ell + c$ consiste à oublier toute information dans $m$ sur la variable $x_k$ (sauf sur la diagonale) et introduire la contrainte $x_k = x_\ell + c$ :

$$(\{\!|x_k := x_\ell + c|\!\}^{zone}(m))_{ij} = \begin{cases} -c & \text{si } i = k \wedge j = \ell \\ c & \text{si } i = \ell \wedge j = k \\ +\infty & \text{si } (i \neq j) \wedge ((i = k \wedge j \neq \ell) \vee (i \neq \ell \wedge j = k)) \\ m_{ij} & \text{sinon} \end{cases} .$$

Cette fois, même si $m$ était en forme normale, elle ne l'est pas nécessairement après l'opération. Pour l'affectation $x_k := c$ on utilise la même définition avec $\ell = 0$.

Quant à l'opérateur d'élargissement, proposé formellement par Antoine Miné [Min04] et précédemment proposé, entre autres, par Bertrand Jeannet [Jea00], il s'agit d'une simple extension matricielle de l'élargissement du domaine abstrait des intervalles (Sec. 2.2.3.1). On a $\forall m' \in \mathcal{D}^{zone}, \bot^{zone} \nabla^{zone} m' = m'$ et pour $m \neq \bot^{zone}$ :

$$(m \nabla^{zone} m')_{ij} = \begin{cases} +\infty & \text{si } m_{ij} < m'_{ij} \\ m_{ij} & \text{sinon} \end{cases} .$$

Bien entendu cet opérateur peut être amélioré en utilisant des paliers avant d'envoyer une contrainte à l'infini (*cf.* Sec. 5.7.1).

**Normalisation incrémentale** Pour clore cette présentation du domaine abstrait des zones, on s'intéresse à la normalisation incrémentale de ses éléments, dont l'utilisation est primordiale pour la complexité des algorithmes que l'on a développé (Sec. 5.5.2.1).

On considère une DBM $m \in \mathcal{D}^{zone}$ en forme normale, dont on modifie certaines bornes. Si on regarde la matrice formée des $u$ colonnes et des $u$ lignes dont aucune des bornes n'a été modifiée, cette matrice est close. Supposons que l'on réordonne les lignes et les colonnes de $m$ de telle manière que ces $u$ lignes et colonnes soient les premières de $m$, une opération qui peut être très peu coûteuse. Alors $\forall i, j, k \in [0, u-1]$ on a toujours $m_{ij} \leq m_{ik} + m_{kj}$. On utilise cette information pour modifier très simplement l'algorithme de Floyd-Warshall (Fig. 3.5) de telle manière qu'il ne fasse aucun calcul lorsque $i, j, k \in [0, u-1]$. Ceci nous donne l'algorithme de la Figure 3.6, ayant pour complexité $\mathcal{O}(n^3 - u^3)$.

On notera que si une seule ligne, et la colonne correspondante, a des bornes qui ont été modifiées, on a $u = n - 1$ et l'opération de normalisation est alors quadratique dans le nombre de variables. On tombe sur ce cas par exemple pour retrouver une forme normale après une affectation non-inversible.

**Function** update$(i, j, k) = m_{ij} \leftarrow \min(m_{ij},\ m_{ik} + m_{kj})$ ;

**for** $k \leftarrow 0$ *to* $u - 1$ **do**
  **for** $i \leftarrow 0$ *to* $u - 1$ **do**
    **for** $j \leftarrow u$ *to* $n$ **do**
      update$(i, j, k)$
  **for** $i \leftarrow u$ *to* $n$ **do**
    **for** $j \leftarrow 0$ *to* $n$ **do**
      update$(i, j, k)$
**for** $k \leftarrow u$ *to* $n$ **do**
  **for** $i \leftarrow 0$ *to* $n$ **do**
    **for** $j \leftarrow 0$ *to* $n$ **do**
      update$(i, j, k)$

Figure 3.6 – L'algorithme de Floyd-Warshall incrémental, où la matrice que forment les $u$ premières lignes et colonnes de $m$ est close pour les plus courts chemins. Sa complexité en temps est de $\mathcal{O}(n^3 - u^3)$.

#### 3.1.1.2 Extension avec des contraintes de zones strictes

Pour la vérification des automates temporisés, [Yov98] est amené à considérer des conjonctions de contraintes de zones et de contraintes de zones strictes, *i.e.* de la forme $x_i - x_j < c$. Si $\mathbb{K}$ est discret, les zones implantées avec des DBM font l'affaire. Lorsque $\mathbb{K}$ est dense, [Yov98] donne un moyen simple pour réutiliser la représentation des matrices de bornes et toute l'algorithmique que l'on vient de présenter. Il utilise pour borne un couple formé de la valeur de la borne et du type (inégalité stricte ou non) de la contrainte de zone. Ces bornes sont des éléments de $(\mathbb{K} \times \{<, \leq\}) \cup (+\infty, <)$. Par exemple les contraintes $x_i \leq 2$ et $x_j \leq x_i$ seront représentées dans une matrice $m$ par $m_{i0} = (2, <)$ et $m_{ji} = (0, \leq)$.

Pour définir ce nouveau domaine abstrait, on réutilise toutes les opérations que l'on a donné à la section précédente (sec. 3.1.1.1), avec les opérateurs suivants sur ces nouvelles bornes en lieu et place des opérateurs sur $\overline{\mathbb{K}}$. On note un type de contrainte par le symbole $\prec \in \{<, \leq\}$ et on note $b()$ la fonction permettant d'extraire la valeur d'une borne et $s()$ celle permettant d'extraire son type : $b((c, \prec)) = c$ et $s((c, \prec)) = \prec$.

– (Addition) Avec l'opération d'addition suivante sur les types :

$$\prec_1 + \prec_2 = \begin{cases} \prec_1 & \text{si } \prec_1 = \prec_2 \\ < & \text{sinon} \end{cases},$$

on définit l'addition sur les bornes par :

$$(c_1, \prec_1) + (c_2, \prec_2) = (c_1 + c_2, \prec_1 + \prec_2).$$

Si on reprend l'exemple de nos deux contraintes ci-dessus, lors du calcul des plus courts chemins par l'algorithme de Floyd-Warshall (Fig. 3.5, p. 36), on va avoir $m_{j0} = (2, <) + (0, \leq) = (2, <)$, *i.e.* $x_j < 2$, ce qui est correct puisque $x_j \leq x_i < 2$.

– (Ordre) On définit l'ordre par :

$$(c_1, \prec_1) < (c_2, \prec_2) \overset{\text{déf}}{\Leftrightarrow} (c_1 < c_2) \text{ ou } (c_1 = c_2 \wedge \prec_1 = < \wedge \prec_2 = \leq).$$

La définition des opérateurs min et max est alors directe.
On remarquera, pour les raisonnements suivants sur ces bornes, que sachant que $x \leq (1, <)$, si on s'intéresse à la borne inférieure de $-x$, celle-ci ne peut-être exprimée : il faudrait pouvoir écrire que $-x \geq (-1, >)$.

Si l'on utilise les mêmes algorithmes, on voit qu'il est tout aussi aisé d'adapter les propriétés données par ces algorithmes à ce nouveau domaine abstrait. Par exemple, pour la propriété de saturation des DBM closes (Thm. 3.2, p. 36), il suffit d'éclater le cas où $b(m_{ij}) < +\infty$ :

– $s(m_{ij}) = \leq \Rightarrow \exists \rho \in \gamma^{zone}(m)$ telle que $\rho(x_i) - \rho(x_j) = m_{ij}$ ;
– $s(m_{ij}) = < \Rightarrow \forall \varepsilon > 0, \exists \rho \in \gamma^{zone}(m)$ telle que $0 < m_{ij} - (\rho(x_i) - \rho(x_j)) \leq \varepsilon$.

Ainsi, le domaine abstrait obtenu ici est plus expressif que celui des zones. Même si ce domaine ne permet pas d'exprimer des contraintes de non-égalité directement, ses connexions avec ces contraintes sont évidentes : $x < y \Rightarrow x \neq y$.

## 3.2 Des ensembles de contraintes incluant des contraintes de non-égalité

On s'intéresse dans cette section à des travaux, relevant ou non de l'interprétation abstraite, où sont manipulées des contraintes de non-égalités, afin de puiser des représentations ou des algorithmes qui pourraient aider à la construction d'un domaine abstrait représentant de telles contraintes.

### 3.2.1 Unions finies d'ensembles convexes

Certains travaux nous interrogent sur l'opportunité d'utiliser un domaine abstrait sur le modèle de la complétion disjonctive, même si notre objectif n'est pas de pouvoir exprimer n'importe quel domaine non-convexe. En effet, pour le *model checking* des systèmes temporisés, il est utilisé en pratique, non pas une DBM, mais une union finie de DBM pour représenter les états accessibles. Dans les faits ces unions finies peuvent grossir très rapidement. Ainsi, vérifier si deux unions de DBM représentent le même domaine, ou si le domaine que représente l'une est inclus dans celui que représente l'autre, est très coûteux. Pour palier à ce problème, plusieurs structures de données ont été proposées, par exemple les CDD [LPWY99] ou les DDD [MLAH99], capables de partager les contraintes utilisées dans différentes DBM de l'union finie.

Autour de ces structures de données les questions principales qui ont été adressées ont été celles de l'équivalence, de l'inclusion, de l'intersection, de la satisfaisabilité, des disjonctions de zones qu'elles représentent. Il serait probablement possible de construire un domaine abstrait à partir de ces briques, en l'occurrence à l'algorithmique exponentielle. Et comme on l'a déjà dit pour la complétion disjonctive, ce domaine serait capable d'exprimer des non-égalités.

Ceci étant, la structure de données utilisée à la base, les BDD, se prête mal à la prise en compte particulière de ces contraintes. Si elle partage les données, elle ne permet pas le

partage des calculs. D'ailleurs rien n'est fait dans ces travaux, par exemple, pour améliorer le test de satisfaisabilité de la conjonction d'une zone $Z$ avec la disjonction des deux zones $x-y<0$ et $y-x<0$. C'est-à-dire de la conjonction de $Z$ et de la contrainte de non-égalité $x \neq y$. Le test consiste à tester la satisfaisabilité d'une part de la conjonction de $Z$ et de la zone $x-y<0$ et d'autre part celle de la conjonction de $Z$ et de la zone $y-x<0$. Lorsque $\mathbb{K}$ est dense ceci est tout à fait sous-optimal comme nous allons le voir à la section suivante.

Ainsi, construire un domaine abstrait à partir de ces travaux est une piste peu engageante pour répondre à notre objectif. Un traitement optimisé des non-égalités ne serait pas aisé au sein des structures de données en question. Il reste toujours que par essence un tel domaine n'introduit pas d'abstraction, et donc que ses algorithmes sont très coûteux.

### 3.2.2 Quelques résultats avec les contraintes d'inégalités

On s'intéresse à des résultats sur des systèmes de contraintes étudiés dans d'autres disciplines, où des contraintes de non-égalité apparaissent. Pour ces nombreux systèmes, c'est principalement, voire exclusivement, le problème de la satisfaisabilité qui est abordé.

#### 3.2.2.1 Cas dense

On commence par le cas où $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$. Afin d'avoir une vue unifiée des différents ensembles de contraintes dont on parle ici, on utilise les notions générales proposées dans [JB96].

On appelle DLR (*Disjunctive Linear Relation* en anglais), une disjonction de contraintes linéaires d'inégalité, d'égalité ou de non-égalité. Une contrainte de Horn est une DLR où au plus une des contraintes linéaires de la disjonction n'est pas une contrainte de non-égalité. Par exemple, les trois contraintes suivantes sont des contraintes de Horn :

- $(x_1 - x_2 \leq 3)$ ;
- $(x_1 + 2x_3 \neq 7)$ ;
- $(3x_1 + x_2 - x_3 = 9 \vee x_2 \neq 3 \vee x_3 - 12x_2 \neq 5)$.

Enfin, une contrainte de Horn est dite négative si elle ne contient aucune contrainte d'inégalité.

Le résultat le plus général est que si $\mathbb{K}$ est dense, alors la satisfaisabilité d'une conjonction de contraintes de Horn peut-être vérifiée en temps polynomial [Kou01], en l'occurrence par des algorithmes de programmation linéaire.

Des contraintes un tout petit peu moins générales, mais intéressantes de par le théorème qui leur est associé, sont les contraintes linéaires généralisées [LM88, LM89] (Déf. 3.6). Il s'agit simplement de contraintes de Horn où contraintes d'inégalité et contraintes de non-égalité ne peuvent pas cohabiter.

Définition 3.6 (contraintes linéaires généralisées). *Une contrainte linéaire généralisée est soit une contrainte linéaire d'inégalité, soit une contrainte de Horn négative.*

Un exemple de conjonction de contraintes linéaires généralisées est : $(x_1 - x_2 \leq 3) \wedge (x_1 + 2x_3 \neq 7) \wedge (x_2 \neq 3 \vee x_3 - 12x_2 \neq 5)$.

On a alors le théorème suivant sur la satisfaisabilité des conjonctions de contraintes linéaires généralisées (Thm 3.3), dû à [LM92], lorsque $\mathbb{K}$ est dense. Il statue l'indépendance

des contraintes de non-égalités, un résultat très important puisqu'il permet d'éviter l'explosion combinatoire que l'on aurait en traduisant naïvement toute contrainte de non-égalité comme la disjonction de deux inégalités pour vérifier la satisfaisabilité.

Théorème 3.3 (indépendance des contraintes de non-égalité). *Soit $C$ une conjonction de contraintes linéaires généralisées. On note $I$ l'ensemble des contraintes de non-égalité et $D$ l'ensemble des contraintes de Horn négatives de $C$.*

*Alors $C$ est satisfaisable si et seulement si $I$ est satisfaisable et pour tout contrainte $d \in D$, la conjonction $C \wedge d$ est satisfaisable.*

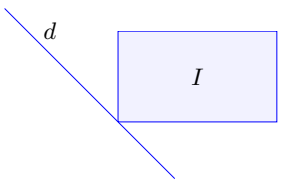
Informellement, ce théorème nous dit que si aucune contrainte de Horn négative $d$ n'est impliquée par $I$, ces contraintes $d$ de $D$ ne peuvent s'additionner pour annuler l'espace représenté par $I$. C'est donc cette indépendance des non-égalités qui permet de développer des algorithmes polynomiaux, à base de programmation linéaire, pour vérifier la satisfaisabilité des conjonctions de contraintes linéaires généralisées et du cas plus général des contraintes de Horn.

Notre attention doit aussi se porter sur des systèmes de contraintes plus simples. Les plus expressifs dont a trouvé l'étude après les systèmes de contraintes linéaires généralisées sont ceux des conjonctions de contraintes temporelles [Kou92]. Ces contraintes sont des contraintes linéaires généralisées dont les contraintes d'inégalités sont des contraintes de zones et les contraintes de non-égalité sont de la forme, $x_i - x_j \neq c$ ou $x_i \neq c$. [Kou92] donne un algorithme en $\mathcal{O}(n^4)$ pour vérifier la satisfaisabilité de conjonctions de contraintes temporelles, grâce au Théorème 3.3 qui s'applique aussi à ces contraintes.

Ensuite, on trouve les conjonctions de contraintes ORD-Horn [NB95], qui sont des contraintes de Horn dont les contraintes d'inégalités sont de la forme $x_i \leq x_j$ et les contraintes de non-égalité de la forme $x_i \neq x_j$. Leur expressivité est donc incomparable avec les contraintes temporelles. La satisfaisabilité d'une conjonction de contraintes ORD-Horn peut être vérifiée en $\mathcal{O}(n^3)$ ([Kou01] améliore légèrement cette complexité).

Enfin, il y a les contraintes de point [Vil82] qui sont soit une contrainte d'inégalité de la forme $x_i \leq x_j$, soit une contrainte de non-égalité de la forme $x_i \neq x_j$. Ces contraintes sont incluses à la fois dans les contraintes temporelles et les contraintes ORD-Horn. On vérifie en $\mathcal{O}(n^2)$ la satisfaisabilité d'une conjonction de contraintes de point : il suffit de vérifier qu'il n'existe pas une égalité et une non-égalité, dans l'ensemble de contraintes, portant sur la même paire de variables.

Au-delà de ces résultats sur la satisfaisabilité de conjonction de différentes contraintes, incluant des contraintes de non-égalité, ces travaux donnent pour les cas les plus expressifs des formes canoniques permettant de tester l'équivalence de deux systèmes ([LM92, Kou01]). Par exemple [LM92] introduit la notion de contrainte de non-égalité « précise » vis-à-vis d'un système d'inégalités linéaires pour définir la forme canonique des systèmes linéaires généralisés (ci-contre la non-égalité précise serait $x \neq 1 \vee y \neq 1$). On trouve également pour ces deux systèmes des algorithmes pour calculer l'élimination d'une variable [Imb93, Kou01].



Une contrainte de non-égalité imprécise : *EnveloppeAffine*$(I \wedge d) \neq d$

Ainsi, à partir des systèmes linéaires généralisés, ou des systèmes de Horn, on peut imaginer construire des domaines abstraits. Cependant il faut bien noter que les formes normales proposées se rapprochent de celles données pour les structures de données décrivant des unions finies de DBM (Sec. 3.2.1), et sont donc très coûteuses.

Par contre, les autres systèmes de contraintes, notamment de contraintes temporelles, laissent à penser que la construction d'un domaine abstrait, exprimant des non-égalités et des inégalités faiblement relationnelles, pourraient avoir une algorithmique « raisonnable » lorsque $\mathbb{K}$ est dense.

#### 3.2.2.2 Cas discret

Lorsque $\mathbb{K} = \mathbb{Z}$ les résultats sur la satisfaisabilité des systèmes de contraintes ORD-Horn ou de contraintes de point, sont toujours vérifiés. Ce n'est plus le cas au-delà de ces conjonctions de contraintes simples. [RH80] montre, par une simple réduction du problème de la coloration d'un graphe par trois couleurs, que le problème devient NP-complet.

Théorème 3.4 (NP-complétude du problème de satisfaisabilité de contraintes d'intervalle en présence de contraintes de non-égalité relationnelles). *Soit des contraintes permettant d'exprimer, au moins, des contraintes de non-égalité de la forme $x_i \neq x_j$ et des contraintes d'inégalités, de la forme $\pm x_i \leq c$.*

*Alors le problème de la satisfaisabilité d'une conjonction de ces contraintes est* NP*-complet.*

*Démonstration.* Soit $G = (V, E)$ un graphe non orienté, une instance du problème de 3-coloration. On construit la conjonction de contraintes suivante, sur l'ensemble de variables que forme $V$ : $x_i \in [1, 3] \Leftrightarrow x_i \in V$ et $x_i \neq x_j \Leftrightarrow (x_i, x_j) \in E$. Ce encodage est polynomial et $G$ admet un coloriage correct si et seulement si cette conjonction de contrainte est satisfaisable. □

Ainsi, lorsque [HS97] conçoit une procédure pour vérifier la satisfaisabilité dans $\mathbb{Z}$ des conjonctions de contraintes d'octogones, et qu'il s'intéresse à résoudre ce problème en présence de contraintes de non-égalités de la forme $\pm x_i \pm x_j \neq c$, il fait tout de suite le choix de l'incomplétude. Il vérifie seulement si une de ces non-égalités contrarie l'une des égalités, implicite ou non, formée par les contraintes d'octogones.

À notre connaissance, tous les autres travaux qui ont à faire face à ce problème, et qui souhaitent être complets, considèrent les $2^d$ problèmes de satisfaisabilité obtenus en voyant les $d$ non-égalités du système comme une disjonction de deux contraintes d'inégalités (comme les CDD et les DDD, à la section précédente). À l'exception de [KJS07], qui pour les solveurs modulo théorie, propose pour les conjonctions de contraintes de zones et de contraintes de non-égalité de la forme $x_i - x_j \neq c$ ou $x_i \neq c$, une amélioration dans le cas très particulier où un ensemble de variables sont toutes non égales deux à deux. Cette situation est connue en programmation logique par contraintes sous le nom de contrainte « *alldifferent* ». On se reportera à la Section 5.5.1 pour de plus amples détails.

## 3.3 Conclusion

Nous avons fait un tour d'horizon des domaines abstraits numériques existants, et noté qu'aucun d'entre eux ne pouvait exprimer directement des contraintes de non-égalité.

Puis, nous avons défini de manière détaillée le domaine abstrait des zones sur lequel nos travaux vont s'appuyer (Ch. 5). Il est donc évident que l'on cherchera à développer un domaine abstrait avec une algorithmique polynomiale, au moins cubique dans le nombre de variables.

Cela dit, nous avons justement appris qu'augmenter n'importe quel des domaines abstraits convexes existant avec des contraintes de non-égalité donnera forcément un domaine abstrait avec une algorithmique exponentielle pour atteindre une précision maximale. Il faudra faire avec cette contradiction dans le cas discret. Par ailleurs, dans le cas dense, nous avons vu des résultats sur la satisfaisabilité de plusieurs classes de contraintes, incluant des contraintes de non-égalité, encourageants. Notamment la classe des contraintes temporelles, qui laisse espérer la possibilité de définir un domaine abstrait avec une complexité en $\mathcal{O}(n^4)$.

# Chapitre 4

# Analyses du contenu des tableaux

Nous terminons cette partie avec l'état de l'art sur le second sujet de cette thèse, les analyses du contenu des tableaux. Il s'agit d'un domaine récent, ces analyses ayant été proposées durant la dernière décénie. Ces propositions émanent de toutes les branches de l'analyse statique : analyses basées sur des logiques décidables (Sec. 4.2.1), sur l'abstraction par prédicats (Sec. 4.2.2), sur l'interprétation abstraite (Sec. 4.2.3 et 4.3.1–4.3.3). On présente ici une dizaine d'analyses différentes et ce faisant, nous sommes proches d'être exhaustif.

On notera que toutes ces analyses s'affranchissent de vérifier que les tableaux des programmes qu'elles analysent sont bien accédés dans leur bornes. En pratique, les programmes que les analyses du contenu de tableaux arrivent à analyser avec succès sont des programmes dont il est très simple de valider les accès aux tableaux, par exemple avec une des analyses numériques présentée à la Section 3.1. D'ailleurs, les invariants découverts par les analyses du contenu des tableaux impliquent très souvent la validité de ces accès.

Avant de présenter ces analyses, il nous a paru important d'essayer de lister quelles étaient ces propriétés sur le contenu des tableaux que l'on rencontre dans les programmes et d'essayer d'unifier dans une même classe l'ensemble des propriétés que découvrent les analyses existantes (Sec. 4.1).

Une caractéristique essentielle de l'analyse que l'on veut concevoir (Sec. 1.2) est son automaticité. Ainsi, c'est selon ce critère que l'on organise la suite de ce chapitre : on commence par présenter les analyses semi-automatiques (Sec. 4.2) où l'intervention de l'utilisateur est plus ou moins nécessaire à travers des annotations du programme. C'est dans cette catégorie qu'on trouve la majorité des analyses existantes. Puis, on présente les analyses automatiques (Sec. 4.3), moins nombreuses, et ce, en détail. D'une part, parce que nous avons trouvé dans ces travaux des techniques que nous avons réutilisées. D'autre part, parce que lorsque nous avons présenté notre analyse (Ch. 6) à d'autres équipes de recherche [1] nous nous sommes rendus compte que les capacités de ces analyses étaient surévaluées par la communauté, or nous souhaitons que le lecteur puisse clairement situer notre travail.

Avant de clore le chapitre, on récapitule, notamment dans une même table, les résultats principaux et les performances de ces dix analyses (Sec. 4.4). Enfin, en conclusion (Sec. 4.5), on s'intéresse succinctement à une analyse du contenu des listes chaînées

1. Notamment au SRI (Menlo Park), à *Microsoft Research* (Redmond) et au LRI (Paris).

intéressante pour notre sujet.

## 4.1 Propriétés sur le contenu des tableaux

On se restreint ici à des tableaux à une dimension. On distingue deux grands types de propriétés selon qu'elles portent ou non sur les multi-ensembles de valeurs que contiennent les tableaux.

Une vaste classe de propriétés du second type (ne portant pas sur des multi-ensembles de valeurs), et qui inclue les propriétés les plus simples que l'on rencontre en général, rassemble les conjonctions de propriétés de la forme suivante (Éq. 4.1). Il s'agit d'implications où certaines variables $\ell_1, \ldots, \ell_n$ sont quantifiées universellement. En partie gauche de l'implication, on trouve une propriété scalaire $\varphi$ portant sur des variables numériques entières $i_1, \ldots, i_m$ du programme, et les variables quantifiées. En partie droite de l'implication, on trouve une propriété scalaire $\psi$ sur le contenu de tableaux $a_1, \ldots, a_p$ du programme, aux indices renvoyés par les fonctions $f_1, \ldots, f_p$ des variables quantifiées. Des variables scalaires du programme $x_1, \ldots, x_q$, ou les variables quantifiées elles-mêmes, peuvent être mises en relation avec le contenu de ces cellules de tableaux.

$$\begin{aligned}&\forall \ell_1, \ldots, \ell_n,\\ &\quad\Big(\varphi(\ell_1, \ldots, \ell_n, i_1, \ldots, i_m) \Rightarrow\\ &\qquad \psi(a_1[f_1(\ell_1, \ldots, \ell_n)], \ldots, a_p[f_p(\ell_1, \ldots, \ell_n)], x_1, \ldots, x_q, \ell_1, \ldots, \ell_n)\Big).\end{aligned} \tag{4.1}$$

Les conjonctions de telles propriétés permettent par exemple d'exprimer qu'un tableau représente un tas binaire maximum : $\forall \ell, (\ell \in [1, \lfloor \frac{n}{2} \rfloor] \Rightarrow a[\ell] \geq a[2\ell] \wedge a[\ell] \geq a[2\ell+1])$, qu'un tableau est trié en ordre croissant : $\forall \ell, (2 \leq \ell \leq n \Rightarrow a[\ell-1] \leq a[\ell])$, ou encore qu'un tableau ne contient pas de doublons : $\forall \ell_1, \ell_2, (1 \leq \ell_1 < \ell_2 \leq n \Rightarrow a[\ell_1] \neq a[\ell_2])$ [2]. Dans cette classe, on trouve aussi la propriété suivante $\forall \ell, (1 \leq \ell \leq 56 \Rightarrow 8\ell + a[\ell] = [-3, 5][63])$, vérifiée par le contenu d'un tableau numérique utilisé pour encrypter des données [3].

Bien entendu, aussi vaste que soit cette classe, elle n'inclue pas de nombreuses propriétés de tableaux que l'on rencontre aussi couramment. C'est le cas des propriétés où les tableaux sont indexés par le contenu d'un autre tableau $(\forall \ell, (1 \leq \ell \leq n \Rightarrow a[b[\ell]] = 0))$. C'est évidemment le cas des propriétés nécessitant des quantifications existentielles pour être exprimées. Par exemple, le fait que le contenu de la variable $x$ est dans le tableau $a$ : $(\exists \ell \in [1, n], a[\ell] = x)$, ou encore le fait que le contenu d'un tableau soit inclus dans celui d'un autre : $\forall \ell_1 (\ell_1 \in [1, n] \Rightarrow (\exists \ell_2 \in [1, m], a[\ell_1] = b[\ell_2]))$. Si on regarde du côté du contenu des matrices, il y aussi le fait qu'une matrice est une matrice de permutation. Pour exprimer la propriété vérifiée par une ligne d'une telle matrice, il faut une alternance de quantificateurs, comme le souligne [HHKV10] : $\forall \ell_1 (\ell_1 \in [1, n] \Rightarrow (\exists \ell_2 \in [1, n], a[\ell_1, \ell_2] = 1 \wedge \forall \ell_3 (\ell_2 \neq \ell_3 \in ([1, n]) \Rightarrow a[\ell_1, \ell_3] = 0))$. Cependant, ces dernières propriétés, utilisant des quantifications existentielles, peuvent

2. Cette propriété peut être très intéressante pour des analyses de dépendance des indices, lorsque le tableau $a$ sert à indicer un autre tableau [PW98].

3. [Mas93] s'intéresse à cette propriété, car comme on le voit, elle peut s'exprimer à l'aide d'une congruence trapézoïdale (Fig. 3.2(c), p. 31), qui est l'un des sujets de sa thèse. Le tableau en question est {60 52 44 36 28 20 12 4 59 51 43 35 27 19 11 3 58 50 42 34 26 18 10 2 57 49 41 33 25 17 9 1 64 56 48 40 32 24 16 8 63 55 47 39 31 23 15 7 62 54 46 38 30 22 14 6}

être exprimées par des propriétés d'une deuxième classe, portant sur des multi-ensembles de valeurs.

On définit cette seconde classe de propriétés informellement comme les conjonctions de propriétés scalaires sur des parties du contenu de tableaux, ou sur des « fonctions » de ces parties. Une partie du contenu d'un tableau représente un multi-ensemble de valeurs, que l'on notera $\{\!\{\ldots\}\!\}$. Si on prend pour fonction le « cardinal » d'un multi-ensemble, on peut alors exprimer la propriété suivante : le tableau $a$, de n cases, contient au plus $i$ valeurs positives $(card(\{\!\{a[\ell] \geq 0 \mid 1 \leq \ell \leq n\}\!\}) \leq i)$ [4]. On peut aussi décrire la propriété que les vecteurs représentés par deux tableaux $a$ et $b$, de taille $n$, sont orthogonaux, grâce à la fonction « somme » : $(sum(\{\!\{a[\ell] \times b[\ell] \mid 1 \leq \ell \leq n\}\!\}) = 0)$. Plus simplement, on peut aussi exprimer la propriété que le contenu d'un tableau est une permutation du contenu d'un autre : $(\{\!\{a[\ell] \mid 1 \leq \ell \leq n\}\!\} = \{\!\{b[\ell] \mid 1 \leq \ell \leq n\}\!\})$.

La première classe de propriétés que l'on a décrite ci-dessus (Éq. 4.1), inclut quasiment tous les invariants découverts par les différentes analyses existantes dans la littérature (Sec. 4.2 et 4.3), à trois exceptions près d'analyses récentes. Deux d'entre elles s'intéressent à montrer qu'un tableau est une permutation d'un autre. La première [SG09] suppose que les tableaux ne contiennent pas deux fois le même élément et ainsi ramène le problème à montrer l'inclusion des ensembles que représentent les deux tableaux. Cette analyse arrive donc à découvrir des propriétés avec une quantification universelle et une quantification existentielle (invariant évoqué ci-dessus). Cependant, son intérêt est diminué de part l'existence de la seconde analyse [PH10], puisqu'elle manipule directement des multi-ensembles. Cette analyse tombe dans la seconde classe. Tout comme la troisième, proposée dans [GLS09] à partir d'une analyse de forme, qui est capable de découvrir des relations entre les cardinaux de telles listes ou tels arbres (par exemple, entre une liste et sa copie, ou entre deux listes et leur concaténation). Même si la problématique est quelque peu différente, cette analyse peut donner des idées pour la définition d'une analyse du contenu des tableaux découvrant des invariants similaires.

Notre analyse (Ch. 6) vise un sous-ensemble bien défini de la première classe. On en restera donc dans la suite à présenter les analyses découvrant des propriétés de cette même classe.

## 4.2 Des analyses semi-automatiques

On trouve dans cette section les analyses basées sur des logiques décidables (Sec. 4.2.1) et principalement les analyses basées sur l'abstraction par prédicats (Sec. 4.2.2). Si en général ces techniques d'analyses nécessitent des informations de la part de l'utilisateur pour découvrir des invariants, elles ont parfois la capacité de découvrir automatiquement certains invariants alors moins précis ou pour une classe de programmes plus restreinte.

4. Cette propriété est utilisée pour prouver qu'un petit programme (Fig. B.1(b), p. 204), où toutes les valeurs positives de $a$ sont copiées dans un tableau $b$ de taille $i$, n'accède pas hors des bornes du tableau $b$ [HM07].

### 4.2.1 Logiques décidables

Quelques travaux portent sur des logiques décidables permettant d'exprimer des propriétés sur le contenu de tableaux numériques. Parmi ceux-ci, [BMS06], en plus de donner une famille de logiques décidables, identifie également plusieurs extensions de ces logiques qui les rendraient indécidables. Pour fixer les idées sur ces travaux, l'intersection du fragment décidable proposé dans [BMS06], avec la classe de propriétés donnée à l'Équation 4.1 (p. 46), est l'ensemble de propriétés suivantes :

$$\forall \ell_1, \ldots, \ell_n, (\varphi(\ell_1, \ldots, \ell_n, i_1, \ldots, i_m) \Rightarrow \psi(a_1[\ell_1], \ldots, a_n[\ell_n], x_1, \ldots, x_q)),$$

où $\varphi$ est formée de conjonctions et/ou de disjonctions de contraintes de zone (où la constante est nulle si la contrainte de zone porte sur deux variables quantifiées). Permettre qu'apparaissent les cellules $a[\ell]$ et $a[\ell+1]$ dans une même propriété, à droite de l'implication, casse le résultat de décidabilité.

Les procédures de décision données dans la littérature avec ces différentes logiques, permettent de vérifier des programmes totalement annotés. C'est-à-dire où pré-conditions, post-conditions et invariants de boucle du programme sont décrits par des propriétés de la logique. Par exemple, [BMS06] vérifie plusieurs programmes de tri ainsi : tri par insertion, à bulle, par fusion, ou encore tri rapide. Bien entendu ces techniques sont éloignées de notre sujet, qui est de découvrir automatiquement ces propriétés. Cependant, récemment [BHI+09] a proposé une méthode pour découvrir, seulement à l'aide des pré-conditions, les post-conditions de certains programmes. La logique décidable sur laquelle ils se basent, permet de représenter des propriétés de la forme suivante :

$$\forall \ell, (\varphi(\ell, i_1, i_2) \Rightarrow \psi(a_1[\ell], a_2[\ell+1], x_1)),$$

où $\varphi$ est une conjonction de contraintes de zone et $\psi$ est une contrainte de zone, où soit $a_1[\ell]$ apparaît, soit $a_1[\ell]$ et $a_2[\ell+1]$. On notera que la présence dans une même propriété de $a[\ell]$ et $a[\ell+1]$ est possible, *a contrario* de la logique de [BMS06]. En contrepartie, il n'y a plus qu'une seule variable quantifiée autorisée [HIV08].

La particularité de cette logique est que pour chacune de ses propriétés, ses modèles correspondent naturellement à l'ensemble de traces d'un automate à compteurs plat [5]. Or ces automates peuvent bien entendu modéliser la sémantique de certaines boucles, non imbriquées, de programmes impératifs. Ce sont justement les boucles qui posent problème dans l'analyse automatique des programmes par ces logiques décidables. Ici, en présence d'une boucle et de sa pré-condition, l'analyse consiste à composer l'automate à compteur plat représentant les modèles de cette pré-condition et l'automate à compteur représentant la relation induite par la boucle, sur les variables scalaires et le contenu des tableaux. L'automate résultant représente l'ensemble des états possibles après n'importe quel nombre d'itérations de la boucle. On peut déduire de cet automate une propriété de la logique, qui n'est autre que la post-condition de la boucle.

**Conclusion** Cette technique est évidemment fort coûteuse. Outre qu'elle ne permet pas de trouver les invariants de boucles, l'expressivité des propriétés pouvant être découvertes

5. Les automates à compteurs contiennent les automates temporisés. Les opérations autorisées sur les compteurs sont des formules de Presburger. Ces automates sont dits plats dès lors que pour tout nœud $q$, il existe au plus un cycle contenant $q$.

est très restreinte. De plus, ces restrictions ne peuvent être levées sans recommencer le travail de recherche : on pense par exemple à la limitation sur les imbrications de boucles. On notera tout de même, comme l'indique la Figure 4.4 (p. 68), que quelques petits exemples intéressants sont analysés avec succès (aucun temps d'analyse n'est donné) : par exemple une boucle insérant en bonne place une valeur dans un tableau trié (autrement dit, la boucle interne du tri par insertion).

### 4.2.2 Abstraction par prédicats

On trouve plusieurs travaux utilisant l'abstraction par prédicats [GS97], une technique particulière d'interprétation abstraite, pour découvrir des invariants sur le contenu des tableaux.

L'abstraction par prédicats considère un domaine abstrait particulier pour chaque programme analysé. Ce domaine est construit à partir d'un ensemble de prédicats définis sur les variables du programme (et selon les points de contrôle). Ces prédicats sont fournis par l'utilisateur et/ou une pré-analyse à base d'heuristiques du programme. Les éléments du domaine abstrait sont des combinaisons booléennes de ces prédicats. Le domaine abstrait est donc fini, il n'y a pas besoin d'opérateur d'extrapolation. L'inconvénient de cette généricité est que le calcul des abstractions des fonctions de transfert doit passer par le domaine concret. Avec les notations du Chapitre 2, une manière basique de calculer $\{\!|f|\!\}^\sharp(x^\sharp)$ consiste à trouver, par énumération, l'ensemble des valeurs abstraites $y^\sharp$ approximant $f \circ \gamma(x^\sharp)$. Pour décider si une valeur abstraite $y^\sharp$ est une telle approximation, on utilise un démonstrateur automatique de théorèmes. Une abstraction correcte de $f$ est alors l'intersection de ces valeurs abstraites :

$$\{\!|f|\!\}^\sharp(x^\sharp) = \sqcap^\sharp\{y^\sharp \in \mathcal{D}^\sharp \mid f \circ \gamma(x^\sharp) \sqsubseteq^\flat \gamma(y^\sharp)\}.$$

Cette abstraction n'est en général pas exacte puisque les démonstrateurs sont en pratique incomplets, et donc certaines valeurs abstraites $y^\sharp$ peuvent manquer au calcul de l'intersection. Surtout, on voit qu'avec cette technique basique, un nombre exponentiel (dans le nombre de prédicats) de requêtes au démonstrateur est nécessaire.

Bien entendu de nombreux travaux de recherche se sont attelés à ce dernier problème, qui n'est autre que celui de l'abstraction (de $f \circ \gamma(x^\sharp)$). Le cas exponentiel est alors devenu le pire cas [DDP99, SS99]. Par exemple l'idée développée dans [DDP99] consiste simplement à regarder d'abord les valeurs abstraites $y^\sharp$ composées d'un seul prédicat. Si une telle valeur abstraite n'approxime pas l'image de la fonction $f$ il en sera de même pour toute autre valeur abstraite ayant la même valuation de ce prédicat. Bien sûr, il existe de nombreuses approximations, par exemple l'analyseur SLAM limite simplement le nombre de prédicats par requêtes au démonstrateur. Cependant, l'abstraction par prédicats reste en pratique une technique très coûteuse.

Lorsque l'on souhaite l'utiliser pour découvrir des invariants quantifiés, la première question qui se pose est celle des prédicats à manipuler. Si l'on prend des prédicats quantifiés, les combinaisons booléennes formant les valeurs abstraites vont s'effectuer entre ces prédicats. Il est alors possible d'exprimer des invariants très complexes, comme $(\forall \ell, a[\ell] = 0) \Rightarrow x = 0$, mais par contre on empêche toute combinaison sous les quantificateurs. Par exemple les prédicats $(\forall \ell, 1 \leq \ell < i \Rightarrow a[\ell] = 0)$ et $(\forall \ell, 1 \leq \ell < i \Rightarrow a[\ell] = 1)$ ne peuvent se combiner pour exprimer l'invariant $(\forall \ell, 1 \leq \ell < i \Rightarrow a[\ell] = 0 \vee a[\ell] = 1)$.

Ainsi, on se retrouve très rapidement à découvrir ou demander à l'utilisateur (et c'est plutôt à lui que cela incombe le plus couramment comme on va le voir) un prédicat qui est l'invariant même que l'on cherche à inférer. L'autre problème soulevé par cette solution, théorique cette fois, est qu'elle oblige la procédure de décision utilisée pour l'abstraction à répondre si des formules logiques du premier ordre sont satisfaisables, ce qui est indécidable. Il est alors nécessaire d'utiliser des procédures d'instanciation des quantificateurs universels [GM09], incomplètes donc et qui vont surtout multiplier le nombre de termes dans les requêtes envoyées au démonstrateur, compliquant d'autant le passage à l'échelle de l'analyse.

#### 4.2.2.1 Prédicats quantifiés implicitement

Une autre approche, proposée par [FQ02] consiste à continuer d'utiliser des prédicats non-quantifiés, mais pouvant porter aussi sur des variables fraîches, non présentes dans le programme. Pour découvrir l'invariant évoqué ci-dessus, l'ensemble de prédicats suivant doit être fourni : $\{\ell \geq 1, \ell < i, a[\ell] = 0, a[\ell] = 1\}$ où $\ell$ est l'une des ces variables fraîches. Dans ces prédicats dit « indicés » [LB04b], la valeur de ces variables est fixée, à une valeur inconnue, pour toute l'exécution du programme. Ainsi, en faisant une analyse par abstraction de prédicats classique, il est correct de quantifier son résultat sur ces variables particulières. Si le résultat de l'analyse est $\ell \geq 1 \wedge \ell < i \Rightarrow a[\ell] = 0 \vee a[\ell] = 1$, alors elle a découvert l'invariant quantifié ci-dessus.

Si cette approche est plus faible que celle utilisant des prédicats quantifiés, elle a l'avantage d'exiger des prédicats simples, ce qui est intéressant lorsque l'utilisateur est impliqué. De plus, ces travaux [FQ02], où seule la découverte des invariants de boucle est visée, ne considèrent qu'un ensemble de prédicats par boucle, ce qui rend l'effort d'annotation du programme raisonnable. Le revers de ce dernier choix est qu'une analyse qui ne considère pas des prédicats en tous points de contrôle perd la possibilité de diminuer fortement sa complexité en considérant des ensembles de prédicats locaux plus restreints mais suffisants [HJMM04].

Un ensemble d'heuristiques très simple est donné par [FQ02] pour découvrir des prédicats adéquats. Cependant, aucun exemple d'invariant inféré grâce à ces heuristiques n'est donné. Pour donner une idée de ce qu'ils pourraient être, on rappelle ici les prédicats indexés que ces heuristiques proposent pour une boucle où le contenu du tableau $a$ est modifié :

- $\ell \geq 1$ et $\ell < i$ pour toute variable entière $i$ dans la portée courante ;

et selon le type du contenu de $a$ :

- $a[\ell] \neq \mathit{null}$ s'il s'agit de pointeurs ;
- $a[\ell] < \ell, a[\ell] \leq 0$ et $a[\ell] \geq 0$ s'il s'agit d'entiers.

Ainsi, probablement seules des propriétés unaires sur les tableaux, très spécifiques, peuvent être trouvées automatiquement par leur analyse. On ne peut argumenter qu'il ne s'agit ici que d'un exemple et qu'il suffirait d'étendre ou de modifier rapidement ces heuristiques pour obtenir de meilleurs résultats. Vu la sensibilité de la précision et des performances des analyses par abstraction de prédicats aux prédicats choisis, leur découverte est un problème complexe [6].

6. À propos des performances, [LB04b] rapporte l'exemple d'un programme conséquent dont les temps d'analyse pour deux ensembles de prédicats, l'un choisi par l'utilisateur, l'autre de manière semi-automatique et ayant la plupart de leurs prédicats en commun, diffèrent d'un facteur 20.

Ceci étant, si on donne manuellement l'ensemble des prédicats adéquats, leur analyse peut trouver des invariants de boucle quantifiés plus complexes (par exemple de tri : $\forall \ell_1, \ell_2, 1 \leq \ell_1 < \ell_2 \leq i \Rightarrow a[\ell_1] \leq a[\ell_2]$). En tout, deux exemples sont donnés. L'un est un algorithme recherchant séquentiellement le premier élément non nul d'un tableau d'entiers (Fig. 7.4(b), p. 186). L'autre est le tri par sélection (Fig. B.3(b), p. 205) pour lequel il faut également donner les prédicats permettant de trouver l'invariant de la boucle interne. En pratique, ces analyses semi-automatiques s'effectuent en un peu moins de vingt secondes en moyenne.

**Conclusions** L'approche proposée par [FQ02] pour découvrir des invariants quantifiés a l'avantage de permettre la réutilisation directe de techniques existantes. Cette approche est d'ailleurs reprise par quasiment tous les travaux suivants.

Elle est notamment formalisée dans [LB04a] et généralisée : les variables quantifiées le sont à chaque point de contrôle et non globalement à toute une boucle. Concrètement cela veut dire que l'opération d'abstraction utilise pour domaine concret des formules universellement quantifiées. En échange d'une précision accrue, on retombe sur le problème évoqué précédemment d'instanciation de ces quantificateurs par le démonstrateur. Cette technique est déjà utilisée dans [LBC03] mais aucun exemple automatique n'est rapporté par les auteurs, ni un nouvel exemple de programmes manipulant des tableaux.

Finalement, le problème n'est pas tant la précision que la découverte automatique des prédicats indexés adéquats. Or, à ce jour, aucune solution satisfaisante n'existe puisque les travaux capables de découvrir de tels prédicats avec succès se basent tous sur un objectif de preuve fourni par l'utilisateur. Bien que ceci les éloigne de notre sujet, on rappelle brièvement les principes et les résultats de ces techniques (Sec. 4.2.2.3), car elles apportent plusieurs résultats d'analyse de programmes manipulant des tableaux, qui sont somme toute peu nombreux dans la littérature.

#### 4.2.2.2 Combinaison des prédicats restreinte par patrons

Toujours dans la lignée des travaux où c'est l'utilisateur qui fournit les ensembles de prédicats, [SG09] ont récemment proposé de demander aussi à l'utilisateur un objectif de preuve et des « patrons », formules logiques décrivant la structure des invariants de boucle à découvrir. Par exemple, le patron suggéré pour le tri par sélection (Fig. B.3(b), p. 205) est $v_1 \wedge (\forall \ell, v_2 \Rightarrow v_3) \wedge (\forall \ell, v_4 \Rightarrow v_5) \wedge (\forall \ell_1, \ell_2, v_6 \Rightarrow v_7)$. Les invariants qui peuvent être découverts sont toutes les instances possibles de ce patron avec des conjonctions de prédicats (issus de l'ensemble de prédicats proposé par l'utilisateur) en lieu et place des variables $v_i$.

Les auteurs réduisent le calcul du point fixe à la recherche systématique d'une valuation des variables $v_i$ telle que la propriété correspondante soit un invariant du programme. Cette valuation doit être optimale, c'est-à-dire que la propriété correspondante renforcée par l'ajout/le retrait d'un prédicat quelconque n'est plus un invariant. Un algorithme basé sur un SMT (*Satisfiability Modulo Theories*) solveur est proposé pour trouver une valuation optimale, en évitant une recherche exhaustive triviale notamment à l'aide de l'objectif de preuve.

En pratique, cette technique permet de découvrir les invariants de boucle de plusieurs algorithmes complexes : trois algorithmes de tri (par insertion, par sélection et à bulle) et

l'algorithme de segmentation utilisé par le tri rapide. Grâce à l'efficacité du SMT solveur, ces programmes sont vérifiés en moyenne en un peu plus d'une seconde. Le problème est qu'il ne s'agit plus vraiment de découvrir des invariants vu la quantité d'informations mises à disposition de l'analyse par l'utilisateur. D'ailleurs, si l'on prend le point de vue de l'utilisateur, il est difficile de croire par exemple pour le tri par sélection qu'il ait pu fournir les prédicats nécessaires suivants : $\{x_i - x_j \leq c \mid x_i, x_j \in \{\ell, \ell_1, \ell_2, i, j, k, n\}, c \in [-1, 1]\} \cup \{x_i - x_j \leq c \mid x_i, x_j \in \{a[t] \mid t \in \{\ell, \ell_1, \ell_2, i, j, k, n\}\}, c \in [-1, 1]\}$ et le patron ci-dessus, sans justement connaître précisément cet invariant ! Cela dit, on peut imaginer que cette technique puisse être automatisée, par exemple avec les techniques que l'on présente maintenant pour découvrir les prédicats à partir d'un objectif de preuve et de nouvelles techniques pour découvrir les patrons adéquats.

#### 4.2.2.3 Raffinement de l'abstraction

On termine cette section sur les techniques d'abstraction par prédicats qui raffinent automatiquement l'ensemble de prédicats utilisés pour l'analyse, jusqu'à ce que cet ensemble permette de vérifier une propriété $P$ fournie par l'utilisateur.

Ces techniques reposent sur l'étude de contre-exemples fournis par *model-checking*. Une itération de raffinement se déroule comme suit [CGJ+00]. Abstraction par prédicats sur l'ensemble de prédicats courants ; *Model-checking* du programme abstrait correspondant : si $P$ est prouvée c'est la fin de l'analyse. Sinon un contre-exemple est obtenu ; Simulation de ce contre-exemple sur le programme (concret), le plus souvent par *model-checking* borné : si $P$ est réfutée c'est la fin de l'analyse. Sinon c'est que le contre-exemple est dû à l'abstraction. Il faut alors choisir à partir des informations apportées par la simulation de ce contre-exemple un ensemble de prédicats à ajouter aux prédicats courants tel qu'à la prochaine itération, ce contre-exemple ne soit plus possible.

En pratique, il faut proposer des prédicats qui élimineront du même coup plusieurs autres contre-exemples similaires, en généralisant les résultats de la simulation. Ces nouveaux prédicats doivent être peu nombreux pour éviter que l'abstraction soit trop détaillée et que l'analyse diverge. On trouve à ce propos des techniques d'élargissement sur l'ensemble de prédicats [BPR02].

Pour des programmes manipulant des tableaux cette généralisation doit résulter en des prédicats quantifiés. On prend comme exemple le programme ci-contre avec son objectif de preuve. Lors de la première analyse, l'ensemble de prédicats est grossier et le résultat est totalement imprécis sur le contenu de $a$. Ainsi un contre-exemple trivial est trouvé : c'est la trace consistant à passer une fois dans la première boucle et enfreindre l'assertion au premier passage dans la seconde boucle. Bien entendu ce contre-exemple est faux et sa simulation informe que savoir que $a[1] = 0$ permet de l'éliminer. Ajouter le prédicat $a[1] = 0$ à l'abstraction n'éliminerait que ce contre-exemple particulier et non les suivants qui viendront du déroulement des boucles et nécessiteront les prédicats $a[2] = 0$, etc, pour être éliminés. Si le processus de raffinement n'infère pas le prédicat quantifié $\forall \ell, 1 \leq \ell < i \Rightarrow a[\ell] = 0$ alors il divergera.

```
1  i := 1 ;
2  while i ≤ n do
3  |   a[i] := 0 ;
4  |   i := i + 1
5  i := 1 ;
6  while i ≤ n do
7  |   assert(a[i] = 0) ;
8  |   i := i + 1
```

On rappelle ci-dessous les résultats de trois travaux à ce sujet, sans entrer dans le détail des heuristiques leur permettant de découvrir ces prédicats quantifiés. On notera

simplement qu'en pratique les parties droites de ces prédicats sont trouvées aisément à partir des simulations des contre-exemples (on le voit pour l'exemple ci-dessus, où $a[\ldots] = 0$ apparaît). C'est un peu plus difficile pour trouver les parties gauches adéquates.

Dans [JM07], il est proposé une technique de généralisation pour découvrir des prédicats « de tranche », qui sont des prédicats quantifiés implicitement sur $\ell$ de la forme suivante :

$$(\varphi_1(i_1) \leq \ell \leq \varphi_2(i_2)) \Rightarrow \psi(a_1[], \ldots, a_p[], x_1, \ldots, x_q, \ell, i_1)),$$

où la notation $a[]$ signifie qu'il n'y a aucune restriction sur l'expressivité de l'index. Pour trouver les parties gauches, les auteurs utilisent d'une part de simples généralisations, par exemple les prédicats suggérés par le contre-exemple ci-dessus $a[1] = 0$ et $i = 1$ sont généralisés en le prédicat de tranche $\ell \in [1, i] \Rightarrow a[\ell] = 0$. Et d'autre part, les auteurs utilisent de simples axiomes, par exemple « les prédicats de tranche $\ell \in [i_1, i_2] \Rightarrow P$ et $\ell \in [i_2 + 1, i_3] \Rightarrow P$ se réduisent au prédicat de tranche $\ell \in [i_1, i_3] \Rightarrow P$ ».

Cette technique est appliquée sur plusieurs exemples avec succès. Ces exemples sont le programme ci-dessus et le programme recherchant le premier élément non nul d'un tableau. L'analyse fonctionne notamment sur un exemple remplissant un tableau $b$ avec les indices auxquels le contenu d'un tableau $a$ est positif : la propriété prouvée $\forall \ell, 1 \leq \ell \leq n \Rightarrow a[b[\ell]] \geq 0$ sort de la classe que l'on avait définie (Éq. 4.1, p. 46). Enfin sur le tri par insertion, l'analyse prouve l'invariant de la boucle interne. Alors que les premiers exemples sont analysés en moyenne en quatre secondes, il faut plus d'une minute et demie pour un résultat partiel sur l'exemple complexe du tri par insertion.

Dans [BHMR07], une approche différente est proposée. À partir d'un contre-exemple, plutôt que de chercher des prédicats quantifiés qui élimineront une infinité de contre-exemples, les auteurs construisent un programme représentant un ensemble infini de contre-exemples similaires. Ils cherchent alors un invariant de ce programme particulier, qu'ils prennent pour prédicat. *A priori* l'avantage de cette technique est que la recherche de l'invariant se fait sur un programme plus simple que le programme original. Dans les faits, c'est discutable. Si on reprend notre exemple de programme initialisant le tableau $a$ à zéro, le programme généré à partir du contre-exemple que l'on avait évoqué a exactement la même première boucle que le programme original. On en revient à la nécessité d'une analyse capable de trouver l'invariant de cette boucle ! En l'occurrence, la technique utilisée pour trouver cet invariant sur le contenu des tableaux est basée sur la logique décidable de [BMS06] présentée à la Section 4.2.1 et l'utilisation de patrons.

Ainsi, les résultats expérimentaux sont faibles : seul le programme qui nous sert d'exemple dans cette section, et un programme recopiant respectivement les valeurs positives et strictement négatives d'un tableau $a$ dans les tableaux $b$ et $c$ (Fig. 7.3(b), p. 182, dénommé « filtre partitionnant » par la suite), ont leurs invariants prouvés.

Enfin, très récemment, [SPW09] propose une autre méthode pour trouver des prédicats quantifiés. En l'occurrence, il s'agit de prédicats indexés qui, *a contrario* de [JM07], peuvent porter sur plusieurs variables quantifiées. Pour trouver les prédicats adéquats sur ces variables particulières (qui formeront les parties gauches des propriétés quantifiées), ils recherchent dans les contre-exemples obtenus les incohérences au niveau des variables d'indices. Toujours sur le programme ci-dessus, pour le même contre-exemple, ils trouvent

qu'il y a une affectation à $a[i] = 0$ suivie d'une violation de l'assertion, *i.e.* $a[\ell] \neq 0$ où $\ell$ est une des variables fraîches. De ces deux faits, ils concluent que $i \neq \ell$. Chaque non-égalité découverte donne naissance à deux nouveaux prédicats, $i < \ell$ et $i > \ell$. Le premier est celui nécessaire pour que l'analyse découvre l'invariant de boucle.

Au niveau des expérimentations, la technique proposée s'applique avec succès sur les exemples de [JM07], avec des temps d'analyse parfois meilleurs, parfois moins bons. C'est la première technique capable de traiter un algorithme de tri complet : même s'il faut quasiment sept minutes d'analyse, le tri par sélection est prouvé. Par ailleurs, les auteurs apportent de nouveaux exemples, issus du noyau Linux. Parmi ceux-ci, on trouve des programmes traversant un tableau en appliquant un traitement différent selon la valeur de l'indice modulo 2 ou modulo 4. Des congruences apparaissent dans les parties gauches des invariants quantifiés découverts. Enfin, un des exemples vérifiés a pour invariant de boucle $\forall \ell, \varphi(a[\ell]) \Rightarrow b[\ell] = c[\ell]$, qui là encore sort de la classe définie à l'Équation 4.1 (p. 46).

**Conclusions** Comparées à l'heuristique *ad hoc* proposée par [FQ02] ces méthodes sont beaucoup plus générales. Cependant, alors qu'elles utilisent un objectif de preuve, il n'y a quasiment pas de résultats sur des programmes complexes manipulant des tableaux. Enfin, l'alternance de raffinements et d'analyses rend ces méthodes très coûteuses.

### 4.2.3 Abstraction par prédicats paramétriques

Comme on l'a vu l'abstraction par prédicats (Sec. 4.2.2) nécessite de définir un domaine abstrait particulier pour chaque programme analysé. À partir de cette critique, [Cou03] propose d'utiliser des prédicats paramétriques $P(p_1, p_2, \ldots)$ indépendants du programme, et montre comment une analyse peut instancier les variables $p_i$ de ces prédicats avec les constantes et variables du programme analysé pour découvrir ses invariants de forme $P$. Cette analyse ne fonctionne pas comme une abstraction par prédicats mais comme une interprétation abstraite classique où la sur-approximation de $\alpha \circ f \circ \gamma$ est donnée par un algorithme dédié. Il n'y a pas d'appel à un démonstrateur de théorèmes et une valeur abstraite n'est pas n'importe quelle combinaison booléenne de prédicats paramétriques instanciés. Là où cette analyse rejoint l'abstraction par prédicats, c'est sur l'objectif de se concentrer sur des propriétés particulières mais complexes.

Ainsi, on est ramené à la conception d'un domaine abstrait classique, visant l'ensemble de propriétés infini représenté par un ou quelques prédicats paramétriques en pratique. Si les valeurs abstraites de ce domaine contiennent bien entendu des instances de ces prédicats, leur forme exacte dépend en grande partie des prédicats utilisés. Enfin, ces domaines ont pour point commun d'utiliser les valeurs abstraites $x^\sharp$ d'un autre domaine $\mathcal{D}^{\mathbb{P}}$ pour représenter l'instanciation des variables formelles $p_i$ des prédicats paramétriques par les constantes, les variables $i, x, \ldots$ et les tableaux $a, \ldots$ du programme analysé.

**Exemple d'analyse** Pour l'analyse de programmes triant le contenu de tableaux, [Cou03] propose entre autres un domaine abstrait $\mathcal{D}^{leq}$ basé sur le prédicat $LEQ(p, p_1, p_2, p_3, p_4)$, ayant pour sémantique :

$$\forall \ell_1, \ell_2, (p_1 \leq \ell_1 \leq p_2 \wedge p_3 \leq \ell_2 \leq p_4) \Rightarrow p[\ell_1] \leq p[\ell_2].$$

Le domaine $\mathcal{D}^{\mathbb{P}}$ utilisé est celui des octogones (Fig. 3.2). Une valeur abstraite se compose alors de prédicats instanciés, de la forme $\langle LEQ(\ldots), x^\sharp \rangle$ et/ou d'un élément $x^\sharp$ de $\mathcal{D}^{oct}$.

Afin de comprendre comment une telle analyse fonctionne, on rapporte ses itérations, données dans l'article, sur la boucle interne du tri à bulle (ci-dessous, où **:=:** est l'opération d'échange). Les définitions des opérations sémantiques sont assez directes et il n'est pas nécessaire de les rappeler. À des fins de simplification, on note un prédicat instancié comme $\langle LEQ(a, \ldots), p_1 = p_3 = p_4 = i \wedge p_2 = i+1 \rangle$, par $a[i, i{+}1] \leq a[i]$. Les itérés aux points de contrôle repérés par des cercles numérotés ⓚ sont donnés à la Figure 4.1. Deux opérations repérées par des étoiles $\star$ retiennent l'attention.

```
1  i := 1 ;
2  ③ while i < n do
3  |   if a[i] > a[i + 1] then
4  |   |   ⑤ a[i] :=: a[i + 1] ⑥ ;
5  |   ⑧ i := i + 1 ⑨
```

- À la première étoile, on trouve le résultat de l'union $(i = 1 \leq n) \sqcup^{leq} (i = 2 \leq n \wedge a[i-1, i] \leq a[i])$. La valeur abstraite de gauche n'ayant aucune information sur le tableau $a$, elle doit être augmentée d'un terme « dégénéré », permettant de se ramener à l'opération d'union générale de deux prédicats instanciés [7]. Le prédicat instancié choisi ici est la tautologie $a[i] \leq a[i]$. Ce choix est basé sur le fait que le prédicat instancié de la valeur abstraite de droite dépend de $i$ et que $i$ est modifiée dans la boucle. Cependant, on notera qu'avec un choix légèrement différent, par exemple une tautologie similaire, $a[i-1] \leq a[i-1]$, l'opération d'union perdrait l'invariant repéré par l'étoile, qui est celui que l'on souhaite découvrir.
- La deuxième étoile réfère à l'opération d'élargissement. Celle-ci a été retardée à la seconde itération parce qu'un prédicat instancié est apparu vis-à-vis du résultat obtenu à la première itération. Lors de la troisième itération, l'opération d'élargissement est : $(1 \leq i \leq 2 \leq n \wedge a[1, i] \leq a[i]) \nabla^{leq} (1 \leq i \leq 3 \leq n \wedge a[1, i] \leq a[i])$. On remarquera simplement qu'elle fait parfaitement le travail souhaité, *i.e.* de générer l'hypothèse d'induction. Ce résultat est d'ailleurs le fait de l'élargissement $\nabla^{\mathbb{K}}$ seul, sur les variables entières du programme.

[Cou03] propose un autre domaine abstrait $\mathcal{D}^{sort}$ basé sur le prédicat paramétrique $SORT(p, p1, p2)$ dont la sémantique est $\forall \ell_1, \ell_2 (p_1 \leq \ell_1 \leq \ell_2 \leq p_2) \Rightarrow p[\ell_1] \leq p[\ell_2]$. Il définit alors un produit réduit (Sec. 2.3) entre $\mathcal{D}^{leq}$ et $\mathcal{D}^{sort}$ qui permet de découvrir les invariants du tri à bulle entier.

**Conclusions** Le principe proposé dans ces travaux mène à la construction d'analyses automatiques *a contrario* de toutes celles vues dans cette section. Les exemples d'analyses donnés, le tri à bulle et le tri rapide, démontrent que l'on peut découvrir des propriétés complexes sur le contenu des tableaux sans construire un domaine abstrait dédié au programme analysé. Cependant, le caractère très spécifique de ces analyses nous a amené à les discuter dans cette section. En pratique, pour analyser un ensemble de programmes effectuant des traitements différents sur le contenu des tableaux, de nombreux domaines abstraits vont être nécessaires, exigeant une implication de l'utilisateur plus forte que communément admise pour choisir lesquels utiliser.

7. Cette opération d'union nécessite une opération de sous-approximation de l'union dans $\mathcal{D}^{\mathbb{P}}$ (*cf.* Sec. 4.3.3, p. 66). Ce problème ne paraît pas être abordé dans ces travaux, où $\sqcup^{\mathbb{P}}$ est utilisée directement.

Quelques autres programmes très simples, d'initialisation de tableaux et notamment de matrices, ont été analysés dans [All08] sur ce même principe. Par exemple, le programme initialisant un tableau à zéro est analysé à l'aide d'un domaine $\mathcal{D}^{zero}$ basé sur le prédicat paramétrique $ZERO(p, p1, p2)$ signifiant $\forall \ell, (p_1 \leq \ell \leq p_2) \Rightarrow p[\ell] = 0$. Enfin, l'auteur propose une solution plus générale au problème soulevé précédemment du choix des cas dégénérés.

## 4.3 Des analyses automatiques

On trouve dans cette section uniquement des analyses basées sur l'interprétation abstraite. Ces analyses, au nombre de trois, sont présentées dans l'ordre croissant d'expressivité des propriétés qu'elles peuvent découvrir. Même si en réalité les capacités automatiques de ces analyses sont faibles, ces analyses apportent des notions intéressantes à ce sujet.

### 4.3.1 Résumés

**Expressivité** Il s'agit ici d'une idée simple, consistant à utiliser une variable scalaire pour abstraire une sous-partie, bornée, de cellules d'un tableau. On prend pour exemple, la variable $y_1$ représentant les deux premières cellules d'un tableau $a$, *i.e.* $a[1]$ et $a[2]$, et la variable $y_2$ représentant les cellules restantes de $a$, *i.e.* les cellules $a[3], \ldots, a[10]$. On appelle les variables $y_i$ des variables résumantes. Si une telle variable vérifie une propriété $\psi$, cela signifie que toute cellule de la partie du tableau qu'elle résume vérifie la propriété $\psi$. Pour découvrir ces propriétés, on utilise simplement une analyse classique. En l'occurrence, les travaux mettant en œuvre ces techniques [BCC+03, GDD+04] s'intéressent aux tableaux numériques. Ainsi, c'est un des domaines abstraits numériques présentés à la Section 3.1, ou un produit de ces domaines (Sec. 2.3), qui est utilisé.

– Si ce domaine abstrait n'est pas relationnel, la propriété attachée à une variable résumante est simplement une approximation de l'ensemble des valeurs que peuvent prendre toutes les cellules attachées à cette variable. Par exemple, si on a la

| | *1* | *2* |
|---|---|---|
| $x^{leq}_{k_3} =$ | $i = 1 \leq n$ | $\star\ 1 \leq i \leq 2 \leq n \wedge a[1,i] \leq a[i]$ |
| $x^{leq}_{k_5} =$ | $i = 1 < n \wedge a[i,i+1] \leq a[i]$ | $1 \leq i \leq 2 < n \wedge a[1,i+1] \leq a[i]$ |
| $x^{leq}_{k_6} =$ | $i = 1 < n \wedge a[i,i+1] \leq a[i+1]$ | $1 \leq i \leq 2 < n \wedge a[1,i+1] \leq a[i+1]$ |
| $x^{leq}_{k_8} =$ | $i = 1 < n \wedge a[i,i+1] \leq a[i+1]$ | $1 \leq i \leq 2 < n \wedge a[1,i+1] \leq a[i+1]$ |
| $x^{leq}_{k_9} =$ | $i = 2 \leq n \wedge a[i-1,i] \leq a[i]$ | $2 \leq i \leq 3 \leq n \wedge a[1,i] \leq a[i]$ |

| | *3* ↻ |
|---|---|
| $x^{leq}_{k_3} =$ | $\star\ 1 \leq i \leq n \wedge a[1,i] \leq a[i]$ |
| $x^{leq}_{k_5} =$ | $1 \leq i < n \wedge a[1,i+1] \leq a[i]$ |
| $x^{leq}_{k_6} =$ | $1 \leq i < n \wedge a[1,i+1] \leq a[i+1]$ |
| $x^{leq}_{k_8} =$ | $1 \leq i < n \wedge a[1,i+1] \leq a[i+1]$ |
| $x^{leq}_{k_9} =$ | $1 \leq i \leq n \wedge a[1,i] \leq a[i]$ |

Figure 4.1 – Analyse avec le domaine abstrait $\mathcal{D}^{leq}$, basé sur le prédicat paramétrique $LEQ(\ldots)$, de la boucle interne du tri à bulle

valeur abstraite ($y_1 \in [0,5] \wedge y_2 \in [0,0]$), on a la propriété $\forall \ell(\ell \in [1,2] \Rightarrow a[\ell] \in [0,5]) \wedge \forall \ell(\ell \in [3,10] \Rightarrow a[\ell] = 0)$.
- Si ce domaine abstrait est relationnel, il est de plus possible d'exprimer des relations entre toutes les valeurs de différentes sous-parties de tableaux, ou encore des relations avec des variables scalaires. Par exemple, on peut déduire de la valeur abstraite ci-dessus que $(y_2 \leq y_1)$, c'est-à-dire la propriété $\forall \ell_1, \ell_2(\ell_1 \in [1,2] \wedge \ell_2 \in [3,10] \Rightarrow a[\ell_2] \leq a[\ell_1])$. L'interprétation de la valeur abstraite $(y_1 = x)$ est la propriété $\forall \ell_1(\ell_1 \in [1,2] \Rightarrow a[\ell_1] = x)$.

Ainsi, si on veut comparer l'expressivité de ces techniques vis-à-vis de celle de la classe de propriétés donnée à l'Équation 4.1 (p. 46), les invariants pouvant être découverts sont de la forme :

$$\forall \ell_1, \ldots, \ell_n, (\varphi_1(\ell_1) \wedge \ldots \wedge \varphi_n(\ell_n) \Rightarrow \psi(a_1[\ell_1], \ldots, a_n[\ell_n], x_1, \ldots, x_q)),$$

où chaque tableau qui apparaît est indicé par une variable quantifiée différente. Plus précisément, dans [BCC+03], où l'analyseur Astrée est présenté, les tableaux sont soit abstraits par une seule variable (« *shrunk array* » en anglais), soit au contraire chaque cellule du tableau est abstraite par une variable (« *expanded array* » en anglais) [8].

Les travaux dans [GDD+04] s'intéressent eux à exprimer toutes les variétés de propriétés données ci-dessus [9].

**Opérations sémantiques** Bien entendu, les opérateurs sémantiques des domaines abstraits utilisés doivent être étendus pour prendre en compte les affectations et les conditionnelles où des cellules de tableaux apparaissent. L'idée consiste à réécrire ces affectations et ces conditionnelles, par une suite d'opérations usuelles des domaines abstraits, sur les variables scalaires et les variables résumantes.

Dès lors, apparaît le phénomène d'*affectation faible*. On distingue deux causes à ce phénomène :
- Les résumés. Lorsque l'on a une affectation à une cellule $a[c] := \langle expr \rangle$, où $c \in \mathbb{Z}$, et que cette cellule est représentée avec d'autres par une variable résumante $y$, cette affectation doit être interprétée comme un choix indéterministe entre garder la propriété sur $y$ inchangée, et effectuer réellement l'affectation $y := \langle expr \rangle$. Une affectation faible consistera donc à utiliser l'opérateur de borne supérieure pour approximer le résultat de ce choix indéterministe. Bien entendu si $y$ n'avait représenté qu'une seule cellule – on parle alors de $y$ comme d'une variable singleton, on aurait pu avoir une affectation « forte ».
- Les alias. Les affectations faibles apparaissent également lors de la retranscription d'affectations de la forme $a[i] := \langle expr \rangle$. Ici, selon la connaissance que l'on a sur $i$, différentes variables résumantes peuvent être potentiellement affectées et on est

8. Les auteurs indiquent que sur leur cas d'étude principal, le nombre de variables globales avant expansion des tableaux est de dix mille, et double après l'expansion des tableaux. Ceci est possible grâce à un travail en amont pour que l'analyse se comporte de manière linéaire dans la taille du code. D'une part, une implantation purement fonctionnelle de leurs valeurs abstraites est utilisée. Ceci permet notamment d'avoir des opérations de treillis dont la complexité dépend du nombre de différences entre les deux opérandes, et ce nombre est souvent petit. D'autre part, l'analyse fait usage des domaines abstraits relationnels sur des paquets (Sec. 3.1).

9. Ces travaux ont été généralisés à une abstraction de fonctions dans [JGR05]. Si on voit un tableau comme une fonction, cette analyse permet donc de mettre en relation les ensembles d'images d'une fonction pour différents ensembles d'antécédents.

devant un même choix indéterministe. Ainsi, à part si la connaissance sur $i$ permet de se ramener au cas précédent ($i = c$), toutes ces variables, qu'elles soient des singletons ou non, doivent être affectées faiblement.

Au-delà, on note que les travaux sur ces analyses du contenu des tableaux apportent peu d'information sur la gestion de l'expression $\langle expr \rangle$ affectée.

- Dans [BCC+03], le principe d'affectation faible, dans les deux scénarios ci-dessus, est présenté. Rien n'est dit quand à la retranscription de $\langle expr \rangle$. On n'en pense pas moins que, si chaque cellule de tableau apparaissant dans l'expression peut être représentée par une variable singleton, l'analyseur fait en sorte d'en déduire une propriété relationnelle. Par exemple, l'affectation $x_1 := a[2] + b[3]$, où les tableaux $a$ et $b$ sont expansés, est probablement traduite par l'affectation : $x_1 := y_2 + y_3$.
- Dans [GDD+04], il n'est question que d'affectations ou de conditionnelles où n'apparaissent pas de cellules de tableaux, mais des variables scalaires et des variables résumantes. Ceci est probablement dû au fait qu'ils veulent utiliser ces techniques aussi sur des graphes de formes. Cela dit, il n'y a pas d'information sur la traduction préalable des cellules de tableaux ou des pointeurs déréférencés, permettant d'arriver à ces opérations exclusivement sur des variables scalaires. Ils ne s'intéressent pas non plus au fait que certaines variables résumantes sont des singletons, et à optimiser en conséquence. Il est juste donné deux opérateurs, très simples, permettant par exemple, après l'affectation $x_1 := y_2 + y_3$, sur la valeur abstraite ($y_2 \in [1, 2] \wedge y_3 \in [5, 7]$), d'obtenir que $x_1 \in [6, 9]$, en évitant que la relation $x_1 = y_2 + y_3$, erronée, n'apparaisse.

Nous aurons donc soin d'expliciter dans nos travaux (Sec. 6.6.2.2) la transcription détaillée des expressions composant les affectations et les conditionnelles, lorsqu'elles contiennent des cellules de tableaux.

Enfin, pour les variables résumantes qui ne sont pas des singletons, on notera que les affectations faibles ne font que perdre de l'information, et que les conditionnelles (portant sur une des cellule résumée) ne peuvent pas apporter d'information. Ainsi, l'analyse ne pouvant qu'affaiblir les propriétés initiales sur ces variables, il se pose la question de comment trouver ces propriétés initiales. Ces travaux ne donnent pas d'explications à ce sujet.

**Conclusion** En conclusion, on notera que les analyses de contenu des tableaux avec des résumés peuvent être mises en œuvre assez simplement. De plus, dans le cas d'Astrée, elles ont montré qu'elles pouvaient être utiles pour prouver l'absence d'erreurs à l'exécution dans des programmes conséquents [10]. Par ailleurs, il ne faut pas se tromper sur leur expressivité, qui est loin d'être faible. Si on prend le cas des tableaux expansés, il est possible d'exprimer des propriétés très complexes : $a[5] = b[12] + 3a[4]$, etc. Cependant, il manque à ces analyses la capacité de résumer un ensemble *symbolique* de cellules. Ceci est nécessaire dès que l'on souhaite analyser un simple parcours d'un tableau par une boucle. Si on note $i$ la variable d'indice utilisée pour un tel parcours, il est fort probable que l'invariant de boucle porte, entre autres, sur la partie des cellules 1 à $i - 1$ du tableau.

10. On notera que ce n'est pas la seule technique utilisée pour analyser le contenu des tableaux. Les tableaux constants sont propagés avant l'analyse et, notamment, certaines parties du code sont analysées avec un partitionnement de traces selon les valeurs de variables indiçant des tableaux [MR05]. Ce partitionnement est proposé par l'utilisateur *via* des directives [SD07].

### 4.3.2 Résumés et partitionnement symbolique

**Expressivité** Une partie des auteurs de [GDD+04] propose justement, dans [GRS05], d'associer des variables résumantes à des ensembles symboliques de cellules. Les invariants qui peuvent être exprimés, vis-à-vis des techniques précédentes (Sec. 4.3.1), ne diffèrent donc que par les parties gauche des implications :

$$\forall \ell_1, \ldots, \ell_n, \\ \big(\varphi_1(\ell_1, i_1, \ldots, i_m) \wedge \ldots \wedge \varphi_n(\ell_n, i_1, \ldots, i_m) \Rightarrow \psi(a_1[\ell_1], \ldots, a_n[\ell_n], x_1, \ldots, x_q)\big), \tag{4.2}$$

où l'on a toujours une variables quantifiée différente pour chaque tableau qui apparaît. De plus $\varphi_k$ est une conjonction de contraintes de la forme $\ell_k \bowtie i_1, \ldots, \ell_k \bowtie i_m$, où $\bowtie \in \{<, =, >\}$.

Des exemples de propriétés pouvant être exprimés sont $\forall \ell, (1 \leq \ell < i \Rightarrow a[\ell] \leq x)$ et $\forall \ell_1, \ell_2, (1 \leq \ell_1 \leq i \wedge i < \ell_2 \leq n \Rightarrow a[\ell_1] = 0 \wedge a[\ell_2] = 1)$. Une propriété ne pouvant être exprimée est la propriété $\forall \ell_1, \ell_2, (1 \leq \ell_1 < \ell_2 \leq n \Rightarrow a[\ell_1] \leq a[\ell_2])$.

**Partitionnement** Cette forme particulière des $\varphi_k$ est la conséquence de l'idée principale de ces travaux : éviter les affectations faibles et éviter de perdre de l'information sur les conditionnelles portant sur des cellules de tableaux. Ainsi, si une affectation ou conditionnelle porte sur $a[i]$ dans le programme, l'ensemble des cellules du tableau $a$ va être partitionné simplement en trois tranches. La tranche $(\ell_1 = i)$, dont le contenu sera représenté par la variable $y_{a,=i}$, non résumante bien entendu, et les tranches $(\ell_2 < i)$ et $(\ell_3 > i)$, dont le contenu sera représenté cette fois par des variables résumantes, les variables $y_{a,<i}$ et $y_{a,>i}$. L'ensemble de ces tranches est appelé une partition. Comme on le voit, une partition se veut couvrante : pour une valeur de $i$ donnée, toute cellule du tableau $a$ appartient à l'une des tranches de la partition.

Lorsque le programme sous analyse affecte ou teste le tableau $a$ selon d'autres variables d'indices, par exemple $j$, les auteurs choisissent une partition permettant de continuer à faire uniquement des affectations fortes. S'ils considéraient simplement la partition formée des trois tranches évoquées ci-dessus selon $i$ et les trois tranches similaires selon $j$, le phénomène d'alias entrerait en jeu. Une affectation à $a[i]$ mènerait bien à une affectation forte à $y_{a,=i}$, mais aussi à des affectations faibles à $y_{a,=j}$, $y_{a,<j}$ et $y_{a,>j}$. À moins de savoir que $i = j$, auquel cas on aurait juste une affectation forte de $y_{a,=j}$, ou alors que $i \neq j$ ce qui éviterait toute affectation de cette variable, etc. La solution consiste à prendre pour tranches toutes les conjonctions possibles des trois contraintes basiques sur $i$ et $j$. La partition est donc faite de neuf tranches : $(\ell_1 < i \wedge \ell_1 < j), (\ell_2 < i \wedge \ell_2 = j)$, …, $(\ell_5 = i \wedge \ell_5 = j)$, …, $(\ell_9 > i \wedge \ell_9 > j)$. On notera les variables scalaires correspondantes par $y_{a,<i,<j}$, etc. Ainsi, la partition choisie est bien une partition, c'est-à-dire que pour toute valeur de $i$ et $j$ donnée, toute cellule du tableau $a$ appartient à une seule tranche.

Cette construction apparaît d'emblée très coûteuse, puisqu'elle est exponentielle dans le nombre d'affectations et de tests du programme portant sur les cellules d'un tableau. Si on note $I$ l'ensemble des variables utilisées pour indicer un tableau, on se retrouve avec une partition de $3^{|I|}$ tranches pour analyser le contenu de ce tableau. Cependant, dans la pratique, de nombreuses tranches peuvent être éliminées d'office, de par les relations entre les différentes variables de $I$. Par exemple, pour un programme de tri par insertion, où la clé à insérer serait à la cellule $i$ et où la variable $j$ serait utilisée pour insérer en

bonne place cette clé, on a l'invariant $j < i$. Celui-ci permet de considérer une partition à cinq tranches au lieu des neuf évoquées ci-dessus [11].

On notera que la notion de partition est intéressante : elle fixe la partie gauche des invariants sur le contenu des tableaux (Éq. 4.1, p. 46) pouvant être découverts. Or, deviner les tranches d'un tableau sur lesquelles il existe des invariants intéressants est plus aisé, à la seule lecture du code source, que de deviner les propriétés vérifiées par le contenu de ces tranches (autrement dit la partie droite des invariants). Cette remarque nous a conduit à reprendre le principe des partitions dans notre analyse du contenu des tableaux (Sec. 6.5). À l'inverse, l'analyse que l'on présente à la Section 4.3.3 a pris le parti contraire, c'est-à-dire de fixer les parties droites : nous verrons que dans la pratique elle faillit sur des programmes avec des boucles imbriquées (Fig 4.4, p. 68).

**Valeurs abstraites** On s'attendrait ici à ce que les valeurs abstraites soient simplement les éléments d'un domaine abstrait numérique relationnel quelconque. Bien entendu, l'analyse devrait prendre soin de la sémantique des variables scalaires, résumantes ou non, représentant les tranches des tableaux. Cependant, si là était l'idée première de [GRS05], elle n'a pas été conservée. Ce n'est qu'un exemple parmi tant d'autres où l'outil utilisé pour vérifier expérimentalement la validité d'une solution initiale, a fortement influencé la définition finale de cette solution. En l'occurrence, l'outil est l'analyseur statique TVLA [LS00], En quelques mots, cet analyseur est approprié pour les analyses de formes, pour lesquelles il fournit le moyen d'abstraire automatiquement par un élément un ensemble non-vide d'éléments ayant les mêmes propriétés, notamment structurelles. Par exemple, la propriété « élément atteignable depuis le pointeur $p$, en suivant le champ *next* » permet d'abstraire par un élément une liste chaînée non-vide pointée par $p$. Ces propriétés sont donc des prédicats d'abstraction. En utilisant TVLA, les auteurs avaient donc un moyen rapide de représenter les tranches d'un tableau : considérer chaque cellule du tableau comme un élément unique, et prendre pour prédicats : « cellule dont l'indice est strictement plus petit que la valeur de $i$ », « ... égal à la valeur de $i$ », etc. De cela, TVLA renvoie pour abstraction un *ensemble* de cas, selon la taille du tableau et la valeur de $i$. En effet, un élément agglomérant représentant un ensemble *non-vide* d'éléments, TVLA doit considérer le cas où le tableau est de taille nulle, considérer la tranche de 1 à $i - 1$ seulement si $i \geq 2$, etc.

On prend pour exemple le programme simple, ci-contre, copiant le tableau $b$ dans le tableau $a$. Seul le tableau $a$ est partitionné puisque c'est le seul à être modifié [12]. À partir des prédicats introduits pour partitionner $a$ selon $i$, TVLA va donc considérer, pour l'analyse de l'intérieur de la boucle du programme, les quatre cas donnés à la Figure 4.2 (les tableaux $a$ et $b$ sont supposés être de la même taille, et celle-ci est supposée non nulle). On trouve le cas où les tableaux $a$ et $b$ ont une seule

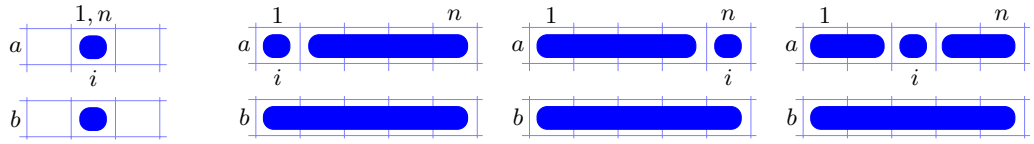
```
1  i := 1 ;
2  while i ≤ n do
3  |  a[i] := b[i] ;
4  |  i := i + 1
```

Recopie du tableau $b$ dans le tableau $a$

11. Si on regarde en détail l'algorithme de tri par insertion (Fig. 7.5(a), p. 189), celui-ci procède à une affectation à la cellule $j + 1$. Les travaux que l'on relate ici ne permettent pas de prendre en compte cette indexation par une expression. Ils réécrivent donc cet algorithme à l'aide d'une variable temporaire, égale à $j + 1$ : des 27 tranches possibles, c'est au final une partition de 7 tranches qui est utilisée.

12. C'est ce qui serait conforme à l'heuristique présentée par les auteurs. Cependant, là encore pour des raisons d'implantation, le succès de leur analyse nécessite que $b$ soit aussi partitionné. On s'affranchit de ce problème.

cellule (a), le cas où les tableaux $a$ et $b$ ont au moins deux cellules, $i$ étant la première (b) ou la dernière (c) cellule, et enfin le cas où les tableaux $a$ et $b$ ont au moins trois cellules, où $i$ n'est ni la première ni la dernière cellule (d).




(a) aucun résumé n'existe : $n = 1 = i$

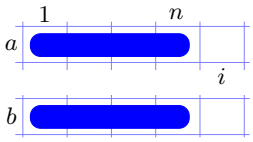
(b) seul le résumé droit de $a$ existe : $1 = i < n$

(c) seul le résumé gauche de $a$ existe : $1 < i = n$

(d) les deux résumés de $a$ existent : $1 < i < n$

Figure 4.2 – L'ensemble des cas composant les valeurs abstraites dans la boucle du programme de copie de tableau

À l'entrée de la boucle, TVLA considérera en plus le cas ci-contre, où la tranche de gauche de $a$ se confond avec le tableau en son entier. C'est lorsque $i = n + 1$. Il s'agit donc aussi du seul cas après la sortie de la boucle.




(e) seul le résumé gauche de $a$ existe : $i = n + 1$

Ainsi, une valeur abstraite est un ensemble de couples $(P_k, X_k)$, appelés « configurations mémoire ».

- $P_k$ est l'un des cas précédents, autrement dit un cas spécial des partitions des tableaux. Ce cas est décrit par l'ensemble des variables scalaires associées aux tranches mises en jeu. Par exemple, le cas (b) est représenté par : $P_k = \{y_{a,=i}\ ;\ y_{a,>i}\ ;\ y_b\}$.
- $X_k$ est un élément d'un domaine abstrait numérique relationnel (Sec. 3.1), que l'on note $\mathcal{D}^{\mathbb{Z}}$. Les variables mises en relation sont les variables scalaires du programme, et les variables de $P_k$ dédoublées : celles suffixées par « *.index* » modélisent l'ensemble des cellules représentées par la variable et celles suffixées par « *.value* » modélisent l'ensemble des valeurs des cellules représentées par la variable (ce qui est le sens originel de cette variable). Par exemple, toujours pour le cas (b), le fait que les valeurs du tableau $a$ sont comprises entre 0 et 5 et celles de $b$ entre -5 et 0, s'abstrait par :

$$\begin{aligned} X_k \quad = \quad & i = 1 \wedge n \geq 2 \\ & \wedge\ y_{a,=i}.index = 1 \wedge 2 \leq y_{a,>i}.index \leq n \wedge 1 \leq y_b.index \leq n \\ & \wedge\ 0 \leq y_{a,=i}.value \leq 5 \wedge 0 \leq y_{a,>i}.value \leq 5 \wedge -5 \leq y_b.value \leq 0 \end{aligned} \quad .$$

On notera la forte redondance de plusieurs informations entre les différentes configurations mémoire $(P_k, X_k)$ d'une valeur abstraite ; redondance de l'information sur le contenu des tableaux, redondance de la modélisation des cellules associées à chaque tranche. Les auteurs, d'ailleurs, en conviennent et estiment qu'une analyse dédiée pourrait s'affranchir de la distinction des différents cas. C'est ce que nous ferons avec l'analyse que nous proposons (Ch. 6).

On termine sur les opérations usuelles de borne supérieure, borne inférieure et élargissement. Elles se font simplement configuration mémoire par configuration mémoire, si tant

est que celles-ci soient définies sur les mêmes cas. Par exemple : $\{(P_1, X_1), \ldots, (P_4, X_4)\} \sqcup \{(P_1, X'_1), \ldots, (P_4, X'_4)\} = \{(P_1, X_1 \sqcup^{\mathbb{Z}} X'_1), \ldots, (P_4, X_4 \sqcup^{\mathbb{Z}} X'_4)\}$.

**Opérations sémantiques** Les opérations qui nous intéressent ici sont les affectations, de deux sortes : celles modifiant le contenu d'un tableau et celles modifiant la valeur d'une variable d'indice.

Pour les premières, tout le travail qui a été fait en amont les rend triviales. Il ne peut s'agir que d'affectations fortes. Si l'on reprend notre exemple, dans chacune des quatre configurations mémoires, l'affectation $a[i] := b[i]$, amène $(-5 \leq y_{a,=i}.\mathit{value} \leq 0)$ le reste des contraintes restant inchangé [13].

Avec les affectations de variables d'indice, on touche à l'approximation. Seul le cas de l'incrémentation est évoqué par les auteurs. Regardons cette opération, sur la variable $i$, avec le simple concept de partition en tête. Pour le tableau $a$, on a conceptuellement les tranches $(\ell_1 < i), (\ell_2 = i)$ et $(\ell_3 > i)$. Donc,

- les propriétés vérifiées, après l'incrémentation, par les cellules à une position inférieure stricte à $i$ sont simplement l'union des propriétés vérifiées, avant l'incrémentation, par les cellules de cette même tranche et la $i^{\text{e}}$ cellule de $a$ ;
- les propriétés vérifiées par la $i^{\text{e}}$ cellule de $a$ après l'incrémentation sont simplement celles vérifiées par les cellules à une position supérieure stricte à $i$ avant l'incrémentation ;
- enfin, les propriétés vérifiées par les cellules à une position supérieure stricte à $i$ restent identiques.

Si l'on revient au concept de configuration mémoire, pour le cas (d), cela reviendrait à effectuer l'affectation faible $y_{a,<i}.\mathit{value} \cup= y_{a,=i}.\mathit{value}$ et l'affectation forte $y_{a,=i}.\mathit{value} := y_{a,>i}.\mathit{value}$ [14].

**Fonctionnement de l'analyse** Supposons comme on l'a évoqué, qu'au point de contrôle initial du programme, le tableau $a$ ne contient que des valeurs dans $[0, 5]$ et le tableau $b$ que des valeurs dans $[-5, 0]$. Puis supposons que l'analyse ait trouvé l'invariant du programme au point de contrôle avant l'affectation à $a[i]$. On a alors, pour la configuration mémoire sur (d) (on omet l'information sur le tableau $b$) :

$$\begin{aligned} X_k \;=\; & 2 \leq i \leq n-1 \\ & \wedge\, 1 \leq y_{a,<i}.\mathit{index} \leq i-1 \wedge y_{a,=i}.\mathit{index} = i \wedge i+1 \leq y_{a,>i}.\mathit{index} \leq n \;. \\ & \wedge\; -5 \leq y_{a,<i}.\mathit{value} \leq 0 \wedge 0 \leq y_{a,=i}.\mathit{value} \leq 5 \wedge 0 \leq y_{a,>i}.\mathit{value} \leq 5 \end{aligned}$$

On voit qu'après l'opération d'affectation à $a[i]$, l'incrémentation de $i$ mène à la même configuration mémoire. En effet l'affectation faible ne perd pas d'information : $y_{a,<i}.\mathit{value} = [-5, 0] \sqcup [-5, 0]$. On a compris qu'il s'agit d'un pas d'induction, dont il faut analyser le cas de base. C'est pourquoi les variables d'indices, en l'occurrence $i$, apparaissent dans

13. Avec le tableau $b$ partitionné l'affectation amène $-5 \leq y_{a,=i}.\mathit{value} = y_{b,=i}.\mathit{value} \leq 0$.

14. Dans les faits c'est plus complexe : TVLA va considérer, pour chaque configuration mémoire, l'incrémentation de $i$ et recalculer les éléments à agglomérer vis-à-vis des prédicats de partitionnement. En faisant cette construction, le calcul des nouvelles valeurs de $y_{a,<i}.\mathit{value}$ et $y_{a,=i}.\mathit{value}$ se fait automatiquement. La valeur abstraite résultante de l'incrémentation est composée des cinq cas (a)–(e) évoqués. La configuration mémoire sur (e) est calculée à partir des configurations sur (a) et (c). La configuration sur (d) à partir d'elle-même et de celle sur (b). La configuration sur (c) à partir de la configuration sur (b). Les configurations sur (a) et (b) restent identiques.

les $X_k$. Effectivement, en faisant une analyse concomitante des variables d'indices et du contenu des tableaux, lors de la première itération du calcul du point fixe, l'analyse va considérer le cas $i = 1$. Alors seulement les deux premières configurations mémoires (a) et (b) existent. Ainsi, après l'incrémentation de $i$, seules les trois autres configurations (c), (d) et (e) existent, et on a pour celles-ci l'assurance que $-5 \leq y_{a,<i}.\mathit{value} \leq 0$, puisque la tranche correspondante représente une seule cellule. C'est l'avantage d'avoir différencié les différents cas (a)–(e) : on évite la question de la valeur initiale de $y_{a,<i}.\mathit{value}$ lorsque $i = 1$ [15].

Enfin, vient l'élargissement qui permet à l'analyse de se rendre directement au cas précédent, où $1 \leq i \leq n$, et où le contenu des cellules de 1 à $i - 1$ du tableau $a$ sont supposées dans $[-5, 0]$. Comme nous l'avons vu sur le cas (d), mais il en est de même pour les autres cas, l'analyse converge à l'itération suivante.

**Résultats** Ainsi, le résultat de l'analyse sur ce programme est que le contenu du tableau $a$ est compris entre $-5$ et 0. Ce résultat est représentatif des capacités *automatiques* de l'analyse proposée dans [GRS05] : découvrir des propriétés simples (la plupart du temps unaires) sur le contenu des tableaux, lorsqu'ils sont simplement traversés. Par exemple, l'analyse du programme d'initialisation, obtenu en remplaçant l'affectation $a[i] := b[i]$ dans le programme précédent par $a[i] := 2i + 3$, montrera que le contenu de $a$ après la boucle est compris entre 5 et $2n + 3$ (du moins en choisissant le domaine abstrait TVPLI ou des polyèdres pour représenter les $X_k$).

Au-delà de ces exemples analysés avec succès, aucun autre n'est donné. On peut cependant penser qu'un algorithme de recherche du maximum, comme celui donné à la Figure 7.4(a) (p. 186) pourrait aussi être analysé automatiquement. D'une part, l'invariant $\forall \ell_1, 1 \leq \ell_1 \leq n \Rightarrow a[\ell_1] \leq max$, cette fois relationnel, s'exprime dans le domaine (Éq. 4.2, p. 59). D'autre part, cela ne paraît pas plus difficile d'obtenir, à la fin du programme, la contrainte $y_{a,<i} \leq max$ que la contrainte $-5 \leq y_{a,<i} \leq 0$ pour le programme de copie de tableau.

**Propriétés plus générales** Ceci étant, on sent que ces résultats sont en deçà du potentiel qu'offre le cadre de cette analyse. Pour aller au-delà de la bride qu'est l'expressivité du domaine utilisé (Éq. 4.2, p. 59), les auteurs permettent que soient associés des prédicats auxiliaires aux configurations mémoires, pour pouvoir trouver des propriétés plus complexes. Par exemple, le prédicat $\delta(y) = \forall \ell \in y.\mathit{index}, a[\ell] = b[\ell]$ leur permet de montrer que le programme de copie de tableau, renvoie deux tableaux égaux cellule par cellule. De fait, on tombe dans l'analyse semi-automatique, l'utilisateur devant à la fois donner ce prédicat, et se charger de définir ses nouvelles valuations pour toute opération sémantique. Les prédicats auxiliaires s'évaluent dans une logique à trois valeurs. Ainsi, si l'on regarde de nouveau l'invariant du programme de copie de tableau découvert avant l'affectation à $a[i]$, on a en plus, pour la configuration mémoire sur (d) :

$$\delta(y_{a,<i}) = 1 \wedge \delta(y_{a,=i}) = \delta(y_{a,>i}) = 1/2.$$

La valeur 1/2 signifiant que l'invariant peut être vérifié ou non par les cellules des deux dernières tranches. Si l'utilisateur a indiqué, qu'après l'affectation $a[i] := b[i]$, si

15. De la même manière, si dans les analyses par prédicats paramétriques (Sec. 4.2.3) on maintenait une disjonction selon les valeurs des variables d'indices (comme à la Fig. 4.2, p. 61), on éviterait le problème des cas « dégénérés ».

$\exists y, y.\mathit{index} = i$ alors $\delta(y) = 1$, l'analyse découvrira l'invariant souhaité, par la même mécanique que précédemment.

Grâce à ces prédicats auxiliaires, l'analyse arrive à trouver l'invariant du tri par insertion (Fig. 7.5(a), p. 189), ce qui est un beau résultat.

**Conclusion** L'apport de [GRS05] est vraiment intéressant : il montre la voie pour une analyse automatique de programmes, non-triviaux, manipulant des tableaux [16]. Cette voie se compose du partitionnement symbolique, dont le lien avec l'analyse concomitante des variables d'indices a été clarifié.

Il reste que l'analyse définie est loin d'être automatique. Il faut être un expert pour pouvoir fournir les prédicats auxiliaires à TVLA. Enfin, les temps d'analyse sont rédhibitoires (Fig. 4.4, p. 68).

### 4.3.3 Domaines quantifiés

**Expressivité** L'analyse que propose [GMT08] ne pose que peu de limites à l'expressivité des invariants sur le contenu des tableaux pouvant être découverts. En effet, leur expressivité englobe même théoriquement celle des propriétés de la première classe (Éq. 4.1). Plus précisément, l'analyse est paramétrée par deux domaines abstraits, que l'on notera $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$. Elle permet de découvrir des conjonctions de propriétés de la forme suivante, où la propriété $\varphi$ doit être un élément du premier domaine, la propriété $\psi$ un élément du second domaine :

$$\forall \ell_1, \ldots, \ell_n, \\ \Big(\varphi(\ell_1, \ldots, \ell_n, i_1, \ldots, i_m) \Rightarrow \psi(a_1[], \ldots, a_p[], x_1, \ldots, x_q, \ell_1, \ldots, \ell_n)\Big).$$

En pratique, le domaine $\mathcal{D}^{\mathbb{K}}$ utilise des fonctions non-interprétées pour représenter les tableaux (d'où la notation $a_1[]$, etc., utilisée ci-dessus). C'est ce qui permet, comme dans [JM07] (Sec. 4.2.2.3) d'exprimer que $\forall \ell, 1 \leq \ell \leq n \Rightarrow a[b[\ell]] = 0$. Un tel domaine $\mathcal{D}^{\mathbb{K}}$ est par exemple le produit défini dans [GT06] de fonctions non-interprétées et de conjonctions d'inégalités linéaires.

Ceci étant, la puissance de l'analyse est restreinte par plusieurs contraintes :

- le nombre de propriétés de tableaux ayant même partie droite dans une valeur abstraite est limité par une constante. Les auteurs n'indiquent pas, même pour leurs expérimentations, la constante utilisée.
- les propriétés $\psi$ doivent être des propriétés atomiques de $\mathcal{D}^{\mathbb{K}}$. Pour le domaine des zones cela signifie que $\psi$ est une contrainte de zone. En pratique cette restriction contraint l'expressivité. La propriété $\varphi \Rightarrow (\psi_1 \wedge \psi_2)$ peut évidemment s'exprimer $(\varphi \Rightarrow \psi_1) \wedge (\varphi \Rightarrow \psi_2)$, mais nous venons de voir que le nombre de propriétés de tableaux avec la même propriété $\varphi$ est limité. Surtout, cela empêche l'analyse de tirer parti du pouvoir de déduction de $\mathcal{D}^{\mathbb{K}}$.
- la propriété $\psi$ doit respecter un des patrons donné par l'utilisateur. Par exemple les propriétés $a_1[\ell_1] = 0$ et $a_2[\ell_1] = 0$ respectent le patron $\mathcal{A}[\mathcal{L}] = 0$.

16. Certains principes de cette analyse sont en fait repris de travaux plus anciens sur les listes chaînées [LRSW00] (*cf.* Section 4.5).

L'usage de domaines abstraits comme paramètres est séduisante. On notera que l'analyse impose des contraintes particulières sur $\mathcal{D}^{\mathbb{Z}}$. Le domaine choisi doit pouvoir exprimer la négation de toute propriété atomique composant ses éléments, et avoir une forme normale. Dans la pratique, le domaine $\mathcal{D}^{\mathbb{Z}}$ doit au moins pouvoir exprimer des contraintes de zones et des contraintes de non-égalité.

**Fonctionnement de l'analyse** Le cœur du fonctionnement de l'analyse est finalement le même que celui de l'analyse proposée par [GRS05]. Si la construction des invariants est ici entièrement dynamique, puisqu'il est question de découvrir la partition idéale plutôt que de la figer avant l'analyse, la découverte de l'invariant se fait sur le même principe d'analyse concomitante du contenu des tableaux et des variables d'indice.

Ainsi, les valeurs abstraites sont la conjonction :
- d'une propriété $\varphi \in \mathcal{D}^{\mathbb{Z}}$ portant sur les variables d'indices ;
- d'un ensemble de propriétés de tableaux, composées comme on l'a vu, d'un ensemble $L_i$ de variables quantifiées, d'une propriété $\varphi_i$ et d'une propriété $\psi_i$. On appellera les propriétés $\varphi_i$ des tranches.

Pour plus de clarté, on notera une valeur abstraite comme suit, plutôt que le simple regroupement de $\varphi$ et des $L_i, \varphi_i$ et $\psi_i$ :

$$\varphi \wedge \bigwedge_i \forall \ell_j \in L_i, \varphi_i \Rightarrow \psi_i.$$

Maintenant que nous avons retranscrit la définition des valeurs abstraites, on décrit le fonctionnement de l'analyse à travers l'exemple de la copie de tableau (reporté ci-contre). On suppose que l'utilisateur a donné le patron $\mathcal{A}[\mathcal{L}] = \mathcal{A}'[\mathcal{L}]$.

```
1 i := 1 ;
2 while i ≤ n do
3 |   a[i] := b[i] ;
4 |   i := i + 1
```

Recopie du tableau $b$ dans le tableau $a$

À la première itération, on a en tête de boucle $(i = 1)$. Après le corps de la boucle on a tout aussi évidemment $(i = 2) \wedge (a[1] = b[1])$. Un premier traitement intervient, consistant à quantifier cette dernière propriété, pour en faire une valeur abstraite. C'est ici que les patrons sont utiles pour restreindre l'introduction de variables quantifiées, sources de complexité. Ici, le patron n'autorise qu'une variable quantifiée pour représenter la propriété (*a contrario* du patron $\mathcal{A}[\mathcal{L}] = \mathcal{A}'[\mathcal{L}']$). On se retrouve donc à faire en tête de boucle l'union de $(i = 1)$ et de $(i = 2) \wedge \forall \ell_1,\ \ell_1 = 1 \Rightarrow a[\ell_1] = b[\ell_1]$. Pour ne pas perdre toute information sur les tableaux, ou autrement dit résoudre le problème de l'initialisation de l'analyse du contenu des tableaux, la valeur abstraite $(i = 1)$ est remplacée par la propriété équivalente $(i = 1) \wedge \forall \ell_1,\ \bot^{\mathbb{Z}} \Rightarrow a[\ell_1] = b[\ell_1]$. L'opération d'union est alors définie de la manière suivante : on a d'un côté l'union des propriétés d'indices, ce qui donne $(1 \leq i \leq 2)$. De l'autre côté, les deux propriétés de tableaux vont être unifiées car elles ont même partie droite. Pour être correcte, cette opération nécessite de calculer une sous-approximation de la disjonction des deux parties gauches. En effet, utiliser l'opérateur d'union de $\mathcal{D}^{\mathbb{Z}}$ si celui-ci n'est pas exact, introduirait de nouvelles valeurs pour lesquelles la propriété de droite serait vérifiée. Pour être correcte, l'opération de sous-approximation se fait dans le contexte de la propriété sur les indices. Ainsi, c'est la disjonction de $(\bot^{\mathbb{Z}} \wedge i = 1)$ et $(1 = \ell_1 \wedge i = 2)$ qui est sous-approximée. On obtient donc la valeur abstraite $(1 \leq i \leq 2) \wedge \forall \ell_1,\ 1 = \ell_1 = i - 1 \Rightarrow a[\ell_1] = b[\ell_1]$.

Si on retarde l'élargissement d'une itération, l'analyse continue et se retrouve après l'affectation à $a[i]$, à avoir ajouté la propriété de tableau $\forall \ell_1, \ell_1 = i \Rightarrow a[\ell_1] = b[\ell_1]$ à la

$$\frac{\begin{array}{l}(1 \leq i \leq 2)\\ \wedge\forall \ell_1,\ 1 = \ell_1 = i-1 \Rightarrow a[\ell_1] = b[\ell_1]\\ \wedge\forall \ell_1,\ \ell_1 = i \Rightarrow a[\ell_1] = b[\ell_1]\end{array}}{\begin{array}{l}(1 \leq i \leq 2)\\ \wedge\forall \ell_1,\ 1 \leq \ell_1 \leq i \Rightarrow a[\ell_1] = b[\ell_1]\end{array}}$$

(a) unification intra valeur abstraite

$$\frac{\begin{array}{l}(1 \leq i \leq 2)\\ \wedge\forall \ell_1,\ 1 = \ell_1 = i-1 \Rightarrow a[\ell_1] = b[\ell_1]\\ (2 \leq i \leq 3)\\ \wedge\forall \ell_1,\ 1 \leq \ell_1 \leq i-1 \Rightarrow a[\ell_1] = b[\ell_1]\end{array}}{\begin{array}{l}(1 \leq i \leq 3)\\ \wedge\forall \ell_1,\ 1 \leq \ell_1 \leq i-1 \Rightarrow a[\ell_1] = b[\ell_1]\end{array}}$$

(b) unification lors d'une union de deux valeurs abstraites

Figure 4.3 – Différentes opérations d'unification de propriétés de tableaux

valeur abstraite précédente. On a donc la valeur abstraite donnée dans la partie supérieure de la Figure 4.3(a). Avant de continuer, l'analyse cherche à unifier les deux propriétés de tableaux au sein de cette nouvelle valeur abstraite, car elles ont même partie droite, travaillant ainsi à constituer une tranche maximale. Une sous-approximation correcte de la disjonction de $(1 = \ell_1 = i - 1)$ et $(\ell_1 = i)$, dans le contexte de la propriété d'indice, est $(1 \leq \ell_1 \leq i)$ [17]. Ainsi, avant l'incrémentation de $i$ on a la propriété donnée dans la partie inférieure de la Figure 4.3(a). L'incrémentation est appliquée simplement à la propriété d'indice et aux tranches des propriétés de tableaux. On se retrouve donc à faire en tête de boucle l'union des valeurs abstraites données dans la partie supérieure de la Figure 4.3(b). Le résultat de cette opération, décrite précédemment, est donné dans la partie inférieure de cette même figure. Ainsi, si on avait procédé à un élargissement suivi d'un rétrécissement, plutôt qu'une union, l'analyse aurait trouvé l'invariant souhaité en tête de boucle :

$$(1 \leq i \leq n) \wedge \forall \ell_1,\ 1 \leq \ell_1 < i \Rightarrow a[\ell_1] = b[\ell_1].$$

On retiendra deux choses de cet exemple d'analyse. Première chose, avec ces valeurs abstraites, une astuce suffit pour régler le problème de l'absence d'information sur le contenu des tableaux pour la trace d'exécution où $i = 1$. C'est un avantage notable sur l'analyse de [GRS05]. Deuxième chose, la construction sémantique des tranches, dont la faisabilité paraît démontrée, nécessite une opération de sous-approximation de l'union pour $\mathcal{D}^{\mathbb{Z}}$. Or, nous allons voir que cette opération est d'une part fort coûteuse, et d'autre part, trop imprécise pour être utilisée dans toutes les situations, comme le fait l'analyse.

**Opération de sous-approximation** C'est l'occasion de prendre encore un exemple concret, que l'on retrouvera dans la suite de la thèse. Il s'agit de deux valeurs abstraites rencontrées lors de l'analyse du tri par insertion (Fig. 7.5(a), p. 189), en tête de la boucle interne :

$$(j = i - 1) \wedge \forall \ell_1,\ 1 \leq \ell_1 < i - 1 \Rightarrow a[\ell_1] \leq a[\ell_1 + 1]$$
$$(j < i - 1) \wedge \forall \ell_1,\ 1 \leq \ell_1 < i \Rightarrow a[\ell_1] \leq a[\ell_1 + 1].$$

La variable $j$ est utilisée pour insérer en bonne place la clé courante, qui est à la position $i$. Ainsi, deux cas se présentent : soit on s'apprête à traiter une nouvelle clé, alors $j = i - 1$

17. Ceci rappelle la fusion de prédicats de tranche utilisé dans [JM07] (Sec. 4.2.2.3). Dans ces travaux le principe était inverse : si la disjonction de deux tranches n'était pas représentable exactement, les deux propriétés de tranches correspondantes étaient conservées. Ici on cherche toujours à les fusionner.

et le tableau est trié jusqu'à cette case ; c'est l'invariant de la boucle externe. Soit on est en train d'insérer une clé, $j$ a traversé le tableau vers la gauche, les cellules à sa droite ont été recopiées ($a[j+1] := a[j]$). Ainsi, le tableau contient un doublon et il est trié jusqu'à la cellule $i$ incluse (quand $j$ valait $i-1$ on a écrasé la cellule $a[i]$).

L'opération d'union de l'analyse que l'on étudie cherche à renvoyer systématiquement pour résultat une seule propriété de tableaux lorsque les parties droites des propriétés de tableaux des deux valeurs abstraites sont identiques. Ainsi l'opération d'union calcule donc, pour les deux valeurs abstraites ci-dessus, une sous-approximation de la disjonction de $\phi_1 = (j = i-1 \wedge 1 \leq \ell_1 < i-1)$ et $\phi_2 = (j < i-1 \wedge 1 \leq \ell_1 < i)$. L'opération fournie par l'analyse trouve la sous-approximation suivante $\phi = (j \leq i-1 \wedge 1 \leq \ell_1 < i \wedge \ell_1 \neq j)$ [GMT08, §4].

Premièrement, on remarque que $\mathcal{D}^{\mathbb{Z}}$ étant un domaine conjonctif, il doit pouvoir exprimer des contraintes de non-égalité pour obtenir un résultat précis. Lorsque l'on sait que l'opération de sous approximation proposée fait appel plusieurs fois à l'opération de normalisation de $\mathcal{D}^{\mathbb{Z}}$, et que cette opération pour un domaine manipulant des non-égalités, dans $\mathbb{Z}$, est exponentielle dans le nombre de non-égalités (Sec. 3.2.2.2), on comprend le coût élevé de cette opération.

Deuxièmement, même avec des contraintes de non-égalité, un domaine conjonctif ne peut représenter précisément tout ensemble non-convexe. C'est le cas ici où $\phi$ est bien la meilleure sous-approximation possible, mais perd l'information que lorsque $i > j+1$ alors $a[j] \leq a[j+1]$. Sans cette information, l'analyse ne peut pas découvrir l'invariant de tri.

Ainsi, la construction des tranches maximales par sous-approximation n'est viable que dans des programmes très simples. En effet, elle fonctionne pour le programme dont on a déroulé l'analyse ci-dessus où la tranche ($1 \leq \ell < i$) est découverte. Mais l'analyse ne traite aucun exemple avec deux boucles imbriquées, où plusieurs cas particuliers (ici, $j = i-1$) apparaissent du fait de l'utilisation de deux indices. *A contrario*, l'analyse de [GRS05], comme la nôtre [HP08], va définir des partitions très détaillées. Pour le tri par insertion, l'union des deux valeurs abstraites que l'on discute ici est représentée dans ces deux analyses sur cinq tranches ($\varphi_4, \varphi_5, \varphi_6, \varphi_7$ et $\varphi_9$, Fig. 6.11, p. 152). Si ces analyses réussissent à trouver l'invariant du tri par insertion, là où l'analyse de [GMT08] échoue, cela n'invalide en rien la faisabilité d'une construction dynamique des tranches. C'est d'ailleurs un des sujets sur lequel travaille Valentin Perrelle dans notre équipe.

**Résultats** L'analyse trouve les invariants d'exemples simples avec une boucle, comme la copie de tableau, une simple initialisation, etc. On notera dans cette catégorie l'exemple du filtre par indirection (Fig. 7.8(a), p. 195), pour lequel l'analyse montre aisément que $\forall \ell \in [0, nb], a[b[\ell]] = 0$ grâce au patron $a[b[\mathcal{L}]] = 0$. Quant aux autres programmes analysés, ce ne sont que les boucles internes de programmes de tri, comme le tri par insertion, une version simple de la phase de segmentation du tri rapide ou encore le tri par sélection.

Ainsi, si l'analyse proposée par [GMT08] est, de toutes les analyses que l'on a décrites, celle qui traite le plus de programmes, elle reste cantonnée aux programmes basés sur une unique traversée des tableaux manipulés. Enfin, les temps d'analyse sont toujours rédhibitoires, alors même que les boucles sont systématiquement déroulées deux fois, afin d'accélérer l'analyse. Il faut par exemple une minute pour analyser un programme qui recherche le minimum d'un tableau (Fig. 4.4, p. 68).

| | ← logique → | ←——— abstraction par prédicats ——— --→ | | |
|---|---|---|---|---|
| | | | ← patrons → | ← raffin. --→ |
| | *[BHI+09]* | *[FQ02]* | *[SG09]* | *[BHMR07]* |
| initialisation (Fig B.1(a)) | + | − | − | 0.27 |
| filtre par indirection (Fig. 7.8(a)) | × | − | − | − |
| filtre partitionnant (Fig. 7.3(b)) | + | − | − | 3.6 |
| premier non nul (Fig. 7.4(b)) | − | ≈ 22 | − | − |
| copie de tableau (Fig. 6.1) | − | − | − | − |
| tri par insertion (Fig. 7.5(a)) | × | − | 2.9 | − |
| insertion seulement | + | − | + | − |
| tri à bulle (Fig. 7.8(b)) | × | − | 0.47 | − |
| tri par sélection (Fig. B.3(b)) | × | ≈ 16 | 1.32 | − |
| minimum seulement | − | − | + | − |
| segmentation tri rapide (Fig. B.3(c)) | × | − | 0.43 | − |
| *machine (cadence proc.)* | | 0.7Ghz | 2.5Ghz | 1.7Ghz |
| *(capacité mém.)* | | | 4Gb | |
| *notes* | post-condition uniquement | estimation selon temps moyen/requête | | |

Figure 4.4 – Performances des analyses existantes

**Conclusions** L'apport principal des travaux présentés dans [GMT08] réside dans les valeurs abstraites qu'ils définissent. En effet, en associant à chaque tranche une propriété propre $\psi_i$, contrairement à [GRS05], il est possible de propager des propriétés de tableaux relationnelles aisément, comme on l'a vu avec l'exemple de la copie de tableau [18].

Mais ces valeurs abstraites sont aussi le problème de cette analyse. En les voulant si expressives, puisqu'elles peuvent potentiellement porter sur un nombre arbitraire de variables quantifiées, il a fallu introduire des patrons. Or, dans la pratique, les patrons utilisés pour les expérimentations imposent tous qu'une seule variable soit quantifiée, et ce, pour des raisons d'efficacité ! On peut d'ailleurs s'interroger sur ce qu'un développeur aurait choisi comme patron pour un algorithme de tri. N'aurait-il pas proposé le patron $a[\mathcal{L}] \leq a[\mathcal{L}']$ (avec en tête l'invariant $\forall \ell_1, \ell_2,\ 1 \leq \ell_1 < \ell_2 \leq n \Rightarrow a[\ell_1] \leq a[\ell_2]$), plutôt que le patron $a[\mathcal{L}] \leq a[\mathcal{L}+1]$ ? Bien entendu, on pourrait prétendre que les patrons pourraient être aisément extraits du code source, rendant ainsi l'analyse automatique. Ceci est faux : par exemple, plusieurs algorithmes de tri, comme le tri par insertion, ne font aucune comparaison explicite entre deux cellules du tableau qu'ils traitent.

Enfin, il est dommage que ces travaux se soient axés principalement sur l'opération de sous-approximation pour construire les tranches, alors que les simples partitions statiques de [GRS05] étaient adéquates pour tous les exemples qu'ils analysent avec succès.

18. Ces travaux ont été publiés en janvier 2008, soit après que soient soumis nos travaux sur ces sujets (Ch. 6) à la conférence PLDI de la même année. Si des idées similaires se retrouvent dans nos deux travaux, notamment le point qui vient d'être évoqué, c'est indépendamment que ces idées se sont développées dans nos deux équipes.

| ←-- abstraction par prédicats → | | ←——— interprétation abstraite ———→ | | | |
|---|---|---|---|---|---|
| | | ← prédicats paramétriques → | | | |
| ←-- raffinement ——→ | | | | ← patrons → | |
| *[JM07]* | *[SPW09]* | *[Cou03]* | *[All08]* | *[GMT08]* | *[GRS05]* |
| 1.19 | 0.83 | + | 0.2 | 3.2 | 1.7 |
| 4.63 | – | – | – | 12.0 | × |
| 7.96 | – | – | 0.4 | 73.0 | – |
| 2.24 | 8.81 | – | – | 24.6 | – |
| 3.65 | 0.84 | – | – | 5.5 | 338.1 |
| – | – | – | – | – | 48.5 |
| 91.22 | 2.45 | – | – | 35.9 | – |
| – | – | + | – | – | – |
| – | 409.87 | – | – | – | – |
| – | – | – | – | 59.2 | – |
| – | – | + | – | 42.2 | – |
| 1,7Ghz | 1.6Ghz | 2Ghz | | 3Ghz | 2.4Ghz |
| 512Mb | 1Gb | 1Gb | | 2Gb | 512Mb |
| | | inclut exemples dans [Čer03] | exemples initialisation sur matrices | | prédicats nécessaires (sauf init.) |

Figure 4.4 – Performances des analyses existantes (suite)

## 4.4 Synthèse des résultats et performances

Parmi toutes les informations techniques sur les analyses existantes présentées dans ce chapitre, il est difficile d'en retirer une vue générale sur les programmes qu'elles analysent et en combien de temps. C'est l'objet de cette section où ces informations sont données de manière synthétique aux Figures 4.4. Cette table est faite pour être comparée directement avec celle donnée à la Figure 7.2 (p. 179) présentant les résultats de notre analyse.

La table regroupe quasiment l'ensemble des programmes présentés dans les publications pour illustrer les capacités des différentes analyses. Ainsi, un *benchmark* petit mais incluant des programmes de natures différentes, dont certains sont complexes, s'est créé au fil du temps [19]. Il n'est pas certain qu'il s'agisse toujours des mêmes programmes, les publications faisant référence à ces programmes le plus souvent par leur nom. Par exemple, pour la phase de segmentation du tri rapide, il existe une version simple à une boucle (Fig. B.3(c), p. 205) et une version optimisée bien plus difficile à analyser avec deux boucles imbriquées (Fig. 7.5(b), p. 189).

La nature de chaque analyse est rappelée par les indications en haut de la table. Globalement, plus on avance vers la droite, plus l'analyse a des capacités automatiques. Pour chaque programme, il est renseigné si l'analyse a découvert les invariants fonctionnels du programme (présence d'un « + » ou d'un temps d'analyse s'il est connu) ; si l'analyse ne peut pas découvrir ces invariants (présence d'un « × ») ; si l'analyse n'a pas été essayée sur ce programme (présence d'un « – »). Pour être précis, les invariants fonctionnels découverts sont donnés à la Section 7.2, mais ce sont ceux auxquels on s'attend hormis

19. Nous proposons d'étendre ce *benchmark* par plusieurs autres programmes (Annexe B).

pour les programmes de tri où l'invariant de permutation n'est pas découvert (le tableau en entrée est une permutation de celui en sortie). Enfin, lorsque les temps d'analyse sont donnés, les caractéristiques de la machine utilisée le sont également.

Toute comparaison précise sur la base de ces performances n'aurait pas de sens. Cependant, on peut remarquer que plusieurs analyses montrent une certaine polyvalence [SG09, JM07, SPW09, GMT08]. Mais aucune d'elles n'est automatique et mis à part [SG09], qui est de loin la moins automatique des trois, les temps d'analyse sont très élevés, même sur les petits exemples de la moitié haute de la table. Il faut une à plusieurs secondes de temps de calcul pour traiter l'initialisation à une constante ou la copie de tableau, alors qu'il ne faudrait pas dépasser quelques centièmes de seconde pour envisager une utilisation pratique de ces analyses. Si on se restreint aux analyses automatiques [Cou03, All08, GRS05] aucune d'elles n'est donc polyvalente. C'est au point où la seule analyse à visée générale, [GRS05], n'a qu'un exemple d'initialisation analysé entièrement automatiquement à son actif.

Ainsi, toutes ces analyses sont bien loin de celle que l'on cherche à concevoir, qui doit être polyvalente, automatique et précise. On peut se demander lesquelles de ces analyses ont le plus de potentiel pour atteindre cet objectif. On écarte immédiatement l'analyse basée sur la logique [BHI+09]. Les analyses par abstraction de prédicats se sont clairement améliorées entre les travaux de 2002 et de 2009, mais la dépendance à l'objectif de preuve est trop forte. De plus, on peut avoir des doutes sur l'amélioration de leurs performances en temps puisqu'elles sont principalement liées à celles particulièrement travaillées de solveurs SAT ou SMT. Restent les analyses par interprétation abstraite où l'objectif paraît possible puisqu'on trouve à la fois des analyses précises [Cou03, All08], polyvalentes [GMT08] et automatiques [Cou03, All08, GRS05]. Mais les obstacles sont nombreux pour enlever la dépendance aux patrons de [GMT08], et aux prédicats de [GRS05] tout en atteignant des temps d'analyse correspondant à ceux des analyses par prédicats paramétriques (les temps que l'on peut attendre pour [Cou03] étant probablement du même ordre que ceux de [All08]).

## 4.5 Conclusion

**Autres analyses symboliques d'intérêt** On s'est limité jusqu'ici aux analyses du contenu des tableaux. On peut probablement trouver des idées et des méthodes applicables à ce problème dans des analyses d'autres structures de données.

On pense par exemple à l'analyse d'alias proposée dans [Deu94], capable de découvrir des invariants quantifiés de la forme $\forall \ell_1, \ell_2,\ (\ell_1 \geq 1 \wedge \ell_2 = 2\ell_1 + 1 \Rightarrow p \rightarrow next^{\ell_1} \equiv q \rightarrow next^{\ell_2})$. On pense aussi à l'analyse d'alias proposée dans [Ven02]. Nous avons retenu de ces analyses une caractéristique commune, et partagée avec plusieurs des analyses présentées dans ce chapitre : elles s'appuyent sur un domaine abstrait numérique, auquel la complexité de l'analyse est transmise [Ven05]. Dans ce dernier article, un formalisme est proposé pour intégrer analyse numérique et analyse symbolique. Cependant, nous n'avons pas su l'instantier.

Toujours sur l'analyse du tas mémoire, on notera l'analyse par interprétation abstraite présentée dans [LRSW00] sur le contenu des listes chaînées. Cette analyse est basée

sur l'analyseur TVLA, que nous avons décrit brièvement à la Section 4.3.2. Elle repose sur l'association de prédicats d'abstraction aux éléments agglomérants (ici ces éléments sont des structures contenant un champ de données $d$ et un champ pointeur $next$), qui permettent d'exprimer un invariant numérique sur le contenu des éléments agglomérés. Ceci est à comparer à l'analyse proposée dans [GRS05] (Sec. 4.3.2) qui associe des valeurs abstraites, d'un domaine abstrait numérique, aux éléments agglomérants pour représenter un invariant numérique des éléments agglomérés. Ces prédicats d'abstraction permettent de définir une analyse spécifique, comme proposé dans [Cou03] (Sec. 4.2.3). Les prédicats utilisés sont par exemple : $SORT(s)$ indiquant que la donnée $d$ de l'élément $s$ est plus petite que la donnée $d$ de l'élément qu'il pointe par $next$. $LEQ(s_1, s_2)$ indiquant que la donnée $d$ de l'élément $s_1$ est plus petite que la donnée $d$ de l'élément $s_2$. Le prédicat unaire $SORT$ associé à un élément où est aggloméré un ensemble d'éléments chaînés par le champ $next$, indique que cette liste est triée. Le prédicat binaire $LEQ$ associé à deux éléments agglomérant $e_1$ et $e_2$ indique que $\forall s_1, s_2(s_1 \in e_1 \wedge s_2 \in e_2 \Rightarrow s_1.d \leq s_2.d)$.

À partir de là, on retrouve les configurations générées par TVLA présentées pour [GRS05] qui scindent la liste chaînée du programme sous analyse selon des cas particuliers. L'analyse fonctionne de manière similaire à celle de [GRS05], dont elle est en fait le précurseur. Cette analyse traite plusieurs exemples complexes pour lesquels elle découvre les invariants de tri souhaités : le tri à bulle, le tri par insertion, et la phase de fusion du tri fusion sont analysés. Elle découvre également que le renversement d'une liste triée est trié en sens inverse, à l'aide du prédicat adéquat $RSORT$ [20]. Les temps d'analyses sont longs : il faut compter d'une à quatre minutes pour ces exemples. Enfin, cette analyse est aussi la première à découvrir la propriété de permutation indiquant que le tri par insertion renvoie la liste triée des éléments de la liste donnée en entrée.

**Conclusion** Dans ce chapitre, nous avons donné une vue précise de l'état de l'art des analyses du contenu des tableaux. Nous avons notamment décrit les classes de propriétés qui ont reçu l'intérêt des différents travaux de recherches de ce domaine, détaillé ces travaux, et comparé les performances en précision comme en temps des analyses qu'ils proposent.

Nous avons vu qu'aucune de ces analyses n'est satisfaisante vis-à-vis de notre objectif. Cependant, plusieurs techniques paraissent pouvoir être réutilisées pour peu qu'on en évite les écueils. On pense notamment au partitionnement symbolique des tableaux.

20. Ce dernier exemple est l'occasion de rappeler que l'implantation d'un algorithme avec une liste chaînée est souvent différente de celle utilisant un tableau. Par conséquent, leurs analyses peuvent être plus ou moins difficiles. Pour implanter le renversement d'une liste chaînée, on peut traverser cette liste en enfilant en tête d'une seconde liste les éléments rencontrés. Pour le renversement d'un tableau $a$, on effectue $\frac{n}{2}$ opérations d'échanges en place, plutôt que d'utiliser un second tableau $b$ que l'on remplirait par un traitement similaire à celui utilisé pour la liste chainée. Il est bien plus difficile d'exprimer et de découvrir l'invariant du contenu du tableau dont on est en train d'échanger des cellules en place que de découvrir l'invariant du tableau $b$.

# Deuxième partie

# Contributions

# Préliminaires

Avant de définir dans les deux chapitres suivants (Ch. 5 et 6) les analyses statiques qui constituent notre contribution, on souhaite donner ici une idée de leurs champs d'application. Pour cela, nous définissons un langage de programmation, tel que pour les programmes écrits dans ce langage nos deux analyses ont des chances de donner des résultats significatifs.

Ce langage éclaire les hypothèses que l'on formule sur nos analyses. Certaines sont orthogonales à la précision de ces analyses : le langage de programmation proposé n'est pas procédural et ne fait pas usage de déclarations. D'autres au contraire les impactent directement : il n'y a pas d'allocation dynamique de mémoire, de pointeurs et aucune instruction ne fait usage d'effets de bord.

Ce langage permet aussi d'exposer les formalismes d'abstraction (Sec. 5.3 et 6.3) et les fonctions de transfert (Sec. 5.7.2 et 6.6.2) à mettre en œuvre pour assurer la correction de chaque analyse. On donne pour cela une sémantique d'atteignabilité au langage, qui sera notre sémantique concrète.

Enfin, ce langage est équivalent au langage d'automates accepté en entrée de notre prototype d'analyseur statique (Ch. 7.1). C'est donc naturellement ce langage que l'on utilisera pour présenter nos exemples de programmes dans les prochains chapitres.

## Programmes considérés pour illustrer les analyses

Notre langage de programmation, décrit à la Figure 4.5, est un langage impératif simple avec tableaux à une dimension.

Il utilise deux types de données : celui des entiers relatifs $\mathbb{Z}$, qui est le type des variables indiçant les tableaux, et un type arbitraire noté $\mathbb{K}$, qui est le type du contenu des tableaux et des autres variables scalaires. Ainsi, un programme donné manipule un ensemble fini de variables dites de contenu, soit des scalaires $x, y, \ldots \in V$ soit des tableaux $a, b, \ldots \in A$, prenant leurs valeurs dans $\mathbb{K}$ et un ensemble fini de variables dites d'indice $i, j, \ldots \in I$, prenant leurs valeurs dans $\mathbb{Z}$. Une restriction, subtile à l'usage, est la stricte séparation entre les variables de contenu et les variables d'indice : l'affectation $a[i] := i + 1$ n'est par exemple pas permise.

Un programme est une séquence d'instructions qui peuvent être une boucle « tant que », une conditionnelle « si-alors-sinon » ou une affectation.

Pour les deux premières, l'expression conditionnant le contrôle est formée d'un ou plusieurs tests, comparant des expressions d'indice ou des expressions de contenu. Ces tests peuvent être assemblés par les opérateurs booléens de conjonction, de disjonction ou de négation. Les opérateurs de comparaisons sur $\mathbb{K}$ (opérateurs $\boxtimes$) sont à définir puisqu'ils dépendent de $\mathbb{K}$ lui-même.

Constantes : $c \in \mathbb{Z}, k \in \mathbb{K}$
Variables : $i \in I, x \in V, a \in A$
Opérateurs : $\bowtie \in \{=, \neq, <, \leq, \}$ et $\diamond \in \{+, -\}$ ; $\boxtimes$ et $\square$ non fixés

$$
\begin{array}{rcl}
\langle \textit{programme} \rangle & ::= & \langle \textit{bloc} \rangle \\
\langle \textit{bloc} \rangle & ::= & \langle \textit{instruction} \rangle \\
& | & \langle \textit{instruction} \rangle \ ; \ \langle \textit{bloc} \rangle \\
\langle \textit{instruction} \rangle & ::= & \langle \textit{affectation} \rangle \\
& | & \textbf{while} \ \langle \textit{condition} \rangle \ \textbf{do} \ \langle \textit{bloc} \rangle \\
& | & \textbf{if} \ \langle \textit{condition} \rangle \ \textbf{then} \ \langle \textit{bloc} \rangle \ \textbf{else} \ \langle \textit{bloc} \rangle \\
\langle \textit{condition} \rangle & ::= & \langle \textit{test} \rangle \\
& | & \textbf{not} \ \langle \textit{condition} \rangle \\
& | & \langle \textit{condition} \rangle \ \textbf{and} \ \langle \textit{condition} \rangle \\
& | & \langle \textit{condition} \rangle \ \textbf{or} \ \langle \textit{condition} \rangle \\
\langle \textit{test} \rangle & ::= & \langle \textit{expression-indice} \rangle \bowtie \langle \textit{expression-indice} \rangle \\
& | & \langle \textit{expression-contenu} \rangle \boxtimes \langle \textit{expression-contenu} \rangle \\
\langle \textit{expression-indice} \rangle & ::= & c \\
& | & i \\
& | & \langle \textit{expression-indice} \rangle \diamond c \\
\langle \textit{expression-contenu} \rangle & ::= & k \\
& | & x \\
& | & a[\langle \textit{expression-indice} \rangle] \\
& | & \langle \textit{expression-contenu} \rangle \ \square \ \langle \textit{expression-contenu} \rangle \\
\langle \textit{affectation} \rangle & ::= & i \ \textbf{:=} \ \langle \textit{expression-indice} \rangle \\
& | & x \ \textbf{:=} \ \langle \textit{expression-contenu} \rangle \\
& | & a[\langle \textit{expression-indice} \rangle] \ \textbf{:=} \ \langle \textit{expression-contenu} \rangle
\end{array}
$$

Figure 4.5 – Syntaxe du langage de programmation

Les expressions d'indice sont fortement contraintes : il s'agit soit d'une constante entière, soit de l'addition d'une variable d'indice et d'une constante. Quant aux expressions de contenu, au lieu de les laisser totalement indéfinies, nous avons choisi d'expliciter le fait que les tableaux n'ont qu'une dimension et qu'ils ne peuvent être indicés que par des expressions d'indice. Cette restriction mise à part, on peut définir librement les constantes de contenu et les opérateurs binaires (opérateurs $\square$) permettant de combiner les expressions de contenu de base : constantes, variables, cellule d'un tableau.

Restent les instructions d'affectations qui respectent simplement les types de données. Dans [HP08], nous avions imposé que les affectations des variables d'indices ne pouvaient être que des incrémentations ou des décrémentations. Cependant, dans les faits l'analyse du contenu des tableaux que l'on propose peut aussi donner des résultats précis sur un parcours non contigu d'un tableau.

On remarquera que ce langage de programmation est complet au sens de Turing. Par conséquence du théorème de Rice [Ric53], aucune analyse statique ne peut décider, pour tous les programmes pouvant être décrits par ce langage, s'ils satisfont une propriété intéressante.

### Sémantique concrète

On décrit un état mémoire – on utilisera le terme d'environnement, par un triplet de fonctions de valuations des variables du programme, noté $\rho = (\rho_i, \rho_v, \rho_a) \in Env$ . Pour les scalaires, on a $\rho_i : I \to \mathbb{Z}$ et $\rho_v : V \to \mathbb{K}$. Pour les tableaux, nous ferons l'hypothèse que tout accès à un tableau est effectué dans ses bornes. Hors de ces bornes, dont la borne inférieure sera 1, les tableaux seront valués à une valeur spéciale $\perp\!\!\!\perp$. En notant $\mathbb{K}_{\perp\!\!\!\perp}$ l'ensemble $\mathbb{K}$ augmenté de cette valeur, on a donc $\rho_a : A \to (\mathbb{Z} \to \mathbb{K}_{\perp\!\!\!\perp})$.

On souhaite profiter du travail exposé à la Section 2.2.2.1 où l'on donne une sémantique d'atteignabilité à des programmes décrits par des automates interprétés (Déf. 2.5, p. 20). Ainsi, pour définir la sémantique d'atteignabilité de nos programmes, on donne simplement une traduction des ces derniers en de tels automates $(K, \delta, k_0)$. Cette traduction est définie par une grammaire attribuée sur notre langage (Fig. 4.6).

La création d'une place $k$ et son ajout à $K$ est décrite par « $\circ(k)$ ». On note $id_g$ la garde toujours vraie ($\rho \to \mathit{vrai}$), $id_a$ l'action sans effet ($\rho \to \rho$) et $id_c$ la commande composée de cette garde et de cette (non-)action $(id_g, id_a)$. Chaque non-terminal a pour attributs hérités la place $k$ auquel il doit se raccrocher et la relation de transition $\delta$ construite jusqu'ici. Il a pour attributs synthétisés la dernière place $k'$ à laquelle sa traduction mène et la relation de transition $\delta'$, incluant $\delta$, permettant de parvenir à cette place. Les schémas de traduction de la boucle « tant que » et de la conditionnelle « si-alors-sinon » expliquent d'eux-mêmes le calcul des attributs.

$$
\begin{array}{rcl}
\langle \mathit{programme} \rangle \uparrow \delta' & ::= & \langle \mathit{bloc} \rangle \downarrow k_0 \downarrow \emptyset \uparrow k' \uparrow \delta' \\
\langle \mathit{bloc} \rangle \downarrow k \downarrow \delta \uparrow k' \uparrow \delta' & ::= & \langle \mathit{instruction} \rangle \downarrow k \downarrow \delta \uparrow k' \uparrow \delta' \\
 & | & \langle \mathit{instruction} \rangle \downarrow k \downarrow \delta \uparrow k'' \uparrow \delta'' \ ; \ \langle \mathit{bloc} \rangle \downarrow k'' \downarrow \delta'' \uparrow k' \uparrow \delta' \\
\langle \mathit{instruction} \rangle \downarrow k \downarrow \delta \uparrow k' \uparrow \delta' & ::= & \langle \mathit{affectation} \rangle \\
 & & \qquad \text{avec } \circ(k'), \delta' = \delta \cup (k, (id_g, [\![\langle \mathit{affectation} \rangle]\!]), k') \\
 & | & \textbf{while } \langle \mathit{condition} \rangle \textbf{ do } \langle \mathit{bloc} \rangle \downarrow k'' \downarrow \delta'' \uparrow k''' \uparrow \delta''' \\
 & & \qquad \text{avec } \circ(k''), \delta'' = (k, ([\![\langle \mathit{condition} \rangle]\!], id_a), k'') \\
 & & \text{et } \circ(k'), \delta' = \delta \cup \delta''' \cup (k''', id_c, k) \cup (k, (\neg[\![\langle \mathit{condition} \rangle]\!], id_a), k') \\
 & | & \textbf{if } \langle \mathit{condition} \rangle \textbf{ then } \langle \mathit{bloc} \rangle \downarrow k'' \downarrow \delta'' \uparrow k'''' \uparrow \delta'''' \\
 & & \qquad \textbf{else } \langle \mathit{bloc} \rangle \downarrow k''' \downarrow \delta''' \uparrow k''''' \uparrow \delta''''' \\
 & & \qquad \text{avec } \circ(k''), \delta'' = (k, ([\![\langle \mathit{condition} \rangle]\!], id_a), k'') \\
 & & \qquad \text{et } \circ(k'''), \delta''' = (k, (\neg[\![\langle \mathit{condition} \rangle]\!], id_a), k''') \\
 & & \text{et } \circ(k'), \delta' = \delta \cup \delta'''' \cup \delta''''' \cup (k'''', id_c, k') \cup (k''''', id_c, k')
\end{array}
$$

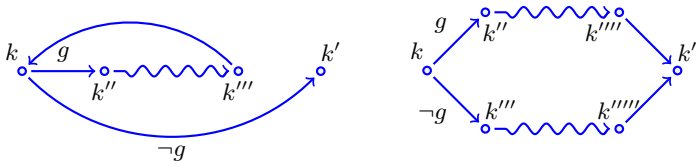



Figure 4.6 – Compilation des programmes en automates interprétés

Ce calcul nécessite bien entendu la définition des gardes et des actions à porter sur les transitions de l'automate. Les premières sont issues des $\langle \mathit{condition} \rangle$ et les secondes sont uniquement des $\langle \mathit{affectation} \rangle$. Pour définir la sémantique des conditions comme celle des affectations, il nous faut la sémantique des expressions. On utilisera la fonction $[\![\cdot]\!]$

pour associer à l'un de ces éléments sa sémantique :

$$\begin{aligned}[\![\cdot]\!] : \quad & \langle \textit{condition} \rangle \to \textit{Env} \to \mathbb{B} \\ & \langle \textit{affectation} \rangle \to \textit{Env} \to \textit{Env} \\ & \langle \textit{expression-indice} \rangle \to \textit{Env} \to \mathbb{Z} \\ & \langle \textit{expression-contenu} \rangle \to \textit{Env} \to \mathbb{K}.\end{aligned}$$

On notera que c'est l'hypothèse formulant que tout accès à un tableau se trouve dans ses bornes qui assure qu'une expression de contenu s'évalue toujours dans $\mathbb{K}$, et non $\mathbb{K}_{\perp\!\!\!\perp}$. La définition de la sémantique des expressions, donnée à la Figure 4.7, se résume à de simples appels aux fonctions de valuations appropriées composant l'environnement.

$$\begin{aligned}
[\![c]\!](\rho_i, \rho_v, \rho_a) &= c \\
[\![i]\!](\rho_i, \rho_v, \rho_a) &= \rho_i(i) \\
[\![\langle \textit{expression-indice} \rangle \diamond \mathrm{c}]\!](\rho_i, \rho_v, \rho_a) &= \\
& [\![\langle \textit{expression-indice} \rangle]\!](\rho_i, \rho_v, \rho_a) \diamond c \\
[\![k]\!](\rho_i, \rho_v, \rho_a) &= k \\
[\![x]\!](\rho_i, \rho_v, \rho_a) &= \rho_v(x) \\
[\![a[\langle \textit{expression-indice} \rangle]]\!](\rho_i, \rho_v, \rho_a) &= \\
& \rho_a(a)([\![\langle \textit{expression-indice} \rangle]\!](\rho_i, \rho_v, \rho_a)) \\
[\![\langle \textit{expression-contenu}_1 \rangle \ \square \ \langle \textit{expression-contenu}_2 \rangle]\!](\rho_i, \rho_v, \rho_a) &= \\
& [\![\langle \textit{expression-contenu}_1 \rangle]\!](\rho_i, \rho_v, \rho_a) \ \square \ [\![\langle \textit{expression-contenu}_2 \rangle]\!](\rho_i, \rho_v, \rho_a)
\end{aligned}$$

Figure 4.7 – Sémantique concrète des expressions

De même, la sémantique des conditions découle simplement des définitions précédentes. Elle est donnée à la Figure 4.8.

$$\begin{aligned}
[\![\textbf{not } \langle \textit{condition} \rangle]\!](\rho_i, \rho_v, \rho_a) &= \\
& \neg \, [\![\langle \textit{condition} \rangle]\!](\rho_i, \rho_v, \rho_a) \\
[\![\langle \textit{condition}_1 \rangle \textbf{ and } \langle \textit{condition}_2 \rangle]\!](\rho_i, \rho_v, \rho_a) &= \\
& [\![\langle \textit{condition}_1 \rangle]\!](\rho_i, \rho_v, \rho_a) \wedge [\![\langle \textit{condition}_2 \rangle]\!](\rho_i, \rho_v, \rho_a) \\
[\![\langle \textit{condition}_1 \rangle \textbf{ or } \langle \textit{condition}_2 \rangle]\!](\rho_i, \rho_v, \rho_a) &= \\
& [\![\langle \textit{condition}_1 \rangle]\!](\rho_i, \rho_v, \rho_a) \vee [\![\langle \textit{condition}_2 \rangle]\!](\rho_i, \rho_v, \rho_a) \\
[\![\langle \textit{expression-indice}_1 \rangle \bowtie \langle \textit{expression-indice}_2 \rangle]\!](\rho_i, \rho_v, \rho_a) &= \\
& [\![\langle \textit{expression-indice}_1 \rangle]\!](\rho_i, \rho_v, \rho_a) \bowtie [\![\langle \textit{expression-indice}_2 \rangle]\!](\rho_i, \rho_v, \rho_a) \\
[\![\langle \textit{expression-contenu}_1 \rangle \barwedge \langle \textit{expression-contenu}_2 \rangle]\!](\rho_i, \rho_v, \rho_a) &= \\
& [\![\langle \textit{expression-contenu}_1 \rangle]\!](\rho_i, \rho_v, \rho_a) \barwedge [\![\langle \textit{expression-contenu}_2 \rangle]\!](\rho_i, \rho_v, \rho_a)
\end{aligned}$$

Figure 4.8 – Sémantique concrète des conditions

Enfin, on donne à la Figure 4.9 la sémantique des affectations dont la définition, là encore, est directe : selon le type d'affectation, la fonction de valuation correspondante est modifiée.

À présent, la traduction de nos programmes vers les automates interprétés est complètement définie. Nous pouvons utiliser la sémantique d'atteignabilité donnée à l'Équation 2.1 (p. 21) comme sémantique concrète de nos programmes, sur le domaine $\mathcal{D}^\flat = \mathcal{P}(\textit{Env})$.

$$\begin{aligned}
[\![i := \langle \textit{expression-indice} \rangle]\!](\rho_i, \rho_v, \rho_a) \quad &= \\
(\rho_i[i \mapsto z],\ \rho_v,\ \rho_a) \text{ où } z &= [\![\langle \textit{expression-indice} \rangle]\!](\rho_i, \rho_v, \rho_a) \\
[\![x := \langle \textit{expression-contenu} \rangle]\!](\rho_i, \rho_v, \rho_a) \quad &= \\
(\rho_i,\ \rho_v[x \mapsto c],\ \rho_a) \text{ où } c &= [\![\langle \textit{expression-contenu} \rangle]\!](\rho_i, \rho_v, \rho_a) \\
[\![a[\langle \textit{expression-indice} \rangle] := \langle \textit{expression-contenu} \rangle]\!](\rho_i, \rho_v, \rho_a) \quad &= \\
(\rho_i,\ \rho_v,\ \rho_a[a \mapsto \rho_a(a)[z \mapsto c]]) \text{ où } z &= [\![\langle \textit{expression-indice} \rangle]\!](\rho_i, \rho_v, \rho_a) \\
\text{et } c &= [\![\langle \textit{expression-contenu} \rangle]\!](\rho_i, \rho_v, \rho_a)
\end{aligned}$$

Figure 4.9 – Sémantique concrète des affectations

On donne sa définition ajustée aux notations propres du chapitre, où $x_k^\flat \in \mathcal{D}^\flat$ :

$$x_{k_0}^\flat = \mathit{Env} \quad \wedge \bigwedge_{k' \in (K \setminus k_0)} x_{k'}^\flat = \bigsqcup_{(k,(g,a),k') \in \delta}^{\flat} \{\!|a|\!\}^\flat \circ \{\!|g|\!\}^\flat(x_k^\flat)$$

et où $\{\!|a|\!\}^\flat(x_k^\flat) = \{\rho' \in \mathit{Env} \mid \exists \rho \in x_k^\flat,\ \rho' = a(\rho)\}$ et $\{\!|g|\!\}^\flat(x_k^\flat) = \{\rho \in \mathit{Env} \mid \rho \in x_k^\flat,\ g(\rho) \text{ est vrai}\}$. On rappelle que la fonction $a$ est soit $id_a$ soit $[\![\langle \textit{affectation} \rangle]\!]$ et la fonction $g$ est soit $id_g$ soit $[\![\langle \textit{condition} \rangle]\!]$.

**Programmes sans tableaux**

La première analyse que l'on propose (Ch. 5) est une analyse numérique qui ne porte par sur les tableaux. On considérera donc pour ce chapitre, d'une part, que les programmes ne font pas usage de ces derniers ($I = A = \emptyset$) et d'autre part, que l'ensemble des valeurs de contenu $\mathbb{K}$ est un des ensembles mathématiques parfaits $\mathbb{Z}$, $\mathbb{Q}$, ou $\mathbb{R}$. On prendra alors pour opérateurs de comparaison sur $\mathbb{K}$, les opérateurs $\{=, \neq, <, \leq, \}$, et pour opérateurs binaires sur $\mathbb{K}$ les opérateurs $\{+, -\}$. Notre environnement sera simplifié à $\mathit{Env} = V \to \mathbb{K}$.

Pour la seconde analyse (Ch. 6), les programmes considérés seront ceux présentés dans ces préliminaires, sans restriction.

# Chapitre 5

# Le domaine abstrait sémantique des zones précisant alias

Dans de nombreuses situations, des variables numériques entières sont utilisées pour adresser des objets. C'est le cas des adresses mémoires, des pointeurs dans les langages comme le langage C, des adresses de composants dans les descriptions haut niveau de systèmes sur puces, ou encore des indices de tableaux. Il est bien rare alors que le problème d'alias ne survienne pas en présence de ce mécanisme d'adressage. Source d'erreurs complexes dans les programmes, le phénomène d'alias complique aussi très sérieusement leur analyse. Pour les tableaux par exemple, qui sont notre motivation première, savoir que $i = j$ permet d'assurer que $a[i]$ et $a[j]$ sont des alias d'une même cellule du tableau $a$ et donc que tout changement sur $a[i]$ touche implicitement $a[j]$. Mais tout aussi importante est la connaissance que $i \neq j$, qui permet de garder $a[j]$ inchangée lorsque $a[i]$ change. Étonnamment la communauté de l'interprétation abstraite n'a jusque-là pas abordé la définition d'un domaine abstrait numérique capable d'exprimer directement des contraintes de non-égalité entre paires de variables. C'est probablement parce que lorsque la définition d'un domaine abstrait repose sur un ensemble convexe, il est plus aisé d'en maîtriser la complexité. Nous avons vu des travaux de la communauté du *model checking* où de tels invariants peuvent être représentés plus ou moins directement mais dans des structures à l'algorithmique fort coûteuse (Sec. 3.2.1).

De nos motivations (Sec. 5.1) et de ces remarques est venue la définition du domaine des zones précisant alias. Ce domaine combine des contraintes de zones à des contraintes de non-égalité de la forme $x_i \neq x_j$ et $x_i \neq 0$ (Sec. 5.2). Une fois une connexion de Galois donnée avec le domaine concret (Sec. 5.3), on s'intéresse longuement, car cela s'avère loin d'être évident, à la forme normale de ces conjonctions de contraintes. Cette forme normale, canonique, est radicalement dépendante de l'ensemble numérique dans lequel les solutions sont recherchées. Aussi en est-il de la complexité en temps pour calculer cette forme normale : dans le cas dense (Sec. 5.4) elle est obtenue en $\mathcal{O}(n^4)$ et dans le cas discret (Sec. 5.5) en temps exponentiel. Pour ce dernier cas, on donne une solution incomplète s'exécutant en $\mathcal{O}(n^4)$ également. Enfin, on donne les opérateurs de treillis (Sec. 5.6) de ce domaine et les opérateurs sémantiques (Sec. 5.7) nécessaires à l'analyse des programmes que nous avons présentés en préliminaires. Nous concluons ce chapitre (Sec. 5.8) en dressant un bilan en demi-teinte. D'un côté, nous n'avons pas trouvé plus que quelques exemples où des invariants de non-égalités sont découverts (Sec. 5.8).

De plus, des analyses de congruences (Fig. 3.2(c), p. 31) s'avèrent plus pertinentes sur des programmes pour lesquels on aurait souhaité découvrir, à un point de contrôle, la non-égalité de deux variables. D'un autre côté, le pouvoir d'expression de ce domaine est intéressant en tant que tel, répondant au besoin de représenter des non-égalités.

Une partie des travaux relatés dans ce chapitre a fait l'objet d'une publication à la conférence VMCAI [PH07].

## 5.1 Objectifs

Comme nous l'avons dit, savoir que deux variables numériques ont toujours des valeurs différentes, à un point de contrôle, peut être très utile à la précision d'une autre analyse. On prend pour exemple le programme suivant qui cherche simplement à permuter trois valeurs du tableau $a$ :

$$x := a[i]\,;\; y := a[j]\,;\; z := a[k]\,;\; a[i] := y\,;\; a[j] := z\,;\; a[k] := x.$$

L'analyse proposée dans [PH10] s'intéresse aux multi-ensembles de valeurs que contiennent les tableaux, et va chercher sur ce programme à montrer que le tableau $a$ en sortie est une permutation du tableau $a$ en entrée. En l'occurrence elle conclut que ce n'est pas forcément le cas parce que lors de l'affectation à $a[k]$, ne sachant pas si $i = k$ ou si $i \neq k$, l'analyse a conclu que la valeur dans la variable $y$ pouvait être écrasée (cas où $k = i$). Si une autre analyse avait prouvé, auparavant sur le programme, que $i \neq k$, qui n'est autre que l'hypothèse de correction de ce programme, cette seconde analyse aurait prouvé la propriété de permutation.

On souhaite donc concevoir un domaine abstrait pour déterminer ces cas d'alias, ou autrement dit, qui puisse déterminer que deux variables sont égales ou (et surtout) non égales. En l'exprimant ainsi, on remarque simplement que ce domaine abstrait pourrait découvrir ces propriétés qu'il s'agisse de variables d'adresse ou non. Dans tous les cas, les propriétés de non-égalités permettraient d'exprimer des invariants intéressants. Toujours autour des tableaux, on donne trois exemples. Dans [GMT08], il est remarqué que lors de l'analyse du tri par insertion, il arrive qu'une partie du tableau soit triée strictement sauf deux cases consécutives particulières. Un tel invariant peut s'exprimer par $\forall \ell, 1 \leq \ell < i \wedge \ell \neq j \Rightarrow a[\ell] < a[\ell + 1]$. Dans [HHKV10], il est proposé une analyse permettant de classifier des matrices. On y trouve entre autres les matrices diagonales pour lesquelles l'invariant à montrer est $\forall \ell, k, 1 \leq \ell, k \leq n \wedge \ell \neq k \Rightarrow a[\ell][k] = 0$. Enfin, lorsqu'on recherche une valeur $x$ particulière dans un tableau, on souhaite aussi identifier l'ensemble des cellules visitées où l'élément n'apparaît pas. Ici, l'invariant intéressant est par exemple $\forall \ell, 1 \leq \ell < i \Rightarrow a[\ell] \neq x$. Ces invariants pourraient très bien être exprimés à l'aide du domaine que nous proposons : d'ailleurs ce dernier invariant est inféré, par l'analyse que l'on présente au chapitre suivant (Ch. 6), grâce à ce domaine (Sec. 7.2.2.7).

On s'intéresse donc maintenant au domaine à définir pour permettre d'inférer ces cas d'alias : ici nous aurons une exigence de complexité « raisonnable », comme celle qui a conduit à la définition des domaines faiblement relationnels (Sec. 3.1).

## 5.2 Domaine et représentations considérés

On pourrait être tenté de définir un domaine combinant seulement des contraintes d'égalités et de non-égalités. En se restreignant à des contraintes entre paire de variables, ce domaine aurait une complexité cubique, probablement moindre. Cependant, son expressivité serait trop restreinte pour donner une analyse intéressante. Les seules relations que l'on pourrait déduire viendraient de la transitivité de l'égalité ($x_i = x_j \wedge x_j = x_k \Rightarrow x_i = x_k$) et du fait que deux variables égales sont non égales aux mêmes variables ($x_i = x_j \wedge x_i \neq x_k \Rightarrow x_j \neq x_k$). C'est donc naturellement que l'on se tourne vers des contraintes plus expressives. Les contraintes de zones (Déf. 3.1, p. 33) enrichissent notamment le pouvoir de déduction précédent : des inégalités strictes peuvent être propagées ($x_i \leq x_j \leq x_k \wedge x_i \neq x_j \Rightarrow x_i \neq x_k$), des ensembles non convexes peuvent être exprimés ($x_i < x_j \vee x_j < x_i \Leftrightarrow x_i \neq x_j$).

En couplant ainsi des contraintes de non-égalité représentant des situations de non-alias ($x_i \neq x_j$) à des contraintes de zones ($x_i - x_j \leq c, c \in \mathbb{K}$), on pourrait être tenté d'ajouter également des contraintes de non-égalité plus complexes, comme $x_i - x_j \neq c$. Mais on aurait alors un domaine avec un nombre arbitraire de contraintes ($x_i - x_j \neq 1, x_i - x_j \neq 3, x_i - x_j \neq 4$, etc). Ceci entrerait en contradiction avec notre objectif de complexité raisonnable. En en restant à une représentation des cas de non-alias seulement, on peut imaginer réutiliser l'algorithmique cubique du domaine abstrait des zones (Sec. 3.1.1).

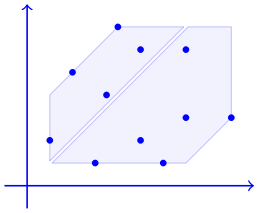
Ainsi, le domaine que nous proposons est la conjonction de contraintes particulières qui sont, soit une contrainte de zone, soit une contrainte de non-égalité de la forme $x_i \neq x_j$. On note que les éléments de ce domaine permettent de préciser tous les cas d'alias possibles pour une paire de variables. En effet, les variables $x_i$ et $x_j$ :

- forment une paire d'alias si leurs valeurs sont égales ($0 \leq x_i - x_j \leq 0$) ;
- ne peuvent pas former une paire d'alias si leurs domaines de valeurs ne s'intersectent pas ($c_1 \leq x_i - x_j \leq c_2$ où $c_1, c_2 \in \overline{\mathbb{K}}$ et $0 \notin [c_1, c_2]$) [1] ;
- et ne peuvent pas former une paire d'alias si ($x_i \neq x_j$), quand bien même leurs domaines s'intersectent ($-c_1 \leq x_i - x_j \leq c_2$) où $c_1, c_2 \in \overline{\mathbb{K}}^+$.

On réunit donc sous le nom de « contrainte de zones et alias » (Déf. 5.1) ces contraintes de zone et ces contraintes de non-égalité.

Définition 5.1 (contraintes de zone et alias). *Une contrainte de zone et alias est, soit une contrainte de zone sur $\mathbb{K}$ (Déf. 3.1), soit une contrainte de non-égalité $x_i \neq x_j$, où $x_i, x_j \in V$. Cette définition inclut les contraintes de la forme $x_i \neq 0$, par $x_i \neq x_0$ où $x_0$ est la variable toujours égale à 0 introduite pour définir les contraintes de zone.*

L'ensemble des points de $\mathbb{K}^n$ satisfaisant un ensemble de contraintes de zone et alias, sera appelé une *zone précisant alias*. Ci-contre, l'abstraction par une zone précisant alias d'un ensemble de points dans deux dimensions. On pourra la comparer aux abstractions de ce même ensemble par les domaines abstraits



Zones précisant alias [PH07] : $x_i - x_j \leq c_{ij}$, $x_i \neq x_j$

1. On rappelle que $\overline{\mathbb{K}} = \mathbb{K} \cup \{+\infty\}$.

existants dans la littérature (Fig. 3.2(a) et 3.2(c)).

On définit une représentation matricielle des conjonctions de contraintes de zone et alias (Sec. 5.2.1) C'est sur cette représentation que reposera le domaine abstrait des zones précisant alias. Une représentation en terme de graphe est également donnée (Sec. 5.2.2) pour raisonner et ainsi faciliter le développement des algorithmes. Ces représentations étendent simplement celles utilisées pour les zones.

### 5.2.1 Les *d*DBM : matrices de bornes et de non-égalités

Pour représenter un ensemble de contraintes de zone et alias, on étend la structure de données des DBM (Déf. 3.3, Sec. 3.1.1.1) par une seconde matrice de même taille représentant les contraintes de non-égalité. On utilisera l'acronyme *d*DBM pour désigner ces matrices de bornes et de non-égalités. Cet acronyme vient de l'anglais *disequality Difference-Bound Matrices*.

Définition 5.2 (matrice de bornes et de non-égalités). *Une matrice de bornes et de non-égalités est un couple, noté* $(m^{\leq}, m^{\neq})$*, formé d'une matrice de bornes (Déf. 3.3) et d'une matrice symétrique de booléens.*

L'occupation mémoire de ces matrices de bornes et de non-égalités est de $\mathcal{O}(n^2)$. Cette représentation n'est pas optimale : nous l'avons choisie parce qu'elle permet d'exprimer la plupart des opérateurs de manière syntaxique. L'opportunité d'une représentation creuse est discutée à la fin du chapitre (Sec. 5.8).

On associe à un ensemble $C$ de contraintes de zone et alias une *d*DBM le représentant de la manière suivante. Seules les contraintes imposant les restrictions les plus fortes sont considérées :

$$m^{\leq}_{ij} = \inf\{c \in \mathbb{K} \mid (x_i - x_j \leq c) \in C\} \qquad m^{\neq}_{ij} = \exists (x_i \neq x_j) \in C,$$

où $\inf\{\emptyset\} = +\infty$. On trouve aux Figures 5.1(a)–(b) un exemple d'ensemble de contraintes de zone et alias et la *d*DBM associée ainsi.

### 5.2.2 Graphes mixtes de potentiels et de non-égalités

Un ensemble de contraintes de zone et alias peut également être représenté par un graphe mixte, les contraintes de zones étant représentées par leur graphe orienté de potentiels (Déf. 3.2, p. 33) et les contraintes de non-égalité par un graphe non orienté. Dans ce dernier graphe, les sommets représentant les variables non égales sont adjacents.

Soit $L = \{0, \ldots, |V|\}$ les labels des sommets du graphe. On note $\mathcal{G}(m) = (L, E^{\leq}, E^{\neq}, w)$ ce graphe mixte, que l'on définit pour plus de simplicité à partir d'une *d*DBM :

$$\begin{array}{ll} E^{\leq} \subseteq L \times L & E^{\neq} \subseteq L \times L \\ E^{\leq} = \{(i,j) \mid m^{\leq}_{ij} < +\infty\} & E^{\neq} = \{(i,j) \mid m^{\neq}_{ij} = \mathit{vrai}\} \\ w \in E^{\leq} \to \mathbb{K} & \forall e = (i,j) \in E^{\leq} \quad w(e) = m^{\leq}_{ij} \end{array}$$

La Figure 5.1(c) montre le graphe mixte associé à l'ensemble de contraintes de zone et alias de la Figure 5.1(a).

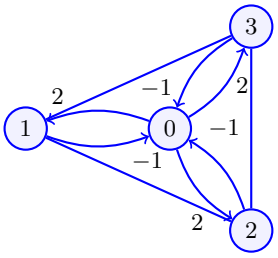
On notera un chemin simple orienté de $\mathcal{G}$ par $\langle i_1, \ldots, i_\ell \rangle$ et un chemin simple mixte, que l'on définit comme passant par au moins une arête non orientée, par $\langle\langle i_1, \ldots, i_\ell \rangle\rangle$. Cette arête peut-être fixée en notant $\langle\langle i_1, \ldots, i_k - i_{k+1}, \ldots, i_\ell \rangle\rangle$.

$$\begin{cases} 1 \leq x_1, x_2, x_3 \leq 2 \\ x_1 \neq x_2 \\ x_2 \neq x_3 \\ x_1 \neq x_3 \end{cases}$$

(a) Un ensemble $C$ de contraintes de zone et alias définies sur $V = \{x_1, x_2, x_3\}$

$$\begin{array}{c} \begin{matrix} x_0 & x_1 & x_2 & x_3 \end{matrix} \\ \begin{pmatrix} +\infty & -1 & -1 & -1 \\ 2 & +\infty & +\infty & +\infty \\ 2 & +\infty & +\infty & +\infty \\ 2 & +\infty & +\infty & +\infty \end{pmatrix} \leq \\ \begin{pmatrix} \mathit{faux} & \mathit{faux} & \mathit{faux} & \mathit{faux} \\ \mathit{faux} & \mathit{faux} & \mathit{vrai} & \mathit{vrai} \\ \mathit{faux} & \mathit{vrai} & \mathit{faux} & \mathit{vrai} \\ \mathit{faux} & \mathit{vrai} & \mathit{vrai} & \mathit{faux} \end{pmatrix} \neq \end{array}$$

(b) Une *d*DBM m représentant $C$




(c) Le graphe mixte de potentiels et de non-égalités $\mathcal{G}(m)$ associé à $m$

Figure 5.1 – Les deux représentations d'une conjonction de contraintes de zone et alias

## 5.3 Abstraction par connexion de Galois

On établit aisément une fonction de concrétisation de l'ensemble des *d*DBM à notre domaine concret (*cf.* Préliminaires – Sémantique concrète). Il s'agit de l'ensemble des valuations des variables de $V$ satisfaisant l'ensemble des contraintes représentées par la *d*DBM :

$$\gamma^{ddbm}(m) = \{(x_1, \ldots, x_n) \to (c_1, \ldots, c_n) \in \mathit{Env} \mid \forall i, j \in [0, n] \quad c_i - c_j \leq m_{ij}^{\leq} \\ \wedge\, m_{ij}^{\neq} \Rightarrow c_i \neq c_j, \text{où } c_0 = 0\}.$$

On notera que $\gamma^{ddbm}$ n'est pas injective, ce qui signifie que différentes *d*DBM peuvent représenter un même élément concret. L'ensemble des images des valuations composant un élément concret, est une zone précisant alias. Par la suite, on confondra donc ces deux notions. On trouve un exemple à la Figure 5.2 de trois *d*DBM représentant la même zone précisant alias (attention, la contrainte $x_j \geq x_i - 1$ est celle représentée par la *d*DBM en (c) et non pas $x_j \geq x_i + 1$ ; l'équivalence ne serait alors vraie que pour $\mathbb{K} = \mathbb{Z}$).

Donc, si c'est bien sur les *d*DBM que l'on va définir le domaine abstrait des zones précisant alias, il faut bien différencier ces deux notions. La définition du domaine abstrait des zones précisant alias, que l'on notera $\mathcal{D}^{zoal}$ (*zoal* pour *zo*ne *al*ias), passe par la définition du treillis des *d*DBM que l'on notera $\mathcal{D}^{ddbm}$.

### 5.3.1 Treillis des matrices de bornes et de non-égalités

Une première nécessité est d'ordonner les *d*DBM de telle manière que $\gamma^{ddbm}$ soit monotone : $m \sqsubseteq^{ddbm} m' \Rightarrow \gamma^{ddbm}(m) \subseteq \gamma^{ddbm}(m')$.

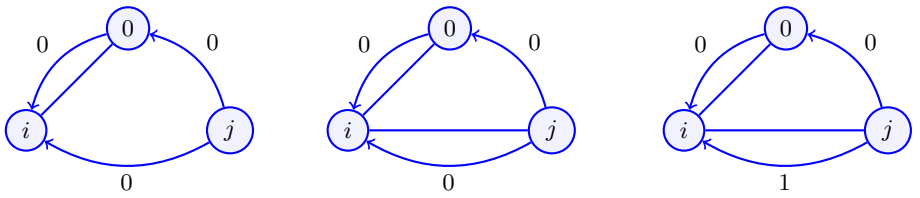
Pour ce qui est des DBM, l'extension matricielle de l'ordre sur $\overline{\mathbb{K}}$ conférait cette propriété (Déf. 3.5, p. 36). La conserver pour les *d*DBM nécessite de contraindre cet ordre par l'ordre inverse d'inclusion de leurs ensembles de non-égalités. Si l'on note $D(m)$ l'ensemble des non-égalités décrites par $m$, on définit :

$$m \sqsubseteq^{ddbm} m' \overset{\text{déf}}{\Leftrightarrow} (m^{\leq} \sqsubseteq^{dbm} m'^{\leq}) \wedge (D(m') \subseteq D(m)).$$

On exprime cet ordre de manière entièrement matricielle en utilisant l'implication pour tester l'inclusion :

$$m \sqsubseteq^{ddbm} m' \overset{\text{déf}}{\Leftrightarrow} \forall i, j \in [0, n], \quad m_{ij}^{\leq} \leq m_{ij}^{\prime\leq} \wedge m_{ij}^{\prime\neq} \Rightarrow m_{ij}^{\neq}.$$

La Figure 5.2 illustre cet ordre. La *d*DBM en (b) est incluse dans les deux autres : elle diffère de celle en (a) d'une contrainte de non-égalité et de celle en (c) d'une contrainte de zone. Ces deux dernières sont par contre incomparables.




(a) graphe mixte d'une *d*DBM représentant $x_j \geq 0 > x_i, x_j \geq x_i$

(b) graphe mixte d'une *d*DBM représentant $x_j \geq 0 > x_i, x_j > x_i$

(c) graphe mixte d'une *d*DBM représentant $x_j \geq 0 > x_i, x_j \geq x_i + 1$

Figure 5.2 – Trois *d*DBM, représentées par leurs graphes de potentiels et de non-égalités, ayant même concrétisation : $x_j \geq 0 > x_i$. Les *d*DBM présentées en (a) et (c) sont incomparables et plus grandes que celle présentée en (b) selon $\sqsubseteq^{ddbm}$.

Selon cet ordre, l'ensemble des *d*DBM a un plus grand élément naturel que l'on notera $\top^{ddbm}$, défini par $(\top^{ddbm})^{\leq} = \top^{dbm}$ et $\forall i, j,\ (\top^{ddbm})_{ij}^{\neq} = \mathit{faux}$, mais n'a pas de plus petit élément. On l'introduit par un élément noté $\bot^{ddbm}$. On définit le domaine des *d*DBM par

$$\mathcal{D}^{ddbm} = \bot^{ddbm} \cup \{(m^{\leq}, m^{\neq}) \mid m^{\leq} \in \mathcal{D}^{dbm}, m^{\neq} \in S(\mathbb{B})\},$$

où $S(\mathbb{B})$ représente l'ensemble des matrices symétriques booléennes. Il ne nous reste plus qu'à définir les opérateurs de borne inférieure et de borne supérieure, dont la définition va de soi, pour parachever la structure de treillis du domaine des matrices de bornes et de non-égalités :

Théorème 5.1 (treillis des matrices de bornes et de non-égalités). *Pour une dimension $n \geq 0$ donnée, l'ensemble $(\mathcal{D}^{ddbm}, \sqsubseteq^{ddbm}, \sqcup^{ddbm}, \sqcap^{ddbm}, \bot^{ddbm}, \top^{ddbm})$ dont les opérateurs sont définis comme suit, pour tout $m, m' \in \mathcal{D}^{ddbm}$,*

- $m \sqsubseteq^{ddbm} m' \Leftrightarrow \begin{cases} \textit{soit} & m = \bot^{ddbm} \\ \textit{soit} & m, m' \neq \bot^{ddbm}, \forall i, j \in [0, n], \quad m_{ij}^{\leq} \leq m_{ij}^{\prime\leq} \wedge m_{ij}^{\prime\neq} \Rightarrow m_{ij}^{\neq} \end{cases}$
- $m \sqcup^{ddbm} m' = \begin{cases} m & \textit{si } m' = \bot^{ddbm} \\ m' & \textit{si } m = \bot^{ddbm} \\ m'' \textit{ défini par } \begin{cases} m_{ij}^{\prime\prime\leq} = \max(m_{ij}^{\leq}, m_{ij}^{\prime\leq}) \\ m_{ij}^{\prime\prime\neq} = m_{ij}^{\neq} \wedge m_{ij}^{\prime\neq} \end{cases} & \textit{sinon} \end{cases}$

– $m \sqcap^{ddbm} m' = \begin{cases} \bot^{ddbm} & \text{si } m = \bot^{ddbm} \\ & \text{ou } m' = \bot^{ddbm} \\ m'' \text{ défini par } \begin{cases} m''^{\leq}_{ij} = \min(m^{\leq}_{ij}, m'^{\leq}_{ij}) \\ m''^{\neq}_{ij} = m^{\neq}_{ij} \vee m'^{\neq}_{ij} \end{cases} & \text{sinon} \end{cases}$

*est un treillis que l'on nomme le treillis des matrices de bornes et de non-égalités. Ce treillis est complet si* $\mathbb{K} = \mathbb{Z}$ *ou* $\mathbb{R}$.

*Démonstration.* On démontre que $\mathcal{D}^{ddbm}$ est un treillis.

- *La relation* $\sqsubseteq^{ddbm}$ *est un ordre partiel.* Par réflexivité, antisymétrie (équivalence) et transitivité des relations $\leq$ et $\Rightarrow$ ;
- *Il existe une borne supérieure pour chaque couple d'éléments de* $\mathcal{D}^{ddbm}$. On démontre que $\sqcup^{ddbm}$ est un opérateur de borne supérieure. Soit $m'' = m \sqcup^{ddbm} m'$ :
  - max est l'opérateur de borne supérieure sur $\mathbb{K}$ donc $m''^{\leq}_{ij} \geq m^{\leq}_{ij}, m'^{\leq}_{ij}$ et c'est le plus petit élément plus grand que $m^{\leq}_{ij}$ et $m'^{\leq}_{ij}$ ;
  - la règle d'élimination de la conjonction logique amène $m''^{\neq}_{ij} \Rightarrow m^{\neq}_{ij}, m'^{\neq}_{ij}$. D'autre part, si un élément implique $m^{\neq}_{ij}$ et $m'^{\neq}_{ij}$, il implique leur conjonction, *i.e.* $m''^{\neq}_{ij}$.
- *Il existe une borne inférieure pour chaque couple d'éléments de* $\mathcal{D}^{ddbm}$. On démontre de la même manière que $\sqcap^{ddbm}$ est un opérateur de borne inférieure. Soit $m'' = m \sqcap^{ddbm} m'$ :
  - min est l'opérateur de borne inférieure sur $\mathbb{K}$ donc $m''^{\leq}_{ij} \leq m^{\leq}_{ij}, m'^{\leq}_{ij}$ et c'est le plus grand élément plus petit que $m^{\leq}_{ij}$ et $m'^{\leq}_{ij}$ ;
  - la règle d'introduction de la disjonction logique amène $m^{\neq}_{ij}, m'^{\neq}_{ij} \Rightarrow m''^{\neq}_{ij}$. D'autre part, si un élément est impliqué par $m^{\neq}_{ij}$ et $m'^{\neq}_{ij}$, leur disjonction, *i.e.* $m''^{\neq}_{ij}$, l'implique.

Enfin, on se rappelle que les limites de suites rationnelles peuvent être irrationnelles. Le treillis n'est donc pas complet pour $\mathbb{K} = \mathbb{Q}$. *A contrario*, la propriété de borne supérieure et inférieure est connue pour $\mathbb{R}$, $\mathbb{Z}$ et $\mathbb{B}$. Pour $\mathbb{K} = \mathbb{Z}$ ou $\mathbb{R}$ le treillis est donc complet. □

### 5.3.2 Domaine abstrait et connexion de Galois

Comme attendu, on définit les éléments du domaine abstrait des zones précisant alias comme ceux du treillis des *d*DBM : $\mathcal{D}^{zoal} = \mathcal{D}^{ddbm}$. On confondra $\bot^{zoal}$ et $\bot^{ddbm}$.

On peut alors donner la fonction de concrétisation suivante, où $m \in \mathcal{D}^{zoal}$, par extension naturelle de $\gamma^{ddbm}$ :

$$\gamma^{zoal}(m) = \begin{cases} \emptyset & \text{si } m = \bot^{zoal} \\ \gamma^{ddbm}(m) & \text{sinon} \end{cases} .$$

En ordonnant les zones précisant alias selon $\sqsubseteq^{ddbm}$ – cet ordre sera étendu par la suite (Sec. 5.6.1) sans remettre en cause les résultats suivants, et sera alors noté $\sqsubseteq^{zoal}$ – on peut définir la fonction d'abstraction suivante, qui associe selon cet ordre la plus petite *d*DBM représentant un ensemble de valuations $x^\flat \in \mathcal{D}^\flat$. Cette propriété est une conséquence de la connexion de Galois, énoncée ensuite, que cette fonction d'abstraction forme avec $\gamma^{zoal}$.

$$\alpha^{zoal}(x^\flat) = \begin{cases} \bot^{zoal} & \text{si } x^\flat = \emptyset \\ \left( m^{\leq}_{ij} = \begin{cases} 0 & \text{si } i = j = 0 \\ \max\{\rho(x_j) \mid \rho \in x^\flat\} & \text{si } i = 0 \\ \max\{-\rho(x_i) \mid \rho \in x^\flat\} & \text{si } j = 0 \\ \max\{\rho(x_i) - \rho(x_j) \mid \rho \in x^\flat\} & \text{si } i, j \neq 0 \end{cases} \right. & \\ \left. m^{\neq}_{ij} = \begin{cases} \mathit{faux} & \text{si } i = j = 0 \\ \forall \rho \in x^\flat, \rho(x_j) \neq 0 & \text{si } i = 0 \\ \forall \rho \in x^\flat, \rho(x_i) \neq 0 & \text{si } j = 0 \\ \forall \rho \in x^\flat, \rho(x_i) \neq \rho(x_j) & \text{si } i, j \neq 0 \end{cases} \right) & \text{si } x^\flat \neq \emptyset \end{cases}$$

On notera que pour $\mathbb{K} = \mathbb{Q}$ cette fonction n'est pas totale. Il suffit de considérer un élément concret tel que, pour une certaine variable $x_i$, toute valuation $\rho$ qu'il contient est telle que $\rho(x_i)$ est irrationnel : max est alors indéfinie. On parlera dans ce cas d'une connexion de Galois partielle – une définition formelle de cette notion est donnée dans [Min04].

Théorème 5.2 (connexion de Galois). *Soit $m \in \mathcal{D}^{zoal}$ et $x^\flat, \in \mathcal{D}^\flat$. Pour $\mathbb{K} \in \{\mathbb{Z}, \mathbb{R}\}$, le couple $(\alpha^{zoal}, \gamma^{zoal})$ forme une connexion de Galois (Déf. 2.1, p. 14) :*

$$\alpha^{zoal}(x^\flat) \sqsubseteq^{ddbm} m \Leftrightarrow x^\flat \sqsubseteq^\flat \gamma^{zoal}(m).$$

*Pour $\mathbb{K} = \mathbb{Q}$ cette connexion est partielle.*

*Démonstration.* Soit $m' = \alpha^{zoal}(x^\flat)$ et $x^{\flat\prime} = \gamma^{zoal}(m)$ :

- $\Rightarrow$ Si $m' = \bot^{zoal}$ la preuve est directe. Sinon : $m^{\leq}_{ij} \geq m'^{\leq}_{ij} = \max\{\rho(x_i) - \rho(x_j) \mid \rho \in x^\flat\}$ et $m^{\neq}_{ij} \Rightarrow m'^{\neq}_{ij} = \forall \rho \in x^\flat, \rho(x_i) \neq \rho(x_j)$. Soit $\rho \in x^\flat$ on a $\rho(x_i) - \rho(x_j) \leq m'^{\leq}_{ij} \leq m^{\leq}_{ij}$ et $m^{\neq}_{ij} \Rightarrow m'^{\neq}_{ij} \Rightarrow \rho(x_i) \neq \rho(x_j)$ ce qui est la définition de $\rho \in \gamma^{zoal}(m)$.
- $\Leftarrow$ Si $x^{\flat\prime} = \emptyset$ la preuve est directe. Sinon, $\forall \rho \in x^\flat$ on a $\rho \in x^{\flat\prime}$ : $\rho(x_i) - \rho(x_j) \leq m^{\leq}_{ij}$ et $m^{\neq}_{ij} \Rightarrow \rho(x_i) \neq \rho(x_j)$. Par définition de $\alpha$ on obtient $\max\{\rho(x_i) - \rho(x_j) \mid \rho \in x^\flat\} = m'^{\leq}_{ij} \leq m^{\leq}_{ij}$ et $m^{\neq}_{ij} \Rightarrow m'^{\neq}_{ij} = \forall \rho \in x^\flat, \rho(x_i) \neq \rho(x_j)$ ce qui est la définition de $m' \sqsubseteq^{ddbm} m$.

□

On en déduit la monotonicité de $\gamma^{zoal}$ que l'on avançait à la section précédente.

Dans cette même section, il avait été remarqué qu'une zone précisant alias avait plusieurs *d*DBM la représentant, qui pouvaient même être incomparables selon $\sqsubseteq^{ddbm}$ (Fig. 5.2, p. 86). Cependant, on remarque que $\alpha^{zoal} \circ \gamma^{zoal}$ est toujours définie quel que soit l'ensemble $\mathbb{K}$, ce qui par rétraction ($(\alpha \circ \gamma)(x^\sharp) \sqsubseteq^\sharp x^\sharp$) de la connexion de Galois (Sec. 2.1.1.1) assure l'existence d'une représentation canonique aux zones précisant alias, minimale pour $\sqsubseteq^{ddbm}$. On notera $\overline{m}$ la forme normale de $m \in \mathcal{D}^{zoal}$ que l'on définit comme :

$$\overline{m} = \alpha^{zoal} \circ \gamma^{zoal}(m) = \inf_{\sqsubseteq^{ddbm}} \{m' \mid \gamma^{zoal}(m) = \gamma^{zoal}(m')\}.$$

Les deux sections suivantes (Sec. 5.4 et Sec. 5.5) portent sur le calcul de cette forme normale, la première lorsque $\mathbb{K}$ est dense ($\mathbb{Q}$, $\mathbb{R}$), la seconde lorsque $\mathbb{K}$ est discret ($\mathbb{Z}$). En effet, la classe de complexité de ce problème diffère selon ces deux cas : classe P dans le premier cas, classe NP dans le second.

## 5.4 Représentation canonique : cas dense

Dans cette section, on définit un algorithme calculant la forme normale d'une *d*DBM lorsque $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$. Sa complexité en temps est de $\mathcal{O}(n^4)$ et en espace de $\mathcal{O}(n^3)$.

On commence par étudier le cas particulier de la satisfaisabilité de l'ensemble de contraintes de zone et alias représenté par une *d*DBM, c'est-à-dire de décider si sa forme normale est ou non $\bot^{zoal}$.

### 5.4.1 Test de satisfaisabilité

Bien entendu, le seul ensemble des contraintes de zones représenté par une *d*DBM peut être insatisfaisable, ce que nous savons décider en $\mathcal{O}(n^3)$ en vérifiant que le graphe de potentiels de $m^{\leq}$ ne contient pas de cycle de poids strictement négatif (Thm. 3.1, p. 34).

Quant aux contraintes de non-égalité, une fois écarté le cas trivial où l'une d'entre elles déclare une variable non égale à elle-même, les travaux de Lassez et McAloon [LM92] nous informent de leur indépendance vis-à-vis des systèmes linéaires d'inégalités (Thm. 3.3, p. 41) : si parmi un ensemble fini de contraintes de non-égalité aucune n'élimine à elle seule les solutions d'un système linéaire d'inégalités, alors la conjonction de cet ensemble et de ce système est satisfaisable – ou en d'autres termes, aucune combinaison finie de ces contraintes de non-égalité ne peut mener à son insatisfaisabilité. Ce résultat s'applique aux contraintes d'inégalités et de non-égalités que l'on considère, les contraintes de zone et alias, puisqu'elles sont incluses dans celles des systèmes linéaires généralisés (Déf. 3.6, p. 40) sur lesquels porte le théorème. Il fournit un test de satisfaisabilité simple où il suffit de vérifier que toute contrainte de non-égalité entre deux variables ne contredit pas un ensemble de contraintes de zones portant égalité entre ces deux mêmes variables.

Ainsi, l'insatisfaisabilité due aux contraintes de non-égalité peut aussi s'énoncer sur le graphe de potentiels et de non-égalité. On l'énonce, pour le cas trivial, par l'existence de boucles dans le graphe de non-égalités, pour les autres cas, par l'existence de deux sommets adjacents dans le graphe de non-égalités et fortement connectés par deux chemins de poids nul dans le graphe de potentiels. Le Théorème 5.3 résume ces différents cas.

Théorème 5.3 (Satisfaisabilité d'une conjonction de contraintes de zones et alias dans $\mathbb{Q}$ ou $\mathbb{R}$).

$$\gamma^{zoal}(m) = \emptyset \Leftrightarrow \begin{cases} \mathcal{G}(m) \textit{ contient un cycle orienté de poids strictement négatif} \\ \textit{ou} \\ \mathcal{G}(m) \textit{ contient une boucle non orientée} \\ \textit{ou} \\ \mathcal{G}(m) \textit{ contient une arête non orientée dont les sommets sont} \\ \qquad \textit{fortement connectés par deux chemins de poids nul} \end{cases}$$

*Démonstration.* Conséquence du Théorème 3.3 ou encore du Lemme 5.2 (p. 93). □

On donne à la Figure 5.3 un algorithme testant la satisfaisabilité d'un ensemble de contraintes de zone et alias représenté par une *d*DBM. Pour la première alternative pouvant mener à l'insatisfaisabilité de l'ensemble, il reprend le test de satisfaisabilité des conjonctions de contraintes de zones présenté à la Figure 3.5 (p. 36, lignes **1**–**3**). Pour la seconde, il teste simplement si aucune des valeurs sur la diagonale de $m^{\neq}$ est

*vrai* (ligne **3**). Enfin pour la troisième, il tire parti du calcul de la fermeture du graphe de potentiels selon les plus courts chemins effectué précédemment : l'égalité entre deux variables $x_i$ et $x_j$ se teste alors directement par $m^{\leq}_{ij} = m^{\leq}_{ji} = 0$ (ligne **5**). Ce dernier test n'est effectué que sur les valeurs de la partie inférieure droite (lignes **4**–**5**) de la matrice de non-égalités, celle-ci étant par définition symétrique.

```
1 m ← (FloydWarshall(m^≤), m^≠) ;
2 for i ← 0 to n do
3 |   if m^≤_ii < 0 ∨ m^≠_ii then return ⊥^zoal ;
4 |   for j ← i+1 to n do
5 |   |   if m^≠_ij ∧ m^≤_ij = 0 ∧ m^≤_ji = 0 then return ⊥^zoal
```

Figure 5.3 – Test de satisfaisabilité d'une *d*DBM $m$ dans le cas dense

On obtient un test de satisfaisabilité simple, dont la complexité est dominée par celle de l'algorithme des plus courts chemins, en l'occurrence $\mathcal{O}(n^3)$.

En général, les complexités des algorithmes calculant la forme normale et testant la satisfaisabilité (d'ensembles de contraintes, par exemple des domaines numériques faiblement relationnels, Sec. 3.1) sont équivalentes et ainsi utilise-t-on en pratique le premier en place du second. Il s'avère que dans le cas présent la complexité du calcul de la forme normale est en $\mathcal{O}(n^4)$, comme nous allons le voir à la section suivante (Sec. 5.4.2), encourageant une utilisation courante du test de satisfaisabilité seul.

### 5.4.2 Calcul de la forme normale

Le problème du calcul de la forme normale de $m \in \mathcal{D}^{ddbm}$ peut être rapproché du problème de déterminer toutes les contraintes de zones et alias pouvant être déduites d'un ensemble de contraintes de la même forme.

Si en général pour des contraintes d'inégalités linéaires ce dernier problème peut se résoudre avec le lemme de Farkas [Far02], il n'y a pas à notre connaissance de résultats en présence de contraintes de non-égalité sur lesquels on puisse s'appuyer. Cependant, puisque pour les ensembles de contraintes de zones ce problème est résolu, on s'intéresse simplement à traduire les contraintes de non-égalité selon ces mêmes contraintes :

$$x_i \neq x_j \Leftrightarrow (x_i - x_j < 0 \vee x_j - x_i < 0).$$

Le problème porte alors sur une disjonction d'ensembles de contraintes de zones strictes et non strictes. En effet, la traduction de la contrainte de non-égalité $x_i \neq x_j$ dans une DBM $m^0$, représentant des contraintes de zones strictes et non strictes (Sec. 3.1.1.2), mène à deux DBM $m^1$ et $m^2$ égales à $m^0$, excepté pour $m^1_{ij} = \min(m^0_{ij}, (0, <))$ et $m^2_{ji} = \min(m^0_{ji}, (0, <))$ [2]. Ainsi, sur l'ensemble de contraintes de non-égalité représenté par $m^{\neq}$, on considère l'arbre de leurs traductions successives à partir de $m^{\leq}$ (modifiée pour pouvoir représenter des contraintes strictes et non strictes) ayant pour feuilles l'ensemble $m^1, \ldots, m^d$ de DBM, représenté à la Figure 5.4 :

L'ensemble de contraintes représenté par $m$ est équivalent par construction à la disjonction des ensembles de contraintes que représentent $m^1, \ldots, m^d$. Notre intuition

2. On rappelle les notations sur les bornes de ces DBM : $b((0, <)) = 0$ et $s((0, <)) = <$.

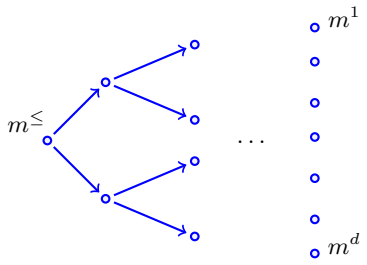



Figure 5.4 – Décomposition des non-égalités de $m$

est que la forme normale de $m$ est la plus petite $d$DBM rassemblant l'ensemble des contraintes représentées par les formes normales des DBM $m^1, \ldots, m^d$, notées $\overline{m^1}, \ldots, \overline{m^d}$. La définition de cette $d$DBM, que l'on note $m^*$, s'apparente logiquement à celle de l'opérateur d'union sur les $d$DBM (Thm. 5.1, p. 86), si ce n'est que les non-égalités sont celles implicites des DBM ($x_i - x_j \leq (0, <) \Rightarrow x_i \neq x_j$). Soit pour $K = \{k \mid \overline{m^k} \neq \bot^{zone}\}$ l'ensemble des DBM satisfaisables, elle se définit comme suit :

$$\begin{cases} \bot^{zoal} & \text{si } K = \emptyset \\ \left( \begin{array}{l} {m^*}_{ij}^{\leq} = \max_{k \in K} \ b(\overline{m^k_{ij}}) \\ {m^*}_{ij}^{\neq} = \bigwedge_{k \in K} \ (\overline{m^k_{ij}} \leq (0, <) \vee \overline{m^k_{ji}} \leq (0, <)) \end{array} \right) & \text{sinon.} \end{cases}$$

On ne cherche pas à montrer à partir de cette définition que l'ensemble de contraintes représenté par $m^*$ est équivalent à celui représenté par $m$. En fait, ce résultat sera obtenu aisément en définissant $m^*$ directement à partir de $m$. Surtout, cette nouvelle définition (Déf. 5.3, p. 97) ne reposera pas sur la construction précédente, coûteuse puisque exponentielle dans le nombre de non-égalités. Cette définition s'appuie sur deux lemmes que nous allons énoncer et démontrer : le premier (Lem. 5.1, p. 92) nous assure que $\overline{m^{\leq}}$ et ${m^*}^{\leq}$ ont exactement les mêmes bornes et le second (Lem. 5.3, p. 94) que toutes les nouvelles non-égalités (vis-à-vis de $m^{\neq}$) que représente ${m^*}^{\neq}$ se déduisent de la conjonction d'une *seule* contrainte de non-égalité et d'un ensemble de contraintes de zones (non strictes donc) :

- $(x_i - x_j \leq c) \wedge (c < 0) \Rightarrow x_i \neq x_j$ ;
- $(x_i = x_{i_0} + c) \wedge (x_j = x_{j_0} + c) \wedge x_{i_0} \neq x_{j_0} \Rightarrow x_i \neq x_j$ ;
- $(x_j = x_{j_0}) \wedge (x_{i_0} \neq x_{j_0}) \Rightarrow x_{i_0} \neq x_j$ ;
- $(x_i \leq x_{i_0} + c \leq x_j) \wedge (x_i \leq x_{j_0} + c \leq x_j) \wedge (x_{i_0} \neq x_{j_0}) \Rightarrow x_i \neq x_j$ ;
- $(x_j \leq x_{j_0} \leq x_{i_0}) \wedge (x_{i_0} \neq x_{j_0}) \Rightarrow x_{i_0} \neq x_j$.

On donne dès à présent ces résultats car le développement du raisonnement et des preuves nécessaires est long. Une fois fait, il ne restera plus qu'à montrer que $m^*$ est la plus petite $d$DBM représentant le même ensemble de contraintes que $m$ (Thm. 5.4, p. 98) : on pourra alors définir un algorithme efficace implantant cet opérateur de normalisation (Sec. 5.4.2.2).

#### 5.4.2.1 Opérateur de normalisation

On considère à présent l'arbre de décomposition de $m^{\leq}$ où toute DBM est mise en forme normale avant de lui appliquer la traduction d'une non-égalité représentée par $m^{\neq}$.

Cet arbre de décomposition n'est plus forcément complet : par exemple la traduction d'une non-égalité $x_i \neq x_j$ dans une DBM où elle est implicite ($2 \leq x_i - x_j \leq 4$) mène pour une branche à la même DBM, pour l'autre à une contradiction ($2 \leq x_i - x_j < 0$) et donc, par normalisation, à $\bot^{zone}$, qui n'est plus sujette à être décomposée.

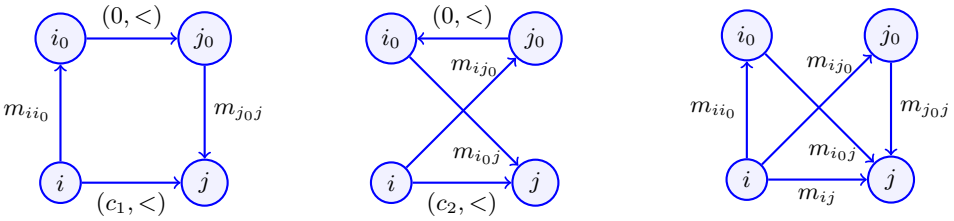
On s'intéresse en premier lieu à déterminer $m^{*\leq}$. Pour cela, on montre qu'à chaque traduction d'une non-égalité dans une DBM, toutes les valeurs (*i.e.* $b(\ldots)$) des bornes de cette DBM sont maintenues dans l'une ou l'autre, si elles existent, des DBM résultantes.

Lemme 5.1 (sauvegarde de la valeur des bornes). *Soit $m$ une DBM close non vide et $x_{i_0} - x_{j_0} \neq 0$ une contrainte de non-égalité. Soit $m^1$ la DBM égale à $m$ excepté pour $m^1_{i_0 j_0} = \min(m_{i_0 j_0}, (0, <))$ et $m^2$ la DBM égale à $m$ excepté pour $m^2_{j_0 i_0} = \min(m_{j_0 i_0}, (0, <))$. Alors pour tout $i, j \in [0, n]$,*

$$\begin{cases} b(\overline{m^1}_{ij}) = b(m_{ij}) \vee b(\overline{m^2}_{ij}) = b(m_{ij}) & \text{si } \overline{m^1}, \overline{m^2} \neq \bot^{zone} \\ b(\overline{m^2}_{ij}) = b(m_{ij}) & \text{si } \overline{m^1} = \bot^{zone}, \overline{m^2} \neq \bot^{zone} \\ b(\overline{m^1}_{ij}) = b(m_{ij}) & \text{si } \overline{m^2} = \bot^{zone}, \overline{m^1} \neq \bot^{zone} \end{cases}$$

*Démonstration.* Si $m_{i_0 j_0} \leq (0, <)$ alors $m^1 = m$ et le lemme est vérifié. De même si $m_{j_0 i_0} \leq (0, <)$ puisque $m^2 = m$. Dans les autres cas,

- Soit $\overline{m^1}, \overline{m^2} \neq \bot^{zone}$, supposons la négation de la propriété : $b(\overline{m^1}_{ij}) = c_1 \wedge b(\overline{m^2}_{ij}) = c_2$ où $c_1, c_2 < c = b(m_{ij})$.




  Puisque la borne entre $i$ et $j$ a été réduite dans $\mathcal{G}^1$ et $\mathcal{G}^2$, les graphes de potentiels de $m^1$ et $m^2$, c'est qu'il existe un chemin entre $i$ et $j$ passant respectivement par l'arête $i_0 \to j_0$ et l'arête $j_0 \to i_0$. En l'occurrence, les valeurs des bornes de ces chemins sont les mêmes dans $\mathcal{G}$ (*e.g.* $m^1_{i_0 j_0} = m_{i_0 j_0}$ car aucun chemin simple de $i$ à $i_0$ ne peut passer par l'arête $i_0 \to j_0$), on a donc : $m_{ii_0} + (0, <) + m_{j_0 j} = (c_1, <)$ et $m_{ij_0} + (0, <) + m_{i_0 j} = (c_2, <)$, donc $b(m_{ii_0}) + b(m_{j_0 j}) = c_1$ et $b(m_{ij_0}) + b(m_{i_0 j}) = c_2$. $\mathcal{G}$ étant clos pour les plus courts chemins, $m_{ij_0} + m_{j_0 j} \geq (c, \prec)$ et $m_{ii_0} + m_{i_0 j} \geq (c, \prec)$, donc $b(m_{ij_0}) + b(m_{j_0 j}) \geq c$ et $b(m_{ii_0}) + b(m_{i_0 j}) \geq c$. Comme $c_1, c_2 < c$ on a $b(m_{ii_0}) + b(m_{j_0 j}) < b(m_{ij_0}) + b(m_{j_0 j})$ et $b(m_{ij_0}) + b(m_{i_0 j}) < b(m_{ii_0}) + b(m_{i_0 j})$. Par simplification on a $b(m_{ii_0}) < b(m_{ij_0})$ et $b(m_{ij_0}) < b(m_{ii_0})$ ce qui est contradictoire.
- Soit $\overline{m^2} \neq \bot^{zone}$ et $\overline{m^1} = \bot^{zone}$. Puisque $m$ est close et non vide, l'insatisfaisabilité de $m^1$ est due à la modification de $m_{i_0 j_0}$ qui a crée un cycle de poids négatif dans $\mathcal{G}^1$, donc il existe un chemin de $j_0$ à $i_0$ de poids au plus nul dans $\mathcal{G}^1$, et en l'occurrence également dans $\mathcal{G}$ : $m_{j_0 i_0} \leq (0, \leq)$. Comme on a supposé que $m_{j_0 i_0} \geq (0, \leq)$, on a $b(m^2_{j_0 i_0}) = 0 = b(m_{j_0 i_0})$, donc aucune valeur de borne n'est modifiée par la normalisation de $m^2$.
- Le raisonnement est identique si $\overline{m^1} \neq \bot^{zone}$ et $\overline{m^2} = \bot^{zone}$.

□

Ce résultat ne permet pas de conclure directement que les bornes de $m^{*\leq}$ sont celles de la DBM initialement décomposée, $\overline{m^{\leq}}$. Il faut pour cela remarquer que si la traduction d'une non-égalité dans une DBM mène à l'insatisfaisabilité des deux branches – et donc que les valeurs des bornes de cette DBM sont perdues – c'est qu'il en sera de même dans toute autre DBM issue de la décomposition de $\overline{m^{\leq}}$. C'est une conséquence directe de l'indépendance des non-égalités (Thm. 3.3, p. 41) : l'insatisfaisabilité ne provient pas de l'addition de différentes non-égalités et donc l'insatisfaisabilité des deux DBM résultantes de la traduction d'une non-égalité ne dépend pas des non-égalités précédemment traduites dans celle-ci. Le lemme suivant exprime ce dernier point en énonçant que les contraintes de zones participant à l'insatisfaisabilité sont non strictes.

Lemme 5.2 (insatisfaisabilité). *Soit $m$ une DBM close non vide et $x_{i_0} - x_{j_0} \neq 0$ une contrainte d'alias. Soit $m^1$ la DBM égale à $m$ excepté pour $m^1_{i_0j_0} = \min(m_{i_0j_0}, (0, <))$ et $m^2$ la DBM égale à $m$ excepté pour $m^2_{j_0i_0} = \min(m_{j_0i_0}, (0, <))$.*

*Alors, si $\overline{m^1} = \overline{m^2} = \bot^{zone}$,*

$$m_{i_0j_0} = m_{j_0i_0} = (0, \leq).$$

*Démonstration.* $m^1$ et $m^2$ étant différentes de $m$ on a : $m_{i_0j_0} \geq (0, \leq)$ et $m_{j_0i_0} \geq (0, \leq)$ et $m^1_{i_0j_0} = (0, <)$ et $m^2_{j_0i_0} = (0, <)$. De plus $m^1$ et $m^2$ étant insatisfaisables on a : $m_{j_0i_0} \leq (0, \leq)$ et $m_{i_0j_0} \leq (0, \leq)$. □

Ainsi, mis à part le cas où toutes les DBM résultantes de la décomposition sont vides, cas que l'on sait déterminer grâce au test de satisfaisabilité (on remarquera que son implantation, présentée à la Figure 5.3 (p. 90), utilise justement pour test $m^{\neq}_{ji} \wedge m^{\leq}_{ji} = m^{\leq}_{ij} = 0$), on est assuré qu'à chaque traduction d'une non-égalité l'une des deux DBM résultantes est non vide. Donc, pour tout $i, j \in [0, n]$ il existe $k' \in K$ tel que $b(\overline{m^{k'}_{ij}}) = \overline{m^{\leq}_{ij}}$ d'après le Lemme sur la sauvegarde de la valeur des bornes (Lem. 5.1) et donc $m^{*\leq}_{ij} = \max_{k \in K} b(\overline{m^k_{ij}}) \geq \overline{m^{\leq}_{ij}}$. Par ailleurs $\forall k \in K, b(\overline{m^k_{ij}}) \leq \overline{m^{\leq}_{ij}}$ par construction, et donc $m^{*\leq}_{ij} = \overline{m^{\leq}_{ij}}$.

On obtient alors le résultat suivant :

$$m^{*\leq} = \overline{m^{\leq}}. \tag{5.1}$$

On s'intéresse maintenant à déterminer $m^{*\neq}$. Pour une DBM $m$ donnée on note le fait qu'elle représente implicitement une non-égalité entre $x_i$ et $x_j$, *i.e.* $(m_{ij} \leq (0, <)) \vee (m_{ji} \leq (0, <))$, par le prédicat $P(i, j, m)$.

On remarque deux points. Premièrement, si le prédicat $P$ est vérifié pour une DBM $m$ de la décomposition, alors il l'est aussi par construction pour toute DBM issue de la décomposition de $m$. Deuxièmement, toute non-égalité représentée par $m^{\neq}$ l'est également par $m^{*\neq}$ : en effet pour chacune d'entre elles, le prédicat $P$ est par définition vérifié dans les deux DBM résultantes de sa traduction.

Il reste à identifier ces nouvelles non-égalités que représente $m^{*\neq}$. Le cas simple c'est celui où la non-égalité en question n'est pas représentée dans $m^{\neq}$ mais l'est implicitement dans $\overline{m^{\leq}}$. On a alors :

$$P(i, j, \overline{m^{\leq}}) \Rightarrow m^{*\neq}.$$

Le cas intéressant est bien sûr celui où $\neg P(i,j,\overline{m^{\leq}})$. Puisque la non-égalité apparaît dans $m^{*\neq}$, on a par définition $\forall k \in K, P(i,j,\overline{m^k})$. Alors nécessairement il existe dans l'arbre de décomposition une DBM $m$ où $\neg P(i,j,m)$ et pour laquelle la traduction de la non-égalité s'est soldée par deux DBM $m^1$ et $m^2$ où, à moins qu'elles soient vides, la non-égalité entre $x_i$ et $x_j$ est apparue implicitement, c'est-à-dire $P(i,j,m^1)$ et $P(i,j,m^2)$.

Nous allons donc nous intéresser à ces traductions précises dans l'arbre de décomposition, identifier les contraintes de zones concernées et montrer que leurs résultats ne dépendent d'aucune autre non-égalité précédemment traduite.

Lemme 5.3 (non-combinaison des non-égalités). *Soit $m$ une DBM close non vide et $x_{i_0} - x_{j_0} \neq 0$ une contrainte de non-égalité. Soit $m^1$ la DBM égale à $m$ excepté pour $m^1_{i_0 j_0} = \min(m_{i_0 j_0}, (0,<))$ et $m^2$ la DBM égale à $m$ excepté pour $m^2_{j_0 i_0} = \min(m_{j_0 i_0}, (0,<))$.*

*Alors, pour tout $i,j \in [0,n]$ tels que $\neg P(i,j,m)$, $\overline{m^1}$ ou $\overline{m^2}$ est non vide, $P(i,j,\overline{m^1})$ si $\overline{m^1} \neq \bot^{zone}$ et $P(i,j,\overline{m^2})$ si $\overline{m^2} \neq \bot^{zone}$ :*

- *si $i,j \neq i_0, j_0$, alors soit :*

$$\begin{cases} b(m_{ii_0}) = -b(m_{i_0 i}) = b(m_{jj_0}) = -b(m_{j_0 j}) \\ s(m_{ii_0}) = s(m_{i_0 i}) = s(m_{jj_0}) = s(m_{j_0 j}) = \leq \end{cases},$$

*soit :*

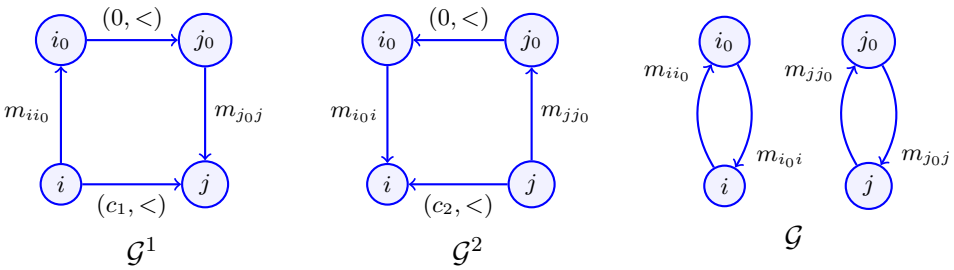
$$\begin{cases} b(m_{ii_0}) = -b(m_{i_0 j}) = b(m_{ij_0}) = -b(m_{j_0 j}) \\ s(m_{ii_0}) = s(m_{i_0 j}) = s(m_{ij_0}) = s(m_{j_0 j}) = \leq \end{cases}.$$

- *Si $i = i_0 \wedge i,j \neq j_0$ alors soit :*

$$\begin{cases} b(m_{jj_0}) = b(m_{j_0 j}) = 0 \\ s(m_{jj_0}) = s(m_{j_0 j}) = \leq \end{cases}, \text{soit} \begin{cases} b(m_{j_0 i}) = b(m_{jj_0}) = 0 \\ s(m_{j_0 i}) = s(m_{jj_0}) = \leq \end{cases}.$$

*Démonstration.* Dans tous les cas, une modification entre $m^1$ et $m$ et entre $m^2$ et $m$ est survenue : $m_{i_0 j_0} \geq (0,\leq)$ et $m_{j_0 i_0} \geq (0,\leq)$ et $m^1_{i_0 j_0} = (0,<)$ et $m^2_{j_0 i_0} = (0,<)$. On différencie le cas où $\overline{m^1}$ et $\overline{m^2}$ sont non vides du cas où l'une de ces deux DBM est vide.

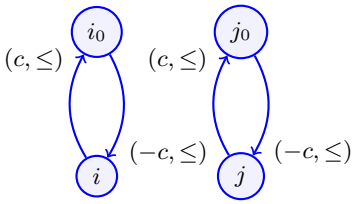
- Soit $\overline{m^1}, \overline{m^2} \neq \bot^{zone}$.
  - On suppose, sans perte de généralité, que $P(i,j,\overline{m^1})$ est vérifié car $\overline{m^1}_{ij} \leq (0,<)$. Deux cas se présentent pour vérifier $P(i,j,\overline{m^2})$ :
    - Soit $\overline{m^2}_{ji} \leq (0,<)$.
      - On s'intéresse au cas plus général où $i,j,i_0,j_0$ sont tous distincts (cas (1a)).




Puisque les bornes entre $i$ et $j$ ont été réduites, on a dans $\mathcal{G}^1$, $m_{ii_0} + (0,<) + m_{j_0 j} = (c_1,<)$ où $c_1 \leq 0$ et dans $\mathcal{G}^2$, $m_{jj_0} + (0,<) + m_{i_0 i} = (c_2,<)$ où $c_2 \leq 0$. Par conséquent $b(m_{ii_0}) + b(m_{j_0 j}) \leq 0$ et $b(m_{jj_0}) + b(m_{i_0 i}) \leq 0$. Comme

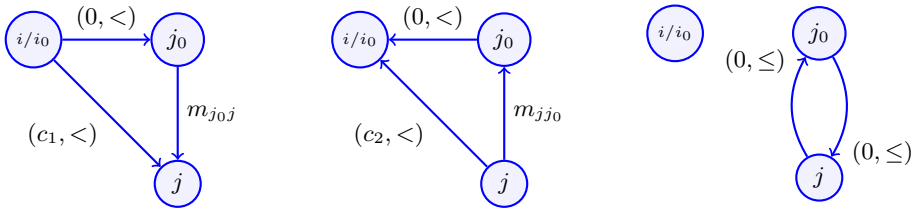
$m \neq \bot^{zone}$ on a $m_{ii_0} + m_{i_0 i} \geq (0, \leq)$ (inéquation (1)) et $m_{jj_0} + m_{j_0 j} \geq (0, \leq)$ (inéquation (2)), donc $b(m_{ii_0}) + b(m_{i_0 i}) \geq 0$ et $b(m_{jj_0}) + b(m_{j_0 j}) \geq 0$.
Le système composé des quatre équations portant sur les valeurs des bornes $m_{ii_0}, m_{i_0 i}, m_{jj_0}, m_{j_0 j}$ a pour solutions :

$$b(m_{ii_0}) = -b(m_{i_0 i}) = b(m_{jj_0}) = -b(m_{j_0 j})$$

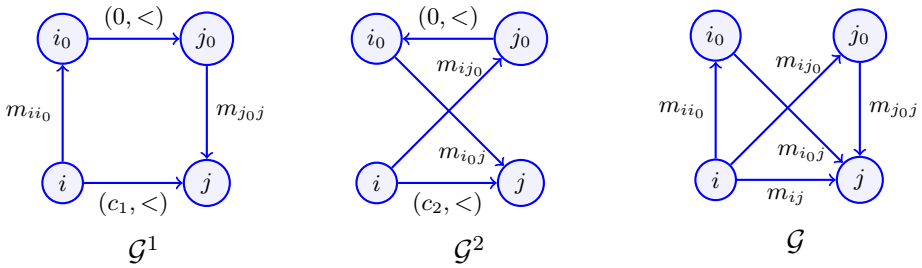



On en tire que $b(m_{ii_0}) + b(m_{i_0 i}) = 0$ et donc, du fait de l'inéquation (1), que $s(m_{ii_0}) = s(m_{i_0 i}) = \leq$. De même, on en tire que $b(m_{jj_0}) + b(m_{j_0 j}) = 0$ et donc, du fait de l'inéquation (2), que $s(m_{jj_0}) = s(m_{j_0 j}) = \leq$.
Le dessin ci-dessus récapitule la situation mise en évidence dans $\mathcal{G}$ où $(c, \leq) = m_{ii_0}$.

– Reste le cas où $i$ et $i_0$ sont confondus (cas (1b)). Par le même raisonnement que celui déployé précédemment on obtient $b(m_{j_0 j}) = b(m_{jj_0}) = 0$ et $s(m_{j_0 j}) = s(m_{jj_0}) = \leq$.

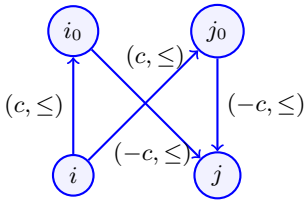



– Soit $m^2_{ij} \leq (0, <)$.
  – Là encore on s'intéresse en premier lieu au cas général où $i, j, i_0, j_0$ sont tous distincts (cas (2a)).




Puisque les bornes entre $i$ et $j$ ont été réduites, on a dans $\mathcal{G}^1$, $m_{ii_0} + (0, <) + m_{j_0 j} = (c_1, <)$ où $c_1 \leq 0$ et dans $\mathcal{G}^2$, $m_{ij_0} + (0, <) + m_{i_0 j} = (c_2, <)$ où $c_2 \leq 0$, donc $b(m_{ii_0}) + b(m_{j_0 j}) \leq 0$ et $b(m_{ij_0}) + b(m_{i_0 j}) \leq 0$. Le fait que $P(i, j, m)$ n'est pas vérifié implique $m_{ij} \geq (0, \leq)$, et comme $m$ est close, entraîne que $m_{ii_0} + m_{i_0 j} \geq (0, \leq)$ (inéquation (1)) et $m_{ij_0} + m_{j_0 j} \geq (0, \leq)$ (inéquation (2)). Par conséquent $b(m_{ii_0}) + b(m_{i_0 j}) \geq 0$ et $b(m_{ij_0}) + b(m_{j_0 j}) \geq 0$.
Le système composé des quatre équations portant sur les valeurs des bornes $m_{ii_0}, m_{ij_0}, m_{i_0 j}, m_{j_0 j}$ a pour solutions :

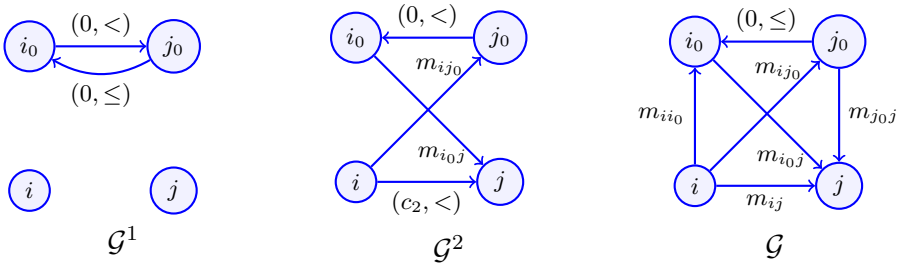
$$b(m_{ii_0}) = -b(m_{i_0j}) = b(m_{ij_0}) = -b(m_{j_0j})$$




On en tire que $b(m_{ii_0}) + b(m_{i_0j}) = 0$ et donc, du fait de l'inéquation (1), que $s(m_{ii_0}) = s(m_{i_0j}) = \leq$. De même, on en tire que $b(m_{ij_0}) + b(m_{j_0j}) = 0$ et donc, du fait de l'inéquation (2), que $s(m_{ij_0}) = s(m_{j_0j}) = \leq$.
Le dessin ci-dessus récapitule la situation mise en évidence dans $\mathcal{G}$ où $(c, \leq) = m_{ii_0}$.

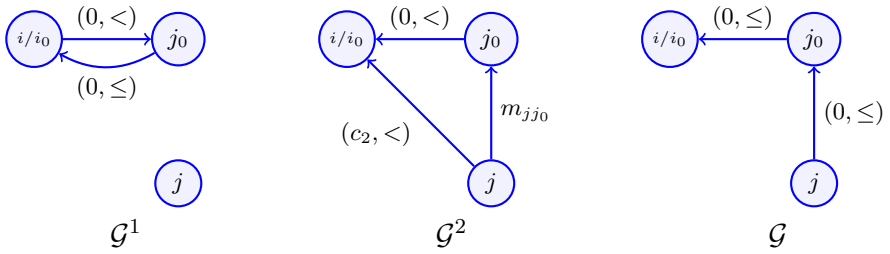
- Le dernier cas où $i$ et $i_0$ sont confondus n'existe pas ! En effet dans $\mathcal{G}^2$ le borne $m^2_{i_0j}$ ne peut-être différente de $m_{i_0j}$ par le simple changement de la borne $m^2_{j_0i_0}$, ce qui est en contradiction avec l'hypothèse que $\overline{m^2}_{i_0j} \leq (0, <)$ alors que $m_{i_0j} \geq (0, \leq)$.

– Soit $\overline{m^2} \neq \bot^{zone}, \overline{m^1} = \bot^{zone}$. Comme $\overline{m^1}$ est vide et que l'on a supposé que $m_{j_0i_0} \geq (0, \leq)$ on a $m_{j_0i_0} = (0, \leq)$. On procède selon les cas pour lesquels $P(i, j, \overline{m^2})$ est vérifié
  – Soit $\overline{m^2}_{ij} \leq (0, <)$.
    – Soit $i, j, i_0, j_0$ distincts.




Puisque la borne entre $i$ et $j$ a été réduite dans $\mathcal{G}^2$ on a $m_{ij_0} + (0, <) + m_{i_0j} = (c_2, <)$ où $c_2 \leq 0$, donc $b(m_{ij_0}) + b(m_{i_0j}) \leq 0$. Comme $m$ est close et que $P(i, j, m)$ n'est pas vérifié, on a $m_{ij_0} + (0, \leq) + m_{i_0j} \geq (0, \leq)$ (inéquation (1)) et donc $b(m_{ij_0}) + b(m_{i_0j}) = 0$ (égalité (1)). Du fait de l'inéquation (1), on a $s(m_{ij_0}) = s(m_{i_0j}) = \leq$.
Pour terminer la preuve il suffit de tirer parti du fait que $m$ est close : on a $b(m_{ii_0}) \leq b(m_{ij_0})$ et $b(m_{j_0j}) \leq b(m_{i_0j})$, et comme $\neg P(i, j, m)$, on a $b(m_{ii_0}) \geq -b(m_{i_0j})$ et donc par l'égalité (1), $b(m_{ii_0}) = b(m_{ij_0})$. De même on obtient $b(m_{j_0j}) = b(m_{i_0j})$ et $s(m_{ii_0}) = s(m_{j_0j}) = \leq$.
On remarquera que le résultat obtenu n'est qu'un cas particulier de ceux où les deux DBM issues de la traduction de la non-égalité entre $x_{i_0}$ et $x_{j_0}$ ne sont pas vides. Vis-à-vis du cas (2a) il diffère juste de la borne $m_{j_0i_0}$ fixée à $(0, \leq)$, et vis-à-vis du cas (1a) l'existence de cette dernière borne ramène ce cas au cas (2a).
Enfin, on notera d'une part que ce résultat serait similaire si on avait supposé que $P(i, j, \overline{m^2})$ était vérifié car $\overline{m^2}_{ji} \leq (0, <)$ et d'autre part que le cas où $i$ et $i_0$ sont confondus n'est pas possible.

– Soit donc le dernier cas possible : $\overline{m^2}_{ji} \leq (0, <)$ et $i$ et $i_0$ sont confondus (cas (2b)).




En appliquant le raisonnement précédent sur $\mathcal{G}^2$ on obtient aisément que $b(m_{jj_0}) = 0$ et que $s(m_{jj_0}) = \leq$.

□

Les conséquences du Lemme 5.3 sont doubles : non seulement il identifie les contraintes de zones présentes dans les DBM apparaissant nécessairement dans une décomposition menant à la déduction d'une nouvelle non-égalité, mais aussi il assure que ces contraintes existent dans $\overline{m^{\leq}}$, c'est-à-dire que leur présence dans cette DBM est une condition suffisante pour déduire cette nouvelle non-égalité.

*A posteriori*, on peut se dire que le fait qu'une non-égalité n'est pas induite par la combinaison de plusieurs autres non-égalités (associées à un ensemble de contraintes de zones) était prévisible de par leur propriété d'indépendance (Thm. 3.3, p. 41).

À des fins de clarté, on donne de nouveau les différentes situations énoncées par le lemme sous forme de contraintes, où $c \in \mathbb{K}$ :
– cas (0) : $(x_i - x_j \leq c) \wedge (c < 0) \Rightarrow x_i \neq x_j$ ;
– cas (1a) : $(x_i = x_{i_0} + c) \wedge (x_j = x_{j_0} + c) \wedge x_{i_0} \neq x_{j_0} \Rightarrow x_i \neq x_j$ ;
– cas (1b) : $(x_j = x_{j_0}) \wedge (x_{i_0} \neq x_{j_0}) \Rightarrow x_{i_0} \neq x_j$ ;
– cas (2a) : $(x_i \leq x_{i_0} + c \leq x_j) \wedge (x_i \leq x_{j_0} + c \leq x_j) \wedge (x_{i_0} \neq x_{j_0}) \Rightarrow x_i \neq x_j$ ;
– cas (2b) : $(x_j \leq x_{j_0} \leq x_{i_0}) \wedge (x_{i_0} \neq x_{j_0}) \Rightarrow x_{i_0} \neq x_j$.

On remarquera que les cas (1b), (2b) ne sont que des cas particuliers des cas (1a), (2a) respectivement, où $j_0 = j$ (ou de manière équivalente $i_0 = i$) et donc où $c = 0$.

Nous avons maintenant tout en main pour décrire $m^*$ selon $m$. La définition matricielle des cas (b), pour $i \neq j$ peut être unifiée simplement :

$$m^{*\neq}_{ij} = \exists i_0 \in \{0, \ldots, n\} \setminus \{i, j\} \text{ tel que } m^{\neq}_{i_0 j} \wedge \overline{m^{\leq}_{ii_0}} = 0 \wedge (\overline{m^{\leq}_{i_0 i}} = 0 \vee \overline{m^{\leq}_{i_0 j}} = 0),$$

tout comme celle des cas (a) :

$$m^{*\neq}_{ij} = \exists i_0, j_0 \in \{0, \ldots, n\} \setminus \{i, j\}, i_0 \neq j_0 \text{ tels que } m^{\neq}_{i_0 j_0} \wedge \overline{m^{\leq}_{ii_0}} = -\overline{m^{\leq}_{j_0 j}} \wedge \begin{pmatrix} \overline{m^{\leq}_{jj_0}} = -\overline{m^{\leq}_{i_0 i}} \\ \vee\ \overline{m^{\leq}_{ij_0}} = -\overline{m^{\leq}_{i_0 j}} \end{pmatrix}.$$

Enfin, notre dernière remarque nous permet d'unifier ces deux définitions car $\overline{m^{\leq}_{ii}} = 0$, $\forall i \in [0, n]$. C'est ainsi que nous définissons l'opérateur de normalisation.

Définition 5.3 (opérateur de normalisation). *Grâce aux Lemmes 5.1–5.3 on peut donner la définition équivalente de $m^*$ suivante :*

– $m^* = \bot^{zoal}$ *si le test de satisfaisabilité de* $m$ *est négatif,*
– *sinon :*

$$\left( \begin{array}{l} m^{*\leq} = \overline{m^{\leq}} \\ m^{*\neq}_{ii} = \mathit{faux} \\ m^{*\neq}_{ij} = \left\{ \begin{array}{ll} m^{\neq}_{ij} & \\ \vee\ \overline{m^{\leq}_{ij}} < 0 \vee \overline{m^{\leq}_{ji}} < 0 & \mathit{cas\ (0)} \\ \vee\ \exists i_0 \in \{0,\ldots,n\} \setminus \{j\}, \exists j_0 \in \{0,\ldots,n\} \setminus \{i\}, i_0 \neq j_0, & \\ \quad (i \neq i_0 \vee j \neq j_0)\ \mathit{tels\ que} & \mathit{cas\ (a,b)} \\ \qquad\qquad m^{\neq}_{i_0 j_0} \wedge \overline{m^{\leq}_{ii_0}} = -\overline{m^{\leq}_{j_0 j}} \wedge \left( \begin{array}{l} \overline{m^{\leq}_{jj_0}} = -\overline{m^{\leq}_{i_0 i}} \\ \vee\ \overline{m^{\leq}_{ij_0}} = -\overline{m^{\leq}_{i_0 j}} \end{array} \right) & \end{array} \right. \end{array} \right).$$

Si l'on veut pouvoir raisonner sur le graphe de potentiels et de non-égalités, on peut traduire simplement les définitions matricielles précédentes en des définitions sur les chemins de ce graphe. Pour les cas (b) l'existence de $i_0$ et le fait que $m^{\neq}_{i_0 j}$ se récrivent par $\exists\langle\langle i, i_0, j\rangle\rangle$. De même, les hypothèses sur $i_0, j_0$ et le fait que $m^{\neq}_{i_0 j_0}$ pour les cas (a) se récrivent par $\exists\langle\langle i, i_0 - j_0, j\rangle\rangle$.

Nous avons d'ores et déjà intitulé cet opérateur comme étant un opérateur de normalisation sans avoir démontré qu'il calcule bien la forme normale d'une *d*DBM. C'est ce à quoi nous nous employons au théorème suivant (Thm. 5.4) pour lequel nous prouvons que l'opérateur est correct, *i.e.* $\gamma^{zoal}(m^*) = \gamma^{zoal}(m)$, puis qu'il est la meilleure abstraction de l'élément concret $\gamma^{zoal}(m)$.

Théorème 5.4 ($m^*$ meilleure abstraction). *Soit* $m$ *une dDBM et* $\mathbb{K} = \mathbb{Q}$ *ou* $\mathbb{R}$. *Alors*
– $\gamma^{zoal}(m^*) = \gamma^{zoal}(m)$ *;*
– *et* $m^* = \overline{m}$.
*Conséquence immédiate :* $(m^*)^* = m^*$ *et* $m^* \sqsubseteq^{ddbm} m$.

*Démonstration.*
– La correction repose sur celle de l'opérateur de normalisation des DBM (Déf. 3.4, p. 35) et une remarque simple : les contraintes de non-égalité introduites sont toutes des contraintes implicites (Lem. 5.3, p. 94).
– Pour montrer qu'il s'agit de la meilleure abstraction, on utilise de nouveau la définition de $m^*$ par décomposition, qui permet d'utiliser la propriété de saturation des DBM closes (Thm. 3.2, p. 36). On a :

$$\gamma^{zoal}(m^*) = \gamma^{zoal}(m) = \bigsqcup_{k \in K}{}^{\flat} \gamma^{zone}(m^k). \tag{5.2}$$

On procède par contradiction. Soit $n$ une *d*DBM telle que $\gamma^{zoal}(n) = \gamma^{zoal}(m)$ et $n \sqsubset^{ddbm} m^*$. Deux cas se présentent :
– premier cas : il existe $i, j \in [0, n]$ tels que $n^{\leq}_{ij} < m^{*\leq}_{ij}$. On sait qu'il existe $k \in K$ tel que $m^{*\leq}_{ij} = b(m^k_{ij})$. La preuve est différente selon que cette borne est stricte ou non :
  – soit $s(m^k_{ij}) = \leq$. Par saturation de $m^k$, on a : $\exists \rho \in \gamma^{zone}(m^k)$ tel que $\rho(x_i) - \rho(x_j) = b(m^k_{ij})$. En utilisant l'Équation 5.2, on obtient $\exists \rho \in \gamma^{zoal}(n)$ tel que

$\rho(x_i)-\rho(x_j) = b(m^k_{ij})$. Et donc $\exists \rho \in \gamma^{zoal}(n)$ tel que $\rho(x_i)-\rho(x_j) = m^{*\leq}_{ij} > n^{\leq}_{ij}$, ce qui est contradictoire.

- soit $s(m^k_{ij}) = <$. Par saturation de $m^k$, on a : $\forall \varepsilon > 0, \exists \rho \in \gamma^{zone}(m^k)$ tel que $0 < b(m^k_{ij}) - (\rho(x_i) - \rho(x_j)) \leq \varepsilon$. Par le même raisonnement que précédemment on obtient : $\forall \varepsilon > 0, \exists \rho \in \gamma^{zoal}(n)$ tel que $\rho(x_i) - \rho(x_j) \geq m^{*\leq}_{ij} - \varepsilon$. En prenant $\varepsilon = \frac{m^{*\leq}_{ij} - n^{\leq}_{ij}}{2}$, il s'en suit $\exists \rho \in \gamma^{zoal}(n)$ tel que $\rho(x_i) - \rho(x_j) \geq \frac{m^{*\leq}_{ij} + n^{\leq}_{ij}}{2}$, et donc tel que $\rho(x_i) - \rho(x_j) > n^{\leq}_{ij}$ ce qui est contradictoire.

- deuxième cas : il existe $i, j \in [0, n]$ tels que $n^{\neq}_{ij} \wedge \neg m^{*\neq}_{ij}$. On sait qu'il existe $k \in K$ tel que $\neg P(i, j, m^k)$. On suppose que $s(m^k_{ij}) = s(m^k_{ji}) = \leq$. Par saturation de $m^k$, on a : $\exists \rho_1, \rho_2 \in \gamma^{zone}(m^k)$ tels que $\rho_1(x_i) - \rho_1(x_j) = b(m^k_{ij}) \geq 0$ et $\rho_2(x_i) - \rho_2(x_j) = -b(m^k_{ji}) \leq 0$. Par convexité de la zone $\gamma^{zone}(m^k)$ il s'en suit, en utilisant l'Équation 5.2, $\exists \rho \in \gamma^{zoal}(n)$ tel que $\rho(x_i) - \rho(x_j) = 0$ ce qui est contradictoire.

  La preuve est similaire pour les autres cas, où les bornes peuvent êtres strictes. □

Nous pouvons maintenant nous intéresser à l'implantation de l'opérateur de normalisation.

#### 5.4.2.2 Algorithme de normalisation

Il existe de nombreuses implantations possibles de l'opérateur de normalisation (Déf. 5.3), et celle que nous présentons à la Figure 5.5 n'est certainement pas optimale.

Cependant, au-delà des choix qui ont été faits, toute implantation s'articule peu ou prou sur le même schéma. En premier lieu, on a la fermeture selon les plus courts chemins de $m^{\leq}$ (ligne **1**).

Puis, un premier choix est à opérer entre inclure tel quel le test de satisfaisabilité (Fig. 5.3, p. 90) ou tirer parti des non-égalités entre mêmes variables que peut générer le cas (1b) (prendre $i_0 = j$) pour le simplifier : on ne testerait alors que la diagonale de la matrice de non-égalités. Bien que la seconde solution soit assez élégante, nous avons choisi la première (lignes **3**–**9**) pour deux raisons. D'une part, on souhaite générer les non-égalités des cas (1a), (2a), (1b) et (2b) de manière unifiée. D'autre part, en s'appuyant sur la structure du test, on peut générer les non-égalités relevant du cas (0) (ligne **13**) et calculer deux ensembles intéressants, notés $D$ et $C$ (ligne **2**), que sont l'ensemble des paires de variables qui peuvent déclencher la génération de non-égalités, *i.e.* deux variables non égales dont la différence est soit négative, soit positive (ligne **9**), et l'ensemble des paires de variables qui peuvent devenir non égales, *i.e.* deux variables dont la différence peut être nulle (ligne **11**).

Reste à choisir comment générer les non-égalités relevant de tous les autres cas. Il y a une palette de possibilités, entre partir des triplets $(i, j, k)$ de variables tels que $m^{\leq}_{ik} = -m^{\leq}_{kj}$ et partir des couples de variables de $D$. Par l'expérimentation, on sait que ces ensembles de triplets et de couples sont en général petits : les arguments plaidant pour ne pas utiliser l'ensemble des triplets sont plutôt ceux du temps, de l'espace mémoire, nécessaires à leur recherche, stockage, de l'ordre de $\mathcal{O}(n^3)$, alors qu'ils sont de $\mathcal{O}(n^2)$

```
   //plus courts chemins
 1 m ← (FloydWarshall(m^≤), m^≠) ;
   //test de satisfaisabilité et non-égalités cas (0)
 2 D ← ∅ ; C ← ∅ ;
 3 for i ← 0 to n do
 4 |  if m^≤_ii ≥ 0 then m^≤_ii ← 0 else return ⊥^zoal ;
 5 |  if m^≠_ii then return ⊥^zoal ;
 6 |  for j ← i+1 to n do
 7 |  |  if m^≤_ij ≥ 0 ∧ m^≤_ji ≥ 0 then
 8 |  |  |  if m^≠_ij then
 9 |  |  |  |  if m^≤_ij = 0 ∧ m^≤_ji = 0 then return ⊥^zoal else D ← D ∪ {(i, j)}
10 |  |  |  else
11 |  |  |  |  C ← C ∪ {(i, j), (j, i)}
12 |  |  else
13 |  |  |  m^≠_ij ← vrai ; m^≠_ji ← vrai
   //non-égalités cas (1a) (2a) (1b) (2b)
14 forall (i0, j0) ∈ D do
15 |  forall (i, j) ∈ C do
16 |  |  if m^≤_ii0 = −m^≤_j0j ∧ (m^≤_jj0 = −m^≤_i0i ∨ m^≤_ij0 = −m^≤_i0j) then
17 |  |  |  m^≠_ij ← vrai ; m^≠_ji ← vrai ;
18 |  |  |  C ← C \ {(i, j), (j, i)}
```

Figure 5.5 – Normalisation d'une *d*DBM $m$ dans le cas dense

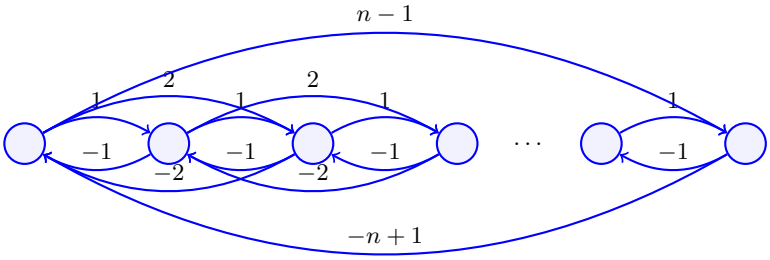
pour l'ensemble $D$.

Nous avons donc choisi d'itérer sur l'ensemble $D$, et de chercher les nouvelles non-égalités parmi les couples de variables de $C$, en s'interrogeant sur les informations susceptibles d'optimiser ce schéma et dont la recherche et le stockage aient un coût raisonnable.

Il est visible que, pour les non-égalités relevant des cas (b), il est possible de repérer et stocker les couples de variables égales ou ordonnées pour un coût en $\mathcal{O}(n^2)$. Cependant traiter ces non-égalités indépendamment des autres n'est pas fructueux, car cela ne permet pas de diminuer le nombre d'opérations nécessaires au traitement des cas (a). Ainsi, nous terminons (lignes **14**–**18**) notre algorithme en générant simplement les non-égalités en regardant si tout couple de $D$ vérifie avec un couple de $C$ les conditions énoncées par la définition de l'opérateur (ligne **16**). Si c'est le cas, la non-égalité est introduite dans la matrice (ligne **17**) et on retire de $C$ le couple correspondant (ligne **18**). Potentiellement, le cardinal de $C$ et $D$ est de $\mathcal{O}(n^2)$, ce qui porte la complexité de l'algorithme de normalisation en $\mathcal{O}(n^4)$.

Nous n'avons pas réussi à prouver que le calcul de la forme normale d'une *d*DBM nécessite au moins $\mathcal{O}(n^4)$ opérations. D'ailleurs, il est assez intéressant de voir qu'il n'y a

pas plus de $\mathcal{O}(n^3)$ instances possibles du cas (1a) ou (2a). En effet, les cas (a) imposent un ordre sur les variables, et donc le nombre de couples de variables tels que, par exemple pour le cas (1a), $x - y = c$ pour $c$ donné est inférieur à $n$. On considère les graphes de potentiels de la forme suivante comme étant ceux générant le maximum de couples pouvant instancier le cas (1a) :




On voit que pour chaque $k \in \{1, \ldots, n-1\}$, il existe $n-k$ couples dont les variables sont distantes de $k$. Pour $k$ donné, les couples correspondants combinés deux à deux, mis à part ceux qui partagent une variable, constituent $(C^2_{n-k} - (\sum_{\ell=1}^{k} \lfloor \frac{n-\ell}{k} \rfloor - 1))$ instances, et donc en sommant sur $k = [1, n-1]$ on trouve $\mathcal{O}(n^3)$ instances possibles. Mais cela n'implique pas l'existence d'un algorithme ayant cette complexité : en effet, repérer et stocker chacun des couples selon une même distance $c$ nécessite $\mathcal{O}(n^4)$ opérations dans le pire des cas. On remarquera que ces graphes particuliers sont traités très rapidement par notre algorithme puisque $C = \emptyset$.

Enfin, on notera que si l'on ne souhaite pas effectuer la normalisation complètement mais s'arrêter à la découverte des non-égalités des cas (b), la complexité de l'algorithme tombe en $\mathcal{O}(n^3)$. Cette idée permet au moins d'expliciter tous les ordres stricts qui représentent en général une majorité des non-égalités représentées par les *d*DBM. Pour trouver ces dernières, qui sont celles du cas (2b), on peut là encore imaginer de nombreux algorithmes, allant d'un décalque de celui de normalisation des DBM représentant des contraintes de zones strictes, à l'algorithme que nous avons proposé dans [PH07] [3], qui consiste en une fermeture transitive du graphe de non-égalité selon le graphe de potentiels, un algorithme basé sur celui de fermeture transitive de [GK79].

**Algorithme de normalisation incrémental** Lorsqu'une sous-partie particulière des valeurs d'une *d*DBM close a été modifiée, on souhaite pouvoir calculer sa forme normale sans recourir à une exécution complète de l'opération de normalisation. Cela permet notamment d'implanter plus efficacement les fonctions de transfert (Sec. 5.7.2).

Dans [Min04], il est proposé un tel algorithme de normalisation incrémental pour les DBM (Sec. 3.1.1.1). Si $c$ colonnes (et les lignes correspondantes) ont été modifiées, l'algorithme s'exécute en $\mathcal{O}(n^3 - (n-c)^3)$. Sa complexité est donc quadratique lorsque seule une colonne (et la ligne correspondante) a été modifiée.

3. Soyons honnête, une erreur importante a été commise dans cette publication où les non-égalités relevant des cas (a) n'étaient pas prises en compte pour le calcul de la forme normale. Ainsi une algorithmique avait été mise en place pour résoudre efficacement les cas (b).

Ainsi, lorsque seule la matrice de bornes d'une *d*DBM close est modifiée, nous pouvons utiliser cet algorithme incrémental. En revanche, ce calcul pouvant modifier n'importe quelle borne, il est nécessaire de procéder au calcul complet de la matrice de non-égalités. Au final, la complexité de la normalisation n'est pas améliorée.

*A contrario*, si seulement $c$ colonnes (et les lignes correspondantes) de la matrice de non-égalités de la *d*DBM close sont modifiées, on a un algorithme de normalisation incrémental en $\mathcal{O}(c.n^3)$. En effet, d'une part la matrice de bornes ne peut être affectée par ces changements, d'autre part l'ensemble $D$ à considérer, qui est celui des non-égalités ajoutées, est au plus de cardinal $c.n$.

## 5.5 Représentation canonique : cas discret

On s'intéresse à présent au problème du calcul de la forme normale d'une *d*DBM lorsque $\mathbb{K} = \mathbb{Z}$. Selon le même raisonnement que celui utilisé pour le cas dense, on définit un algorithme calculant la forme normale. Sa complexité étant exponentielle, il est proposé une version incomplète, dont la complexité en temps est de $\mathcal{O}(n^4)$.

On étudie tout d'abord le cas particulier de la satisfaisabilité de l'ensemble de contraintes représenté par la *d*DBM, en essayant d'évaluer finement sa complexité en donnant une nouvelle réduction polynomiale, et en s'intéressant à des cas particuliers simples.

### 5.5.1 Test de satisfaisabilité

Il est connu depuis longtemps (Thm. 3.4, p. 42) que décider de la satisfaisabilité d'un ensemble de contraintes de potentiel et de non-égalité est un problème NP-complet. La réduction polynomiale alors utilisée pour le prouver était celle du problème de la coloration d'un graphe par trois couleurs (Sec. 3.2.2.2).

Afin de mieux cerner la complexité de notre problème, nous avons cherché d'autres problèmes de coloration plus difficiles à résoudre, que celui de 3-coloration, qui se réduiraient à lui. À notre connaissance, le plus difficile d'entre eux ayant été étudié est le problème de coloration, introduit récemment par [BDM08], où pour chaque sommet les couleurs possibles appartiennent à un intervalle particulier – connu en anglais sous l'appellation « $(\gamma, \mu)$-*coloring* » où les couleurs possibles pour chaque sommets $v$ appartiennent à $[\gamma(v), \mu(v)]$.

La réduction à opérer est similaire à celle utilisée lorsque la restriction porte sur trois, ou $k$, couleurs. Soit $G = (V, E)$ un graphe non orienté et $\gamma, \mu$ deux fonctions de $V \to \mathbb{N}$ telles que $\forall v \in V, \gamma(v) \leq \mu(v)$, une instance du problème $(\gamma, \mu)$-*coloring*. On construit la *d*DBM $m$ suivante, sur l'ensemble de variables que forme $V$, où $\forall v, v' \in V$, $m^{\leq}_{v0} = \mu(v), m^{\leq}_{0v} = -\gamma(v), m^{\leq}_{vv'} = +\infty$ et $m^{\neq}_{vv'} \Leftrightarrow (v, v') \in E$. Clairement, ce codage est polynomial et $G$ admet un coloriage correct si et seulement si $m$ est non vide.

Ce problème de coloration est strictement plus difficile à résoudre que celui de coloration par $k$ couleurs, à moins que P = NP (*cf.* [BDM08]). En effet, ce dernier problème peut être résolu en temps polynomial *a contrario* du premier pour certaines classes de graphes cordaux : graphes bipartis, graphes d'intervalles, graphes scindés. Par contre les auteurs ont établi que pour les graphes scindés complets (tous les sommets de la clique sont reliés à tous les sommets du stable) et les graphes bipartis complets ces deux

problèmes de décision sont polynomiaux.

Au-delà des résultats de complexité qu'apportent ces remarques, on peut s'interroger sur l'opportunité de spécialiser le test de satisfaisabilité pour les instances se traduisant à un des graphes des deux classes précitées, et ainsi utiliser les algorithmes polynomiaux correspondants. Ceci nécessite bien sûr de pouvoir reconnaître ces instances de manière efficace : [RT75] donne un algorithme pour reconnaître un graphe cordal en temps quadratique (dans le nombre de sommets), [DSW88] donne un algorithme permettant de trouver un graphe cordal maximal dans un graphe quelconque, en temps cubique – [PP07] montre que la complexité est quadratique s'il s'agit de découvrir un graphe scindé ou un graphe biparti maximal. Cependant, il est difficile de penser que de tels graphes ou sous-graphes vont être fréquents : on ne peut donc pas raisonnablement penser à les utiliser.

Néanmoins, on notera les travaux relatés dans [KJS07] qui s'essayent à ce type d'optimisations : après identification d'ensembles de variables ayant même $\gamma$ et $\mu$ ils recherchent parmi eux une clique qui rendrait impossible toute valuation des variables. Notre exemple à la Figure 5.1 (p. 85) illustre cette situation : on a $\gamma(x_i) = 1$ et $\mu(x_i) = 2$ pour les variables $x_1, x_2$ et $x_3$ qui forment une clique. Ces cas d'ensembles de variables qui sont non égales deux à deux ont été longuement étudiés dans le cadre de la programmation logique par contrainte comme étant ceux de la contrainte « *alldifferent* » : ils bénéficient ainsi de procédures de décisions efficaces [vH01]. Dans [KJS07], ces procédures ne sont pas utilisées, car inutiles dans le cas très simple où les variables ont les mêmes intervalles de valuations possibles – il suffit que la taille de la clique soit strictement supérieure à $\mu(x_i) - \gamma(x_i) + 1$ pour conclure à l'insatisfaisabilité. Cette restriction leur permet cependant de ne pas rechercher des cliques sur n'importe quel ensemble de variables pour instancier le problème « *alldifferent* » et ainsi garder un test de satisfaisabilité compétitif. Cela dit, la valeur ajoutée de cette optimisation ne se voit que sur quelques exemples très particuliers et de grande taille.

Enfin, si l'on considère le test de satisfaisabilité, évident, consistant à tester la satisfaisabilité des $2^d$ DBM résultant de la décomposition par disjonction des $d$ non-égalités que représente $m^{\neq}$, on peut encore s'intéresser à des hypothèses qui permettraient d'éviter la décomposition de certaines non-égalités. C'est ce à quoi se sont employés les auteurs de [SW02] où ils découvrent ces non-égalités dites « inertes » qui, comme dans le cas dense, ne peuvent s'additionner à aucune autre pour amener à l'insatisfaisabilité de l'ensemble de contraintes. [KJS07] exprime ces mêmes conditions, sans le savoir, d'une toute autre manière, plus simple : toute non égalité qui ne fait pas partie d'une composante fortement connexe du graphe de potentiels et de non-égalités est inerte. En effet, il serait dommage pour l'ensemble de contraintes $\forall i \in [1, n-1], x_i \geq x_{i+1} \wedge x_i \neq x_{i+1}$ de décomposer une seule non-égalité pour conclure à la satisfaisabilité de cet ensemble.

*A contrario* des domaines dont sont issus ces derniers travaux, notre utilisation du test de satisfaisabilité est marginale : on se contentera de donner l'algorithme de calcul de la forme normale des zones précisant alias, qui répond également au problème de décision. Ceux qui souhaiteraient implanter un test optimisé pourront prendre les travaux évoqués précédemment comme des pistes, notamment celle des non-inégalités inertes.

### 5.5.2 Calcul de la forme normale

Pour calculer la forme normale de $m \in \mathcal{D}^{zoal}$, nous en sommes donc réduits, vu la nature du problème, à construire la disjonction de DBM représentant le même ensemble de valuations que $m$. Bien entendu, il s'agit ici de DBM classiques, puisqu'il n'est pas nécessaire de recourir, comme dans le cas dense, à des contraintes de zones strictes pour traduire les non-égalités que représente $m^{\neq}$ :

$$x_i \neq x_j \Leftrightarrow (x_i - x_j \leq -1 \vee x_j - x_i \leq -1).$$

Une fois cette disjonction construite, il ne reste plus qu'à reconstruire une *d*DBM à partir des formes normales des DBM obtenues, qui préserve là encore l'ensemble de valuations. On propose la *d*DBM $m^*$ suivante, très semblable à celle proposée pour le cas dense et dont on reprend les notations :

$$\begin{cases} \bot^{zoal} & \text{si } K = \emptyset \\ \left( \begin{array}{l} {m^*}^{\leq}_{ij} = \max_{k \in K} \overline{m^k_{ij}} \\ {m^*}^{\neq}_{ij} = \bigwedge_{k \in K} (\overline{m^k_{ij}} \leq -1 \vee \overline{m^k_{ji}} \leq -1) \end{array} \right) & \text{sinon.} \end{cases}$$

Le théorème suivant (Thm. 5.5) montre que $m^*$ est bien la forme normale de $m$. La preuve que la première n'approxime pas la seconde repose simplement sur le fait que toute contrainte de zone et toute contrainte de non-égalité présentes dans $m$ sont présentes (pour les secondes sous la forme d'une contrainte plus forte, de la forme $m^k_{ij} \leq -1$) dans toute DBM $m^k$.

Théorème 5.5 ($m^*$ meilleure abstraction). *Soit $m$ une dDBM et $\mathbb{K} = \mathbb{Z}$. Alors*
- *$\gamma^{zoal}(m^*) = \gamma^{zoal}(m)$ ;*
- *et $m^* = \overline{m}$.*

*Conséquence immédiate : $(m^*)^* = m^*$ et $m^* \sqsubseteq^{ddbm} m$.*

*Démonstration.*
- Par construction et correction de l'opérateur de normalisation des DBM (Déf. 3.4, p. 35) on a $\gamma^{zoal}(m) = \bigsqcup^{\flat}_{k \in K} \gamma^{zone}(m^k)$. Cette dernière union est clairement approximée par $\gamma^{zoal}(m^*)$, donc $\gamma^{zoal}(m) \sqsubseteq^{\flat} \gamma^{zoal}(m^*)$. On montre l'inclusion inverse : soit $\rho \in \gamma^{zoal}(m^*)$, par définition de la fonction de concrétisation on a $\forall i, j, \rho(x_i) - \rho(x_j) \leq {m^*}^{\leq}_{ij} \wedge {m^*}^{\neq}_{ij} \Rightarrow \rho(x_i) \neq \rho(x_j)$. D'une part, comme l'introduction des non-égalités ne relâche aucune contrainte de potentiel, on a $\forall k \in K, \overline{m^k_{ij}} \leq m^{\leq}_{ij}$ et donc $m^{\leq}_{ij} \geq \max_{k \in K} \overline{m^k_{ij}} \geq \rho(x_i) - \rho(x_j)$. D'autre part, si $m^{\neq}_{ij}$ alors pour tout $k \in K$, la DBM $m^k$ est soit issue d'une branche où la contrainte de potentiel entre $i$ et $j$ est inférieure à $-1$ soit d'une branche où la contrainte de potentiel entre $j$ et $i$ est inférieure à $-1$, donc $m^{\neq}_{ij} \Rightarrow (\forall k \in K, \overline{m^k_{ij}} \leq -1 \vee \overline{m^k_{ji}} \leq -1) \Rightarrow \rho(x_i) \neq \rho(x_j)$. On a donc $\rho \in \gamma^{zoal}(m)$.
- La preuve est similaire à celle donnée pour le cas dense (Thm. 5.4, p. 98).

□

Si on ne peut s'affranchir de la construction exponentielle de $\overline{m}$, on s'intéresse à des conditions qui permettraient d'éviter la construction de certaines branches de l'arbre de décomposition complet (Fig. 5.4, p. 91). Pour cela, on regarde les différents cas qui

peuvent se présenter localement lors de l'introduction d'une non-égalité $x_{i_0} \neq x_{j_0}$ dans une DBM $m^0$ de l'arbre de décomposition. On suppose que $m^0$ est close et que $m^0 \neq \bot^{zone}$. On a $m^1$ et $m^2$, les deux DBM résultant de l'opération, égales à $m^0$, excepté pour $m^1_{i_0j_0} = \min(m^0_{i_0j_0}, -1)$ et $m^2_{j_0i_0} = \min(m^0_{j_0i_0}, -1)$. On étudie la borne de la contrainte de potentiel $x_{i_0} - x_{j_0}$ :

- Si $m_{i_0j_0} \leq -1$ alors la contrainte de non-égalité n'apporte pas d'information, elle ne joue donc aucun rôle. En effet, on a $m^1 = m^0$ et $\overline{m^2} = \bot^{zone}$ puisque $\langle x_{i_0}, x_{j_0}, x_{i_0} \rangle$ forme un cycle de poids négatif ($m_{i_0j_0} - 1 \leq -2$) dans $m^2$.
- Si $m_{i_0j_0} = 0$, on a une contrainte d'inégalité stricte, il ne sert donc à rien, là encore, d'introduire une disjonction. En effet, on a $\overline{m^2} = \bot^{zone}$ puisque $\langle x_{i_0}, x_{j_0}, x_{i_0} \rangle$ forme un cycle de poids négatif ($m_{i_0j_0} - 1 = -1$). La traduction de la non-égalité se résume donc à réduire la borne $m_{i_0j_0}$ à $-1$, ce qui est cohérent vis-à-vis de l'ordre $\sqsubseteq^{ddbm}$ et de ce qu'est la forme normale. À moins que $m_{j_0i_0} = 0$ et on peut tout de suite conclure à l'inconsistance de $m^0$.
- Si $m_{i_0j_0} \geq 1$, alors créer une disjonction a un intérêt. À part si $m_{j_0i_0} \leq 0$, auquel cas on retombe dans l'un des cas précédent, on a pas d'incohérences triviales.

Ainsi, le calcul de la forme normale consiste à alterner réduction des bornes des inégalités strictes et disjonction de cas, jusqu'à ce que toute DBM n'aie plus que des non-égalités triviales vis-à-vis des contraintes de zones (Fig. 5.6).

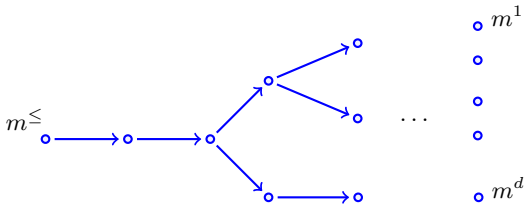
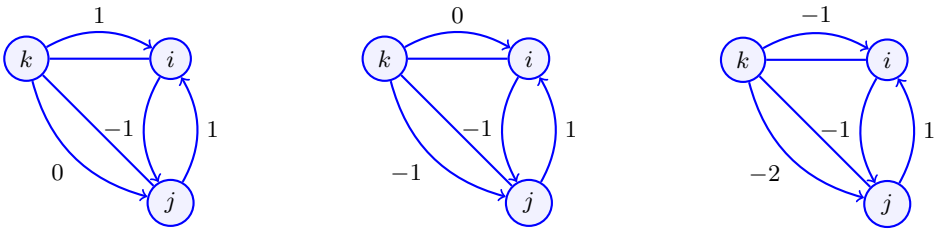



Figure 5.6 – Construction de la forme normale de $m$ par décomposition des non-égalités

Si ces résultats sont basiques, il est cependant intéressant de mentionner que la réduction d'une borne d'une inégalité stricte peut en faire apparaître en cascade de nouvelles, dont on souhaite également réduire les bornes avant d'introduire toute disjonction. On prend pour exemple à la Figure 5.7 la $d$DBM représentée en (a) où $x_i = x_j + 1$, $x_j < x_k$ et $x_i \neq x_k$. La réduction de la borne $m_{jk}$ explicite que $x_k$ est bornée par $x_j + 1$. Or cette borne est égale à $x_i$ et donc ell ne peut pas être atteinte par $x_k$ puisque ces deux variables sont non égales : on a donc une nouvelle inégalité stricte $x_i < x_k$ (b). En réduisant la borne correspondante, il apparaît alors que $x_k \geq x_j + 2$ (c), illustrant du même coup que ces réductions à répétitions peuvent toucher une même borne.

#### 5.5.2.1 Algorithme de normalisation

On donne un premier algorithme qui calcule la forme normale selon le schéma précédemment évoqué. Cependant, son coût est tel qu'on lui préférera en pratique, soit celui donné pour le cas dense (Sec. 5.4.2.2), qui constitue une approximation correcte, soit une version simplifiée consistant à ne réduire que les bornes des inégalités strictes avant d'appliquer l'algorithme du cas dense. En évitant toute disjonction, cette approximation plus précise de la forme normale est obtenue en $\mathcal{O}(n^4)$ dans le pire cas.




(a) représentation close pour les plus courts chemins de $x_i = x_j + 1 \wedge xj < x_k \wedge x_i \neq x_k$

(b) représentation explicitant que $x_i \leq x_k$ et $x_j + 1 \leq x_k$

(c) représentation explicitant que $x_i + 1 \leq x_k$ et $x_j + 2 \leq x_k$

Figure 5.7 – Exemple de réductions à répétition d'une borne lors du traitement des inégalités strictes

Afin d'implanter ces algorithmes, on définit une fonction auxiliaire `ClasserNonEgalites` (Fig. 5.8), qui à une *d*DBM $m$ et à un ensemble $D$ de non-égalités retourne les non-égalités formant des inégalités strictes ($D^0$, lignes **6** et **8**) avec $m^{\leq}$ et les non-égalités pouvant engendrer une disjonction ($D^*$, ligne **10**) de $m^{\leq}$. Par la même occasion, comme l'étude de cas le suggérait en présence d'une inégalité stricte, elle teste si les deux variables correspondantes ne sont pas égales (ligne **9**), auquel cas $m$ est incohérente vis-à-vis de $D$ et la fonction retourne alors en mettant à *vrai* un booléen prévu à cet effet.

```
1  D^0 ← ∅ ; D* ← ∅ ;
2  forall (i, j) ∈ D do
3      if m^≤_ij ≥ 0 ∧ m^≤_ji ≥ 0 then
4          s ← 0 ;
5          if m^≤_ij = 0 then
6              D^0 ← D^0 ∪ {(i, j)} ; s ← 1
7          if m^≤_ji = 0 then
8              D^0 ← D^0 ∪ {(j, i)} ; s ← s + 1 ;
9              if s = 2 then return (vrai, ∅, ∅)
10         if s = 0 then D* ← D* ∪ {(i, j)} ;
11 return (faux, D^0, D*)
```

Figure 5.8 – La fonction `ClasserNonEgalites`$(m, D)$, auxiliaire des algorithmes de normalisations de *d*DBM présentés respectivement aux Figures 5.9 et 5.11

Pour l'algorithme complet, plutôt que d'implanter l'opération $m^*$ telle quelle, on construit récursivement la forme normale de $m$ en combinant les DBM au niveau des nœuds internes de la décomposition : elles sont alors transformées en *d*DBM pour être combinées par l'opérateur d'union $\sqcup^{ddbm}$.

L'algorithme est donc principalement constitué de la fonction récursive `ReduireDisjoindre` (Fig. 5.9) qui prend en entrée une *d*DBM $m$ – dont elle va retourner la forme normale –

```
1  (b, D^0, D^*) ← ClasserNonEgalites(m,D) ;
2  if b then return ⊥^zoal ;
3  if D^0 = ∅ ∧ D^* = ∅ then
4      for i ← 0 to n do
5          for j ← i+1 to n do
6              if m^≤_ij < 0 ∨ m^≤_ji < 0 then m^≠_ij ← vrai ; m^≠_ji ← vrai
7      return m
8  else
9      if D^0 ≠ ∅ then
10         forall (i, j) ∈ D^0 do m^≤_ij ← −1 ;
11         m^≤ ← NormalisationIncrementale(D^0, m^≤) ;
12         if m^≤ = ⊥^zone then
13             return ⊥^zoal
14         else
15             return ReduireDisjoindre((m^≤, m^≠), D^*)
16     else
17         (i, j) ←Extraire(D^*) ;
18         m^1 ← m^≤ ; m^1_ij ← −1 ; m^2 ← m^≤ ; m^2_ji ← −1 ;
19         m^1 ← NormalisationIncrementale({(i, j)}, m^1) ;
20         m^2 ← NormalisationIncrementale({(j, i)}, m^2) ;
21         if m^1 = ⊥^zone then
22             if m^2 = ⊥^zone then
23                 return ⊥^zoal
24             else
25                 return ReduireDisjoindre((m^2, m^≠), D^*)
26         else
27             if m^2 = ⊥^zone then
28                 return ReduireDisjoindre((m^1, m^≠), D^*)
29             else
30                 return ReduireDisjoindre((m^1, m^≠), D^*) ⊔^ddbm
                          ReduireDisjoindre((m^2, m^≠), D^*)
```

Figure 5.9 – `ReduireDisjoindre`$(m, D)$, la fonction principale de l'algorithme de normalisation d'une $d$DBM dans le cas discret (Fig. 5.10)

telle que $m^{\leq}$ est en forme normale et un ensemble $D$ de couples de variables non égales selon $m^{\neq}$, devant inclure toutes les non-égalités pouvant engendrer la réduction d'une borne ou la disjonction de $m^{\leq}$. La fonction commence par appeler `ClasserNonEgalites` (lignes **1** et **2**) pour déterminer parmi les non-égalités de $D$ celles qui engendreront l'une (ensemble $D^0$) ou l'autre (ensemble $D^*$) de ces deux situations.

Vient alors le cas de base (lignes **3** à **7**) où ces deux ensembles sont vides, autrement dit où toute non-égalité de $D$, et donc de $m^{\neq}$, est triviale pour $m^{\leq}$. Comme $m^{\leq}$ est en forme normale, pour normaliser $m$, il ne reste donc plus qu'à s'assurer que toute

contrainte de non-égalité triviale est explicitée (ligne **6**) dans $m^{\neq}$.

Suit la définition récursive : si $D^0$ n'est pas vide, on réduit l'ensemble des bornes correspondantes (ligne **10**) de $m^{\leq}$, dont on recherche alors la forme normale en utilisant l'algorithme de normalisation incrémental (ligne **11**). En effet, toutes les colonnes n'ont peut-être pas été affectées par des réductions de bornes : on passe d'ailleurs à `NormalisationIncrementale` l'ensemble $D^0$ afin qu'il calcule l'ensemble minimum de colonnes (et de lignes correspondantes) dont il doit effectuer la normalisation. Si $m^{\leq}$ s'avère insatisfaisable la fonction retourne, sinon on appelle récursivement `ReduireDisjoindre` avec pour ensemble de non-égalités à prendre en compte $D^*$ : en effet, les bornes des contraintes de potentiel n'ayant pu que diminuer, ce n'est que les non-égalités parmi $D^*$ qui pourront générer de nouvelles réduction de bornes ou des disjonctions.

Si $D^0$ est vide, on procède à une disjonction (lignes **17** à **30**) pour une des non-égalités de $D^*$ : on instancie comme attendu les deux DBM $m^1$ et $m^2$, que l'on normalise là encore par l'algorithme incrémental et qui s'exécute ici en temps quadratique puisqu'une seule borne a été modifiée. Si $m^1$ et $m^2$ sont insatisfaisable, la fonction retourne $\bot^{zoal}$, sinon elle effectue le ou les appels récursifs nécessaires, sur les non-égalités restantes de $D^*$.

Remarque. *S'il n'y a que des non-égalités inertes dans $D^*$ alors aucune disjonction n'est nécessaire (Sec. 5.5.1). Puisque nous ne donnons pas la preuve que dans ce cas m est déjà en forme normale, l'algorithme n'utilise pas ce fait.*

Ainsi, l'algorithme de normalisation (Fig. 5.10) consiste simplement en l'appel de la fonction `ReduireDisjoindre`, en respectant ses hypothèses : $m^{\leq}$ est normalisée et sa satisfaisabilité vérifiée avant de former l'ensemble $D$ des non-égalités représentées par $m^{\neq}$.

```
1 m^≤ ← Normalisation(m^≤) ;
2 if m^≤ = ⊥^zone then return ⊥^zoal ;
3 D ← ∅ ;
4 for i ← 0 to n do
5 |   for j ← i to n do
6 |   |   if m^≠_ij then D ← D ∪ {(i, j)}
7 ReduireDisjoindre((m^≤, m^≠), D)
```

Figure 5.10 – Normalisation d'une *d*DBM $m$ dans le cas discret

**Normalisation partielle** Comme nous l'avions annoncé, nous proposons également un algorithme de normalisation partielle. Il s'agit de l'algorithme que nous venons de donner (Fig. 5.10) où la fonction `ReduireDisjoindre` est remplacée par la fonction `Reduire` présentée à la Figure 5.11. Deux changements sont opérés : d'une part, elle n'introduit aucune disjonction et d'autre part, lorsque toute inégalité stricte a sa borne réduite à $-1$, elle appelle l'algorithme de normalisation dense.

Ceci est cohérent à deux titres : premièrement l'algorithme de normalisation dense permet d'expliciter des non-égalités que la réduction des bornes des inégalités strictes ne peuvent expliciter – ce sont les cas (a) que génère l'algorithme – et deuxièmement il

n'engendre pas de nouvelles inégalités strictes : on montre aisément que si la contrainte de potentiel où apparaît la nouvelle non-égalité est à 0 alors la phase de réduction des bornes précédente l'aurait réduite. Par exemple, pour le cas (1a) on a : $x_i = x_{i_0} + c \wedge x_j = x_{j_0} + c \wedge x_{i_0} \neq x_{j_0} \Rightarrow x_i \neq x_j$. Si $x_i \leq x_j$ alors c'est que $x_{i_0} \leq x_{j_0}$ et le traitement de l'inégalité stricte aurait amené, après normalisation, $x_i - x_j \leq -1$.

$(b, D^0, D^*) \leftarrow$ `ClasserNonEgalites(`$m$`,`$D$`)` ;
**if** $b$ **then return** $\bot^{zoal}$ ;
**if** $D^0 = \emptyset$ **then**
  **return** `NormalisationDense(`$m$`)`
**else**
  **forall** $(i,j) \in D^0$ **do** $m^{\leq}_{ij} \leftarrow -1$ ;
  $m^{\leq} \leftarrow$ `NormalisationIncrementale(`$D^0, m^{\leq}$`)` ;
  **if** $m^{\leq} = \bot^{zone}$ **then return** $\bot^{zoal}$ **else return** `Reduire(`$(m^{\leq}, m^{\neq}), D^*$`)`

Figure 5.11 – La fonction `Reduire`$(m, D)$ qui, appelée en lieu et place de la fonction `ReduireDisjoindre` par l'algorithme de normalisation, fait renvoyer par ce dernier une forme réduite de $m$ en $\mathcal{O}(n^4)$

La complexité de cet algorithme de normalisation partielle est dominée par celle de l'algorithme de normalisation dense, qui est de $\mathcal{O}(n^4)$ et celle de l'application à répétition de l'algorithme de normalisation incrémentale des DBM, également de $\mathcal{O}(n^4)$. Pour le montrer, on note $\mathsf{C}(c)$ la complexité de la normalisation incrémentale où les bornes de $c$ colonnes sont à recalculer. On rappelle que $\mathsf{C}(c) = n^3 - (n-c)^3 = c^3 - 3nc^2 + 3cn^2$. Si on note $c_1, \ldots, c_p$ le nombre de colonnes affectées à chacune des $p$ normalisations incrémentales qu'effectue l'algorithme, on a $1 \leq c_i \leq n$ et donc $c_i^3 \leq nc_i^2 \leq 3nc_i^2$. Il suit $c_i^3 - 3nc_i^2 \leq 0$, donc $0 \leq \mathsf{C}(c_i) \leq 3c_i n^2$ et

$$\sum_{i=1}^{p} \mathsf{C}(c_i) \leq 3n^2 \sum_{i=1}^{p} c_i.$$

Or il ne peut pas y avoir plus de réductions de bornes que de non-égalités, donc $\sum_{i=1}^{p} c_i \leq \frac{n(n-1)}{2} \leq n^2$. La complexité est donc bornée par $3n^4$.

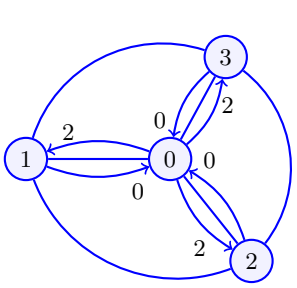
On termine cette section en donnant à la Figure 5.12 une illustration de la précision de l'algorithme de normalisation partielle.

## 5.6 Opérateurs de treillis

On introduit à présent les opérateurs abstraits sur les zones précisant alias nécessaires à la définition du treillis : test d'inclusion, union et intersection.

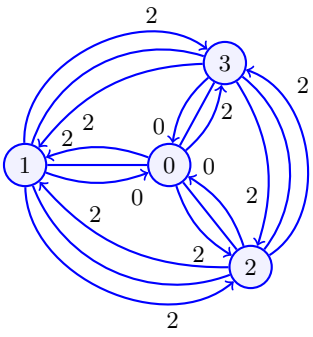
### 5.6.1 Test d'inclusion

Si l'on a la monotonicité de $\gamma^{zoal}$ vis-à-vis de $\sqsubseteq^{ddbm}$, on souhaite avoir un ordre partiel $\sqsubseteq^{zoal}$ qui de plus assure $\gamma^{zoal}(m) \sqsubseteq^{\flat} \gamma^{zoal}(m') \Rightarrow m \sqsubseteq^{zoal} m'$. Pour ce faire, on applique simplement l'opérateur de normalisation sur l'argument gauche du test d'inclusion entre $d$DBM.



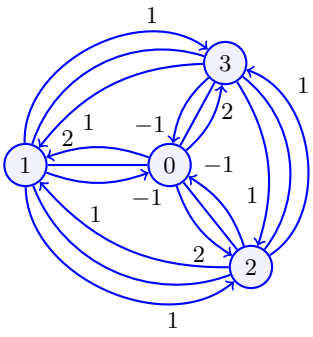

$$\begin{cases} 0 \leq x_1, x_2, x_3 \leq 2 \\ x_1 \neq x_2 \wedge x_1 \neq 0 \\ x_2 \neq x_3 \wedge x_2 \neq 0 \\ x_1 \neq x_3 \wedge x_3 \neq 0 \end{cases}$$

(a) Représentations d'une *d*DBM $m$




$$\begin{cases} 0 \leq x_1, x_2, x_3 \leq 2 \\ x_1 \neq x_2 \wedge x_1 \neq 0 \\ x_2 \neq x_3 \wedge x_2 \neq 0 \\ x_1 \neq x_3 \wedge x_3 \neq 0 \\ -2 \leq x_1 - x_2 \leq 2 \\ -2 \leq x_1 - x_3 \leq 2 \\ -2 \leq x_2 - x_3 \leq 2 \end{cases}$$

(b) Représentations de $m$ une fois $m^{\leq}$ close




$$\begin{cases} 1 \leq x_1, x_2, x_3 \leq 2 \\ x_1 \neq x_2 \wedge x_1 \neq 0 \\ x_2 \neq x_3 \wedge x_2 \neq 0 \\ x_1 \neq x_3 \wedge x_3 \neq 0 \\ -1 \leq x_1 - x_2 \leq 1 \\ -1 \leq x_1 - x_3 \leq 1 \\ -1 \leq x_2 - x_3 \leq 1 \end{cases}$$

(c) Représentations de $m$ normalisée partiellement

Figure 5.12 – Cas discret : illustration de la normalisation partielle d'une *d*DBM $m$. Ce résultat est à comparer avec la forme normale de $m$ : $\bot^{zoal}$.

Définition 5.4 (test d'inclusion). *On définit*

$$m \sqsubseteq^{zoal} m' \overset{déf}{\Leftrightarrow} \overline{m} \sqsubseteq^{ddbm} m'.$$

*On a :* $m \sqsubseteq^{zoal} m' \Leftrightarrow \gamma^{zoal}(m) \sqsubseteq^{\flat} \gamma^{zoal}(m')$.

*Démonstration.*

⇒ Par monotonicité de $\gamma^{zoal}$ vis-à-vis de $\sqsubseteq^{ddbm}$ (conséquence du Thm. 5.2, p. 88) et correction de l'opérateur de normalisation (Thm. 5.4, p. 98)

⇐ Par monotonicité de $\alpha^{zoal}$ on a $(\alpha^{zoal} \circ \gamma^{zoal})(m) \sqsubseteq^{ddbm} (\alpha^{zoal} \circ \gamma^{zoal})(m')$. Comme $(\alpha^{zoal} \circ \gamma^{zoal})(m) = \overline{m}$ par définition, il suit : $\overline{m} \sqsubseteq^{ddbm} \overline{m'}$ et, par correction de l'opérateur de normalisation, que $\overline{m} \sqsubseteq^{ddbm} m'$.

□

### 5.6.2 Union

À l'instar des zones, l'union de deux zones précisant alias n'est pas forcément une zone précisant alias. Cependant, si l'opérateur d'union sur les *d*DBM calcule une approximation correcte, celle-ci est trop large vis-à-vis de $\gamma^{zoal}$. En effet, cet opérateur ne gardant que les contraintes explicites que représentent ses opérandes, toute contrainte laissée implicite dans celles-ci peut être perdue dans l'union. On applique donc sur les représentants des deux zones précisant alias dont on souhaite approximer l'union l'opérateur de normalisation avant d'appliquer l'opérateur $\sqcup^{ddbm}$. On montre que l'on obtient non-seulement une *d*DBM représentant la plus petite zone précisant alias approximant l'union des deux autres, mais en plus qu'elle est en forme normale.

Définition 5.5 (opérateur d'union). *On définit l'union de deux zones précisant alias par :*

$$m \sqcup^{zoal} m' \overset{déf}{\Leftrightarrow} \overline{m} \sqcup^{ddbm} \overline{m}'.$$

*L'opérateur $\sqcup^{zoal}$ est la meilleure abstraction de $\sqcup^\flat$. De plus $m \sqcup^{zoal} m'$ est en forme normale.*

Pour prouver cette dernière propriété, il est nécessaire de s'intéresser à une propriété importante des *d*DBM closes, leur saturation. Celle-ci assure que lorsque $\neg m_{ij}^{\neq}$ il existe une valuation de $\gamma^{zoal}(m)$ où $x_i$ et $x_j$ sont égales, ou encore qu'il existe pour chaque contrainte d'inégalité que représente $m$, si $\mathbb{K} = \mathbb{Z}$, une valuation qui la sature. Lorsque $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$, on peut rencontrer des contraintes d'inégalités strictes. Si l'on suppose qu'il s'agit de la contrainte $x_i - x_j < m_{ij}^{\leq}$, alors la propriété de saturation devient : pour tout $\varepsilon \in \mathbb{K}$ positif aussi petit que l'on souhaite, il existe une valuation telle que la différence $m_{ij}^{\leq} - (\rho(x_i) - \rho(x_j))$ est bornée par $\varepsilon$. Ces contraintes strictes peuvent être explicites : c'est le cas lorsque $m_{ij}^{\neq} \wedge m_{ij}^{\leq} = 0$. Mais elle peuvent aussi être implicites, par exemple l'ensemble de contraintes :

$$x_i \leq x_k + 1 \wedge x_k \leq x_j \wedge x_k \neq x_j \tag{5.3}$$

induit clairement que $x_i < x_j + 1$, une contrainte stricte qu'une *d*DBM ne peut représenter. Pour décider si la propriété de saturation doit être stricte ou non entre deux variables $x_i$ et $x_j$, il suffit de regarder s'il existe un chemin dans le graphe de potentiels et de non-égalités de $i$ à $j$ passant par une inégalité stricte ($m_{k\ell}^{\neq} \wedge m_{k\ell}^{\leq} = 0$), comme nous l'apprend la normalisation des DBM représentant des contraintes strictes (Sec. 3.1.1.2).

Théorème 5.6 (saturation des *d*DBM closes). *Soit $m$ une dDBM en forme normale. Alors $\forall i, j \in [0, n]$ :*

- $\neg m_{ij}^{\neq} \Rightarrow \exists \rho \in \gamma^{zoal}(m)$ *tel que* $\rho(x_i) = \rho(x_j)$ ;

*et*

- $m_{ij}^{\leq} = +\infty \Rightarrow \forall M < +\infty, \exists \rho \in \gamma^{zoal}(m)$ *telle que* $\rho(x_i) - \rho(x_j) \geq M$ ;
- $m_{ij}^{\leq} < +\infty$ :
  - *si* $\mathbb{K} = \mathbb{Z}$ *alors* $\exists \rho \in \gamma^{zoal}(m)$ *telle que* $\rho(x_i) - \rho(x_j) = m_{ij}^{\leq}$ ;
  - *sinon* ($\mathbb{K} = \mathbb{Q}$ *ou* $\mathbb{R}$)
    - *si* $\exists (k, \ell)$ – *pouvant être égal à* $(i, j)$ – *tel que* $m_{k\ell}^{\neq} \wedge m_{kl}^{\leq} = 0 \wedge \exists \langle j, \ldots, \ell, k, \ldots, i \rangle$, *alors* $\forall \varepsilon > 0, \exists \rho \in \gamma^{zoal}(m)$ *telle que* $0 < m_{ij}^{\leq} - (\rho(x_i) - \rho(x_j)) \leq \varepsilon$ ;
    - *sinon,* $\exists \rho \in \gamma^{zoal}(m)$ *telle que* $\rho(x_i) - \rho(x_j) = m_{ij}^{\leq}$.

*Démonstration.* La preuve de cette propriété est très proche bien entendu de celles que l'on a donné pour établir que les matrices $m^*$ sont en formes normales. On se contente donc de donner la preuve de quelques cas :

- si $\neg m_{ij}^{\neq}$, alors par définition de $m^*$, $\exists k \in K$ tel que, pour $\mathbb{K} = \mathbb{Z}$, $m_{ij}^k \geq 0 \wedge m_{ji}^k \geq 0$, pour $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$, $m_{ij}^k \geq (0, \leq) \wedge m_{ji}^k \geq (0, \leq)$. On conclut par saturation des DBM et convexité des zones.
- si $m_{ij}^{\leq} = +\infty$, alors par définition de $m^*$, $\exists k \in K$ tel que $m_{ij}^k = +\infty$. On conclut par saturation des DBM.

□

On revient à la démonstration des propriétés de l'opérateur d'union défini ci-dessus (Déf. 5.5).

*Démonstration.* On souhaite montrer que $m \sqcup^{zoal} m' = \inf_{\sqsubseteq^{ddbm}}\{n \mid \gamma^{zoal}(n) \sqsupseteq^\flat \gamma^{zoal}(m) \sqcup^\flat \gamma^{zoal}(m')\}$.

- Si $\gamma^{zoal}(m) = \emptyset$ alors $\overline{m} = \bot^{zoal}$ et la propriété est vérifiée par définition de la forme normale de $m'$. Même raisonnement si $\gamma^{zoal}(m') = \emptyset$.
- Sinon, on montre d'abord que $\sqcup^{zoal}$ approxime $\sqcup^\flat$. Par correction de l'opérateur de normalisation on a $\gamma^{zoal}(m) \sqcup^\flat \gamma^{zoal}(m') = \gamma^{zoal}(\overline{m}) \sqcup^\flat \gamma^{zoal}(\overline{m}')$, et comme $\overline{m}, \overline{m}' \sqsubseteq^{ddbm} \overline{m} \sqcup^{ddbm} \overline{m}'$ par monotonicité de $\gamma^{zoal}$ on obtient $\gamma^{zoal}(\overline{m}) \sqcup^\flat \gamma^{zoal}(\overline{m}') \sqsubseteq^\flat \gamma^{zoal}(\overline{m} \sqcup^{ddbm} \overline{m}')$. Donc $\gamma^{zoal}(m \sqcup^{zoal} m') \sqsupseteq^\flat \gamma^{zoal}(m) \sqcup^\flat \gamma^{zoal}(m')$.
  Maintenant on suppose l'existence d'un *d*DBM $n$ telle que $\gamma^{zoal}(n) \sqsupseteq^\flat \gamma^{zoal}(m) \sqcup^\flat \gamma^{zoal}(m')$ et on montre que $m \sqcup^{zoal} m' \sqsubseteq^{ddbm} n$. On procède par cas :
  - Soit $i, j$ tels que $\overline{m}^{\leq}_{ij} < +\infty \wedge \overline{m}'^{\leq}_{ij} < +\infty \wedge \neg\overline{m}^{\neq}_{ij} \wedge \neg\overline{m}'^{\neq}_{ij}$. Par saturation des *d*DBM (Thm. 5.6) on a d'une part l'existence de $\rho \in \gamma^{zoal}(\overline{m})$ et de $\rho' \in \gamma^{zoal}(\overline{m}')$ telles que $\rho(x_i) - \rho(x_j) = \overline{m}^{\leq}_{ij}$ et $\rho'(x_i) - \rho'(x_j) = \overline{m}'^{\leq}_{ij}$. Comme $\rho, \rho' \in \gamma^{zoal}(n)$ alors $n^{\leq}_{ij} \geq \max(\overline{m}^{\leq}_{ij}, \overline{m}'^{\leq}_{ij})$. D'autre part, on a l'existence de $\rho'' \in \gamma^{zoal}(\overline{m})$ telle que $\rho''(x_i) = \rho''(x_j)$. Comme $\rho'' \in \gamma^{zoal}(n)$ alors $\neg n^{\neq}_{ij}$ et donc $n^{\neq}_{ij} \Rightarrow \overline{m}^{\neq}_{ij}, \overline{m}'^{\neq}_{ij}$.
  - les autres cas ($\overline{m}^{\leq}_{ij} = +\infty$, $\overline{m}^{\neq}_{ij}$, etc) sont similaires : l'appel à la propriété de saturation des *d*DBM permet de conclure au même résultat.

  Cette étude de cas permet d'établir que $\overline{m} \sqsubseteq^{ddbm} n$ et $\overline{m}' \sqsubseteq^{ddbm} n$ et donc que $\overline{m} \sqcup^{ddbm} \overline{m'} \sqsubseteq^{ddbm} n$.

□

### 5.6.3 Intersection

L'intersection de deux zones précisant alias est également une zone précisant alias. En utilisant simplement l'opérateur d'intersection sur les *d*DBM, on obtient une *d*DBM qui représente exactement l'intersection des deux zones précisant alias, qu'elles soient représentées par des *d*DBM closes ou non.

Définition 5.6 (opérateur d'intersection). *On définit l'intersection de deux zones précisant alias par :*

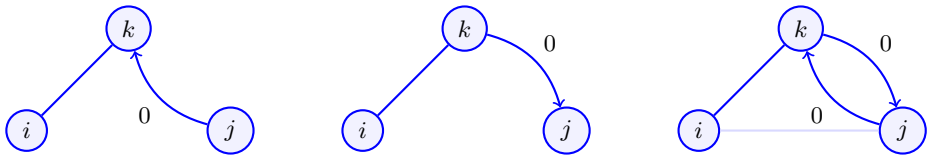
$$m \sqcap^{zoal} m' \overset{déf}{\Leftrightarrow} m \sqcap^{ddbm} m'.$$

*L'opérateur $\sqcap^{zoal}$ est l'abstraction exacte de $\sqcap^\flat$.*

*Démonstration.* Il est aisé de vérifier que $\gamma^{ddbm}(m \sqcap^{ddbm} m') = \gamma^{ddbm}(m) \sqcap^\flat \gamma^{ddbm}(m')$. □

La *d*DBM résultant de cette opération d'intersection n'est pas forcément en forme normale, même lorsqu'on l'applique à des *d*DBM qui le sont comme l'illustre la Figure 5.13.

Il est assez clair que les opérateurs décrits dans cette section donnent une structure de treillis aux zones précisant alias. Nous ne le démontrerons donc pas. Nous allons maintenant nous intéresser à la définition des opérations sémantiques du domaine abstrait.




(a) $x_i \neq x_k \leq x_j$ (b) $x_i \neq x_k \geq x_j$ (c) $x_i \neq x_k = x_j$

Figure 5.13 – Deux *d*DBM closes en (a) et (b) représentées par leurs graphes de potentiels et de non-égalités, suivies du résultat de leur intersection selon $\sqcap^{zoal}$ en (c). On notera que cette dernière n'est pas close : pour ce faire, elle doit indiquer que $x_i \neq x_j$, contrainte apparaissant ici d'une couleur plus claire.

## 5.7 Opérateurs sémantiques

On commence par donner un opérateur d'élargissement. On ne donne pas d'opérateur de rétrécissement dont la définition est aisée. On termine en définissant les fonctions de transfert.

### 5.7.1 Opérateur d'élargissement

Comme le domaine des zones, le domaine des zones précisant alias est de profondeur infinie. Il est donc nécessaire de fournir un opérateur d'élargissement pour assurer la convergence en un temps fini de l'itération approchant le point fixe des équations sémantiques du programme (Thm. 2.4, p. 17).

Nous nous contentons de reprendre l'opérateur d'élargissement des zones (Sec. 3.1.1.1) dont l'extension aux *d*DBM est triviale : toute contrainte de non-égalité qui n'est pas présente dans les deux zones précisant alias est oubliée.

On remarque que selon cette définition, l'élargissement $(x_i < x_j)\nabla(x_i \leq x_j)$ garde cette dernière contrainte lorsque $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$ alors qu'il la perd lorsque $\mathbb{K} = \mathbb{Z}$, la première contrainte s'écrivant $x_i - x_j \leq -1$. On souhaite donc instaurer dans ce cas un palier à l'élargissement en zéro, autrement dit que l'élargissement teste la stabilité de cette borne pour chaque différence $x_i - x_j$. Dans [CC92b], cette même idée est présentée pour permettre à l'opérateur d'élargissement des intervalles d'essayer d'inférer au moins le signe des variables. Elle est même généralisée à un ensemble fini de paliers, que des heuristiques simples peuvent déterminer, dont on teste la stabilité les uns après les autres en espérant améliorer la précision de l'analyse.

Bien que l'on utilise directement la notion de palier pour définir notre opérateur d'élargissement ici, nous ne proposons pas d'heuristiques pour trouver ceux-ci. Ce problème ne relevant pas particulièrement des contraintes de zones précisant alias, nous invitons le lecteur à se reporter à [BCC+02] et [BCC+03] pour des exemples de telles heuristiques.

Définition 5.7 (opérateur d'élargissement). *Soit $m, m'$ deux zones précisant alias et $T \in \mathcal{P}(\mathbb{K})$ un ensemble fini de paliers tel que $0 \in T$ si $\mathbb{K} = \mathbb{Z}$. On définit l'élargissement de $m$ par $m'$ comme suit :*

$$m \nabla^{zoal} m' = \begin{cases} m' & \text{si } m = \bot^{zoal} \\ (m''^{\leq}, m''^{\neq}) & \text{sinon} \end{cases},$$

$$\text{où } m''^{\leq}_{ij} = \begin{cases} m^{\leq}_{ij} & \text{si } m'^{\leq}_{ij} \leq m^{\leq}_{ij} \\ \min\{c \in T \cup \{+\infty\} \mid c \geq m'^{\leq}_{ij}\} & \text{sinon} \end{cases} \quad \text{et } m''^{\neq}_{ij} = m^{\neq}_{ij} \wedge m'^{\neq}_{ij}.$$

*Démonstration.* Selon la Définition 2.3 (p. 16) des opérateurs d'élargissement, deux propriétés sont à vérifier :

1. Il est aisé de vérifier que $m, m' \sqsubseteq^{zoal} m \nabla^{zoal} m'$.
2. Soit une suite $(m'^n)_{n\geq 0}$ de *d*DBM et la suite définie par $m^0 = m'^0$ et $m^{n+1} = m^n \nabla^{zoal} m'^{n+1}$ dont la convergence doit être assurée. On montre aisément par induction sur $n$ que chaque élément $m^n_{ij}$ de la *d*DBM $m^n$ ne peut prendre ses valeurs que dans l'ensemble fini $(T \cup \{m^0_{ij}, +\infty\}) \times \mathbb{B}$. Ainsi, la suite $(m^n)_{n\geq 0}$ ne peut prendre ses valeurs que dans un ensemble fini de matrices, et comme par conséquence du point 1 elle est croissante, sa convergence en un temps fini est assurée.

□

On notera qu'une suite d'applications de l'opérateur d'élargissement est au plus de hauteur $n^2 \times |T + 1|$. Celle hauteur maximale de l'opérateur est atteinte lorsque chaque borne est amenée l'une après l'autre à l'infini, en butant sur chacun des paliers.

**Normalisation des opérandes et convergence** Comme le montre très justement [Min06] pour les zones, l'utilisation de l'opérateur de normalisation interagit mal avec celui d'élargissement. En effet, si l'on est tenté pour améliorer la précision de l'analyse de normaliser les valeurs abstraites avant de les élargir, c'est-à-dire d'effectuer aux points d'élargissement la séquence $m^{n+1} = \overline{m^n} \nabla^{zoal} m'^n$, alors la convergence de l'analyse n'est plus garantie (Intuitivement, l'élargissement ne fait pas nécessairement disparaître des contraintes en repoussant à l'infini des coefficients : des contraintes implicites peuvent subsister et amener la normalisation à rétablir ces coefficients, annulant l'effet d'accélération). Bien entendu, notre domaine souffre du même problème. Comme l'auteur l'indique, on peut toujours normaliser $m'^n$, bien que ceci n'implique pas nécessairement une amélioration du résultat de l'analyse.

### 5.7.2 Fonctions de transfert

Il reste à définir l'ensemble des fonctions de transfert nécessaires pour l'analyse de nos programmes (*cf.* Préliminaires – Programmes considérés). On propose pour chacun d'eux différentes possibilités illustrant des compromis différents entre précision et coût. Enfin, quand cela est pertinent, on discute également l'effet sur la précision lorsque l'argument est précédemment normalisé.

#### 5.7.2.1 Opérateur « d'oubli »

L'opérateur d'oubli correspond, comme nous l'avions suggéré à la suite de la définition des domaines abstraits sémantiques (Déf. 2.6, p. 22), à une affectation non-déterministe d'une variable $x_k$. Cette affectation, que l'on note $x_k$ := ?, se définit simplement en enlevant de la zone précisant alias toute contrainte où $x_k$ apparaît.

Définition 5.8 (opérateur d'oubli). *Soit $m \in \mathcal{D}^{zoal}$. L'opération d'oubli laisse invariant $m = \bot^{zoal}$. Sinon elle est définie par :*

$$\{\!|x_k := ?|\!\}^{zoal}(m) = m' \text{ défini par } \begin{cases} m'^{\leq}_{ij} = \begin{cases} +\infty & si\ k = i\ xor\ k = j \\ 0 & si\ k = i = j \\ m^{\leq}_{ij} & sinon \end{cases} \\ m'^{\neq}_{ij} = \begin{cases} faux & si\ k = i \vee k = j \\ m^{\neq}_{ij} & sinon \end{cases} \end{cases}.$$

*Si $m$ est en forme normale, soit $\mathbb{K} = \mathbb{Z}$ et l'opérateur est exact, soit $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$ et l'opérateur n'est pas toujours exact. Par contre, quel que soit $\mathbb{K}$ le résultat de l'opération sera lui aussi en forme normale.*

*Démonstration.* La correction de l'opérateur, définie par $\gamma^{zoal}(\{\!|x_k := ?|\!\}^{zoal}(m)) \supseteq \{\rho \in Env \mid \exists c \in \mathbb{K}, \rho[x_k \mapsto c] \in \gamma^{zoal}(m)\}$ est évidente. Pour les autres propriétés :

- Lorsque $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$, comme certaines contraintes induites par une zone précisant alias ne peuvent être traduites dans le formalisme des *d*DBM, et donc que la normalisation ne peut les expliciter, l'opération d'oubli perdra parmi ces contraintes implicites celles qui dépendent des contraintes portant sur $x_k$. On prend pour exemple l'ensemble de contraintes à l'Équation 5.3 (p. 111) où l'on applique l'affectation indéterministe à $x_k$ : on perd alors le fait que $x_i \neq x_j + 1$.
- Si $\mathbb{K} = \mathbb{Z}$ on montre aisément que $\gamma^{zoal}(\{\!|x_k := ?|\!\}^{zoal}(m)) \subseteq \{\rho \in Env \mid \exists c \in \mathbb{K}, \rho[x_k \mapsto c] \in \gamma^{zoal}(m)\}$.
- Si $m$ est en forme normale, il est trivial que $m'$ l'est aussi.

□

Clairement, la précision maximum de l'opérateur d'oubli est obtenue au prix d'une opération de normalisation. Cependant, il suffirait d'expliciter uniquement les contraintes qu'impliquent les contraintes portant sur $x_k$ pour s'assurer de cette précision maximum. C'est ce que fait par exemple l'opérateur d'oubli des zones qui s'exécute en $\mathcal{O}(n^2)$, à comparer au coût de la normalisation en $\mathcal{O}(n^3)$ pour ce domaine.

De la même manière, pour les zones précisant alias, on peut dans le cas dense utiliser l'algorithme de normalisation (Fig. 5.5, p. 100) en restreignant l'ensemble $D$ des non-égalités visitées aux couples $(i, j)$ où $k = i \vee k = j$. L'opérateur d'oubli atteint alors la précision maximum, pour une complexité en temps de $\mathcal{O}(n^3)$.

#### 5.7.2.2 Affectations

Au vu des contraintes que peuvent représenter les *d*DBM, seules les abstractions des affectations $x_k := \langle expression \rangle$, où $\langle expression \rangle$ est de la forme $x_i + c$ ou $c$, peuvent être précises. Pour les autres affectations, l'opérateur d'oubli (Déf. 5.8) est une abstraction correcte que l'on améliorera dans certains cas particuliers.

Parmi les cas que l'on peut traiter précisément, on différencie les affectations inversibles des autres. Pour ces dernières, la définition de l'opérateur (Déf. 5.9) est simple : faire appel à celui d'oubli sur $x_k$, puis encoder respectivement l'égalité $x_k = x_\ell + c$ ou $x_k = c$. On voit que l'on ne peut pas procéder de la sorte pour les affectations inversibles, puisque l'on commence par oublier les informations sur $x_k$. On utilise une variable fraîche $x_\ell \notin V$

qui va représenter la variable $x_k$ après l'affectation : ainsi, on étend la *d*DBM pour inclure $x_\ell$ et l'on ajoute la contrainte $x_\ell = x_k + c$, *i.e.* $m^{\leq}_{k\ell} = c \wedge m^{\leq}_{\ell k} = -c$. On appelle alors l'opération de normalisation avant d'éliminer $x_k$ de la *d*DBM. Il ne reste qu'à renommer $x_\ell$ en $x_k$.

Définition 5.9 (opérateurs d'affectations simples). *Soit $m \in \mathcal{D}^{zoal}$. L'opération d'affectation laisse invariant $m = \bot^{zoal}$. Sinon elle est définie comme suit, où l'opération de normalisation peut être remplacée par la normalisation partielle autour de $x_k$ évoquée précédemment.*

*Pour les affectations non-inversibles, on a :*

- $\{\!|x_k := x_\ell + c|\!\}^{zoal}(m) = m'$ *où* $\left\{ \begin{array}{l} m'^{\leq}_{ij} = \left\{ \begin{array}{ll} c & si\ i = k \wedge j = \ell \\ -c & si\ i = \ell \wedge j = k \\ (\{\!|x_k := ?|\!\}^{zoal}(\overline{m}))^{\leq}_{ij} & sinon \end{array} \right. \\ m'^{\neq}_{ij} = \{\!|x_k := ?|\!\}^{zoal}(\overline{m}))^{\neq}_{ij} \end{array} \right.$
- $\{\!|x_k := c|\!\}^{zoal}(m) = m'$ *où* $\left\{ \begin{array}{l} m'^{\leq}_{ij} = \left\{ \begin{array}{ll} c & si\ i = k \wedge j = 0 \\ -c & si\ i = 0 \wedge j = k \\ (\{\!|x_k := ?|\!\}^{zoal}(\overline{m}))^{\leq}_{ij} & sinon \end{array} \right. \\ m'^{\neq}_{ij} = \{\!|x_k := ?|\!\}^{zoal}(\overline{m}))^{\neq}_{ij} \end{array} \right.$ .

*Pour les affectations inversibles, on a :*

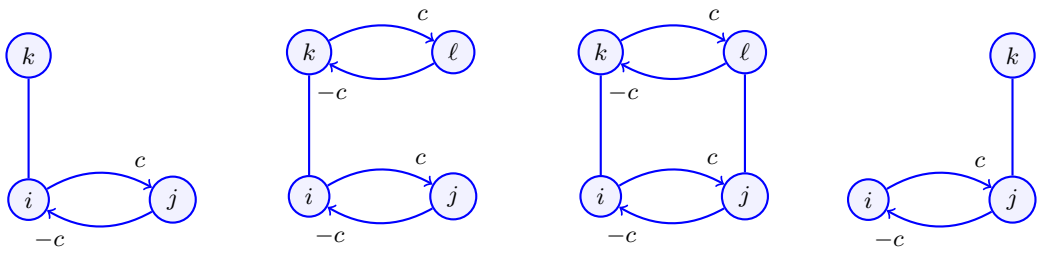
- $\{\!|x_k := x_k + c|\!\}^{zoal}(m) = \left(\exists x_k(\overline{m[k\ell \mapsto c, \ell k \mapsto -c]})\right)[x_\ell \to x_k]$ .

*Si $\mathbb{K} = \mathbb{Z}$ l'opération d'affectation est exacte. Pour les affectation inversibles elle renvoi une dDBM en forme normale.*

La preuve de la correction de l'opérateur et celles des remarques associées ne soulevant pas de problème particulier, on ne les présente pas.

Si l'on se souvient du traitement des affectations inversibles pour les zones (Sec. 3.1.1.1), on pourrait être surpris de ne pas trouver un algorithme similaire, ne nécessitant pas de variable supplémentaire, ni d'opération de normalisation. En effet, pour les zones on peut définir localement les contraintes portant sur $x_k$ (si $x_i \leq x_k + 5$ alors après affectation $x_i \leq x_k - c + 5$) telles que l'opérateur soit exact. Or les interactions possibles dans les zones précisant alias entre les inégalités et les non-égalités, rendent impossible pour $\mathbb{K} = \mathbb{Z}$ toute définition locale : on prend pour exemple celui de la Figure 5.7 (p. 106) où il faudrait trouver directement que $x_j \leq x_k - c - 2$. De même, pour $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$ une normalisation de $m^{\leq}$ est nécessaire avant de pouvoir espérer définir localement les non-égalités induites par l'affectation. Ainsi, le coût de notre opérateur pour les affectations inversibles n'est pas exagéré : tout au plus pourrait-il se passer de l'utilisation d'une variable fraîche.

On illustre ces propos par l'exemple de la Figure 5.14 où l'on voit les différentes étapes de l'opération d'affectation inversible appliquée à la zone précisant alias représentée en (a). Une fois $x_\ell$ introduite et la relation $x_k = x_\ell + c$ encodée (b), on appelle la normalisation qui explicite que $x_\ell$ est non-égal à $x_j$ (c) Enfin, on élimine $x_k$ et on renomme $x_\ell$ en $x_k$ (d).




(a) $x_i = x_j - c \wedge x_k \neq x_i$

(b) $x_k = x_\ell - c \wedge x_i = x_j - c \wedge x_k \neq x_i$

(c) $x_k = x_\ell - c \wedge x_i = x_j - c \wedge x_k \neq x_i \wedge x_\ell \neq x_j$

(d) $x_i = x_j - c \wedge x_k \neq x_j$

Figure 5.14 – Une zone précisant alias à laquelle on applique l'opération d'affectation $x_k := x_k + c$ (Déf. 5.9)

Enfin, il est à noter que le résultat de l'opérateur pour les affectations non-inversibles n'est pas en forme normale. Cependant, s'il a été appliqué sur une *d*DBM en forme normale, l'opérateur d'oubli préservant cette caractéristique (Déf. 5.8, p. 115), on peut faire usage de la normalisation incrémentale (Sec. 5.4.2.2) lorsque $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$, puisque seules les valeurs de la colonne et de la ligne $k$ ont été modifiées. On récupère ainsi une forme normale en $\mathcal{O}(n^3)$.

Lorsque $\langle expression \rangle$ est plus complexe, deux choix s'offrent à nous. Soit perdre toute information sur $x_k$ : les opérations d'affectation et d'oubli se confondent. Soit utiliser un domaine abstrait plus précis momentanément, dans lequel on procède à l'opération d'affectation et dont est extrait du résultat toutes les contraintes de zones précisant alias. On prend pour exemple l'ensemble de contraintes $x_i \in [-5, 5] \wedge x_i \neq 0 \wedge x_k = x_\ell$ et l'affectation $x_k := x_k + x_i$. En utilisant le domaine abstrait sémantique des polyèdres (Sec. 3.1), on obtient entre autres que $x_k - x_\ell \in [-5, 5]$. Si l'on veut également découvrir des contraintes de non-égalité – hors de celles que des contraintes de potentiel induisent trivialement, au vu des domaines disponibles actuellement, on en est réduit au cas par cas. Pour notre exemple, il est aisé de voir que $x_k - x_\ell \neq 0$.

C'est le premier cas que l'on expose ci-dessous. On donne également trois autres cas particuliers :

- si $m^{\leq}_{k\ell} = m^{\leq}_{\ell k} = 0 \wedge m^{\neq}_{i0}$ alors $(\{\!|x_k := x_k + x_i|\!\}^{zoal}(m))^{\neq}_{k\ell} = vrai$ ;
- si $m^{\neq}_{ij}$ alors $(\{\!|x_k := x_i - x_j|\!\}^{zoal}(m))^{\neq}_{k0} = vrai$ ;
- si $c \neq 0$ alors $(\{\!|x_k := c.x_i|\!\}^{zoal}(m))^{\neq}_{k0} = m^{\neq}_{i0}$ ;
- si $m^{\neq}_{\ell 0}$ alors $(\{\!|x_k := -x_\ell|\!\}^{zoal}(m))^{\neq}_{k\ell} = vrai$.

#### 5.7.2.3 Conditionnelles

Il reste à définir les abstractions des conditionnelles afin d'être exhaustif. On donne la définition de l'opérateur pour les conditionnelles (Déf. 5.10) qui peuvent être précisément abstraites pour les zones précisant alias : $\pm x_k \leq c, x_k \neq 0, x_k - x_\ell \leq c$ et $x_k \neq x_\ell$. Il s'agit simplement d'ajouter la contrainte correspondante dans la *d*DBM.

Définition 5.10 (opérateur de conditionnelles simples). *Soit $m \in \mathcal{D}^{zoal}$. L'abstraction de tout test laisse invariant $m = \bot^{zoal}$. Sinon, on définit les cas simples suivants :*

- $\{\!|x_k \leq c|\!\}^{zoal}(m) = m'$ *où* $\begin{cases} m'^{\leq}_{ij} = \begin{cases} \min(m^{\leq}_{ij}, c) & si\ i = k \wedge j = 0 \\ m^{\leq}_{ij} & sinon \end{cases} \\ m'^{\neq}_{ij} = m^{\neq}_{ij} \end{cases}$ ;
- $\{\!|x_k \neq 0|\!\}^{zoal}(m) = m'$ *où* $\begin{cases} m'^{\leq}_{ij} = m^{\leq}_{ij} \\ m'^{\neq}_{ij} = \begin{cases} vrai & si\ i = k \wedge j = 0 \\ m^{\neq}_{ij} & sinon \end{cases} \end{cases}$ ;
- $\{\!|x_k - x_\ell \leq c|\!\}^{zoal}(m) = m'$ *où* $\begin{cases} m'^{\leq}_{ij} = \begin{cases} \min(m^{\leq}_{ij}, c) & si\ i = k \wedge j = \ell \\ m^{\leq}_{ij} & sinon \end{cases} \\ m'^{\neq}_{ij} = m^{\neq}_{ij} \end{cases}$ ;
- $\{\!|x_k \neq x_\ell|\!\}^{zoal}(m) = m'$ *où* $\begin{cases} m'^{\leq}_{ij} = m^{\leq}_{ij} \\ m'^{\neq}_{ij} = \begin{cases} vrai & si\ i = k \wedge j = \ell \\ m^{\neq}_{ij} & sinon \end{cases} \end{cases}$ .

*Ces opérateurs sont exacts.*

Là encore, si $m$ était en forme normale précédemment à l'opération, et si $\mathbb{K} = \mathbb{Q}$ ou $\mathbb{R}$, on peut utiliser l'opération de normalisation incrémentale.

Pour ce qui est des conditionnelles plus complexes, bien entendu l'identité est toujours correcte. *A contrario*, on peut comme pour les affectations, avoir recours à un domaine abstrait plus expressif pour améliorer la précision.

Entre ces deux extrêmes, l'opérateur d'union $\sqcup^{zoal}$ sera utilisé pour abstraire les disjonctions de conditionnelles, celui d'intersection $\sqcap^{zoal}$ pour abstraire les conjonctions, notamment celles que cachent $x_k = x_\ell + c$ ou encore $x_k < x_\ell$. Plus intéressant, en introduisant une nouvelle variable fraîche $x_\ell \in V$, on peut également prendre en compte les conditionnelles $x_k \neq c$ et $x_k - x_i \neq c$ où $c \neq 0$. Pour les premières, on les transforment en $x_k \neq x_\ell \wedge x_\ell = c$, pour les secondes, en $x_k \neq x_\ell \wedge x_\ell = x_i + c$.

On a à présent tout en main pour analyser un programme avec le domaine abstrait des zones précisant alias. On termine ce chapitre en donnant de rapides exemples et en discutant certains points et les résultats obtenus.

## 5.8 Conclusion

Nous avons défini dans le détail un nouveau domaine numérique abstrait manipulant à la fois des contraintes de potentiel et des contraintes de non-égalité entre paires de variables. Généralement, la convexité est une des clés d'une complexité maitrisée : si l'on échappe pas à cette règle empirique, puisque le test de satisfaisabilité et la normalisation sont exponentielles dans le cas arithmétique, on a montré que ces opérations s'effectuaient en $\mathcal{O}(n^4)$ dans le cas dense et donné des algorithmes avec la même complexité pour approximer le premier cas.

Si on détient à présent un domaine répondant apparemment à nos objectifs (Sec. 5.1), c'est-à-dire inférer des non-égalités et ce, avec une algorithmique « raisonnable », nous n'avons que quelques exemples de programmes où de tels invariants sont découverts, et encore ne s'agit-il quasiment que d'exemples *ad hoc*. Seul un exemple d'algorithme d'exclusion mutuelle de deux programmes, donné à la Figure 5.16, est tiré de la littérature [MP95] et relève d'une forme d'adressage.

Ainsi, plutôt que de discuter par exemple des bénéfices qu'aurait une représentation creuse de nos matrices, et pour lesquels on pourra se reporter à [Min04] pour la matrice de bornes et dont on ne doute pas du bénéfice pour la matrice de non-égalités, c'est bien du pouvoir de déduction et d'expressivité qu'il nous faut discuter.

**Déduction et expressivité** On pourrait augmenter l'expressivité en espérant pouvoir déduire pour plus de programmes des invariants de non-égalités. Sans doute aurions-nous pu nous coupler à des contraintes de zones strictes, et ainsi avoir une plus jolie propriété de saturation (Thm. 5.6, p. 111) en pouvant représenter directement que $x_i < 4$ lorsque $x_i < x_j \wedge x_j = 4$. Ou encore aurions-nous pu nous coupler à des contraintes d'octogones, et exprimer du même coup des non-égalités de la forme $x_i \neq -x_j$, probablement pour une même complexité. On aurait alors découvert, autrement que par des cas spéciaux de l'affectation, que $x \neq y \wedge x \neq 0$ en tête de boucle (ligne **5**) du programme *ad hoc* donné à la Figure 5.15(a).

```
1 y := 1 ; x := f() ;
2 if x = y ∨ x = 0 then
3 |  ...
4 else
5 |  while vrai do
6 |  |  x := −x + 1
```

(a)

```
1 if x = y then
2 |  ...
3 else
4 |  while vrai do
5 |  |  z := f() ;
6 |  |  if x ≤ y then
7 |  |  |  if y ≤ z then y := z else x := z
8 |  |  else if x ≥ y then
9 |  |  |  if x ≤ z then x := z else y := z
```

(b)

Figure 5.15 – Deux exemples *ad hoc* de programmes où un invariant de non-égalité est inféré – $x \neq y$ en tête de boucle – par le domaine des zones précisant alias

Ceci étant, aucun nouveau programme intéressant ne serait traité par ces domaines à l'expressivité enrichie. De plus, si on se penche sur leur pouvoir de déduction on voit qu'il est sur-dimensionné. En effet, lorsqu'on regarde les quelques exemples que l'on a, ils déduisent pratiquement tous une non-égalité $x_i \neq x_j$ à un point de contrôle cible de plusieurs chemins, pour lesquels il est montré soit que $x_i < x_j$, soit que $x_i > x_j$. Cette situation se retrouve dans l'exemple *ad hoc* donné à la Figure 5.15(b) en tête de boucle (ligne **4**). La variable $z$ reçoit une valeur indéterministe et selon différents cas est affectée

à $x$ ou $y$ de telle manière qu'après la conditionnelle à la ligne **7**, on a $x < y$ et après celle de la ligne **9**, on a $y < x$. Les non-égalités déduites dans ces conditionnelles viennent uniquement de la transitivité des inégalités strictes, comme pour la plupart de nos exemples d'ailleurs, ce qui laisserait entendre qu'une déduction affaiblie, en l'occurrence aux cas (b), serait suffisante en pratique pour l'expressivité que l'on a et on aurait alors un domaine dont les opérations seraient en $\mathcal{O}(n^3)$.

Il reste que dans des exemples plus proches de la réalité, l'apparition d'une non-égalité est souvent la conséquence d'invariants plus forts, par exemple de l'union de $x_i - x_j = -1$ et $x_i - x_j = 1$. C'est le cas pour l'algorithme d'exclusion mutuelle (Fig. 5.16), dont on analyse le produit asynchrone des automates interprétés représentant ses deux processus en parallèle. À la première itération de l'analyse, on a $(1 = x = y + 1)$ au point de contrôle (①,❶). Seule la trace menant à l'exécution de la section critique du premier processus est possible. Alors, on a $(0 = x = y - 1)$ en (②,❶). L'exécution de la section critique du second processus est alors possible et on trouve $(1 = x = y + 1)$ en (①,❷). Au final, notre analyse découvre l'invariant $(0 \leq x, y \leq 1 \wedge x \neq y)$ en (①,❶), qui assure que les processus ne sont pas tous les deux bloqués et qu'ils ne peuvent entrer tous les deux dans leurs sections critiques.

```
x := 1 ;                                  y := 0 ;
while vrai do                             while vrai do
  | ① [ while x ≤ 0 do ; x := x − 1 ] ;  ‖  | ❶ [ while y ≤ 0 do ; y := y − 1 ] ;
  | //section critique                     | //section critique
  | y := y + 1                             | x := x + 1
```

Figure 5.16 – Un algorithme assurant l'exclusion mutuelle de deux processus avec des sémaphores [MP95]. Les crochets indiquent une instruction atomique.

Si on peut être satisfait de ce résultat, on remarque qu'un produit réduit (Sec. 2.3) entre le domaine des zones et celui des congruences de zones (Sec. 3.1, Fig. 3.2(c)) aurait permis d'exprimer le résultat de l'union de $x - y = -1$ et $x - y = 1$ avec autant de précision que le domaine des zones précisant alias : $(x_i - x_j \in [-1, 1] \wedge x_i \neq x_j) \Leftrightarrow (x_i - x_j \in [-1, 1] \wedge x_i - x_j = 0[2])$. Si cela ne veut pas dire que les domaines de congruences existant permettent toujours de capturer l'invariant de non-égalité que notre domaine infère (prendre pour exemple $k \neq 0$ et l'union de $x_i - x_j = -k$ et de $x_i - x_j = k$), on remarque que les invariants complexes qui nous manquent, dans les exemples que l'on n'arrive pas à traiter, pour montrer une contrainte de non-égalité, sont souvent des invariants de congruence : par exemple pour un buffer circulaire de capacité $n$ et dont on note $b$ la position de la première donnée et $t$ celle de la première cellule disponible, on souhaiterait montrer que lorsque le buffer n'est ni vide ni plein, on a $b \neq t$. Nous arrivons à prouver cet invariant seulement sur une version éclatée du programme, bien loin de celle qu'écrirait un programmeur, où toutes les positions possibles de $b$ et de $t$ vis-à-vis des bornes du buffer sont explicitées. Or, prouver que $t - b$ est égal au nombre d'éléments dans le buffer modulo $n$, sur une version classique du programme *i.e.* codée à l'aide de l'opération modulo, est à l'inverse envisageable.

En conclusion, plutôt que de chercher à augmenter l'expressivité, il serait intéressant de regarder les résultats de différents produits réduits entre le domaine des zones précisant alias, dont on aurait réduit le pouvoir de déduction, et notamment des do-

maines de congruences. Il est certain que ces derniers sont beaucoup plus appropriés pour l'analyse fine des indices de tableaux (*cf.* par exemple la thèse de Masdupuy [Mas93]).

Enfin, comme nous allons le voir dans le chapitre suivant et *a contrario* de notre vision de départ, nous ne nous servirons pas du domaine des zones précisant alias pour notre analyse du contenu des tableaux, pour laquelle on va préférer considérer la disjonction de tous les relations possibles entre les variables qui indicent un tableau ($i < j \vee i = j \vee i > j$), et ainsi contourner le problème des alias.

# Chapitre 6

# Un domaine abstrait sémantique du contenu des tableaux

Bien que l'analyse des accès aux tableaux, pour vérifier qu'ils en respectent les bornes, ait été une des motivations des premiers travaux sur l'interprétation abstraite [CC76], l'analyse du contenu des tableaux n'a été considérée que récemment (Ch. 4). La raison est bien sûr que ce problème est en général très difficile : la sémantique des tableaux est complexe, en particulier de par le phénomène d'alias des variables d'indices, et les tableaux peuvent représenter un grand nombre de variables, voire un nombre inconnu de variables. Parmi les analyses proposées ces huit dernières années (Sec. 4.4), aucune n'est à la fois entièrement automatique et suffisamment polyvalente et précise pour donner des résultats intéressants sur des programmes effectuant des traitements de nature différente sur le contenu des tableaux.

## 6.1 Objectifs

Comme on l'a déjà exposé en introduction (Sec. 1.2), on souhaite donc concevoir une analyse capable de découvrir automatiquement les invariants sur le contenu des tableaux de programmes qui initialisent, trient des tableaux, ou encore recherchent un élément dans des tableaux.

Bien entendu, on restreint cet objectif à une classe de programmes. Cela ne veut pas dire que notre analyse devra être précise sur tout programme de cette classe mais qu'elle sera sûrement très imprécise sur tout programme n'appartenant pas à cette classe. Nous avons déjà présenté cette classe de programmes dans les préliminaires (Sec. « Programmes considérés ») : notamment, absence de mémoire dynamique et tableaux à une seule dimension, pouvant être indexés uniquement par une constante ou l'addition d'une variable et d'une constante. Ces restrictions éliminent des programmes complexes, comme par exemple ceux qui utilisent un tableau pour stocker un arbre binaire. Par contre, dans cette classe de programmes simples, il reste la majorité des traitements effectués en pratique sur des tableaux. Notamment, on retrouve dans cette classe l'ensemble des programmes utilisés dans la littérature pour valider expérimentalement les analyses du contenu des tableaux (Sec. 4.4). Si l'on jette de nouveau un œil à des programmes de cette classe, en feuilletant le Chapitre 7, on se souvient que certains sont loin d'être simples à analyser (Fig. 7.3, 7.4, 7.5 et 7.8).

Enfin, comme on l'a déjà évoqué, on ne cherchera pas à découvrir des invariants de la seconde classe de propriétés de tableaux, que l'on avait définit à la Section 4.1 (propriétés s'exprimant sur des mutliensembles de valeurs). L'objectif sera donc de choisir l'expressivité la plus forte possible au sein de la première classe de propriétés de tableaux (Éq. 4.1, p. 46) sans compromettre les caractères automatique et polyvalent de l'analyse, qui sont prioritaires à la précision. Cela dit, au vu des programmes que l'on souhaite analyser, il est certain que l'analyse devra être en mesure de découvrir des propriétés unaires comme relationnelles sur le contenu des tableaux.

Pour illustrer comment on compte atteindre notre objectif, on déroule ici l'analyse définie dans ce chapitre sur un des programmes les plus simples que l'on souhaite analyser avec succès.

### 6.1.1 Exemple illustrant le fonctionnement de l'analyse

On utilise des formalismes quelque peu différents et un peu moins puissants que par la suite, mais l'idée générale de l'analyse est respectée. On déroule l'analyse du programme qui copie un tableau $b$ dans un tableau $a$ de même taille (Fig. 6.1). L'objectif est de montrer que $a$ et $b$ sont égaux, c'est-à-dire que le contenu du tableau $a$ est égal à celui du tableau $b$, *cellule par cellule*.

```
1 i := 1 ;
2 while i ≤ n do
3 |   a[i] := b[i] ;
4 |   i := i + 1
```

Figure 6.1 –

Comme proposé dans [GRS05] (Sec. 4.3.2), on commence par considérer une découpe des tableaux en tranches symboliques, permettant de représenter exactement le résultat de l'affectation à la cellule $a[i]$. Voyant que la variable d'indice $i$ va potentiellement accéder à toutes les cellules dans l'intervalle $[1, n]$, on considère les trois tranches suivantes : $\varphi_1 = [1, i-1]$, $\varphi_2 = [i, i]$ et $\varphi_3 = [i+1, n]$, partitionnant l'ensemble de ces cellules.

Une fois ce travail de découpage effectué, vient l'analyse d'atteignabilité elle-même. L'idée consiste alors, en un point de contrôle, à associer à chacune de ces tranches $\varphi_p$ une propriété $\psi_p$ qui exprime les valeurs possibles dans le tableau $a$ et dans le tableau $b$ sur cette tranche, à ce point de contrôle. On utilise un domaine abstrait $\mathcal{D}$ pour représenter ces propriétés $\psi_p$. Ainsi, puisqu'on a pas d'information au début du programme sur le contenu des tableaux $a$ et $b$, on prend $\psi_1 = \psi_2 = \psi_3 = \top$ au point de contrôle initial.

Déroulons la première itération. Après l'initialisation de $i$ à 1, on a $(i = 1)$ et donc que $\varphi_2$ représente la première cellule des tableaux $a$ et $b$. Ce qui veut aussi dire que $\varphi_1 = [1, i-1]$ ne représente alors aucune cellule. Ainsi, le contenu du tableau n'a pas été modifié, on a toujours $\psi_2 = \psi_3 = \top$, mais on pourrait prendre pour $\psi_1$ n'importe quelle propriété. On prend la plus forte, $\bot$. L'entrée dans la boucle nous donne $(i = 1 \leq n)$ et les mêmes propriétés $\psi_p$, puis vient l'affectation $a[i] := b[i]$. Comme prévu, on ne modifiera que $\psi_2$, laissant inchangée l'information obtenue sur les autres cellules du tableau. Pour exprimer que $a[i] = b[i]$ au point de contrôle suivant, on utilise la propriété relationnelle $\psi_2 = (a = b)$ – où les variables de tableaux $a$ et $b$ sont considérées comme des variables scalaires, car la sémantique que l'on donne aux propriétés $\psi_p$ est en fait :

$$\forall \ell, \ell \in \varphi_p \Rightarrow \psi_p[a[\ell]/a][b[\ell]/b].$$

Ainsi, la sémantique de la propriété $\psi_p$ s'entend *cellule par cellule* sur la tranche $\varphi_p$, ce qui est exactement ce dont on a besoin pour exprimer les invariants de ce programme :

par exemple à ce point de contrôle que $\forall \ell, \ell \in [1, i-1] \Rightarrow a[\ell] = b[\ell]$ ou encore que $\forall \ell, \ell \in [i, i] \Rightarrow a[\ell] = b[\ell]$, *i.e.* $a[i] = b[i]$. Si on reprend le calcul itératif, on avait donc avant l'incrémentation de $i$, $(i = 1 \leq n)$ et $\psi_1 = \bot, \psi_2 = (a = b)$ et $\psi_3 = \top$. Dans le cas général (*i.e.* où $1 \leq i \leq n$) après l'incrémentation de $i$, les cellules que représente $\varphi_1$ sont celles que $\varphi_1$ et $\varphi_2$ représentaient ; et les cellules que représentent $\varphi_2$ et $\varphi_3$ sont celles que représentait $\varphi_3$. Ainsi, après une telle instruction on associe à la tranche $\varphi_1$ la borne supérieure (dans $\mathcal{D}$) des propriétés $\psi_1$ et $\psi_2$ et aux tranches $\varphi_2$ et $\varphi_3$ la propriété $\psi_3$. Ce qui nous donne, au dernier point de contrôle de la boucle, $(i = 2)$, $\psi_1 = \bot \sqcup (a = b) = (a = b)$ et $\psi_2 = \psi_3 = \top$. En clair que $a[1] = b[1]$, et que d'avoir choisi $\bot$ pour $\psi_1$ lorsque $\varphi_1$ ne représentait pas de cellule, était astucieux.

On applique une stratégie récursive, on entre dans une seconde itération de la boucle. La valeur abstraite $(i = 1), \psi_1 = \bot, \psi_2 = \psi_3 = \top$ est élargie par celle que l'on vient de donner, la tête de boucle étant un point d'élargissement. La propriété sur $i$ devient $(1 \leq i \leq n+1)$ et comme il s'agit d'élargir simplement tranche par tranche les propriétés $\psi_p$, on a $(1 \leq i \leq n+1), \psi_1 = (a = b)$ et $\psi_2 = \psi_3 = \top$ en tête de boucle. En clair, $(1 \leq i \leq n+1) \wedge \forall \ell, \ell \in [1, i-1] \Rightarrow a[\ell] = b[\ell]$. Cette valeur abstraite est stable : après l'affectation à $a[i]$ on aura $(1 \leq i \leq n+1), \psi_1 = \psi_2 = (a = b)$ et $\psi_3 = \top$ et après l'incrémentation de $i$ on retombe sur la même valeur abstraite. De plus il s'agit de l'invariant de boucle exact du programme.

Enfin, au point de contrôle final du programme, atteignable lorsque $i \geq n+1$, on a $(1 \leq i = n+1), \psi_1 = (a = b)$ et $\psi_2 = \psi_3 = \top$, ce qui s'interprète par $\forall \ell, 1 \leq \ell \leq n \Rightarrow a[\ell] = b[\ell]$. L'analyse a atteint son objectif.

Remarque 6.1. *(Déjà formulée sur les travaux de [Cou03] et [GRS05]) On notera que l'analyse a porté non seulement sur le contenu des tableaux mais aussi sur les variables d'indices du programme : on voit, lors de la première itération, que si on n'avait pas traité la trace particulière où $i = 1$, l'analyse n'aurait pas pu découvrir l'invariant de boucle sur le contenu des tableaux $a$ et $b$. Les analyses numériques et du contenu ne peuvent se faire l'une après l'autre. Ici la pré-analyse numérique nous aurait informés que dans la boucle $i \in [1, n]$. L'analyse de contenu aurait bien trouvé après l'affectation à $a[i]$ que $a[i] = b[i]$ pour $i \in [1, n]$, mais n'ayant aucune information sur les cellules $[1, i-1]$ elle aurait perdu toute information sur le contenu des tableaux après l'incrémentation de $i$.*

**Plan explicatif** Ainsi notre analyse se compose de deux analyses consécutives, sur le modèle proposé dans [GRS05] : une première analyse de partitionnement des tableaux en tranches, suivie d'une seconde analyse, l'analyse principale, qui associe à chacune de ces tranches une propriété, relationnelle, vérifiée par le contenu des tableaux sur cette tranche. On commence par discuter ce choix de méthode, et définir la sémantique des propriétés attachées aux tranches (Sec. 6.2) par l'analyse principale. On fait en sorte que les tranches des partitions et les propriétés attachées à ces tranches, puissent être représentées par les éléments de domaines abstraits (Sec. 6.2.1), ceux-ci devenant des paramètres de l'expressivité de l'analyse. Ensuite, on définit l'analyse principale : d'abord ses valeurs abstraites (Sec. 6.2.2), définies pour une partition donnée, puis la concrétisation de ses valeurs abstraites (Sec. 6.3), et leurs normalisation (Sec. 6.4). Le résultat de cette normalisation n'est pas une forme canonique, mais s'est avéré en pratique suffisamment précis. Enfin, il ne reste plus que la définition des opérateurs sémantiques de

cette analyse principale (Sec. 6.6). Elle intervient après la définition de notre analyse de partitionnement (Sec. 6.5), là aussi par interprétation abstraite, car cette analyse génère des partitions qui facilitent l'extraction d'informations des valeurs abstraites de l'analyse principale, ce qui est utile pour définir les opérations sémantiques de cette dernière.

Nous concluons ce chapitre (Sec. 6.7) en donnant un exemple détaillé et complet de l'analyse d'un programme simple. C'est au chapitre suivant (Sec. 7.2) que l'on discutera des résultats obtenus par notre analyse. Ces résultats sont très bons et répondent à notre objectif.

Les travaux relatés dans ce chapitre ont fait l'objet d'une publication à la conférence PLDI [HP08].

## 6.2 Domaine et représentations considérés

Les classes de propriétés utilisées par les analyses du contenu des tableaux existantes (Chapitre 4) sont soit très peu expressives, et l'analyse est alors trop imprécise pour être satisfaisante, soit très expressives, au point que l'analyse ne peut s'affranchir d'une aide humaine. Ainsi, avant même de définir notre domaine abstrait nous souhaitons choisir soigneusement la nouvelle classe de propriétés sur laquelle il sera bâtit. De notre point de vue, l'objectif n'est pas tant de construire une classe rassemblant des propriétés très complexes que de choisir judicieusement leur expressivité pour obtenir une analyse entièrement automatique. Nous avons choisi :

– d'une part de se restreindre à des propriétés de tableaux exprimés avec une seule variable quantifiée $\ell$. Dès lors on s'empêche par exemple d'analyser le tri à bulle, qui fait partie de ceux proposés dans la littérature [1]. Par contre on remarque que les invariants de nombreux programmes, comme des programmes de tri, peuvent être découvert avec une seule variable quantifiée dès lors que l'on peut exprimer des relations entre des cellules distantes d'une constante. Si on reprend la classe générale de propriétés portant sur le contenu des tableaux présentée à l'Équation 4.1, cette restriction se définie en fixant $f_i = \ell + c_i, c_i \in \mathbb{Z}$. À des fins de clarté, nous ajoutons une autre restriction : en maintenant une séparation stricte entre les variables de contenu et les variables d'indices dans nos propriétés, on simplifie la définition du domaine abstrait sémantique. Le cas où cette séparation est gênante est d'ailleurs un cas particulier où $\mathbb{K} = \mathbb{Z}$, pour lequel on discutera l'extension de notre analyse pour le traiter (Sec. 7.2.4). Cette séparation se traduit par l'interdiction d'utiliser $\ell$ dans nos propriétés sur le contenu.

  Définition 6.1 (Propriétés de tableaux). *Une propriété sur le contenu des tableaux est une implication de la forme suivante, où $i_k \in I$, $a_k \in A$ et $x_k \in V$ et $\varphi$ et $\psi$ sont des propriétés relationnelles :*

$$\forall \ell, \varphi(\ell, i_1, \ldots, i_m) \Rightarrow \psi(a_1[\ell + c_1], \ldots, a_q[\ell + c_q], x_1, \ldots, x_t).$$

  Des exemples de tels invariants sont, à la fin des programmes suivants :
  – $\forall \ell, 1 \leq \ell \leq n \Rightarrow a[\ell] \leq max$ pour le programme de recherche du maximum ;
  – $\forall \ell, 1 \leq \ell \leq n \Rightarrow a[\ell] = b[\ell]$ pour le programme de copie de tableaux ;

1. La boucle interne du tri à bulle a pour invariant $\forall \ell_1, \ell_2, 1 \leq \ell_1 < i \wedge \ell_2 = i \Rightarrow a[\ell_1] \leq a[\ell_2]$. Or notre formalisme ne saura pas traité $a[i]$ à part.

– $\forall \ell, 2 \leq \ell \leq n \Rightarrow a[\ell - 1] \leq a[\ell]$ pour le programme de tri par insertion.
Un exemple d'invariant notable n'appartenant pas à notre classe de propriétés de tableaux est : $\forall \ell, 1 \leq \ell \leq n \Rightarrow A[\ell] = \ell$.

– d'autre part nous avons choisi, comme on l'a dit dans notre exemple d'analyse (Sec. 6.1.1), de fixer *a priori* les propriétés $\varphi$ qui pourront apparaître en chaque point de contrôle dans les propriétés de tableaux. Nous avons vu au Chapitre 4 que les analyses polyvalentes s'appuyaient toutes sur l'aide de l'utilisateur, sous des formes différentes, pour obtenir les propriétés $\psi$ des invariants à découvrir. Nous faisons l'hypothèse qu'il est beaucoup plus facile d'extraire du code source les tranches de tableaux (par exemple, $1 \leq \ell \leq i$) sur lesquelles une propriété $\psi$ intéressante va être vérifiée que cette propriété elle-même. Si cette hypothèse est correcte, on peut concevoir une pré-analyse simple qui fixe avec succès l'ensemble des propriétés $\{\varphi_p\}_{p \in P}$ apparaissant en partie gauche des propriétés de tableaux à découvrir. Ainsi l'analyse peut se concentrer sur le problème plus difficile de découvrir les parties droites de ces propriétés, et alors s'affranchir de l'aide de l'utilisateur. La seule contrainte que nous imposons sur l'ensemble $\{\varphi_p\}_{p \in P}$ est qu'aucun de ses éléments ne se chevauchent :

  Définition 6.2 (Partition). *Nous appelons une partition $P$ un ensemble fini de formules $\{\varphi_p\}_{p \in P}$ portant sur les variables d'indices et une variable spécifique notée $\ell$, telles que :*

$$\forall p, p' \in P, (p \neq p') \Rightarrow \neg(\varphi_p \wedge \varphi_{p'}).$$

  *Chaque élément d'une partition est appelée « tranche ».*

  Bien entendu le choix des partitions pour un programme donné peut faire que la classe de propriété ainsi décidée pour son analyse n'est pas suffisamment expressive pour représenter certains de ses invariants (problème similaire à celui du domaine abstrait des *templates* (Sec. 3.1) par exemple). Nous n'aurons pas de résultat de complétude de notre analyse à ce propos.

Ces choix d'expressivité fait, on peut dessiner la définition d'une valeur abstraite de notre domaine abstrait. La Remarque 6.1 rappelait qu'une analyse concomitante des variables d'indice (Sec. 6.1.1) était nécessaire pour la précision de l'analyse. Pour être précis nous aurons aussi besoin d'une analyse concomitante des variables scalaires de contenu. Ainsi les éléments de notre domaine abstrait seront la conjonction d'une propriété sur les indices, d'une propriété sur les scalaires de contenu et d'un ensemble de propriétés de tableaux (Déf. 6.1) telles que leurs parties gauches forment une partition (Déf. 6.2). Avant de pouvoir définir formellement les valeurs abstraites que l'on vient d'évoquer et leur sémantique (Équation. 6.4, p. 131) la question préalable est celle de l'expressivité exacte et de la représentation des propriétés composant ces valeurs abstraites. On rappelle que la question du partitionnement symbolique est traitée à la section 6.5.

### 6.2.1 Une représentation paramétrique

Pour représenter les éléments de notre domaine abstrait, nous choisissons un point de vue algorithmique plutôt que logique. On paramètre l'analyse par des domaines abstraits sémantiques permettant de représenter et manipuler les différentes propriétés qui composent les valeurs abstraites. L'expressivité de nos valeurs abstraites dépend donc

de celle des domaines abstraits choisis pour paramètres. Cette approche permet de tirer parti des nombreux domaines abstraits sémantiques existants, et nous paraît être celle s'articulant le mieux avec la philosophie de l'interprétation abstraite.

Lorsqu'on considère l'utilisation d'éléments d'autres domaines abstraits pour représenter nos propriétés, celles en parties droites des propriétés de tableaux (Déf. 6.1) soulèvent une difficulté très pratique, celle de la présence des termes de la forme $a[\ell + c]$. Pour représenter ceux-ci on introduit de nouvelles variables, dites de tranches, où la référence à $\ell$ est implicite.

Notation 6.1 (variables de tranches). *Une variable de tranche est une variable de la forme $a^z$ où $z \in \mathbb{Z}$ et $a \in A$. Dans une valeur abstraite $\psi$ elle représente la cellule $a[\ell + z]$.*

*On note $S(A)$ l'ensemble infini (si $A \neq \emptyset$) des variables de tranches du programme.*

Si l'on reprend les propriétés de tableaux pris en exemple au début de la section, on souhaite donc représenter leurs parties droites par (les éléments d'un domaine abstrait représentant) les propriétés suivantes : $a^0 \leq max$ pour la recherche du maximum, $a^0 = b^0$ pour la copie de tableau et $a^{-1} \leq a^0$ pour le tri par insertion. De nombreux domaines permettent de représenter ces propriétés, on peut citer parmi les plus simples d'entre eux le domaine des zones (Sec. 3.1.1).

On considère pour paramètres de l'analyse deux domaines abstraits sémantiques, l'un permettant d'abstraire des éléments de $\mathbb{Z}^n$, l'autre des éléments de $\mathbb{K}^n$, que l'on note respectivement $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$. Nos valeurs abstraites sont alors définies formellement comme la conjonction d'une part, d'un élément $\eta$ de $\mathcal{D}^{\mathbb{Z}}$ représentant les propriétés des variables d'indices et d'un élément $\mu$ de $\mathcal{D}^{\mathbb{K}}$ représentant les propriétés des variables de contenu et, d'autre part, d'un ensemble de propriétés de tableaux dont les parties gauches sont représentées par des éléments $\varphi_p$ de $\mathcal{D}^{\mathbb{Z}}$ et les parties droites par des éléments $\psi_p$ de $\mathcal{D}^{\mathbb{K}}$. La sémantique de ces valeurs abstraites est :

$$\eta(I) \wedge \mu(V) \bigwedge_p \forall \ell \; \varphi_p(\ell \cup I) \Rightarrow \psi_p(S(A) \cup V)[a[\ell + z]/a^z]. \tag{6.1}$$

Pour être plus général on pourrait considérer deux domaines abstraits sémantiques supplémentaires pour représenter respectivement les parties droites et gauches des propriétés de tableaux. Sachant que de nombreuses fonctions sémantiques du domaine nécessiteront des opérations impliquant, soit $\eta$ et certains $\varphi_p$, soit $\mu$ et certains $\psi_p$, alors de nombreuses conversions d'un domaine abstrait vers l'autre devraient être ajoutées à leurs définitions. Là encore, afin de ne pas compliquer ces définitions pour ce qui relève seulement d'une difficulté technique, nos éléments abstraits ne seront paramétrés que par $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$.

Il reste d'ailleurs à indiquer formellement les hypothèses que nous faisons sur ces domaines abstraits sémantiques pour qu'ils soient des paramètres valides de l'analyse.

#### 6.2.1.1 Opérateurs exigés des domaine abstraits sémantiques utilisés en paramètres de l'analyse

Nous donnons ces hypothèses tôt afin que le lecteur puisse se convaincre rapidement qu'elles ne sont pas impraticables. Cependant, comme il sera fait mention à chaque fois qu'une opération de l'analyse reposera sur l'une de ces hypothèses, cette section peut

aussi être lue au fil du chapitre.

Bien entendu, les domaines abstraits $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$ respectent en premier lieu la Définition 2.6 (p. 22) des domaines abstraits sémantiques avec pour valeur de $Q$ respectivement $\mathbb{Z}$ et $\mathbb{K}$. À cette définition s'ajoutent des contraintes et des algorithmes supplémentaires exposés ci-dessous (Déf. 6.3).

Définition 6.3 (domaine abstrait sémantique pour l'analyse de tableaux). *Un domaine abstrait $\mathcal{D}^\sharp$ pouvant être utilisé en paramètre de l'analyse du contenu de tableaux est composé :*

- *des cinq points énoncés par la définition 2.6 ;*
- *et :*
  6. *d'un opérateur de normalisation. Une valeur abstraite suivie d'une étoile$^*$ notera sa forme normalisée ;*
  7. *d'un algorithme effectif pour calculer la borne inférieure $\sqcap^\sharp$, qui soit l'abstraction exacte de $\cap$ (Déf. 2.2, p. 15) ;*
  8. *d'un opérateur d'élimination de variable, noté $\exists^\sharp.v$, lui aussi exact.*
- *de plus si $\mathcal{D}^\sharp$ est utilisé comme le paramètre $\mathcal{D}^{\mathbb{Z}}$ il doit :*
  9. *être capable d'exprimer toute contrainte de zone et de rendre compte exactement des affectations de la forme $i := j + c$ ou $i := c$ ;*
  10. *fournir un algorithme effectif pour calculer une approximation de l'opération de soustraction $x^\sharp \wedge \neg x'^\sharp$, notée $x^\sharp \setminus^\sharp x'^\sharp$ ;*
  11. *fournir un algorithme effectif (possiblement incomplet) indiquant si l'union $\sqcup^{\mathbb{Z}}$ de deux éléments de $\mathcal{D}^{\mathbb{Z}}$ est exacte. Si c'est le cas le résultat de cette union est renvoyé. On note $\between^{\mathbb{Z}}$ cet opérateur.*
- *sinon, si $\mathcal{D}^\sharp$ est utilisé comme le paramètre $\mathcal{D}^{\mathbb{K}}$ il doit :*
  12. *permettre de renommer des variables dans un de ses éléments (si possible à coût constant).*

Les hypothèses 6, 7 et 8 portent sur des opérations usuelles, dont on imagine bien qu'elles seront utiles par la suite (pour la définition des opérations sémantiques, etc). Les hypothèses 10 et 11 introduisent deux opérations particulières qui sont nécessaires pour une précision accrue de l'opération de normalisation (Déf. 6.9, p. 144). Enfin, l'hypothèse 9 et 12 permettent de calculer exactement la translation (Déf. 6.7, p. 136) d'une tranche et de la propriété associée par une constante.

Afin de montrer qu'il existe des instances possibles de $\mathcal{D}^{\mathbb{Z}}$, on donne des algorithmes implantant les opérateurs $\setminus^{\mathbb{Z}}$ et $\between^{\mathbb{Z}}$ pour les zones, car c'est en pratique le domaine que l'on utilisera. Il ne fait aucun doute que le domaine des zones $\mathcal{D}^{zone}$ remplisse les autres hypothèses que nous venons de formuler pour $\mathcal{D}^{\mathbb{Z}}$ (1 – 9 et 12). Ce sont des algorithmes plus coûteux que les opérations usuelles, leur complexité étant pour $\setminus^{zone}$ comme pour $\between^{zone}$ en $\mathcal{O}(n^4)$.

Pour calculer $m^1 \setminus^{zone} m^2$, une méthode simple consiste à construire, pour chaque contrainte $(x_i - x_j \leq m^2_{ij})$ représentée par $m^2$, la DBM $m'_{ij}$ qui soit la conjonction de $m^1$ et de la négation de cette contrainte, *i.e.* $(x_j - x_i < -m^2_{ij})$ ou encore $(x_j - x_i \leq -m^2_{ij} - 1)$ puisqu'on est dans $\mathbb{Z}$, et calculer l'union de ces DBM $m'_{ij}$.

```
1  E ← ∅ ; b ← vrai ;
2  m¹ ← closure(m¹) ; m² ← closure(m²) ;
3  for i ← 1 to n do
4  |  for j ← 1 to n do
5  |  |  if i ≠ j then
6  |  |  |  if m¹_ji ≤ −(m²_ij + 1) then
7  |  |  |  |  return m¹
8  |  |  |  else if m¹_ij ≥ m²_ij + 1 then
9  |  |  |  |  E ← E ∪ {(i, j)}
10 |  |  |  else
11 |  |  |  |  b ← faux
12 if b then
13 |  return m¹
14 else
15 |  m ←_zone ⊥ ;
16 |  forall (i, j) ∈ E do
17 |  |  m′ ← m¹ ; m′_ji ← −(m²_ij + 1) ;
18 |  |  m ← m ⊔_dbm closure(m′) ;
19 return m
```

(a) Algorithme implantant l'opération $m^1 \setminus^{zone} m^2$ où $m^1, m^2 \in \mathcal{D}^{zone}$

```
1  E ← ∅ ; b ← vrai ;
2  m¹ ← closure(m¹) ; m² ← closure(m²) ;
3  m ← m¹ ⊔_dbm m² ;
4  for i ← 1 to n do
5  |  for j ← 1 to n do
6  |  |  if i ≠ j then
7  |  |  |  if m_ij ≥ m¹_ij + 1 then
8  |  |  |  |  E ← E ∪ {(i, j)}
9  |  |  |  else
10 |  |  |  |  b ← faux
11 if b then
12 |  return Some(m)
13 else
14 |  forall (i, j) ∈ E do
15 |  |  m′ ← m ; m′_ji ← −(m¹_ij + 1) ;
16 |  |  if ¬(m′ ⊑^zone m²) then
17 |  |  |  return None
18 return Some(m)
```

(b) Algorithme implantant l'opérateur $m^1 \ \mathbb{W}^{zone} \ m^2$ où $m^1, m^2 \in \mathcal{D}^{zone}$

Figure 6.2 – Algorithmes implantant les opérateurs $\setminus^{\mathbb{Z}}$ et $\mathbb{W}^{\mathbb{Z}}$ pour le domaine abstrait des zones. Leur complexité est de $\mathcal{O}(n^4)$ en temps et de $\mathcal{O}(n^3)$ en espace.

L'algorithme que l'on donne à la Figure 6.2(a) suit ce schéma en l'optimisant. En effet si dans $m^1$ on a $x_j - x_i \leq m^1_{ji} \leq -m^2_{ij} - 1$ alors on a $m'_{ij} = m^1$ et on peut tout de suite s'arrêter, l'union retournera $m^1$ (lignes **6**–**7**). Si dans $m^1$ on a $x_i - x_j \leq m^1_{ij} \leq m^2_{ij}$ alors on a $m'_{ij} = \bot^{zone}$ et donc ce terme ne sert à rien dans l'union (lignes **8**–**9**). Enfin si l'on a pour tout $i, j$ que $m^1_{ij} \geq m^2_{ij} + 1$ c'est que $m^2 \sqsubset^{zone} m^1$ et comme une zone représente un ensemble convexe, on peut tout de suite retourner $m^1$ (lignes **10**–**13**). Ainsi construit-on l'ensemble $E$ de couples $(i, j)$ pour lequel il est utile de calculer l'union des $m'_{ij}$ (lignes **15**–**18**). Les $m'_{ij}$ sont normalisés préalablement à l'union par normalisation incrémentale. Ainsi, avec potentiellement $\mathcal{O}(n^2)$ couples dans $E$ appelant une opération d'union et une opération de normalisation incrémentale en $\mathcal{O}(n^2)$, on a bien une complexité en temps de $\mathcal{O}(n^4)$ pour cet opérateur.

L'algorithme calculant $m^1 \ \mathbb{W}^{zone} \ m^2$ est donné à la Figure 6.2(b). Il consiste à calculer $m = m^1 \sqcup^{zone} m^2$, l'unique candidat possible, puis pour chaque contrainte que représente $m^1$, regarder la conjonction de $m$ et de la négation de cette contrainte. L'espace représenté par cette conjonction est soit vide (la contrainte est une face des espaces représentés par $m$ et par $m^1$), soit il doit être couvert par $m^2$ pour que $m$ représente l'union exacte de $m^1$ et $m^2$. Cet algorithme est implanté de la même manière que l'opérateur précédent, ce

qui porte sa complexité en temps à $\mathcal{O}(n^4)$ également.

### 6.2.2 Une famille de domaines abstraits

En préambule nous indiquions qu'une partition serait attachée à chaque point de contrôle du programme, restreignant ainsi les valeurs abstraites possibles. Les conséquences de cette approche sont doubles.

La première est que la partition, puisqu'elle est connue et invariante, n'a pas besoin de figurer dans l'élément abstrait. Ce point de vue s'accommode bien avec notre vision algorithmique : nous n'avons pas besoin de recourir à un domaine abstrait générique pour représenter l'implication (*e.g.* celui proposé dans [MM07]) et nous pouvons définir nos éléments abstraits simplement comme la juxtaposition de propriétés : celles sur les variables scalaires et celles des parties droites des propriétés de tableaux. On définit ainsi une *famille* de domaines abstraits, dont les éléments sont tous de la même forme mais dont la sémantique diffère selon la partition qui leurs est associée.

Définition 6.4 (Domaines abstraits du contenu des tableaux). *On définit l'ensemble suivant de domaines abstraits pour l'analyse du contenu des tableaux (repérés par l'exposant « arco » pour* array contents*). Ces domaines abstraits diffèrent par la partition $P = \{\varphi_p(\ell \cup I)\}$ (Déf. 6.2, p. 127) selon laquelle la sémantique de leurs éléments abstraits est entendue :*

$$\mathcal{D}_P^{arco} = \bot^{arco} \cup \Big\{ (\eta(I), \mu(V), \{\psi_p(S(A) \cup V)\}_{p \in P}) \mid \eta \in \mathcal{D}^{\mathbb{Z}} \wedge \mu \in \mathcal{D}^{\mathbb{K}} \wedge \forall p \in P, \psi_p \in \mathcal{D}^{\mathbb{K}} \Big\}$$

*où $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$ sont deux domaines abstraits vérifiant les hypothèses données à la Déf. 6.3.*

À la Figure 6.3, on donne, entre autres, trois exemples de valeurs abstraites, la première appartenant à $\mathcal{D}_{P_1}^{arco}$, les deux autres à $\mathcal{D}_{P_2}^{arco}$. Pour ces exemples, $\mathcal{D}^{\mathbb{Z}}$ est le domaine des zones et $\mathcal{D}^{\mathbb{K}}$ celui des zones précisant alias.

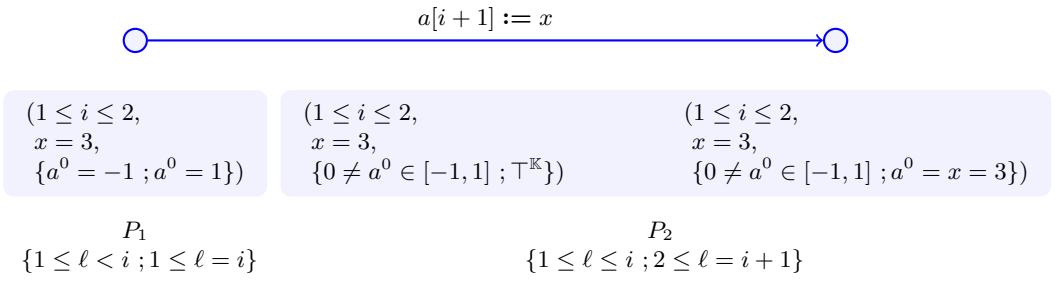



Figure 6.3 – Exemple de changement dynamique du domaine abstrait $\mathcal{D}_{P_1}^{arco}$ au domaine abstrait $\mathcal{D}_{P_2}^{arco}$ durant une opération d'affectation abstraite ($\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{zone}$ et $\mathcal{D}^{\mathbb{K}} = \mathcal{D}^{zoal}$)

Une partition pouvant changer d'un point de contrôle à un autre, l'analyse doit passer d'un domaine abstrait à un autre. On parle alors d'adaptation de la valeur abstraite consistant simplement à construire une nouvelle valeur abstraite, basée sur la partition cible, qui soit une sur-approximation de la valeur abstraite source, dont la sémantique s'entend, elle, selon la partition source. Si chaque adaptation respecte ce critère conservatif, la terminaison de l'analyse est toujours assurée malgré ces changements

de domaines abstraits dynamiques. La définition et la preuve de correction de cette opération nécessitant la définition d'autres opérateurs – dont bien entendu la fonction de concrétisation associée aux domaines abstraits (Sec. 6.3) pour pouvoir discuter le caractère conservatif – est présentée à la section 6.5.2.

Afin de comprendre cette opération dès à présent, on donne à la Figure 6.3 un exemple d'opération d'affectation abstraite durant laquelle elle est utilisée. À gauche, on trouve la valeur abstraite, définie sur $P_1$ qui doit être adaptée à la partition $P_2$. Outre le fait que les propriétés scalaires restent inchangées, le résultat amène deux remarques : pour les cellules que représente la tranche $(2 \leq \ell = i+1)$ de $P_2$, aucune propriété de tableaux de la première valeur abstraite n'apporte d'information. On leur associe donc l'élément $\top^{\mathbb{K}}$. Par contre pour les cellules que représente la tranche $(1 \leq \ell \leq i)$, ces cellules appartenant soit à la tranche $(1 \leq \ell < i)$ soit à la tranche $(1 \leq \ell = i)$ de $P_1$, on peut leur associer l'union des propriétés correspondantes $(a^0 = -1) \sqcup^{\mathbb{K}} (a^0 = 1) = (a^0 \neq 0 \wedge a^0 \in [-1, 1])$. Une union ici exacte grâce au domaine $\mathcal{D}^{zoal}$ utilisé.

#### 6.2.2.1 Structure de treillis des domaines abstraits

Deuxième conséquence de notre approche, les opérateurs de treillis peuvent être définis indépendamment de la partition : en effet lorsqu'ils sont utilisés, en un point de contrôle, leurs opérandes partagent de fait la même partition. Il est donc naturel de ne pas en définir un seul mais un pour chaque domaine abstrait $\mathcal{D}_P^{arco}$ de la famille de domaines précédemment définie (Déf. 6.4) et ainsi donner à chacun une structure de treillis.

On définit l'ordre sur les valeurs abstraites d'un domaine ainsi : une valeur précède une autre si pour chaque élément abstrait qui les composent (un $\eta$, un $\mu$ et un ensemble de $\psi_p$ dont la cardinalité dépend de $P$), celui de la première valeur est plus petit, selon l'ordre $\mathcal{D}^{\mathbb{Z}}$ ou $\mathcal{D}^{\mathbb{K}}$ correspondant à la nature de l'élément, que celui de la seconde valeur. Les opérateurs de bornes supérieures et inférieures sont définis de la même manière, paire à paire.

Définition 6.5 (opérateurs de treillis). *Soit $P$ une partition. On définit les opérateurs suivants, pour tout $\Phi, \Phi' \in \mathcal{D}_P^{arco}$ :*

- $\Phi \sqsubseteq_P^{arco} \Phi' \Leftrightarrow \begin{cases} soit\ \Phi = \bot^{arco} \\ soit\ (\eta \sqsubseteq^{\mathbb{Z}} \eta' \wedge \mu \sqsubseteq^{\mathbb{K}} \mu' \wedge \forall p \in P, \psi_p \sqsubseteq^{\mathbb{K}} \psi'_p) \end{cases}$ ;
- $\Phi \sqcup_P^{arco} \Phi' = \begin{cases} \Phi & si\ \Phi' = \bot^{arco} \\ \Phi' & si\ \Phi = \bot^{arco} \\ (\eta \sqcup^{\mathbb{Z}} \eta', \mu \sqcup^{\mathbb{K}} \mu', \{\psi_p \sqcup^{\mathbb{K}} \psi'_p\}_{p \in P}) & sinon \end{cases}$ ;
- $\Phi \sqcap_P^{arco} \Phi' = \begin{cases} \bot^{arco} & si\ \Phi = \bot^{arco}\ ou\ \Phi' = \bot^{arco} \\ (\eta \sqcap^{\mathbb{Z}} \eta', \mu \sqcap^{\mathbb{K}} \mu', \{\psi_p \sqcap^{\mathbb{K}} \psi'_p\}_{p \in P}) & sinon \end{cases}$ .

*Au vu de l'ordre $\sqsubseteq_P^{arco}$ on peut définir un plus grand élément naturel, $\top_P^{arco} = (\top^{\mathbb{Z}}, \top^{\mathbb{K}}, \{\top^{\mathbb{K}}\}_{p \in P})$. L'ensemble $(\mathcal{D}_P^{arco}, \sqsubseteq_P^{arco}, \sqcup_P^{arco}, \sqcap_P^{arco}, \bot^{arco}, \top_P^{arco})$ forme alors un treillis, qui est complet lorsque $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$ le sont.*

*Démonstration.*

- la relation $\sqsubseteq_P^{arco}$ est un ordre partiel, par réflexivité, antisymétrie et transitivité des relations $\sqsubseteq^{\mathbb{Z}}$ et $\sqsubseteq^{\mathbb{K}}$.
- Les preuves sont directes pour les autres opérateurs, ainsi que pour la complétude.

□

La simplicité de l'opération d'union est à comparer à celle du domaine abstrait proposé dans [GMT08] où des opérations complexes, notamment de sous-approximations, sont nécessaires (Sec. 4.3.3) pour inférer une partition sur laquelle le résultat sera exprimé. Quand bien même une opération d'union peut cacher ici une opération d'adaptation, ce qui est rare dans la pratique, on tire ici le bénéfice de fixer *a priori* les partitions. Si l'on reprend les exemples d'opérations d'unions de leur domaine abstrait quantifié (Fig. 4.3, p. 66) à la tête d'une boucle copiant simplement un tableau dans un autre, et que l'on convient que pour un tel programme une simple analyse peut deviner qu'une tranche représentant les cellules à gauche de la variable d'itération ($1 \leq \ell < i$) est une tranche intéressante, on verrait la simple opération d'union suivante lors de notre analyse : $(1 \leq i \leq 2, \top^{\mathbb{K}}, \{a^0 = b^0\}) \sqcup^{arco} (2 \leq i \leq 3, \top^{\mathbb{K}}, \{a^0 = b^0\}) = (1 \leq i \leq 3, \top^{\mathbb{K}}, \{a^0 = b^0\})$.

## 6.3 Abstraction sous fonction de concrétisation

Notre domaine abstrait défini, nous choisissons d'utiliser le formalisme d'abstraction sous fonction de concrétisation (Sec. 2.1.1.2) pour assurer la correction de l'analyse. Nous devons donc définir une telle fonction, en l'occurrence une pour chaque domaine $\mathcal{D}_P^{arco}$, qui soit monotone vis-à-vis de $\sqsubseteq_P^{arco}$.

On se réfère à l'Équation 6.1 (p. 128) où l'on a défini la sémantique d'un élément abstrait en termes logiques pour définir l'élément concret (*cf.* Préliminaires – Sémantique concrète) qu'un élément $\Phi \in \mathcal{D}_P^{arco}$ abstrait. Nous allons avoir recours aux fonctions de concrétisation des domaines $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$ (Déf. 6.3) bien entendu, ainsi qu'à la notion de contexte d'une tranche. Il s'agit simplement de l'ensemble des propriétés sur les variables d'indices qu'une tranche impose : on le définit par $\overline{\varphi_p} = \exists \ell.\varphi_p$ – on rappelle que l'hypothèse 8 sur $\mathcal{D}^{\mathbb{Z}}$ assure que cet ensemble n'est pas approximé par l'opération d'élimination.

On peut maintenant définir l'ensemble $\gamma_P^{arco}(\Phi)$ de valuations $(\rho_i, \rho_v, \rho_a)$ en compréhension (Déf. 6.6).

Définition 6.6 (fonction de concrétisation $\gamma_P^{arco}$). *Soit $P$ une partition et $\Phi \in \mathcal{D}_P^{arco}$. On définit la fonction de concrétisation suivante : si $\Phi = \bot^{arco}$ alors $\gamma_P^{arco}(\Phi) = \emptyset$ ; sinon $\Phi = (\eta, \mu, \{\psi_p\})$ et*

$$\begin{aligned}\gamma_P^{arco}(\Phi) = \{(\rho_i, \rho_v, \rho_a) \in Env \mid \quad & \rho_i \in \gamma^{\mathbb{Z}}(\eta) \wedge \forall p \in P, (\gamma^{\mathbb{K}}(\psi_p) = \emptyset \Rightarrow \rho_i \notin \gamma^{\mathbb{Z}}(\overline{\varphi_p})) \\ & \wedge \rho_v \in \gamma^{\mathbb{K}}(\mu) \wedge \forall p \in P, (\rho_i \in \gamma^{\mathbb{Z}}(\overline{\varphi_p}) \Rightarrow \rho_v \in \gamma^{\mathbb{K}}(\overline{\psi_p})) \\ & \wedge \forall p \in P, \forall k \in K(\varphi_p), \exists \rho \in R(\psi_p), \forall a^z \in Dom(\rho), \rho_a(a)(k+z) = \rho(a^z)\}.\end{aligned}$$

On explique comment on arrive à cette définition de la fonction de concrétisation.

- Pour ce qui est des valuations possibles des variables d'indices, $\rho_i$ est contrainte par la propriété $\eta$ : on a $\rho_i \in \gamma^{\mathbb{Z}}(\eta)$. Elle peut l'être également par certaines propriétés de tableaux : lorsque $\psi_p$ est incohérente on déduit par contraposition que la propriété $\varphi_p$ est impossible. Ainsi pour toute propriété de tableau où $\gamma^{\mathbb{K}}(\psi_p) = \emptyset$, les valuations des variables d'indices ne peuvent appartenir à $\gamma^{\mathbb{Z}}(\overline{\varphi_p})$.
- Pour ce qui est des variables scalaires de contenu, leurs valuations possibles respectent évidemment la propriété $\mu$. De plus les propriétés de tableaux, où ces variables peuvent apparaître en partie droite, doivent être prises en compte. On remarque alors que les propriétés de $\psi_p$ portant uniquement sur les variables scalaires

$$P = \{1 \leq \ell < i \ ; \quad 1 \leq \ell = i\} \qquad \Phi = (1 \leq i \leq 3, \top^{\mathbb{K}}, \{b^1 = a^0 \in [7,8] \wedge x = 6 \ ; \ a^0 = 7 \wedge a^{-1} = 8 \wedge x \in [5,6]\})$$

| $\rho_i$ | $\rho_v$ | $\rho_a$ | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| $i \to 1$ | $x \to 5$ | $a \to$ | 7 | ? | ? | ? | $b \to$ | ? | ? | ? | ? |
| $i \to 1$ | $x \to 6$ | $a \to$ | 7 | ? | ? | ? | $b \to$ | ? | ? | ? | ? |
| $i \to 2$ | $x \to 6$ | $a \to$ | 8 | 7 | ? | ? | $b \to$ | ? | 8 | ? | ? |
| $i \to 3$ | $x \to 6$ | $a \to$ | 7 | 8 | 7 | ? | $b \to$ | ? | 7 | 8 | ? |
| $i \to 3$ | $x \to 6$ | $a \to$ | 8 | 8 | 7 | ? | $b \to$ | ? | 8 | 8 | ? |

Figure 6.4 – Exemple de valeur abstraite et de l'ensemble $\gamma_P^{arco}$ de valuations $(\rho_i, \rho_v, \rho_a)$ qu'elle abstrait ($\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{\mathbb{K}} = \mathcal{D}^{zone}$)

ne dépendent pas de la variable quantifiée $\ell$. Si on note $\overline{\psi_p}$ ces propriétés, qu'on définit simplement par $\exists(S(A)).\psi_p$, on voit que l'on a l'implication $\overline{\varphi_p} \Rightarrow \overline{\psi_p}$. Ainsi si $\rho_i$ appartient aux valuations que représente l'antécédent de cette implication, $\rho_v$ doit appartenir à celles que représente le conséquent.
On prend pour exemple la valeur abstraite donnée en haut de la Figure 6.4, définie sur une partition telle que $\overline{\varphi_1} = (i \geq 2)$ et $\overline{\varphi_2} = (i \geq 1)$. Comme $\overline{\psi_1} = (x = 6)$ et $\overline{\psi_2} = (x \in [5,6])$, si $\rho_i(i) = 1$ alors $\rho_v(x)$ peut être égal à 5 ou 6, par contre $\rho_v(x)$ est égal à 6 pour toute valuation où $\rho_i(i) \geq 2$. On remarque alors que la propriété $\mu$ pourrait être remplacée par la propriété plus précise $x \in [5,6]$ puisque $\rho_i(i) \in [1,3]$ d'après $\eta$ : ce sera l'objet de l'opération de normalisation (Sec. 6.4.2).

– Enfin, pour ce qui est des valuations des cellules des tableaux, on explicite pour chaque propriété de tableaux les restrictions qu'elle impose. Tout d'abord on souhaite identifier sur quelles cellules portent ces restrictions : l'ensemble des valeurs possibles de la variable $\ell$, qui dépendent des valuations choisies des variables d'indices apparaissant dans $\varphi_p$, constitue le socle de référence. On définit cet ensemble de valeurs par :

$$K(\varphi) = \{\rho_{|\{\ell\}}(\ell) \mid \forall i \in I, \rho(i) = \rho_i(i) \wedge \rho \in \gamma^{\mathbb{Z}}(\varphi)\}.$$

Cet ensemble peut être vide : on prend pour exemple la valeur abstraite de la Figure 6.4 où l'on a $K(\varphi_1) = \emptyset$ si $\rho_i(i) = 1$. Dans ce cas la propriété de tableaux n'exprime aucune restriction sur les valuations des tableaux. Dans le cas contraire, les cellules qui seront affectées dépendent des variables de tranches présentes dans $\psi_p$ : ce sont simplement les cellules à la position $k + z$, pour tout $k \in K(\varphi_p)$ et pour tout $a^z \in Var(\psi_p)$. Reste à identifier les propriétés que ces cellules devront respecter. Il s'agit de l'une des valuations possibles des variables de tranches, dépendantes des valuations des variables scalaires de contenu apparaissant dans $\psi_p$ choisies. On définit ces valuations possibles par : $R(\psi) = \{\rho_{|S(A)} \mid \forall x \in V, \rho(x) = \rho_v(x) \wedge \rho \in \gamma^{\mathbb{K}}(\psi)\}$. Ainsi, il doit exister une valuation $\rho$ de $R(\psi_p)$ telle que pour tout $k$ et tout $z$, $\rho_a(a)(k + z) = \rho(a^z)$.
Si on reprend l'exemple de la Figure 6.4, pour $\rho_i(i) = 2$ on a $K(\varphi_1) = \{1\}$, $K(\varphi_2) = \{2\}$, $R(\psi_1) = \{b^1 \to 7, a^0 \to 7 \ ; b^1 \to 8, a^0 \to 8\}$ et $R(\psi_2) = \{a^0 \to 7, a^{-1} \to 8\}$.

Selon la première propriété de tableau, $\rho_a(a)(1+0)$ peut être égal à 7 ou 8, par contre la seconde impose qu'il $(\rho_a(a)(2-1))$ soit égal à 8. Ainsi c'est la seconde valuation de $R(\psi_1)$ qui est la seule possible : on a alors $\rho_a(b)(1+1)$ égal à 8 également. On remarque que pour $\rho_i(i) = 3$ (on a alors $K(\varphi_1) = \{1, 2\}$ et $K(\varphi_2) = \{3\}$) cette fois les deux valuations de $R(\psi_1)$ sont possibles pour $k = 1$. Ainsi $\rho_a(a)(1+0)$ et $\rho_a(b)(1+1)$ peuvent être égaux soit à 7, soit à 8.

Théorème 6.1 (monotonicité de $\gamma_P^{arco}$). *La fonction $\gamma_P^{arco}$ est monotone vis-à-vis de l'ordre $\sqsubseteq_P^{arco}$ (Déf. 6.5, p. 132).*

*Démonstration.* Soit $\Phi^1, \Phi^2 \in \mathcal{D}^{arco}$ tel que $\Phi^1 \sqsubseteq^{arco} \Phi^2$. Soit $(\rho_i^1, \rho_v^1, \rho_a^1) \in \gamma_P^{arco}(\Phi^1)$.
- par monotonicité de $\gamma^{\mathbb{Z}}$ on a $\rho_i^1 \in \gamma^{\mathbb{Z}}(\eta^2)$. Soit $p \in P$ tel que $\gamma^{\mathbb{K}}(\psi_p^2) = \emptyset$. La monotonicité de $\gamma^{\mathbb{K}}$ implique $\gamma^{\mathbb{K}}(\psi_p^1) = \emptyset$ et donc par hypothèse on a $\rho_i^1 \notin \gamma^{\mathbb{Z}}(\overline{\varphi_p})$.
- par monotonicité de $\gamma^{\mathbb{K}}$ on a $\rho_v^1 \in \gamma^{\mathbb{K}}(\mu^2)$. Soit $p \in P$ tel que $\rho_i^1 \in \gamma^{\mathbb{Z}}(\overline{\varphi_p})$. Par hypothèse on a $\rho_v^1 \in \gamma^{\mathbb{K}}(\overline{\psi_p^1})$. On conclut par monotonicité de l'opération d'élimination sur $\mathcal{D}^{\mathbb{K}}$ et de la monotonicité de $\gamma^{\mathbb{K}}$ que $\rho_v^1 \in \gamma^{\mathbb{K}}(\overline{\psi_p^2})$.
- soit $p$ et $k \in K(\varphi_p)$. On peut conclure si $R(\psi_p^1) \subseteq R(\psi_p^2)$, ce qu'assure la monotonicité de $\gamma^{\mathbb{K}}$.

Donc $(\rho_i^1, \rho_v^1, \rho_a^1) \in \gamma_P^{arco}(\Phi^2)$. □

On termine cette section par un exemple (Fig. 6.5) montrant une opération d'union $\sqcup_P^{arco}$ et les concrétisations de ses opérandes et de son résultat. Bien que l'union des propriétés $\psi_1$, comme celles des propriétés $\eta$, des valeurs abstraites $\Phi^1$ et $\Phi^2$ n'ait pas donné lieu à une sur-approximation, la valeur abstraite résultante, elle, est une approximation de l'union exacte comme le montre sa concrétisation.

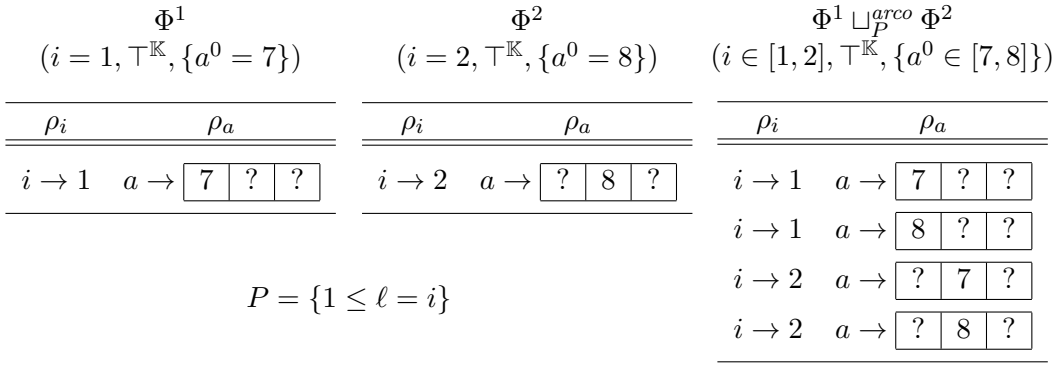



Figure 6.5 – Exemple d'union de valeurs abstraites et concrétisation ($\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{\mathbb{K}} = \mathcal{D}^{int}$)

Bien entendu, il n'y a aucune raison pour que la fonction $\gamma_P^{arco}$ soit injective, d'autant plus que cette propriété n'est pas imposée aux fonctions $\gamma^{\mathbb{Z}}$ et $\gamma^{\mathbb{K}}$. On s'intéresse donc naturellement à la normalisation de nos valeurs abstraites.

## 6.4 Normalisation

### 6.4.1 Test de satisfaisabilité

On étudie dans cette section un ensemble de conditions qui implique l'insatisfaisabilité d'une valeur abstraite. En effet, sauf pour des cas simples, auquel la normalisation

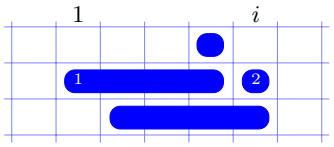
(Sec. 6.4.2) pourra nous ramener, on ne peut donner de test de satisfaisabilité. Ainsi, cette étude est surtout l'occasion d'introduire des notations pour la suite du chapitre et de comprendre dans le détail la sémantique de nos valeurs abstraites.

Parmi les contraintes portant sur les valuations des tableaux, nous avons vu que les variables de tranches contraignent des cellules de tableaux au-delà de celles associées à la tranche elle-même. Pour identifier ces cellules on est amené à considérer les translatés des tranches $\varphi_p$ selon les variables de tranches apparaissant dans la propriété associée $\psi_p$. C'est donc naturellement selon cette dernière que l'on définit l'ensemble de ces translations possible : $\mathit{Trans}(\psi) = \{z \in \mathbb{Z} \mid a^z \in \mathit{Var}(\psi)\}$. À cette opération de translation de tranche on ajoute celle de translation de la propriété associée :

Définition 6.7 (opérateurs de translation). *Soit $z \in \mathbb{Z}, \varphi \in \mathcal{D}^{\mathbb{Z}}$ et $\psi \in \mathcal{D}^{\mathbb{K}}$. On définit :*
- *$\varphi \oplus z = [\ell := \ell + z]^{\mathbb{Z}}(\varphi)$. Cet opérateur s'encode simplement par application de la fonction de transfert pour les affectations de $\mathcal{D}^{\mathbb{Z}}$ ;*
- *$\psi \boxplus z = \psi[a^{y-z}/a^y]$, c'est-à-dire les variables de tranche $a^y$ sont remplacées par les variables de tranche $a^{y-z}$. Cet opérateur peut s'encoder par application successive de celui de renommage de $\mathcal{D}^{\mathbb{K}}$ (Hypothèse 12, Déf. 6.3, p. 129).*

Remarque. *Sur l'opérateur $\oplus$ : la 9e hypothèse portant sur $\mathcal{D}^{\mathbb{Z}}$ (Déf. 6.3) assure que l'affectation $\ell := \ell + z$ n'est pas sujette à une sur-approximation, ce qui couplé à la 8e hypothèse permet d'affirmer que $\forall z \in \mathbb{Z}, \gamma^{\mathbb{Z}}(\overline{\varphi}) = \gamma^{\mathbb{Z}}(\overline{\varphi \oplus z})$ ; c'est-à-dire que l'opérateur laisse invariant les contraintes portant sur les variables d'indices. Enfin on remarquera que pour $\rho_i$ donné, on a $K(\varphi \oplus z) = \{k + z \mid k \in K(\varphi)\}$. Sur l'opérateur $\boxplus$ : celui-ci est tel que pour tout $z$ : $\forall \ell\ \varphi_p \oplus z \Rightarrow \psi_p \boxplus z[a[\ell + w]/a^w]$, propriété que l'on notera $\varphi_p \oplus z \Rrightarrow \psi_p \boxplus z$.*

La Figure 6.6 illustre ces définitions pour les variables de tranches de la valeur abstraite présentée à la Figure 6.4 (p. 134). Les ensembles de cellules de $a$ et de $b$ que représentent les variables de tranches $a^0, b^1 \in \mathit{Var}(\psi_1)$ et $a^0, a^{-1} \in \mathit{Var}(\psi_2)$ sont, dans l'ordre d'apparition sur la Figure 6.6 et pour $\rho_i(i) = 4$ : les propriétés $\varphi_2 \oplus -1 = (1 \leq \ell = i - 1), \varphi_1, \varphi_2$ et $\varphi_1 \oplus 1 = (2 \leq \ell \leq i)$. Si l'on se place vis-à-vis des ces référentiels, les propriétés que vérifient les cellules de $a$ sont respectivement $\psi_2 \boxplus -1 = (a^0 = 8)$,$\psi_1$,$\psi_2$ et les cellules de $b$, $\psi_1 \boxplus 1 = (b^0 = a^{-1} \in [7, 8])$.




$\varphi_1 = (1 \leq \ell < i)$
$\varphi_2 = (1 \leq \ell = i)$

Figure 6.6 – Schéma illustrant les cellules représentées, tous tableaux confondus, par différentes variables de tranches (pour une valuation fixée des variables d'indices, ici $i = 4$), en l'occurrence celles de la valeur abstraite présentée à la Figure 6.4

On peut à présent proposer (Prop. 6.1), pour une partition P donnée, un ensemble de conditions assurant l'insatisfaisabilité d'une valeur abstraite $\Phi \in \mathcal{D}_P^{arco}$, *i.e.* tel que $\gamma_P^{arco}(\Phi) = \emptyset$. Hors des deux conditions évidentes, l'un des ensembles de valuations $\gamma^{\mathbb{Z}}(\eta)$ ou $\gamma^{\mathbb{K}}(\mu)$ est vide, ces conditions reposent toutes sur le même principe, annuler l'ensemble des valuations possibles des variables d'indices. En effet les autres incohérences, que

peuvent générer des ensembles de propriétés de tableaux, mènent rarement à l'incohérence globale de la valeur abstraite. Nous l'avons vu lors de la définition de $\gamma_P^{arco}$ (Sec. 6.3) où, lorsque l'incohérence saute aux yeux ($\gamma^{\mathbb{K}}(\psi_p) = \emptyset$), cela ne fait que réduire l'espace des valuations des variables d'indices. Ce n'est donc que lorsque $\gamma^{\mathbb{Z}}(\eta)$ est complètement couvert que l'insatisfaisabilité est acquise : ainsi, les conditions que nous donnons sont à associer pour concourir à cet objectif.

La première est celle que nous venons d'évoquer. Toute propriété de tableaux dont la tranche $p$ implique une propriété insatisfaisable soustrait l'ensemble $\overline{\varphi_p}$ à celui des valuations possibles des variables d'indices.

La deuxième condition porte sur la cohérence des valuations possibles des variables scalaires selon différentes propriétés de tableaux. Comme $\overline{\varphi_p} \Rightarrow \overline{\psi_p}$ et, qu'*a contrario* des tranches qui, elles, ne se chevauchent pas (Déf. 6.2), un ensemble de propriétés $\overline{\varphi_p}$ peuvent s'intersecter, si la conjonction des propriétés $\overline{\psi_p}$ correspondantes est fausse, cette implication soustrait, par contraposition, l'intersection des $\overline{\varphi_p}$ à l'ensemble des valuations possibles des variables d'indices. On prend pour exemple la valeur abstraite $\Phi$ suivante

$$(i \geq 1, x \in [5,6], \{a^0 = 8 \wedge a^1 = 7 \wedge x = 6 \ ; a^{-1} = 7 \wedge x = 5 \ ; \top^{\mathbb{K}}\})$$

sur la partition donnée à la Figure 6.4 (p. 134). On a toujours $\overline{\varphi_1} = (i \geq 2)$, $\overline{\varphi_2} = (i \geq 1)$ mais ici $\overline{\psi_1} \sqcap^{\mathbb{K}} \overline{\psi_2} = (x = 6) \sqcap^{\mathbb{K}} (x = 5) = \bot^{\mathbb{K}}$ : ainsi une valuation où $\rho_i(i) \geq 2$ (*i.e.* où $\rho_i \in \gamma^{\mathbb{Z}}(\overline{\varphi_1}) \cap \gamma^{\mathbb{Z}}(\overline{\varphi_2})$) ne peut appartenir à $\gamma_P^{arco}(\Phi)$ puisque pour cela $\rho_v(x)$ devrait respecter deux contraintes incompatibles. Par contre des valuations où $\rho_i(i) = 1$ et $\rho_v(x) = 5$ peuvent exister.

Enfin, la dernière condition porte sur la cohérence des valuations possibles des cellules des tableaux. Comme nous l'avons vu, ces valuations peuvent dépendre de plusieurs variables de tranche $a^{z_p^j}, z_p^j \in \mathit{Trans}(\psi_p)$ et on a $\varphi_p \oplus z_p^j \Rrightarrow \psi_p \boxplus z_p^j$. Ainsi pour les cellules appartenant à l'intersection de propriétés $\varphi_p \oplus z_p^j$, les valuations possibles doivent appartenir aux valuations représentées par l'intersection des propriétés $\psi_p \boxplus z_p^j$. Si l'intersection de ces propriétés est insatisfaisable, alors les valuations des variables d'indices telles que des cellules peuvent appartenir à l'intersection des propriétés $\varphi_p \oplus z_p^j$ sont impossibles, par contraposition.

On donne, sur l'exemple précédent, deux situations où cette condition est vérifiée. Pour la première, l'incohérence est due à des variables de tranche présentes dans une même propriété de tableau : il s'agit de $a^0$ et $a^1 \in \mathit{Var}(\psi_1)$. On voit immédiatement sur le schéma ci-dessus (Fig. 6.6) que les cellules du tableau $a$ de 2 à $i-1$ sont contraintes par ces deux variables de tranches (on a bien $\varphi_1 \sqcap^{\mathbb{Z}} \varphi_1 \oplus 1 = (2 \leq \ell < i)$), et que ces contraintes sont incompatibles : $a^0 = 7$ de par $\psi_1$ et $a^{1-1} = 8$ de par $\psi_1 \boxplus 1$. Ainsi si $\rho_i(i) \geq 3$ il n'existe pas de valuation possible pour $\rho_a(a)(2), \ldots, \rho_a(a)(\rho_i(i) - 1)$. Par contre si $\rho_i(i) = 2$ la tranche $\varphi_1$ ne représente qu'une cellule, la première, et donc on peut avoir $\rho_a(a)(1) = 7$ et $\rho_a(a)(2) = 8$. Pour la seconde situation, l'incohérence est due à des variables de tranches présentes dans différentes partitions : il s'agit de $a^0 \in \mathit{Var}(\psi_1)$ et $a^{-1} \in \mathit{Var}(\psi_2)$. Cette fois, c'est la cellule du tableau $a$ en $i-1$ qui est contrainte par ces deux variables, et là encore ces contraintes sont incompatibles, si $\rho_i(i) \geq 2$. On notera que l'on a donné deux exemples d'incohérence, pour cette troisième condition, où aucune variable scalaire n'apparaît. Il ne faut pas en conclure qu'en présence de variables scalaires c'est la seconde condition qui jouera : considérer par exemple l'intersecion des propriétés $a^0 = x \sqcap^{\mathbb{K}} a^{-1+1} < x$.

Proposition 6.1 (conditions suffisantes d'insatisfaisabilité). *Soit $P$ une partition et $\Phi \in \mathcal{D}_P^{arco}$. Si $\gamma^{\mathbb{Z}}(\eta) = \emptyset$ ou $\gamma^{\mathbb{K}}(\mu) = \emptyset$, ou encore :*

1. *$\exists P_0 \in \mathcal{P}(P)$ tel que $\forall p \in P_0, \gamma^{\mathbb{K}}(\psi_p) = \emptyset$ ;*
2. *et $\exists P_1, \ldots, P_n \in \mathcal{P}(P)$ tels que $\gamma^{\mathbb{K}}(\mu) \bigcap_{p \in P_i} \gamma^{\mathbb{K}}(\overline{\psi_p}) = \emptyset$ ;*
3. *et $\exists P_{n+1}, \ldots, P_m \in \mathcal{P}(P)$ tels que $\forall p \in P_i, \exists \{z_p^1, \ldots, z_p^{t(p)}\} \subseteq Trans(\psi_p)$ tel que $\gamma^{\mathbb{K}}(\mu) \bigcap_{\substack{p \in P_i \\ j \in [1,t(p)]}} \gamma^{\mathbb{K}}(\psi_p \boxplus z_p^j) = \emptyset$ ;*

*et* $$\gamma^{\mathbb{Z}}(\eta) \subseteq \Big( \underbrace{\bigcup_{p \in P_0} \gamma^{\mathbb{Z}}(\overline{\varphi_p})}_{1} \underbrace{\bigcup_{i \in [1,n]} \big( \bigcap_{p \in P_i} \gamma^{\mathbb{Z}}(\overline{\varphi_p}) \big)}_{2} \underbrace{\bigcup_{i \in [n+1,m]} \{\rho_{|I} \mid \rho \in \bigcap_{\substack{p \in P_i \\ j \in [1,t(p)]}} \gamma^{\mathbb{Z}}(\varphi_p \oplus z_p^j)\}}_{3} \Big),$$

*alors $\gamma_P^{arco}(\Phi) = \emptyset$.*

*Démonstration.* Pour les deux premiers cas, l'insatisfaisabilité est évidente. Pour le dernier cas, on donne pour chacune des trois conditions qui le forment, l'ensemble des valuations des variables d'indices $\rho_i$ qu'elles interdisent. Alors si l'union de ces ensembles inclut $\gamma^{\mathbb{Z}}(\eta)$, on peut directement conclure à l'insatisfaisabilité.

1. Si $P_0 = \emptyset$ alors aucune contrainte sur $\rho_i$ n'est induite. Sinon, pour tout $p \in P_0$, l'hypothèse implique, selon la définition de la fonction de concrétisation, que $\rho_i \notin \gamma^{\mathbb{Z}}(\overline{\varphi_p})$.
2. On suppose qu'il existe au moins une sous-partition vérifiant l'hypothèse. Soit $i \in [1, n]$, supposons qu'il existe $\rho \in \gamma_P^{arco}(\Phi)$ telle que $\rho_i \in \bigcap_{p \in P_i} \gamma^{\mathbb{Z}}(\overline{\varphi_p})$. Alors par définition de la fonction de concrétisation on a $\rho_v \in \bigcap_{p \in P_i} \gamma^{\mathbb{K}}(\overline{\psi_p})$ et $\rho_v \in \gamma^{\mathbb{K}}(\mu)$, ce qui contredit l'hypothèse qui dit que l'intersection de ces deux ensembles est vide.
3. On suppose là encore qu'il existe au moins une sous-partition vérifiant l'hypothèse. Toujours par contradiction, soit $i \in [n+1, m]$, supposons qu'il existe $\rho \in \gamma_P^{arco}(\Phi)$ telle que $\rho_i \in \{\rho_{|I} \mid \rho \in \bigcap_{\substack{p \in P_i \\ j \in [1,t(p)]}} \gamma^{\mathbb{Z}}(\varphi_p \oplus z_p^j)\}$. Alors $\exists k \in \bigcap_{\substack{p \in P_i \\ j \in [1,t(p)]}} K(\varphi_p \oplus z_p^j)$. Soit $p \in P_i$, et $j \in [1, t(p)]$, on définit $k_p^j = k - z_p^j$. On a $k_p^j \in K(\varphi_p)$ et donc par définition de la fonction de concrétisation il existe $\rho_p^j \in R(\psi_p)$ telle que $\rho_a(a)(k_p^j + z_p^j) = \rho_p^j(a^{z_p^j})$, donc $\exists \rho_p^{j'} \in R(\psi_p \boxplus z_p^j)$ telle que $\rho_a(a)(k) = \rho_p^{j'}(a^0)$. Appliqué à tout $i \in [n+1, m]$, on a ainsi que l'intersection $\bigcap_{\substack{p \in P_i \\ j \in [1,t(p)]}} R(\psi_p \boxplus z_p^j)$ n'est pas vide, ce qui contredit l'hypothèse pour peu qu'il existe une valuation des variables scalaires de contenu pour cet ensemble de $\psi_p$, cas particulier couvert par la seconde hypothèse.

□

Remarque. *De cette proposition, on retiendra aussi que notre domaine abstrait peut représenter, via les propriétés de tableaux, un ensemble non-convexe de valuations des variables d'indices, même si $\mathcal{D}^{\mathbb{Z}}$ représente lui des ensembles convexes. La particularité étant que ces « trous » sont prédéfinis, ce sont des intersections de $\overline{\varphi_p}$, puisque la partition, les $\varphi_p$, sont prédéfinis.*

Si l'on souhaitait implanter un tel test de satisfaisabilité, tester l'incohérence des propriétés $\eta$, $\mu$, $\psi_p$ ou leurs intersections ne poserait pas de problème. Il y a par contre deux obstacles. Un majeur, celui de tester l'inclusion de $\gamma^{\mathbb{Z}}(\eta)$ dans la série d'union d'ensembles concrets décrite. Un second obstacle, plus mineur, est qu'il faut trouver ces tranches et ces variables de tranches vérifiant ces conditions, ce qui fait beaucoup de cas à considérer.

Ainsi, on ne donne pas de test de satisfaisabilité, pour lequel on s'en remettra à l'opérateur de normalisation, qui évite simplement ces deux obstacles, au prix de la précision.

### 6.4.2 Opération de normalisation

On aura compris que l'objectif n'est pas ici de calculer une représentation canonique de nos valeurs abstraites, représentation que nous n'avons pas même définie. Dans cette section on donne un opérateur renvoyant *une* certaine forme normalisée de nos valeurs abstraites, qui s'est avérée suffisamment précise pour amener l'analyse à découvrir des invariants intéressants dans plusieurs programmes complexes.

Cet opérateur (Déf. 6.8, p. 142) est donné sous la forme de règles réécrivant la valeur abstraite, dont on illustre le pouvoir de déduction par des exemples. Pour gagner en précision, on effectue deux traitements préalables sur la valeur abstraite $\Phi$ :

- on propage aux propriétés de tableaux où apparaissent des variables scalaires, l'information fournie par $\mu$ sur ces variables *seulement*. Par exemple si l'on a $\psi_p = (a^0 = x)$ et $\mu = (x \leq y \wedge z = 1)$ on aura après cette phase $\psi_p = (a^0 = x \leq y)$. Ainsi, seules les variables scalaires liées au contenu des cellules de tableaux ont vocation à se retrouver dans les propriétés de tableaux, pour des raisons d'efficacité. Les fonctions de transfert portant sur des variables scalaires concourent à ce même objectif (Sec. 6.6.2) et ce sont elles qui initient la présence des variables scalaires dans les propriétés de tableaux : par exemple l'affectation $a[i] := x$ fera apparaître $x$ dans l'une d'entre elles ;
- on normalise les éléments abstraits qui composent $\Phi$ : $\eta$, $\mu$ et les $\psi_p$ obtenus par la propagation précédente.

Les deux premières règles appliquées propagent de l'information entre les propriétés de tableaux.

- La règle 1 s'intéresse aux variables scalaires présentes dans les propriétés de tableaux.

$$\frac{\overline{\varphi_p} \sqsubseteq^{\mathbb{Z}} \overline{\varphi_{p'}}}{\Phi[\psi_p \to \psi_p \sqcap^{\mathbb{K}} \overline{\psi_{p'}}]}\,1$$

  Les propriétés sur ces variables ne dépendent pas de $\ell$ et peuvent donc être propagées à d'autres propriétés de tableaux. En effet, si on a $\overline{\varphi_p} \Rightarrow \overline{\varphi_{p'}}$, ce que l'on teste avec $\sqsubseteq^{\mathbb{Z}}$, comme $\overline{\varphi_{p'}} \Rightarrow \overline{\psi_{p'}}$ par définition, on a $\overline{\varphi_p} \Rightarrow \overline{\psi_{p'}}$ : ainsi la règle propage-t-elle dans ce cas $\overline{\psi_{p'}}$ à $\psi_p$.

  Exemple. *Soit la valeur abstraite* $(i \geq 1, \top^{\mathbb{K}}, \{x \leq b^0 \,; x > a^0 = 1\})$ *définie sur la partition donnée à la Figure 6.4 (p. 134) : les propriétés de tableaux qu'elle représente sont donc* $\forall \ell, 1 \leq \ell < i \Rightarrow x \leq b[\ell]$ *et* $\forall \ell, \ell = i \geq 1 \Rightarrow x > a[\ell] = 1$. *On peut imaginer que cette valeur abstraite résulte de l'application dernière de*

*la conditionnelle* $a[i] = 1$. *Ainsi l'information apportée sur* $a[i]$ *entraîne* $x > 1$, *information qui peut-être propagée à* $\psi_1$ *puisque* $\overline{\varphi_1} = (i \geq 2) \sqsubseteq^{\mathbb{Z}} (i \geq 1) = \overline{\varphi_2}$. *La règle renvoie donc la valeur abstraite* $(i \geq 1, \top^{\mathbb{K}}, \{1 < x \leq b^0 \ ; x > a^0 = 1\})$.

– La règle 2 porte sur la propagation des informations portées par les variables de tranches au sein des propriétés de tableaux.

$$\frac{\varphi_p \oplus z \sqsubseteq^{\mathbb{Z}} \varphi_{p'} \oplus z'}{\Phi[\psi_p \to \psi_p \sqcap^{\mathbb{K}} (\psi_{p'} \boxplus (z' - z))]}\ 2$$

De la même manière que pour les variables scalaires, si on a $\varphi_p \oplus z \Rrightarrow \varphi_{p'} \oplus z'$, comme on sait que $\varphi_{p'} \oplus z' \Rrightarrow \psi_{p'} \boxplus z'$, on a $\varphi_p \oplus z \Rrightarrow \psi_{p'} \boxplus z'$, et donc $\varphi_p \Rrightarrow \psi_{p'} \boxplus (z' - z)$. Cette dernière information peut donc être propagée à $\psi_p$. La question qui se pose est celle des constantes $z, z' \in \mathbb{Z}$ pour lesquelles on souhaite tester que $\varphi_p \oplus z \sqsubseteq^{\mathbb{Z}} \varphi_{p'} \oplus z'$ : on choisit de restreindre la règle de déduction uniquement aux $z$ et $z'$ telles que $z \in \mathit{Trans}(\psi_p)$ et $z' \in \mathit{Trans}(\psi_{p'})$.Cette heuristique suppose simplement que si les fonctions de transferts n'ont fait apparaître que des variables de tranches de la forme $a^0$, il y a peu de chance qu'un invariant portant sur des cellules représentées par différentes tranches existe. *A contrario*, si elles ont fait apparaître des variables de tranches de la forme $a^z, z \neq 0$ (*e.g.*, l'affectation $a[i] := a[i-1] + 1$) il y a de fortes chances pour qu'une propagation soit impérative pour trouver un invariant intéressant.

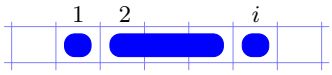
Exemple. *Soit la partition* $\{\varphi_1 = (\ell = 1) \ ; \varphi_2 = (2 \leq \ell \leq 10)\}$. *Si l'on a la propriété de tableaux* $\forall \ell, 2 \leq \ell \leq 10 \Rightarrow a[\ell - 1] \leq a[\ell]$, *i.e.* $\psi_2 = (a^{-1} \leq a^0)$, *on souhaite que soit explicité que* $a[1] \leq a[2]$, *ce que l'application de la règle nous donne : on a* $\varphi_1 \oplus 0 = (\ell = 1) \sqsubseteq^{\mathbb{Z}} (1 \leq \ell \leq 9) = \varphi_2 \oplus -1$ *(inclusion testée puisque* $-1 \in \mathit{Trans}(\psi_2)$*), et donc la propriété* $\psi_2 \boxplus -1 = (a^{-1+1} \leq a^{0+1})$ *est propagée à* $\psi_1$.

On notera que chaque application de cette règle peut introduire de nouvelles variables de tranches $a^z$ dans des propriétés de tableaux. Cependant ces changements ne seront pas pris en compte pour pouvoir appliquer cette même règle. Autrement dit, les $z, z'$ permettant de vérifier la prémisse de la règle 2 doivent être présents dans les $\psi_p$ obtenus après l'application de la règle 1.

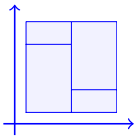
On remarque que les définitions exactes de ces deux premières règles (Déf. 6.8, p. 142) utilisent la propriété $\eta$ dans leurs prémisses. Dans la pratique ceci permet de gagner souvent en précision : on illustre donc ce point par un exemple réel.

Exemple 6.1. *Le programme présenté à la Figure 7.3(d) (p. 182), qui initialise un tableau, a les tranches suivantes dans sa partition en tête de boucle :* $\varphi_1 = (\ell = 1), \varphi_2 = (2 \leq \ell < i \leq n)$ *et* $\varphi_3 = (2 \leq \ell = i \leq n)$ *(illustrées Fig. 6.7(a)). Très souvent on retrouve des tranches comme* $\varphi_2$ *dans les partitions, pouvant être vides selon la valeur d'une variable d'indice (ici les variables* $i$ *ou* $n$*). Par conséquent,* $\varphi_3 \oplus -1$ *peut soit représenter la même cellule que* $\varphi_1$ *(si* $i = 2$*) soit la dernière cellule que représente* $\varphi_2$ *(si* $i \geq 3$*). Il n'y a donc pas d'inclusion entre ces trois propriétés, à moins de prendre en compte* $\eta$ *: lors de la première itération de l'analyse on a* $\eta = (i = 2)$ *en tête de boucle, ce qui permet d'activer la règle 2 et de propager de l'information à* $\psi_3$ *(en l'occurrence que* $a^{-1} = 7$*). Cependant* $\eta$ *va rapidement se stabiliser à* $2 \leq i \leq n$ *à ce point de contrôle, et aucune propagation ne sera plus possible. Or ceci est primordial pour montrer que toutes les*

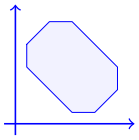
*valeurs du tableau sont supérieures ou égales à 7 : de nouvelles règles seront proposées pour couvrir ce besoin, fréquent en pratique.*




(a) Tranches $\varphi_1, \varphi_2$ et $\varphi_3$ de l'Exemple 6.1



(b) Une valeur abstraite imaginaire : $\eta$ représente l'espace carré extérieur et $\overline{\varphi_1}, \ldots, \overline{\varphi_4}$ représentent les quatre espaces rectangulaires internes



(c) Le résultat de $(\eta \setminus^{\mathbb{K}} \overline{\varphi_1}) \setminus^{\mathbb{K}} \ldots \setminus^{\mathbb{K}} \overline{\varphi_4}$

Figure 6.7 – Opération de normalisation : quelques schémas utiles à la compréhension de certains exemples

Suivent les règles propageant de l'information des propriétés de tableaux aux propriétés sur les scalaires.

– Lorsqu'une propriété de tableau a pour propriété $\psi_p$ l'élément $\bot^{\mathbb{K}}$, on sait que la propriété sur les variables d'indices peut être renforcée par $\neg\overline{\varphi_p}$. La règle 3 utilise l'opérateur $\setminus^{\mathbb{Z}}$ défini précédemment (Sec. 6.2.1.1), pour obtenir une sur-approximation de cette propagation.

$$\frac{\psi_p = \bot^{\mathbb{K}}}{\Phi[\eta \to (\eta \setminus^{\mathbb{Z}} \overline{\varphi_p})]}\,3$$

Exemple 6.2. *Le programme présenté à la Figure 7.4(c) (p. 186) utilise une « sentinelle » pour s'assurer de ne pas parcourir un tableau au delà de ses bornes. En tête de boucle, l'analyse trouve justement à partir d'une certaine itération la propriété* $\eta = (1 \leq i \leq n)$*. Après la conditionnelle de la boucle, l'analyse découvre aussi la propriété de tableau suivante :* $(1 \leq \ell = n = i) \Rightarrow \bot^{\mathbb{K}}$*. La règle* 3 *est alors activée, avec* $\overline{\varphi_p} = \overline{(1 \leq \ell = n = i)} = (i = n \geq 1)$ *et son application (l'analyse utilise en l'occurrence l'opérateur* $\setminus^{zone}$ *donné à la Figure 6.2(a), p. 130) amène* $\eta = (1 \leq i < n)$*. C'est exactement la propriété nécessaire pour qu'à la suite de l'incrémentation de* $i$ *la contrainte* $i \leq n$ *soit conservée en tête de boucle.*

– La règle 4 repose simplement sur le fait que si $\eta \Rightarrow \overline{\varphi_p}$, alors $\overline{\psi_p}$ est vérifiée. Cette dernière propriété peut donc être propagée à $\mu$.

$$\frac{\eta \sqsubseteq^{\mathbb{Z}} \overline{\varphi_p}}{\Phi[\mu \to \mu \sqcap^{\mathbb{K}} \overline{\psi_p}]}\,4$$

Les deux règles suivantes, les règles 5 et 6, concluent à l'insatisfaisabilité de la valeur abstraite dès lors que $\eta$, respectivement $\mu$, est $\bot^{\mathbb{Z}}$, respectivement $\bot^{\mathbb{K}}$.

Des différentes conditions d'insatisfaisabilité que l'on a énoncé précédemment (Sec. 6.4.1), on voit que seul un sous-ensemble d'entre elles sont couvertes par l'opération de normalisation. En effet, pour que celle-ci explicite l'insatisfaisabilité d'une valeur abstraite due à des incohérences entre ses propriétés de tableaux, ces incohérences doivent

pouvoir être exprimées par les règles 1 ou 2 dans la valeur abstraite (des $\psi_p$ sont alors réécrits à $\bot^{\mathbb{K}}$), puis la règle 3, en ôtant de $\eta$ les $\overline{\varphi_p}$ correspondants les uns après les autres, doit réussir à réécrire $\eta$ à $\bot^{\mathbb{Z}}$. Alors seulement la règle 5 peut réécrire la valeur abstraite à $\bot^{arco}$. On donne des exemples où l'une de ces étapes failli, illustrant ainsi les limites, choisies, de l'opérateur de normalisation sur ce sujet. Bien entendu, au prix d'une plus grande complexité, de nouvelles règles pourraient très bien prendre en compte ces exemples.

Exemple. *Si on reprend la partition donnée à l'exemple 6.1 (et représentée à la Figure 6.7(a)) on se rappelle que si* $\eta = (i \geq 2)$ *alors* $\varphi_1$, $\varphi_2 \oplus -1$ *et* $\varphi_3 \oplus -1$ *s'intersectent, mais qu'il n'y a pas de relation d'inclusion entre ces trois propriétés. Si* $\psi_1 = (a^0 = 7)$, $\psi_2 = (a^{-1} = 9)$ *et* $\psi_3 = (a^{-1} = 5)$, *on a* $\psi_1 \sqcap^{\mathbb{K}} \psi_2 \boxplus -1 = \bot^{\mathbb{K}}$ *et* $\psi_1 \sqcap^{\mathbb{K}} \psi_3 \boxplus -1 = \bot^{\mathbb{K}}$. *La Proposition 6.1 (p. 138) des conditions d'insatisfaisabilité affirme donc que* $\neg(\overline{\varphi_1 \sqcap^{\mathbb{Z}} \varphi_2 \oplus -1})$, *i.e. que* $\neg(n+1 \geq i \geq 3)$ *et que* $\neg(\overline{\varphi_1 \sqcap^{\mathbb{Z}} \varphi_3 \oplus -1})$, *i.e. que* $\neg(i = 2 \leq n)$. *Ici c'est la règle 2 qui n'est pas faite pour repérer les deux incohérences ci-dessus et donc* $\eta$ *ne sera pas remplacé par* $(\eta \setminus^{\mathbb{Z}} (i = 2 \leq n)) \setminus^{\mathbb{Z}} (n+1 \geq i \geq 3) = \bot^{\mathbb{Z}}$ *et l'insatisfaisabilité de cette valeur abstraite ne sera pas révélée.*

*Pour illustrer le fait que la règle* 3 *peut faillir à montrer que la conjonction de* $\eta$ *et plusieurs propriétés* $\overline{\varphi_i}$ *est fausse, on prend pour exemple les propriétés représentées graphiquement à la Figure 6.7(b) : si* $\mathcal{D}^{\mathbb{Z}}$ *représente des ensembles convexes, quel que soit la propriété* $\overline{\varphi_i}$ *que la règle traite en premier, l'opération* $(\eta \setminus^{\mathbb{Z}} \overline{\varphi_i})$ *induit une approximation puisque l'espace résultant n'est pas convexe. Si* $\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{oct}$, *le domaine des octogones, après les applications successives de la règle 3 on trouve la propriété représentée à la Figure 6.7(c) : on est loin d'avoir trouvé* $\bot^{\mathbb{Z}}$.

Enfin, la dernière règle provient de la remarque faite en introduction, lors de l'analyse prospective du programme de copie de tableaux (Sec. 6.1.1). On remarquait d'une part que si la propriété $\overline{\varphi_p}$ est fausse vis-à-vis de $\eta$, alors $\psi_p$ peut-être n'importe qu'elle propriété, l'implication $\varphi_p \Rrightarrow \psi_p$ est toujours vérifiée, et d'autre part que pour la précision de l'analyse il était intéressant de choisir $\psi_p = \bot^{\mathbb{K}}$. Ce choix est d'ailleurs cohérent par rapport à $\sqsubseteq^{arco}$ ; il est mis en œuvre par la règle 7.

Définition 6.8 (opération de normalisation). *Soit* $\Phi \in \mathcal{D}_P^{arco}$. *On définit* $\mu_p$ *comme la propriété* $\mu$ *restreinte à l'ensemble de variables* $V_p = \{v \in V \mid \exists v' \in Var(\psi_p)$ *telle que* $v = v' \vee v$ *contraint* $v'$ *dans* $\mu\}$. *L'opération de normalisation consiste à :*

- *calculer* $\Phi' = (\overline{\eta}, \overline{\mu}, \{\overline{\psi_p \sqcap^{\mathbb{K}} \mu_p}\}_{p \in P})$ ;
- *appliquer les règles suivantes, dans l'ordre, sur* $\Phi'$ :

$$\frac{\eta \sqcap^{\mathbb{Z}} \overline{\varphi_p} \sqsubseteq^{\mathbb{Z}} \overline{\varphi_{p'}}}{\Phi[\psi_p \to \psi_p \sqcap^{\mathbb{K}} \overline{\psi_{p'}}]}\,1 \qquad \frac{\eta \sqcap^{\mathbb{Z}} \varphi_p \oplus z \sqsubseteq^{\mathbb{Z}} \varphi_{p'} \oplus z'}{\Phi[\psi_p \to \psi_p \sqcap^{\mathbb{K}} (\psi_{p'} \boxplus (z'-z))]}\,2 \quad \begin{array}{l} \text{où } z \in Var(\psi_p, \text{ tel qu'après la règle 1}), \\ z' \in Var(\psi_{p'}, \text{ tel qu'après la règle 1}) \end{array}$$

$$\frac{\psi_p = \bot^{\mathbb{K}}}{\Phi[\eta \to (\eta \setminus^{\mathbb{Z}} \overline{\varphi_p})]}\,3 \qquad \frac{\eta \sqsubseteq^{\mathbb{Z}} \overline{\varphi_p}}{\Phi[\mu \to \mu \sqcap^{\mathbb{K}} \overline{\psi_p}]}\,4$$

$$\frac{\eta = \bot^{\mathbb{Z}}}{\bot^{arco}}\,5 \qquad \frac{\mu = \bot^{\mathbb{K}}}{\bot^{arco}}\,6$$

$$\frac{\eta \sqcap^{\mathbb{Z}} \overline{\varphi_p} = \bot^{\mathbb{Z}}}{\Phi[\psi_p \to \bot^{\mathbb{K}}]}\,7$$

Proposition 6.2 (correction de l'opération de normalisation). *La valeur abstraite résultante de l'opération de normalisation (Déf. 6.8), que l'on note* $\Phi^*$*, est plus précise et a la même concrétisation que* $\Phi$*.*

*Démonstration.* La correction des deux traitements préalables à l'application des règles ne pose pas de problème. Il reste à montrer que l'application de chaque règle sur une valeur abstraite $\Phi^1$ renvoie une valeur abstraite plus précise $\Phi^2 \sqsubseteq_P^{arco} \Phi^1$ et représentant la même valeur concrète $\gamma^{arco}(\Phi^1) = \gamma_P^{arco}(\Phi^2)$. La propriété de précision étant évidente pour chaque règle, il suffit, par monotonicité de $\gamma_P^{arco}$ (Thm. 6.1, p. 135), de montrer que $\gamma^{arco}(\Phi^1) \subseteq \gamma^{arco}(\Phi^2)$ pour prouver leur correction.

1. Si $\gamma^{\mathbb{K}}(\psi_p) = \emptyset$ alors l'opération est invariante. Sinon, soit $\rho \in \gamma^{arco}(\Phi^1)$ : si on suppose que $\rho_i \notin \gamma^{\mathbb{Z}}(\overline{\varphi_p})$ alors on conclut de la même manière que pour la règle 7. Sinon, si $\rho_i \in \gamma^{\mathbb{Z}}(\overline{\varphi_p})$ alors l'hypothèse amène $\rho_i \in \gamma^{\mathbb{Z}}(\overline{\varphi_{p'}})$ et donc par définition on a $\rho_v \in \gamma^{\mathbb{K}}(\overline{\psi_p})$ et $\rho_v \in \gamma^{\mathbb{K}}(\overline{\psi_{p'}})$. Ainsi on ne peut pas avoir $\gamma^{\mathbb{Z}}(\psi_p^2) = \emptyset$, puisque $\gamma^{\mathbb{K}}(\psi_p) \neq \emptyset$ et $\gamma^{\mathbb{K}}(\overline{\psi_p}) \cap \gamma^{\mathbb{K}}(\overline{\psi_{p'}}) \neq \emptyset$, et donc il n'y a pas de contradiction sur $\rho_i$. On a bien $\rho_v \in \gamma^{\mathbb{K}}(\overline{\psi_p^2}) = \gamma^{\mathbb{K}}(\overline{\psi_p}) \cap \gamma^{\mathbb{K}}(\overline{\psi_{p'}})$. Résultat : $\rho_a$ n'est pas plus contraint dans $\Phi^2$.
2. Soit $\rho \in \gamma^{arco}(\Phi^1)$. Là encore, si $\gamma^{\mathbb{K}}(\psi_p) = \emptyset$ l'opération est invariante et on ne regarde que le cas, où $\rho_i \in \gamma^{\mathbb{Z}}(\overline{\varphi_p})$. Donc $K(\varphi_p) \neq \emptyset$. Soit $k \in K(\varphi_p)$ on a $k + z \in K(\varphi_p \oplus z)$ et par hypothèse $k + z \in K(\varphi_{p'} \oplus z')$ et donc $k + z - z' \in K(\varphi_{p'})$. Donc par définition, $\exists \rho' \in R(\psi_{p'})$ tel que $\forall a^w \in Dom(\rho), \rho_a(a)(k + z - z' + w) = \rho'(a^w)$, donc $\exists \rho' \in R(\psi_{p'} \boxplus (z' - z))$ tel que $\forall a^w \in Dom(\rho), \rho_a(a)(k + w) = \rho'(a^w)$. On conclut donc que $\rho_a$ n'est pas plus contraint dans $\Phi^2$ et, du fait de l'existence de $\rho'$ et que $\gamma^{\mathbb{K}}(\psi_p) \neq \emptyset$, que $\rho_v \in \gamma^{\mathbb{K}}(\overline{\psi_p^2})$ et que $\gamma^{\mathbb{Z}}(\psi_p^2) \neq \emptyset$. Donc $\rho_v$ et $\rho_i$ ne sont pas plus contraint dans $\Phi^2$.
3. Soit $\rho \in \gamma^{arco}(\Phi^1)$. Comme $\gamma^{\mathbb{K}}(\psi_p) = \emptyset$, par définition $\rho_i \notin \gamma^{\mathbb{Z}}(\overline{\varphi_p})$. On conclut par correction de l'opérateur $\setminus^{\mathbb{Z}}$ que $\rho_i \in \gamma^{\mathbb{Z}}(\eta^2)$. Il n'y a aucun doute sur le fait que $\rho_v$ et $\rho_a$ ne sont pas plus contraint dans $\Phi^2$.
4. Soit $\rho \in \gamma^{arco}(\Phi^1)$. On a aucune restriction nouvelle sur $\rho_i$. Comme $\rho_i \in \gamma^{\mathbb{Z}}(\overline{\varphi_p})$ par hypothèse, on a $\rho_v \in \gamma^{\mathbb{K}}(\overline{\psi_p})$ et donc $\rho_v \in \gamma^{\mathbb{K}}(\overline{\psi_p}) \sqcap^{\mathbb{K}} \gamma^{\mathbb{K}}(\mu)$. Ainsi $\rho_v \in \gamma^{\mathbb{K}}(\mu^2)$ et $\rho_a$ n'est pas plus contraint dans $\Phi^2$.
5. L'hypothèse induit que $\gamma^{arco}(\Phi^1) = \emptyset$. Or par définition $\gamma^{arco}(\Phi^2) = \emptyset$.
6. Idem.
7. Soit $\rho \in \gamma^{arco}(\Phi^1)$. Comme $\sqcap^{\mathbb{Z}}$ est exact, on a $\gamma^{\mathbb{Z}}(\eta^1) \cap \gamma^{\mathbb{Z}}(\overline{\varphi_p}) = \emptyset$. Par définition $\rho_i \in \gamma^{\mathbb{Z}}(\eta^1)$ donc $\rho_i \notin \gamma^{\mathbb{Z}}(\overline{\varphi_p})$. Donc le fait que $\gamma^{\mathbb{Z}}(\psi_p^2) = \emptyset$ ne fait qu'expliciter cette dernière contrainte. De plus comme $\rho_i \notin \gamma^{\mathbb{Z}}(\overline{\varphi_p})$, aucune restriction supplémentaire ne porte sur les valeurs de $\rho_v$ possibles. Enfin, cela implique également que $K(\varphi_p) = \emptyset$ et donc aucune restriction sur $\rho_a$ n'est perdue.

□

Aucune de ces règles n'a pu être affaiblie, sans sacrifier la capacité de l'analyse à découvrir des invariants intéressants de programmes effectuant des traitements sensiblement différents sur des tableaux. Comme nous l'avons vu à l'Exemple 6.2 (p. 141), l'analyse d'un programme utilisant une « sentinelle » impose l'existence de la règle 3. La nécessité des règles 4 à 7 n'est pas discutable, la dernière notamment puisqu'elle est

primordiale pour l'analyse des programmes traversant des tableaux. Les règles 1 et 2 enfin sont nécessaires à l'analyse de programmes de tri ou d'initialisation un peu complexe de tableaux. Cependant, elles ne sont pas suffisantes, comme nous l'avons sous-entendu à la fin de l'Exemple 6.1 (p. 140), que l'on reprend.

Il s'agissait d'ailleurs d'un programme d'initialisation (Fig. 7.3(d), p. 182), pour lequel on s'intéresse toujours aux trois mêmes tranches (Fig. 6.7(a), p. 141). À une certaine itération, l'analyse a calculé la valeur abstraite suivante au point de contrôle précédent l'affectation $a[i] := a[i-1]+1$ (ligne **4**) : $\eta = (2 \leq i \leq n)$, $\psi_1 = (a^0 = 7)$, $\psi_2 = (a^0 = a^{-1} + 1 \geq 8)$, $\psi_3 = \top^{\mathbb{K}}$. En clair $a[1] = a[2] - 1 = 7$ et $\forall \ell, 2 \leq \ell < i \leq n$ on a $a[\ell] = a[\ell-1] + 1 \geq 8$. Vient l'affectation, dont le traitement (*cf.* Sec. 6.6.2.2) amène, comme on peut s'y attendre $\psi_3 = (a^0 = a^{-1} + 1)$, avant de faire appel à l'opération de normalisation. Et celle-ci, pour trouver que $a^0 \geq 8$, en clair $a[i] \geq 8$, doit s'apercevoir que $(\varphi_3 \oplus -1) \Rightarrow (\varphi_1 \vee \varphi_2)$, ce qui permet de conclure car $(\varphi_1 \vee \varphi_2) \Rrightarrow a^0 = 7 \vee a^0 \geq 8$.

Si on généralise cet exemple, on en déduit une nouvelle règle de propagation, que l'on donne à la Définition 6.9 suivante, et qui utilise l'opérateur $\mathbb{M}^{\mathbb{Z}}$, défini Section 6.2.1.1, pour tester la disjonction en question. Cette règle 2*b* est accompagnée de sa sœur, la règle 1*b*, pendant de la règle 1.

Définition 6.9 (opération de normalisation améliorée). *Cet opérateur consiste en l'opérateur de normalisation (Déf. 6.8) auquel sont ajoutées les deux règles suivantes :*

$$\frac{\eta \sqcap^{\mathbb{Z}} \overline{\varphi_p} \sqsubseteq^{\mathbb{Z}} \overline{\varphi_{p'}} \quad \mathbb{M}^{\mathbb{Z}} \overline{\varphi_{p''}}}{\Phi[\psi_p \to \psi_p \sqcap^{\mathbb{K}} (\overline{\psi_{p'}} \sqcup^{\mathbb{K}} \overline{\psi_{p'}})]} 1b$$

$$\frac{\eta \sqcap^{\mathbb{Z}} \varphi_p \oplus z \sqsubseteq^{\mathbb{Z}} \varphi_{p'} \oplus z' \quad \mathbb{M}^{\mathbb{Z}} \varphi_{p''} \oplus z''}{\Phi[\psi_p \to \psi_p \sqcap^{\mathbb{K}} (\psi_{p'} \boxplus (z'-z) \sqcup^{\mathbb{K}} \psi_{p''} \boxplus (z''-z))]} 2b$$

*où* $z \in Var(\psi_p$, *tel qu'après les règles 1, 1b*$)$,
$z' \in Var(\psi_{p'}$, *tel qu'après les règles 1, 1b*$)$,
$z'' \in Var(\psi_{p''}$, *tel qu'après les règles 1, 1b*$)$

*L'opérateur de normalisation amélioré est correct.*

*Démonstration.* Les preuves de correction de ces deux règles reposent sur celle de l'opérateur $\mathbb{M}^{\mathbb{Z}}$. Pour le reste, elle sont similaires à celles données pour les règles 1 et 2. □

On considéra dans la suite du manuscrit que l'opérateur de normalisation est l'opérateur de normalisation amélioré.

## 6.5 Partitionnement symbolique

Jusqu'à présent nous avons pu faire abstraction du calcul des partitions symboliques en chaque point de contrôle. On peut imaginer de nombreuses analyses différentes pour calculer ces partitions, passer d'une analyse à une autre sur telle ou telle partie du programme, etc. Quoi qu'il en soit, toute analyse doit prendre en compte que la précision de l'analyse de contenu des tableaux sera très impactée par un partitionnement trop peu détaillé et que son coût augmentera drastiquement avec un partitionnement trop détaillé (Sec. 7.2.3).

On décrit dans cette section une analyse de partitionnement implantée comme une analyse par interprétation abstraite, qui dans la plupart des cas se comporte comme une simple propagation. Les partitions générées par cette analyse ont permis l'analyse du

contenu des tableaux de nombreux programmes.

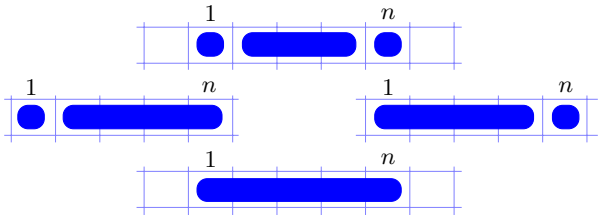
Les travaux relatés dans [GRS05], sur lesquels on s'est appuyé pour définir nos valeurs abstraites à base de partitions statiques, définissent ces dernières à partir d'une heuristique très simple (Section 4.3.2) : si une cellule de tableau $\langle expr_i \rangle$ est affectée ou testée, on souhaite garder le maximum d'informations possible sur cette cellule. Pour toutes les autres cellules, on se contente de les abstraire ensemble, plus précisément d'abstraire ensemble celles avant la cellule $\langle expr_i \rangle$ et celles après cette cellule. Dès lors que le programme affecte ou teste plus de deux cellules de tableaux, par exemple les cellules aux indices $i$ et $j$, pour pouvoir garder le maximum d'informations sur ces cellules, il faut séparer les différents cas d'alias possibles entre les variables d'indices $i$ et $j$. Ainsi, la partition donnée pour le programme est $(\ell < j, i), (\ell = j < i), (j < \ell < i), (\ell = i < j)$, $(\ell = i = j), (j < \ell = i), (i < \ell < j), (i < \ell = j), (i, j < \ell)$. Si on redonne ici le principe de cette construction, c'est parce qu'il reste au cœur de la construction de nos partitions (Déf. 6.11, p. 149). Cependant, hors de ce principe, aucune analyse n'est donnée dans [GRS05] pour construire véritablement ces partitions : de nombreuses hypothèses sont cachées, la partition est attachée à un programme et non pas à un point de contrôle, etc. Ainsi, lors d'un stage à *Microsoft Research* nous avons définit un domaine abstrait pour construire, avec une approche plus sémantique, les partitions symboliques, et ce à chaque point de contrôle.

### 6.5.1 Domaine abstrait sémantique des partitions

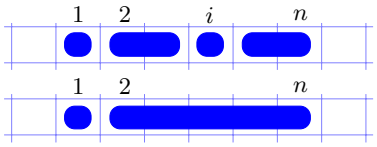
Ce domaine abstrait, noté $\mathcal{D}^{arpa}$ (*arpa* pour *ar*ray *pa*rtitions), aura simplement pour éléments toutes les partitions pouvant être exprimées sur $\mathcal{D}^{\mathbb{Z}}$. La sémantique de ces éléments abstraits $P \in \mathcal{D}^{arpa}$ est, à un point de contrôle : « les tranches $\varphi_p \in P$ sont probablement nécessaires pour représenter l'invariant sur le contenu des tableaux ». D'autre part, cette définition fait qu'à un point de contrôle $c$ cible de deux transitions, l'une partant du point de contrôle $c_1$, l'autre du point de contrôle $c_2$, la partition en $c$ nécessite probablement toutes les tranches des partitions en $c_1$ et $c_2$ pour représenter l'invariant qui n'est autre que la disjonction des invariants en $c_1$ et $c_2$. Cependant, si on prend pour exemple les partitions $P_1 = \{(\ell = 1); (2 \leq \ell \leq n)\}$ et $P_2 = \{(1 \leq \ell < n); (2 \leq \ell = n)\}$ comme étant celles nécessaires pour représenter les invariants en $c_1$ et $c_2$ (Fig. 6.8(a)), prendre pour partition en $c$ les quatre tranches ci-dessus enfreint la définition de nos partitions (Déf. 6.2), imposant qu'aucune des tranches qui les composent ne se chevauchent. La solution consiste à prendre la partition $P$ ayant pour tranches toutes les intersections entre les tranches de $P_1$ et $P_2$ : $P = \{(\ell = 1 < n); (2 \leq \ell < n); (2 \leq \ell = n)\}$. Il est évident que les invariants sur $P_1$ et sur $P_2$ peuvent alors s'exprimer sur $P$. On remarque alors deux choses : premièrement, l'opération d'union du treillis $P_1 \sqcup^{arpa} P_2$ renverra une partition plus détaillée que $P_1$ et $P_2$. Deuxièmement, bien qu'elle soit plus détaillée, cette partition n'est pas forcément suffisamment détaillée, comme c'est le cas pour la partition $P$ ci-dessus. Pour s'en rendre compte il faut regarder au niveau des propriétés de tableaux. Si le premier invariant sur $P_1$ était « la première cellule contient 1, toutes les cellules suivantes contiennent zéro », et le second invariant sur $P_2$ « la dernière cellule (position $n$) contient 1, toutes les cellules précédentes contiennent zéro », l'invariant le plus précis qui pourra être exprimé sur $P$ sera « la première et la dernière cellules contiennent zéro ou 1, toutes les autres zéro ». Ainsi l'opération d'union du treillis $P_1 \sqcup^{arpa} P_2$ renverra une

partition insuffisamment détaillée, autrement dit introduira une approximation, lorsque les tranches de $P_1$ et $P_2$ se chevauchent.

On définit alors un treillis des partitions (Déf. 6.10). On ordonne donc nos partitions de telle manière que $P_1 \sqsubseteq^{arpa} P_2$ implique que $P_2$ soit plus détaillée que $P_1$ [2]. Cette notion de détail n'a de sens que si $P_1$ et $P_2$ représentent, toutes tranches confondues, le même ensemble de cellules (ce qui est le cas dans notre exemple précédent). La relation définie par « toute tranche de $P_2$ est incluse dans une tranche de $P_1$ » est un ordre qui implique cette relation de détail. La Figure 6.8 donne des exemples de partitions ordonnées ainsi.




(a) Un exemple de partitions illustrant les opérations $\sqcup^{arpa}$ et $\sqcap^{arpa}$




(b) Les deux partitions que l'on trouve, tous points de contrôle confondus, dans le programme de recherche du maximum

Figure 6.8 – Des exemples de partitions ordonnées selon $\sqsubseteq^{arpa}$

Nos éléments abstraits sont des ensembles de tranches formant une partition :

$$\mathcal{D}^{arpa} = \Big\{ \{\varphi_1, \ldots, \varphi_n\} \mid \varphi_p \in \mathcal{D}^{\mathbb{Z}} \wedge \gamma^{\mathbb{Z}}(\varphi_p) \neq \emptyset \\ \wedge \forall p, p' \, (p \neq p) \Rightarrow (\gamma^{\mathbb{Z}}(\varphi_p) \cap \gamma^{\mathbb{Z}}(\varphi_{p'}) = \emptyset) \Big\}.$$

Selon l'ordre $\sqsubseteq^{arpa}$ que l'on vient de définir, il y a un plus petit élément naturel, $\bot^{arpa}$, qui est l'ensemble contenant simplement la tranche $\top^{\mathbb{Z}}$, la moins détaillée qu'il soit, et un plus grand élément naturel, $\top^{arpa}$, qui est l'ensemble vide.

Enfin, pour définir les opérations de borne supérieure $P_1 \sqcup^{arpa} P_2$ et de borne inférieure $P_1 \sqcap^{arpa} P_2$, on utilise le graphe non-orienté $G(P_1, P_2)$ ayant un sommet pour chaque tranche de $P_1$ et $P_2$ et une arrête reliant $\varphi_{p_1}$ à $\varphi_{p_2}$ si ces deux tranches s'intersectent. La définition de la borne supérieure est alors directe. Celle de la borne inférieure utilise les composantes connexes (notées $CC$) de $G$ car celles-ci constituent un ensemble de tranches de $P_1$ et de $P_2$ tel que toute autre tranche de $P_1$ et $P_2$ n'intersecte pas cet ensemble : ainsi, en construisant une tranche pour chaque composante connexe, qui soit l'union des tranches de la composante, on obtient bien une partition, où chaque tranche de $P_1$ et $P_2$ est inclut dans une tranche de cette partition.

Définition 6.10 (treillis des partitions). *L'ensemble* $(\mathcal{D}^{arpa}, \sqsubseteq^{arpa}, \sqcup^{arpa}, \sqcap^{arpa}, \bot^{arpa}, \top^{arpa})$ *dont les opérateurs sont défini, pour* $\forall P_1, P_2 \in \mathcal{D}^{arpa}$*, par :*

– $P_1 \sqsubseteq^{arpa} P_2 \Leftrightarrow \forall \varphi_q \in P_2 \, \exists \varphi_p \in P_1$ *tel que* $\varphi_q \sqsubseteq^{\mathbb{Z}} \varphi_p$ *;*

2. Le domaine de partitionnement de traces [MR05] ordonne de la même manière ses partitions (de traces).

– $P_1 \sqcup^{arpa} P_2 = \left\{ \{\varphi_{p_1} \sqcap^{\mathbb{Z}} \varphi_{q_1}, \ldots, \varphi_{p_m} \sqcap^{\mathbb{Z}} \varphi_{q_m}\} \mid (\varphi_{p_k}, \varphi_{q_k}) \in G(P_1, P_2) \right\}$ ;
– $P_1 \sqcap^{arpa} P_2 = \left\{ \{\bigsqcup^{\mathbb{Z}}_{\varphi \in C_1}, \ldots, \bigsqcup^{\mathbb{Z}}_{\varphi \in C_m}\} \mid C_1, \ldots, C_m \in CC(G(P_1, P_2)) \right\}$

*forme un treillis.*

*Démonstration.* La démonstration est sans difficultés. Par exemple, pour ce qui est de l'ordre, la réflexivité de $\sqsubseteq^{arpa}$ est assurée par celle de $\sqsubseteq^{\mathbb{Z}}$ et sa transitivité est clairement impliquée par celle de $\sqsubseteq^{\mathbb{Z}}$. Enfin, son antisymétrie est garantie par le fait que les tranches d'une partition ne se chevauchent pas (Déf. 6.2) : soit $\varphi_{p_2} \in P_2$ alors $P_1 \sqsubseteq^{arpa} P_2$ implique l'existence de $\varphi_{p_1} \in P_1$ telle que $\varphi_{p_2} \sqsubseteq^{\mathbb{Z}} \varphi_{p_1}$ et $P_2 \sqsubseteq^{arpa} P_1$ implique l'existence d'une tranche de $P_2$ incluant $\varphi_{p_1}$, qui ne peut-être que $\varphi_{p_2}$ puisque sinon elle chevaucherait $\varphi_{p_2}$. Par antisymétrie de $\sqsubseteq^{\mathbb{Z}}$ on conclut que $\varphi_{p_1} = \varphi_{p_2}$. □

La Figure 6.8(a) illustre les opérations du treillis des partitions. On définit à présent les opérations sémantiques du domaine abstrait.

#### 6.5.1.1 Opérations sémantiques

Avant de comprendre la définition des fonctions de transfert du domaine abstrait des partitions, on décrit le comportement que l'on attend de l'analyse de partitionnement sur le programme de recherche du maximum, donné ci-contre.

```
x := a[1] ;
i := 2 ;
while i ≤ n do
    if x < a[i] then
        x := a[i]
    i := i + 1
```

Figure 6.9 –

On donne pour partition au point de contrôle initial la partition $\{(\ell \geq 1)\}$, puisqu'on sait que nos tableaux s'indicent à partir de 1, et pour les autres points de contrôle, la partition $\bot^{arpa}$, *i.e.* pas de partitionnement. On regarde la première itération. La première instruction consiste à définir le maximum, stocké dans la variable $x$ comme étant le contenu de la première cellule. On souhaite donc retenir, au point de contrôle suivant, l'invariant que $x = a[1]$ et l'invariant que l'on avait sur l'ensemble du tableau, donc il est nécessaire à ce point de contrôle d'avoir la partition $\{(\ell = 1); (\ell \geq 2)\}$. Celle-ci sera construite, comme toutes les autres, en faisant l'union ($\sqcup^{arpa}$) de la partition au point de contrôle précédent et celle avec les tranches nécessaires pour représenter exactement, comme dans [GRS05], l'information que fournie l'instruction : ici la partition $\{(\ell < 1); (\ell = 1); (\ell > 1)\}$. La partition obtenue, $P = \{(\ell = 1); (\ell \geq 2)\}$, reste inchangée jusqu'à la conditionnelle vérifiant si la cellule $i$ du tableau contient une valeur plus grande que le maximum des cellules précédentes. Si on choisit d'effectuer simplement l'union avec la partition $\{(\ell < i); (\ell = i); (\ell > i)\}$ pour obtenir la partition au point de contrôle suivant, le résultat est assez imprécis [3]. En effet, sachant ici que $2 \leq i \leq n$, on sait que l'information fournie par la garde portera sur les cellules $2 \leq \ell \leq n$. Ainsi on fera plutôt l'union avec la partition suivante, où les deux dernières tranches sont introduites pour que cette partition soit couvrante : $\{(2 \leq \ell \leq n \wedge \ell < i); (2 \leq \ell \leq n \wedge \ell = i); (2 \leq \ell \leq n \wedge \ell > i); (\ell < 2); (2 \leq \ell \wedge \ell > n);\}$.

3. Il s'agirait de la partition $\{(\ell = 1 < i); (2 \leq \ell < i); (2 \leq \ell = i); (2 \leq \ell \wedge \ell > i)\}$ une fois enlevées les tranches insatisfaisables avec l'information fournie par la pré-analyse ($2 \leq i \leq n$). Ces tranches suffiraient pour cette version du programme, mais pas pour une version où la boucle parcourrait le tableau en sens inverse (de $n$ à 2) puisqu'il faudrait la tranche ($2 \leq i < \ell \leq n$) pour exprimer l'invariant de boucle. On souhaite une analyse de partitionnement plus robuste en pratique.

On obtient alors, pour cette branche du programme, la partition suivante, illustrée Figure 6.8(b) [4] :

$$P' = \{(\ell = 1); (2 \leq \ell \leq n \wedge \ell < i); (2 \leq \ell = i \leq n); (2 \leq \ell \leq n \wedge \ell > i); \\ (\ell > n \wedge \ell \geq 2)\}. \tag{6.2}$$

Avant l'instruction d'incrémentation, l'analyse effectue l'union des deux branches formées par la conditionnelle, *i.e.* de $P'$ et $P'$. L'instruction d'incrémentation ne modifie pas cette partition. Vient la seconde itération de la boucle, avec l'union en tête de boucle de $P$ et $P'$, donnant $P'$, qui apparaît alors comme une partition stable. L'analyse passe donc au point de contrôle final, pour lequel on trouve la partition $P'$. À ce point de contrôle cette partition n'est plus satisfaisante. En effet toutes ses tranches ne sont pas nécessaires pour représenter l'invariant ($x$ est le maximum du tableau) qu'une seule propriété de tableau suffit à exprimer : $\forall \ell, 1 \leq \ell \leq n \Rightarrow a[\ell] \leq x$. Cependant, si on note que $i$ est une variable non vivante à ce point de contrôle, on peut en conclure, non pas que le partitionnement selon $i$ n'est jamais nécessaire pour représenter l'invariant, mais que de toute façon l'information dépendante de $i$ ne sera pas utilisée. Par contre, on considère que la variable $n$ est un paramètre du programme et donc qu'elle sera réutilisée par la suite. On suppose donc que $n$ est vivante au dernier point de contrôle. Ainsi on regroupera dans $P'$ au moins les tranches dépendant de $i$, ce qui nous donnera la partition $P'' = \{(\ell = 1); (2 \leq \ell \leq n); (\ell > n \wedge \ell \geq 2)\}$ (illustrée Fig. 6.8(b)) au dernier point de contrôle du programme.

Au final la taille des partitions crée pour ce programme simple est plutôt correcte. En général, après l'obtention du point fixe, on peut diminuer la taille d'une partition à un point de contrôle en ôtant les tranches qui sont insatisfaisables selon l'invariant des variables d'indice.

Ainsi on a identifié à travers cet exemple deux autres analyses nécessaires pour effectuer notre partitionnement.

- une analyse d'atteignabilité des variables d'indice. On supposera qu'elle est implantée en utilisant le domaine abstrait $D^{\mathbb{Z}}$. Son résultat permettra de créer des partitions au plus juste.
- une analyse de durées de vie des variables, en l'occurrence des variables d'indice. Son résultat permettra, lorsqu'une variable d'indice utilisée pour définir des tranches de la partition ne s'avère plus être vivante, de simplifier cette partition. En effet il y a de fortes chances qu'il n'y ait plus aucun intérêt à maintenir ces tranches.

On supposera que ces deux analyses sont effectuées avant celle que l'on est en train de définir. Il ne reste plus d'ailleurs qu'à présenter les fonctions de transfert de cette analyse de partitionnement.

**Fonctions de transferts** En analysant le contenu des tableaux de plus en plus de programmes, il est apparu nécessaire de prendre en compte plus d'instructions que celles proposées dans [GRS05] pour partitionner :

- nous avons vu lorsque nous avons déroulé l'analyse ci-dessus, qu'il était nécessaire de détailler la partition après l'instruction $x := a[1]$. Ainsi, lorsqu'une et une seule cellule de tableau $\langle expr_i \rangle$ apparaîtra en partie droite d'une affectation à un scalaire, on découpera également la partition selon $\langle expr_i \rangle$.

4. On omet dans cette figure la tranche où $\ell > n$.

– les programmes qui recherchent un certain contenu dans un tableau, peuvent utiliser une variable d'indice spécifiquement pour se souvenir de la cellule $\langle expr_i \rangle$ où le contenu a été trouvé (par exemple la variable $j$ dans le programme, donné à la Figure 7.4(b), p. 186, cherchant le premier élément non nul). S'il s'avère que la partition courante permet de représenter précisément l'information en $\langle expr_i \rangle$, autrement dit qu'il existe une tranche $(\ell = \langle expr_i \rangle)$ dans cette partition, alors il est pertinent, après une affectation $j := \langle expr_i \rangle$ de découper également selon $j$ : on a toutes les chances de pouvoir représenter précisément un invariant du programme.

Les définitions 6.11 et 6.12 donnent l'abstraction des fonctions de transferts en conséquence de tous ces choix heuristiques. Elles utilisent les trois opérations suivantes :

– l'opération *Split* qui pour une expression d'indice $\langle expr_i \rangle$ renvoie la partition générique : $Split(\langle expr_i \rangle) = \{(\ell < \langle expr_i \rangle); (\ell = \langle expr_i \rangle); (\ell > \langle expr_i \rangle)\}$.
– une opération plus complexe *ContextSplit* qui renvoie une paire de partitions pour une expression d'indice $\langle expr_i \rangle$ en fonction de l'invariant $\eta^0$ sur les variables numériques calculé par la pré-analyse. Les tranches de cette paire de partitions $(P^{in}, P^{out})$ doivent être disjointes deux à deux et former ensemble une couverture de $\mathbb{Z}^n$, comme c'est le cas de l'ensemble des tranches de la partition renvoyée par *Split*. $P^{in}$ n'est formée que d'une seule tranche représentant l'espace de valeurs possible de $\langle expr_i \rangle$, qui donne les limites dans lesquelles on va partitionner (avec *Split*). Dans l'exemple précédent, sur le programme de recherche du maximum, $P^{in}$ est $\{(2 \leq \ell \leq n)\}$. $P^{out}$ est donc $\{(\ell < 2); (2 \leq \ell \wedge \ell > n)\}$. On voit plus loin à travers un exemple comment cette opération peut être définie en pratique pour le domaine abstrait $\mathcal{D}^{zone}$.
– enfin l'opération *Forget* qui à une partition et une variable d'indice associe le multi-ensemble de tranches $Forget(P)(i) = \{\exists i.\varphi_p \mid \varphi_p \in P\}$. Cet ensemble ne forme pas une partition. Pour la partition $P'$ de notre exemple l'opération renvoie $\{(\ell = 1); (2 \leq \ell \leq n); (2 \leq \ell \leq n); (2 \leq \ell \leq n); (\ell > n \wedge \ell \geq 2)\}$.

Définition 6.11 (Partitionnement selon l'affectation ou la conditionnelle). *Soit* $P \in \mathcal{D}^{arpa}$. *Si* $\langle expr_i \rangle$ *est une constante* $c$ *:*

$$\left.\begin{array}{l} \{\!|a[\langle expr_i \rangle] := \langle expr \rangle|\!\}^{arco}(P) \\ \{\!|x := a[\langle expr_i \rangle]|\!\}^{arco}(P) \\ \{\!|g|\!\}^{arco}(P) \text{ où } a[\langle expr_i \rangle] \in g \end{array}\right\} = \; P \sqcup^{arpa} Split(c) \; .$$

*Sinon,*

$$\left.\begin{array}{l} \{\!|a[\langle expr_i \rangle] := \langle expr \rangle|\!\}^{arco}(P) \\ \{\!|x := a[\langle expr_i \rangle]|\!\}^{arco}(P) \\ \{\!|g|\!\}^{arco}(P) \text{ où } a[\langle expr_i \rangle] \in g \end{array}\right\} = P \sqcup^{arpa} P'$$

$$\text{où} \quad \begin{array}{l} P' = (P^{in} \sqcup^{arpa} Split(\langle expr_i \rangle)) \cup P^{out} \\ \text{et } (P^{in}, P^{out}) = ContextSplit(\langle expr_i \rangle) \end{array}$$

*et si* $j \notin Var(\langle expr_i \rangle)$ *et* $\exists \varphi_p \in P$ *telle que* $\varphi_p \Rightarrow \langle expr_i \rangle$ *:*

$$\left.\begin{array}{l} \{\!|j := \langle expr_i \rangle|\!\}^{arco}(P) \end{array}\right\} = P \sqcup^{arpa} P'$$

$$\text{où} \quad \begin{array}{l} P' = (P^{in} \sqcup^{arpa} Split(j)) \cup P^{out} \\ \text{et } (P^{in}, P^{out}) = ContextSplit(\langle expr_i \rangle) \end{array} .$$

Toute affectation ou conditionnelle peut également changer la partition si une variable « meurt » après elle. Ce qui veut dire qu'une même fonction de transfert peut modifier

de deux manières différentes (selon la Définition 6.11 précédente et la Définition 6.12 suivante) la partition : c'est par exemple le cas pour l'affectation $a[j+1] := x$ à la ligne **8** du tri par insertion (Fig. 6.10). N'appliquer que la seconde définition, qui simplifie la partition, ferait perdre de la précision à l'analyse de contenu. Afin d'apporter le plus de précision possible à cette analyse, un point de contrôle doit être ajouté à l'automate interprété analysé lorsque ce cas particulier est détecté. Ce point de contrôle $k_2$ scinde la transition $k_0 \to k_1$ portant la fonction de transfert en deux : une première transition avec la même fonction de transfert et une seconde transition avec la fonction identité. La première définition est appliquée sur la transition $k_0 \to k_2$ (modifiant $P$ en $P''$) et la seconde définition sur la transition $k_2 \to k_1$ (modifiant $P''$ en $P'$). Ainsi, lors de l'analyse du contenu, la partition $P''$ en $k_2$ sera suffisamment détaillée pour recueillir l'information apportée par la fonction de transfert, permettant à cette analyse de trouver un résultat plus fort, qui sera lui adapté à la partition simplifiée $P'$ en $k_1$.

Définition 6.12 (Simplification selon la durée de vie). *Soit $P \in \mathcal{D}^{arpa}$. Soit $i \in Var(P)$ dont c'est la dernière utilisation dans une action $a$ ou une garde $g$. Alors*

$$\left.\begin{matrix} \{\!|a|\!\}^{arco}(P) \\ \{\!|g|\!\}^{arco}(P) \end{matrix}\right\} = P' \sqcap^{arpa} P' \text{ où } P' = Forget(P)(i).$$

Remarque 6.2. *Avec ces définitions l'analyse $\mathcal{D}^{arpa}$ crée des partitions ayant la propriété de couvrir l'espace ($\ell \geq 1$), autrement dit elle crée des partitions telles que :*

$$(\ell \geq 1) \Rightarrow \bigvee_{p \in P} \varphi_p.$$

*Cette propriété est la conséquence : des partitions initiales, qui sont $\{(\ell \geq 1)\}$ ; des partitions renvoyées par Split évidemment couvrantes ainsi que celles renvoyées par ContextSplit puisque la spécification de cette opération l'exige ; enfin du fait que la propriété de couverture est conservée par l'opération d'union $\mathcal{D}^{arpa}$ et l'opération Forget.*

*Dans le cas où les partitions sont nettoyées après l'analyse des tranches insatisfaisables selon $\eta^0$, cette propriété de couverture est affaiblie à $(\ell \geq 1) \wedge \eta^0$ pour chaque point de contrôle.*

On termine cette section en déroulant le tout début de l'analyse de partitionnement que l'on vient de définir sur le tri par insertion, donné ci-contre. On prend $\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{zone}$. Après l'affectation $x := a[i]$ (l. **3**) on trouve la même partition $P'$ que pour l'exemple précédent (Éq. 6.2, p. 148). Suit l'affectation $j := i - 1$ qui laisse inchangée la partition puisqu'aucune de ses tranches ne porte sur la cellule $i - 1$. C'est la conditionnelle suivante qui va modifier $P'$ car elle porte sur $a[j]$ (l. **5**). L'invariant numérique calculé par l'analyse précédente de zones est $\eta^0 = (2 \leq i \leq n \wedge 1 \leq j < i \leq n)$ à ce point de contrôle. On applique la définition 6.11. Notre implantation de $ContextSplit(\langle expr_i \rangle)^{zone}$ cherche dans $\eta^0$ une borne inférieure et une borne supérieure de $\langle expr_i \rangle$, bornes qui peuvent être symboliques. Parfois plusieurs bornes sont possibles, comme ici $i - 1$ et $n$ pour la borne supérieure. On isole

```
i := 2 ;
while i ≤ n do
    x := a[i] ;
    j := i − 1 ;
    while j ≥ 1 and
    a[j] > x do
        a[j + 1] := a[j]
        j := j − 1
    a[j + 1] := x ;
    i := i + 1
```

Figure 6.10 –

parmi ces bornes celles qui sont les plus « serrées ». Ici c'est $i-1$ car $i-1 < n$. Si plusieurs bornes sont ainsi isolées, notre algorithme choisit de préférence une borne déjà présente dans la partition, sinon elle effectue un choix par défaut sur un critère permettant de conserver le déterminisme de l'analyse. Une fois ces bornes $\langle expr_i \rangle_{\text{inf}}$ et $\langle expr_i \rangle_{\text{sup}}$ choisies, la définition de $\mathit{ContextSplit}(\langle expr_i \rangle)^{zone}$ est directe [5] :

$$\begin{aligned}\mathit{ContextSplit}(\langle expr_i \rangle)^{zone} \quad = \quad & (\{(\langle expr_i \rangle_{\text{inf}} \leq \ell \leq \langle expr_i \rangle_{\text{sup}})\}^{in}, \\ & \{(\ell < \langle expr_i \rangle_{\text{inf}}); (\ell \geq \langle expr_i \rangle_{\text{inf}} \wedge \ell > \langle expr_i \rangle_{\text{sup}})\}^{out}).\end{aligned}$$

On reprend le calcul de la partition après la conditionnelle qui a pour résultat $P''' = P' \sqcup^{arpa} P''$. Selon ce qui vient d'être décrit on a $P'' = \{(1 \leq \ell < i \wedge \ell < j)\,;\,(1 \leq \ell = j < i)\,;\,(1 \leq \ell < i \wedge \ell > j); (\ell < 1); (\ell \geq 1 \wedge \ell \geq i)\}$. $P'''$ contient alors 14 tranches. Si on ne regarde que celles qui sont satisfaisables selon $\eta^0$ on a les tranches suivantes, présentées de manière arborescente (de gauche à droite) :

$(\ell = 1 \wedge \ell < i \wedge \ell < j)$ $\quad$ $(\ell = 1 \wedge \ell < i \wedge \ell = j)$
$(2 \leq \ell \leq n \wedge \ell < i \wedge \ell < j)$ $\quad$ $(2 \leq \ell \leq n \wedge \ell < i \wedge \ell = j)$ $\quad$ $(2 \leq \ell \leq n \wedge \ell < i \wedge \ell > j)$
$(2 \leq \ell = i \leq n)$
$(2 \leq \ell \leq n \wedge \ell > i)$
$(\ell \geq 2 \wedge \ell \geq i \wedge \ell > n)$

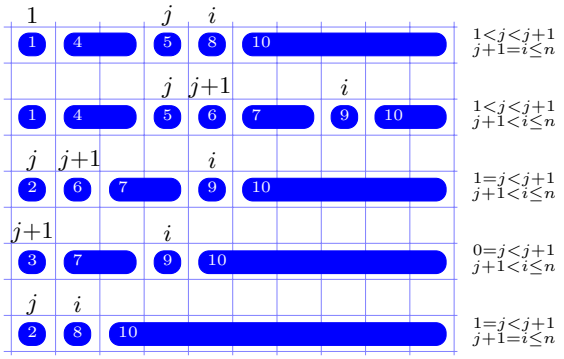
Dernière instruction à modifier la partition de la boucle interne, l'affectation $a[j+1] := a[j]$ (l. **6**) va introduire dans l'espace $2 \leq \ell \leq i$ un partitionnement selon $j+1$ de la même manière que précédemment. On arrête ici notre suivi de l'analyse. Son résultat final en tête de boucle interne est donné à la Figure 6.11(a). Ce résultat se comprend directement à partir de la partition donnée ci-dessus et de l'opération de partitionnement selon $j+1$ évoquée ci-dessus. On pourrait être étonné de l'absence dans la partition donnée ci-dessus de la tranche $\varphi_3$. C'est parce que l'on a pas donné l'ensemble des 14 tranches dans lesquelles elle apparaît. Si cette tranche est insatisfaisable dans la boucle, elle ne l'est pas en tête de boucle où $j$ peut valoir zéro. Cette tranche n'a pas été oubliée car le nettoyage des partitions est fait *a posteriori*. On donne à la Figure 6.11(b) une représentation intéressante de la partition selon une partition de $\eta^0$. Chaque « configuration » est telle que toute cellule du tableau est représentée par une seule tranche. La seconde configuration représente le cas général de l'algorithme. Les autres sont des cas particuliers, que l'analyse de contenu va probablement considérer à certains points de contrôle ou étapes de son calcul (notre analyse considère chacune de ces configurations au moins une fois durant son calcul). Par exemple la dernière configuration, cas particulier où $(j = 1 \wedge i = 2)$, sera considérée par notre analyse lors de la première itération.

En pratique toutes ces tranches ne sont pas nécessaires pour découvrir l'invariant de tri. Notamment l'heuristique de partitionnement a considéré la première cellule à part supposant qu'un invariant plus fort existerait sur la partie $2 \leq \ell \leq n$ du tableau que sur la partie $1 \leq \ell \leq n$. Il n'en est rien, et d'ailleurs cet algorithme du tri par insertion pourrait très bien débuter avec l'initialisation $i := 1$ sans que son résultat final en soit changé.

5. C'est exagéré. D'une part, on ne donne ici que le cas où une des deux bornes est symbolique. D'autre part, on aurait pu choisir pour ce cas d'autres tranches pour l'ensemble *out* : $\{(\ell < \langle expr_i \rangle_{\text{inf}} \wedge \ell \leq \langle expr_i \rangle_{\text{sup}})\,;\,(\ell > \langle expr_i \rangle_{\text{sup}})\}^{out}$, ou encore l'ensemble plus détaillé $\{(\ell < \langle expr_i \rangle_{\text{inf}} \wedge \ell \leq \langle expr_i \rangle_{\text{sup}})\,;\,(\ell \geq \langle expr_i \rangle_{\text{inf}} \wedge \ell > \langle expr_i \rangle_{\text{sup}}); (\ell < \langle expr_i \rangle_{\text{inf}} \wedge \ell > \langle expr_i \rangle_{\text{sup}})\}^{out}$. Il s'agit d'un choix heuristique.

$\varphi_1 = (\ell = 1 \wedge \ell < i \wedge \ell < j)$
$\varphi_2 = (\ell = 1 \wedge \ell < i \wedge \ell = j)$
$\varphi_3 = (\ell = 1 \wedge \ell < i \wedge \ell > j)$
$\varphi_4 = (2 \leq \ell \leq n \wedge \ell < i \wedge \ell < j)$
$\varphi_5 = (2 \leq \ell \leq n \wedge \ell < i \wedge \ell = j)$
$\varphi_6 = (2 \leq \ell \leq n \wedge \ell < i \wedge \ell = j + 1)$
$\varphi_7 = (2 \leq \ell \leq n \wedge \ell < i \wedge \ell > j + 1)$
$\varphi_8 = (2 \leq \ell = i \leq n \wedge \ell = j + 1)$
$\varphi_9 = (2 \leq \ell = i \leq n \wedge \ell > j + 1)$
$\varphi_{10} = (2 \leq \ell \leq n \wedge \ell > i)$
$\varphi_{11} = (\ell \geq 2 \wedge \ell > i \wedge \ell > n)$

(a) Partition




(b) Représentation schématique (sans $\varphi_{11}$)

Figure 6.11 – En (a) la partition calculée par $\mathcal{D}^{arpa}$ en tête de boucle interne (ligne **5**) du tri par insertion (Fig. 6.10). À ce point de contrôle $\eta^0 = (0 \leq j \leq i-1 \wedge 2 \leq i \leq n)$. En (b) une représentation de cette même partition selon différents cas particuliers (dont la disjonction implique $\eta^0$) qui montre comment les différentes tranches couvrent le contenu du tableau.

**Conclusion** Nous verrons encore à quel point le coût de l'analyse de contenu est fortement influencé par le nombre de tranches dans la partition (Sec. 7.2.3). Cependant nous n'avons pas d'objectif de preuve qui pourrait nous aider à générer des partitions moins détaillées. On ne peut pas non plus s'appuyer sur l'analyse du contenu qui, si elle était concomitante, pourrait permettre de rattraper dynamiquement un partitionnement trop détaillé.

En pratique, l'analyse de partitionnement que nous venons de présenter, a permis l'analyse de nombreux exemples et s'est montrée plutôt robuste. Il reste encore beaucoup de travail pour qu'elle puisse fournir des partitions permettant à l'analyse de contenu de passer à l'échelle.

### 6.5.2 Adaptation des valeurs abstraites de $\mathcal{D}^{arco}$

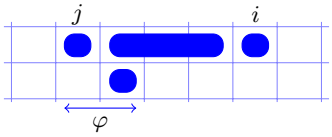
On s'intéresse ici aux opérations nécessaires pour adapter une valeur abstraite de $\mathcal{D}^{arco}$ d'un point de contrôle à un autre lorsque les partitions en ces deux points de contrôle sont différentes. Nous avions déjà évoqué cette opération d'adaptation à la Section 6.2.2. Elle repose, comme plusieurs opérations sémantiques définies ensuite (Sec. 6.6), sur un opérateur permettant d'extraire d'une valeur abstraite les propriétés de tableaux associées à une tranche symbolique quelconque.

#### 6.5.2.1 Extraction de propriétés de tableaux

Étant donné une valeur abstraite $\Phi \in \mathcal{D}_P^{arco}$ et une propriété $\varphi$ sur $I \cup \{\ell\}$, on souhaite extraire de $\Phi$ une propriété $\psi$ sur $V \cup S(A)$, aussi forte que possible, telle que $\Phi$ implique $\varphi \Rrightarrow \psi$.

À la Figure 6.12, on donne un exemple de valeur abstraite, définie sur la partition $\varphi_1 = (\ell = j < i)$, $\varphi_2 = (j < \ell < i)$ et $\varphi_3 = (j < \ell = i)$, représentée en (a). On

$$\Phi = ((j < i), \top^{\mathbb{K}}, \{\psi_1 = (a^0 = a^1 = x); \psi_2 = (a^0 \leq a^1 \leq x); \psi_3 = (a^0 \leq x)\})$$




(a) Tranches $\varphi_1, \varphi_2, \varphi_3$ suivi de la tranche $\varphi_1 \oplus 1$ puis de la propriété $\varphi$ pour laquelle on calcule $\psi^{\Phi}(\varphi)$

$$\mathit{Inter}^{\Phi}(\varphi) = \{1; 2; 3\},\ \mathit{Trans}^{\Phi}(\varphi) = \{(1, 1)\}$$

$$\begin{aligned} \psi^{\Phi}(\varphi) &= (a^0 = a^1 = x) \\ &\quad \sqcup^{\mathbb{K}}((a^0 \leq a^1 \leq x) \sqcap^{\mathbb{K}} (a^{-1} = a^0 = x)) \\ &\quad \sqcup^{\mathbb{K}}((a^0 \leq x) \sqcap^{\mathbb{K}} (a^{-1} = a^0 = x)) \\ &= (a^0 = x) \end{aligned}$$

(b) Les différents ensembles calculés par l'opérateur d'extraction et son résultat pour $\varphi$ sur $\Phi$

Figure 6.12 – Exemple d'extraction de propriétés de tableaux pour $\varphi = (j \leq \ell \leq j + 1)$

souhaiterait connaître les propriétés de tableaux vérifiées par les cellules de $j$ à $j + 1$, cellules représentées, toujours en (a), par la propriété $\varphi = (j \leq \ell \leq j + 1)$.

Une première approximation consiste à trouver un ensemble de tranches $p \in P$ telles que $\varphi \Rightarrow \bigvee \varphi_p$ : on pourrait alors proposer $\psi = \sqcup^{\mathbb{K}}\psi_p$. Nous avons vu à quel point il peut-être hasardeux de se reposer sur l'opérateur $\mathbb{M}^{\mathbb{Z}}$ pour repérer une disjonction, et, de plus, coûteux. C'est alors que l'on tire parti de la propriété de couverture (à $\eta$ près) des partitions proposées dans cette section (Rem. 6.2, p. 150) : si $(\varphi \sqcap^{\mathbb{Z}} \eta) \sqsubseteq^{\mathbb{Z}} \sqcup^{\mathbb{Z}}_{p' \in P} \varphi_{p'}$ alors, en prenant toute tranche $p$ intersectant $\varphi$, on est assuré que $(\varphi \sqcap^{\mathbb{Z}} \eta) \Rightarrow \bigvee \varphi_p$. Ainsi la définition de l'opérateur d'extraction (Déf. 6.13) utilise l'ensemble des tranches qu'intersecte $\varphi$, noté $\mathit{Inter}^{\Phi}(\varphi)$. Si on reprend notre exemple on trouve $\mathit{Inter}^{\Phi}(\varphi) = \{1; 2; 3\}$ et donc l'opérateur pourrait proposer la propriété $\psi = (a^0 = a^1 = x) \sqcup^{\mathbb{K}} (a^0 \leq a^1 \leq x) \sqcup^{\mathbb{K}} (a^0 \leq x) = (a^0 \leq x)$.

Ce résultat ne serait pas très satisfaisant puisque la première propriété de tableau indique que $\forall \ell, \ell = j < i \Rightarrow a[\ell] = a[\ell + 1] = x$, ce qui implique, avec $\eta = (j < i)$, que $\forall \ell, j \leq \ell \leq j + 1 \Rightarrow a[\ell] = x$ *i.e.* $\varphi \Rrightarrow (a^0 = x)$. Si l'on souhaite aussi tirer parti des informations portées par les variables de tranches on ne veut pas, pour les mêmes raisons que précédemment, chercher des tranches $p'$ et des variables de tranches $a^{z_{p'}}$ dans ces tranches telles que $\varphi \Rightarrow \bigvee \varphi_p \oplus z_p$. Par contre, si on revient à l'ensemble des tranches $p$ de $\mathit{Inter}^{\Phi}(\varphi)$, on a bien entendu $\varphi \Rightarrow \bigvee(\varphi_p \wedge \varphi)$, et pour certains $p$ qu'il peut exister une tranche $p'$ et une variable de tranches $a^{z_{p'}}$ telles que $(\varphi_p \wedge \varphi) \Rightarrow \varphi_{p'} \oplus z_{p'}$. On propose alors comme seconde approximation $\psi = \sqcup^{\mathbb{K}}(\exists p' \ ?\ \psi_p \sqcap^{\mathbb{K}} \psi_{p'} \boxplus z_{p'} : \psi_p)$ [6]. Sur notre exemple on a $\varphi_2 \sqcap^{\mathbb{Z}} \varphi = (\ell = j + 1 < i)$ et $\varphi_3 \sqcap^{\mathbb{Z}} \varphi = (\ell = j + 1 = i)$, toutes les deux incluses dans $\varphi_1 \oplus 1 = (\ell = j + 1 \leq i)$. Les étapes du calcul où apparaît donc des intersections avec la propriété $\psi_1 \boxplus 1 = (a^{-1} = a^0 = x)$ sont données à la Figure 6.12(b). Le résultat de l'opération d'extraction, notée $\psi^{\Phi}(\varphi)$, est alors celui souhaité : $(a^0 = x)$.

Ainsi, la définition de l'opérateur (Déf. 6.13) utilise aussi l'ensemble des tranches $\varphi_{p'} \oplus z'$ intersectant $\varphi$. Cet ensemble, celui des candidats potentiels pour couvrir une intersection entre $\varphi$ et d'autres tranches, est noté $\mathit{Trans}^{\Phi}(\varphi)$. L'opérateur assemble alors, comme proposé en seconde approximation, les différentes propriétés correspondantes aux tranches de $\mathit{Inter}^{\Phi}(\varphi)$ et $\mathit{Trans}^{\Phi}(\varphi)$.

6. On prend dans le premier cas l'intersection de $\psi_p$ et $\psi_{p'} \boxplus z'$ car on ne sait pas laquelle des deux propriétés est la plus précise, même lorsque $\Phi$ est normalisée.

Définition 6.13 (extraction d'une propriété de tableaux). *Soit $\Phi \in \mathcal{D}_P^{arco}$ et $\varphi \in \mathcal{D}^{\mathbb{Z}}$ définies sur $I \cup \{\ell\}$. Soit les deux ensembles suivants :*

$$Inter^{\Phi}(\varphi) = \{p \in P \mid \varphi_p \sqcap^{\mathbb{Z}} \varphi \sqcap^{\mathbb{Z}} \eta \neq \bot^{\mathbb{Z}}\}$$
$$Trans^{\Phi}(\varphi) = \{(p, z) \mid a^z \in Var(\psi_p) \wedge \varphi_p \oplus z \sqcap^{\mathbb{Z}} \varphi \sqcap^{\mathbb{Z}} \eta \neq \bot^{\mathbb{Z}}\}.$$

*On définit la propriété $\psi^{\Phi}(\varphi)$ telle que sachant $\Phi$, $\varphi \Rrightarrow \psi^{\Phi}(\varphi)$ :*

$$\psi^{\Phi}(\varphi) = \begin{cases} \bot^{\mathbb{K}} & \text{si } \varphi = \bot^{\mathbb{Z}} \\ \bigsqcup^{\mathbb{K}}_{p \in Inter^{\Phi}(\varphi)} \left( \psi_p \bigsqcap^{\mathbb{K}}_{\substack{(q,z) \in Trans^{\Phi}(\varphi) \\ \varphi_p \sqcap^{\mathbb{Z}} \varphi \sqcap^{\mathbb{Z}} \eta \sqsubseteq^{\mathbb{Z}} \varphi_q \oplus z}} \psi_q \boxplus z \right) & \text{sinon, si } (\varphi \sqcap^{\mathbb{Z}} \eta) \sqsubseteq^{\mathbb{Z}} \bigsqcup^{\mathbb{Z}}_{p \in P} \varphi_p \\ \top^{\mathbb{K}} & \text{sinon} \end{cases}$$

*Bien entendu, si $\Phi$ est normalisée avant l'opération, la propriété extraite est plus précise.*

*Démonstration.* Pour prouver la correction de cet opérateur, on montre que d'ajouter la propriété de tableaux $\varphi \Rrightarrow \psi$ à $\Phi$ ne change pas sa concrétisation. Ainsi on définit la valeur abstraite $\Phi' = (\eta, \mu, \{\psi_p\}_{p \in P} \cup \{\psi^{\Phi}(\varphi)\})$ et on montre que $\gamma^{arco}_P(\Phi) \subseteq \gamma^{arco}_{P \cup \{\varphi\}}(\Phi')$, l'inclusion inverse étant évidente. Même si $P \cup \{\varphi\}$ ne représente pas une partition au sens de la définition 6.2, puisque $\varphi$ peut intersecter une tranche de $P$, la définition de la fonction de concrétisation est toujours valide car elle ne repose pas sur cette propriété des partitions, qu'aucune tranche ne s'intersecte.

Soit $\rho \in \gamma^{arco}(\Phi)$. On abrégera le test « $\varphi_p \sqcap^{\mathbb{Z}} \varphi \sqcap^{\mathbb{Z}} \eta \sqsubseteq^{\mathbb{Z}} \varphi_q \oplus z$ » par $Incl(p, (q, z))$. On a $\rho_i \in \gamma^{\mathbb{Z}}(\eta')$ et $\rho_v \in \gamma^{\mathbb{K}}(\mu')$. Soit $p$ tel que $\gamma^{\mathbb{K}}(\psi'_p) = \emptyset$. Si $p \in P$ on a bien $\rho_i \notin \gamma^{\mathbb{Z}}(\overline{\varphi_p})$. Sinon, pour tout $p \in Inter^{\Phi}(\varphi)$, on a $\gamma^{\mathbb{K}}(\psi_p) \bigcap_{(q,z) \in Trans^{\Phi}(\varphi), Incl(p,(q,z))} \gamma^{\mathbb{K}}(\psi_q \boxplus z) = \emptyset$. Si l'on reprend la preuve donnée pour la troisième condition d'insatisfaisabilité d'une valeur abstraite (Prop. 6.1, p. 138) on en conclut que pour tout $p$ : $\rho_i \notin \{\rho_{|I} \mid \rho \in \gamma^{\mathbb{Z}}(\varphi_p) \bigcap_{(q,z) \in Trans^{\Phi}(\varphi), Incl(p,(q,z))} \gamma^{\mathbb{Z}}(\varphi_q \oplus z)\}$. Or, par construction, on a :

$$\gamma^{\mathbb{Z}}(\varphi) \subseteq \bigcup_{p \in Inter^{\Phi}(\varphi)} (\gamma^{\mathbb{Z}}(\varphi_p) \bigcap_{(q,z) \in Trans^{\Phi}(\varphi), Incl(p,(q,z))} \gamma^{\mathbb{Z}}(\varphi_q \oplus z)), \tag{6.3}$$

et donc $\gamma^{\mathbb{Z}}(\overline{\varphi}) \subseteq \bigcup_{p \in Inter^{\Phi}(\varphi)} \{\rho_{|I} \mid \rho \in \gamma^{\mathbb{Z}}(\varphi_p) \bigcap_{(q,z) \in Trans^{\Phi}(\varphi), Incl(p,(q,z))} \gamma^{\mathbb{Z}}(\varphi_q \oplus z)\}\}$. On conclut donc que $\rho_i \notin \gamma^{\mathbb{Z}}(\overline{\varphi})$. Pour montrer que si $\rho_i \in \gamma^{\mathbb{Z}}(\overline{\varphi})$ alors $\rho_v \in \gamma^{\mathbb{Z}}(\overline{\psi^{\Phi}(\varphi)})$, il suffit, en reprenant l'Équation 6.3, de réutiliser la preuve de correction de la règle 1 de l'opérateur de normalisation (Prop. 6.2, p. 143). De la même manière, pour montrer que $\forall k \in K(\varphi), \exists \rho \in R(\psi^{\Phi}(\varphi)), \forall a^z \in Dom(\rho), \rho_a(a)(k + z) = \rho(a^z)$, il est aisé de réutiliser la preuve de correction de la règle 2 du même opérateur. □

#### 6.5.2.2 Opération d'adaptation

Avec l'opération d'extraction de propriétés que l'on vient de définir, il est aisé d'adapter une valeur abstraite d'une partition $P_1$ à une partition $P_2$. On associe à chaque tranche de $P_2$ la propriété extraite de la valeur abstraite évaluée selon $P_1$ sur cette tranche.

Définition 6.14 (adaptation d'une valeur abstraite). *Soit $P_1$ et $P_2$ deux partitions et $\Phi \in \mathcal{D}^{arco}_{P_1}$. On définit l'adaptation $\Phi_{(P_1 \to P_2)} \in \mathcal{D}^{arco}_{P_2}$ de $\Phi$ sur la partition $P_2$ par :*

$$\Phi_{(P_1 \to P_2)} = (\eta, \mu, \{\psi^{\Phi}(\varphi_p)\}_{p \in P_2}).$$

*On a $\gamma^{arco}_{P_1}(\Phi) \subseteq \gamma^{arco}_{P_2}(\Phi_{(P_1 \to P_2)})$.*

*Démonstration.* Soit $\Phi' = (\eta, \mu, \{\psi_p\}_{p\in P_1} \cup \{\psi^\Phi(\varphi_p)\}_{p\in P_2})$. Si on montre que $\gamma^{arco}_{P_1}(\Phi) = \gamma^{arco}_{P_1\cup P_2}(\Phi')$ la preuve de correction est directe : en effet, retirer une propriété de tableau d'une valeur abstraite, c'est probablement augmenter sa concrétisation. En enlevant de $\Phi'$ toutes celles dont les tranches appartiennent à la partition $P_1$, on obtient $\Phi_{(P_1\to P_2)}$, donc $\gamma^{arco}_{P_1\cup P_2}(\Phi') \subseteq \gamma^{arco}_{P_2}(\Phi_{(P_1\to P_2)})$ et on conclut.

La preuve de l'égalité des concrétisations de $\Phi$ et $\Phi'$ est tout aussi aisée : pour tout $\varphi_p \in P_2$, on a par correction de l'opérateur d'extraction de propriétés de tableaux, que $\gamma^{arco}_{P_1}(\Phi) = \gamma^{arco}_{P_1\cup\{\varphi_p\}}(\Phi_p)$ où $\Phi_p$ est $\Phi$ à laquelle on a ajouté $\psi^\Phi(\varphi_p)$. Si $|P_2| = 1$ on conclut, sinon soit $\varphi_{p'}$ une autre tranche de $P_2$, on a $\gamma^{arco}_{P_1\cup\{\varphi_{p'}\}}(\Phi_{p'}) = \gamma^{arco}_{P_1\cup\{\varphi_p\}}(\Phi_p)$. Cette égalité assure que le transfert d'une propriété de tableaux, par exemple de $\Phi_{p'}$ à $\Phi_p$ ne modifie pas la concrétisation de $\Phi_p$. Construire ainsi $\Phi'$ assure que sa concrétisation est égale à celle de $\Phi$. □

Remarque 6.3. *Jusqu'à présent, on a vu uniquement des situations où l'opération d'adaptation était effectuée* avant *la fonction de transfert. En fait, il en était ainsi parce que la partition $P_2$ au point de contrôle suivant était plus fine (Déf. 6.10, p. 146) que la partition $P_1$ au point de contrôle précédent. Quand ce n'est pas le cas, la fonction de transfert doit être appliquée* avant*, car l'adaptation de la partition $P_1$ à la partition $P_2$ peut mener à une perte d'information. On prend pour exemple la copie de tableau. En tête de boucle, si on omet la tranche où $(\ell > n)$, on a la partition $\{\varphi_1, \varphi_2, \varphi_3\} = \{(1 \le \ell \le n)\} \sqcup^{arpa} Split(i)$, et au dernier point de contrôle la partition $\{(1 \le \ell \le n)\}$. On se place au moment où l'on a la valeur abstraite stable suivante en tête de boucle : $(1 \le i) \wedge \forall \ell,\ (1 \le \ell \le n \wedge \ell < i) \Rightarrow a[\ell] = b[\ell]$. Si on adapte cette valeur abstraite avant d'appliquer la condition de sortie $i > n$, comme on n'a pas de relation entre $i$ et $n$, on a pour l'opération d'extraction (Déf. 6.13) $Inter((1 \le \ell \le n)) = \{1, 2, 3\}$. Or $\psi_2 = \psi_3 = \top^{\mathbb{K}}$ et donc on obtient la valeur abstraite $(1 \le i)$ au dernier point de contrôle. Si on applique d'abord la condition de sortie, on obtient par normalisation une valeur abstraite où $\psi_2 = \psi_3 = \bot^{\mathbb{K}}$. On a pour cette valeur abstraite $Inter((1 \le \ell \le n)) = \{1\}$ et donc on obtient au dernier point de contrôle $(1 \le i \wedge i > n) \wedge \forall \ell,\ (1 \le \ell \le n) \Rightarrow a[\ell] = b[\ell]$.*

## 6.6 Opérateurs Sémantiques

### 6.6.1 Opérateur d'élargissement

Il ne fait aucun doute que les domaines que l'on souhaite utiliser pour $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$ seront de profondeur infinie. Ainsi, on donne un opérateur d'élargissement (Déf. 6.15) pour assurer la convergence de l'analyse (*cf.* Thm. 2.4, p. 17).

Il s'agit simplement d'un élargissement élément par élément, des différents éléments de domaine abstraits qui composent nos valeurs abstraites. Un opérateur de rétrécissement peut-être défini de la même manière.

Définition 6.15 (opérateur d'élargissement). *Soit $\Phi^1, \Phi^2 \in \mathcal{D}^{arco}(P)$. On définit l'opération d'élargissement de $\Phi^1$ par $\Phi^2$ par :*

$$\Phi^1 \nabla^{arco} \Phi^2 = (\eta^1 \nabla^{\mathbb{Z}} \eta^2, \mu^1 \nabla^{\mathbb{K}} \mu^2, \{\psi^1_p \nabla^{\mathbb{K}} \psi^2_p\}_{p\in P})$$

*Démonstration.* Les propriétés à démontrer (Déf. 2.3, p. 16) sont directement déduites de celles des opérateurs $\nabla^{\mathbb{Z}}$ et $\nabla^{\mathbb{K}}$ : $\Phi^1, \Phi^2 \sqsubseteq^{arco} \Phi^1 \nabla^{arco} \Phi^2$ et propriété de chaîne ascendante. □

Si l'on se remémore les problèmes d'interactions entre les opérations d'élargissement et de normalisation des domaines des zones et des zones précisant alias (Sec. 5.7.1), et ne sachant pas si $\mathcal{D}^{\mathbb{Z}}$ ou $\mathcal{D}^{\mathbb{K}}$ est un domaine sujet à ces problèmes, on évitera également toute normalisation qui pourrait compromettre la convergence de l'analyse. Il s'agit bien entendu de l'opération de normalisation sur $\mathcal{D}_P^{arco}$. Si on évitait seulement les normalisations des éléments $\eta, \mu$ et $\psi_p$, on oublierait que des propriétés peuvent être propagées par l'opération de normalisation sur $\mathcal{D}_P^{arco}$ d'un de ces éléments à un autre, et donc générer le même problème.

On donne un exemple qui illustre ce danger potentiel, et par la même occasion l'opérateur. Si on élargit $\Phi^1 = ((i = 1), \top^{\mathbb{K}}, \{\psi_1 = \bot^{\mathbb{K}}\})$ par $\Phi^2 = ((i = 2), \top^{\mathbb{K}}, \{\psi_1 = \bot^{\mathbb{K}}\})$, où $\varphi_1 = (\ell = i \geq 3)$, on obtient avec $\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{zone}$, $\Phi^1 \nabla^{arco} \Phi^2 = ((i \geq 1), \top^{\mathbb{K}}, \{\psi_1 = \bot^{\mathbb{K}}\})$. Or, si on note $\Phi$ ce résultat, on a $\Phi^* = ((i \leq 2), \top^{\mathbb{K}}, \{\psi_1 = \bot^{\mathbb{K}}\})$ – résultat dû principalement à la règle 3 de normalisation (Déf. 6.8, p. 142), ce qui annule l'effet de l'élargissement de $\eta^1$ par $\eta^2$.

### 6.6.2 Fonctions de transfert

Dans toutes les définitions suivantes de fonctions de transfert, nécessaires à l'analyse de nos programmes (Préliminaires – Sémantique concrète), on omettra le cas où la valeur abstraite est $\bot^{arco}$, car ces fonctions ne la font pas varier. Enfin, on distinguera systématiquement le cas des variables d'indices de celui des variables de contenu.

#### 6.6.2.1 Opérateurs d'oubli

**Variable d'indice** Soit $i \in I$. Bien entendu, on applique à la propriété $\eta$ l'opération d'affectation indéterministe à $i$. Ce n'est pas la seule propriété qui doit être modifiée : en effet, les cellules représentées par des tranches $\varphi_p$ dont la définition dépend de $i$, ne sont probablement plus les mêmes cellules et donc les propriétés $\psi_p$ associées à ces tranches $\varphi_p$ sont à modifier. Ayant perdu toute information sur $i$, on peut seulement assurer que ces tranches $\varphi_p$ représentent une ou des cellules parmi $\exists i.\varphi_p$. On utilise alors l'opérateur d'extraction de propriétés de tableau (Déf. 6.13, p. 154) sur cette dernière propriété pour calculer une propriété correcte $\psi_p$ qu'implique $\varphi_p$.

Définition 6.16 (opérateur d'oubli d'une variable d'indice). *Soit $\Phi \in \mathcal{D}_P^{arco}, \Phi \neq \bot^{arco}$. On définit l'opération d'oubli sur $i \in I$ par :*

$$\{\!|i := ?|\!\}^{arco}(\Phi) = (\{\!|i := ?|\!\}^{\mathbb{Z}}(\eta), \mu, \{\psi_p\}_{p \in P \setminus P_i} \cup \{\psi^{\Phi}(\exists i.\varphi_p)\}_{p \in P_i}),$$

*où $P_i = \{p \in P \mid i \in Var(\varphi_p)\}$.*

*Démonstration.* La preuve de correction est aisée, par correction de l'opérateur d'extraction de propriétés de tableaux. □

On prend pour exemple la propriété de tableau suivante, définie sur la partition $P$ donnée à la Figure 6.14(a) (p. 160) : $\Phi = ((i = 1), \top^{\mathbb{K}}, \{\bot^{\mathbb{K}}; (a^0 = 5); (a^0 = 7)\}_{p \in P})$. On aura remarqué que $\psi_1 = \bot^{\mathbb{K}}$ car $\varphi_1$ représente un ensemble de cellules vide vis-à-vis de $\eta$. Ce ne sera plus le cas après l'opération d'oubli $i :=$ ?. On a, et il ne s'agit pas là d'un cas particulier, au vue de notre construction des partitions, $\exists i.\varphi_1 = \exists i.\varphi_2 = \exists i.\varphi_3 = (1 \leq i \leq n)$. Dans la pratique on recherchera ces égalités afin d'optimiser le nombre

d'appels à l'opération d'extraction de propriétés de tableaux, dont le coût est loin d'être négligeable. Ainsi, comme $\psi^{\Phi}((1 \leq i \leq n)) = (5 \leq a^0 \leq 7)$, le résultat de l'opération est $\Phi' = (\top^{\mathbb{Z}}, \top^{\mathbb{K}}, \{(5 \leq a^0 \leq 7); (5 \leq a^0 \leq 7); (5 \leq a^0 \leq 7)\}_{p \in P})$.

**Variable de contenu** Dans le cas où il s'agit d'une variable scalaire, $x \in V$, l'opération d'oubli est triviale : appelé l'opérateur d'oubli de $\mathcal{D}^{\mathbb{K}}$ sur $\mu$ et sur chaque $\psi_p$.

Dans le cas où c'est la valeur d'une cellule d'un tableau $a \in A$ qui est oubliée, il est nécessaire de modifier, dans toutes les propriétés de tableaux, toute variable de tranche portant information sur des cellules dont l'une peut être touchée par l'affectation indéterministe. Pour trouver ces variables de tranches, on crée un élément de $\mathcal{D}^{\mathbb{Z}}$ représentant la cellule affectée : pour une affectation $a[i+1] := ?$ il s'agit simplement de construire $\varphi = (\ell = i + 1)$. En ayant un élément de $\mathcal{D}^{\mathbb{Z}}$, on a à notre disposition l'opérateur d'intersection $\sqcap^{\mathbb{Z}}$ pour connaître les variables de tranches $a^z$, de diverses propriétés de tableaux $p$, qui doivent être modifiées : ce sont toutes celles telles que $\varphi$ intersecte $\varphi_p \oplus z$. L'opérateur n'a plus qu'à appliquer à ces variables de tranches l'opérateur d'oubli de $\mathcal{D}^{\mathbb{K}}$.

Définition 6.17 (opérateur d'oubli d'une variable de contenu). *Soit $\Phi \in \mathcal{D}_P^{arco}, \Phi \neq \bot^{arco}$. On définit l'opération d'oubli sur $x \in V$ par :*

$$\{\!|x := ?|\!\}^{arco}(\Phi) = (\eta, \{\!|x := ?|\!\}^{\mathbb{K}}(\mu), \{\{\!|x := ?|\!\}^{\mathbb{K}}(\psi_p)\}_{p \in P}).$$

*Pour définir l'opération d'oubli sur $a[\langle expr \rangle]$ où $\langle expr \rangle \equiv i + c$ ou $\langle expr \rangle \equiv c$, et où $i \in I$ et $a \in A$, on construit d'abord l'ensemble $E = \{(p, z) \mid \varphi_p \in P \wedge a^z \in Var(\psi_p) \wedge (\ell = \langle expr \rangle) \sqcap^{\mathbb{Z}} \eta \sqcap^{\mathbb{Z}} \varphi_p \oplus z \neq \bot^{\mathbb{Z}}\}$. Alors, si on note $E_p = \{z \in \mathbb{Z} \mid (p, z) \in E\}$, on définit :*

$$\{\!|a[\langle expr \rangle] := ?|\!\}^{arco}(\Phi) = (\eta, \mu, \{\psi'_p\}_{p \in P}),$$

$$\text{où } \psi'_p = \begin{cases} \psi_p & \text{si } E_p = \emptyset \\ \{\!|a^{z_1} := ?|\!\}^{\mathbb{K}}(\ldots \{\!|a^{z_n} := ?|\!\}^{\mathbb{K}}(\psi_p)) & \text{sinon, où } \{z_1, \ldots, z_n\} = E_p \end{cases}.$$

*Démonstration.* La preuve de correction de l'opérateur ne soulevant pas de problème particulier, on ne la présente pas. □

Nous verrons à la section suivante, où sont décrits les opérateurs d'affectation, des exemples illustrant les différentes variables de tranches devant être modifiées.

Comme toujours, avant d'appliquer ces opérateurs à une valeur abstraite, on peut normaliser cette dernière pour gagner en précision. Nous avions vu que pour le domaine des zones et des zones précisant alias dans le cas dense (Sec. 5.7.2.1), il n'était pas nécessaire de faire appel à une normalisation complète de la valeur abstraite pour obtenir une précision maximum, ce qui permettait de réduire notamment la complexité. Ici, les interactions entre propriétés scalaires et propriétés de tableaux sont telles qu'il n'existe pas de telle alternative. Cela dit, si pour des raisons de performances on ne souhaite pas normaliser préalablement, et si $\mathcal{D}^{\mathbb{Z}}$ ou $\mathcal{D}^{\mathbb{K}}$ est l'un des deux domaines précités, utiliser la version de leurs opérateurs d'oubli incluant les normalisations partielles adéquates préalables, constitue un bon compromis entre efficacité et précision.

#### 6.6.2.2 Affectations

Pour cette section – et la suivante, il est important de noter que les définitions des diverses opérations abstraites qui y sont décrites ne sont pas indépendantes du choix de nos partitions. Cependant, il serait aisé, pour une construction différente des partitions, de les adapter.

**Affectations de variables d'indice** On définit d'abord un opérateur pour le cas des affectations non-inversibles, donc de la forme $i := j + c$ (ou $i := c$). Ici, les cellules représentées par les tranches $\varphi_p$ de la partition après l'affectation, sont aussi celles que représentent les propriétés $\varphi_p$ dans lesquelles $i$ a été remplacée par $j+c$, avant l'affectation. On note ces propriétés $\varphi_p[j + c/i]^{\mathbb{Z}}$ dont le calcul est aisé : $\exists i.(\varphi_p \sqcap^{\mathbb{Z}} (i = j + c))$. En appelant l'opérateur d'extraction de propriétés de tableaux sur ces propriétés, pour calculer la propriété qu'implique $\varphi_p$ après l'affectation, on se donne une chance de récupérer une information précise, par exemple si la définition de la partition dépend de $j$.

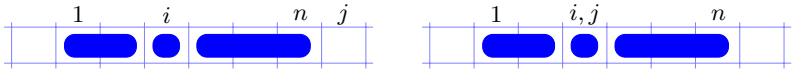
Définition 6.18 (affectation non-inversible d'une variable d'indice). *Soit $\Phi \in \mathcal{D}_P^{arco}$, $\Phi \neq \bot^{arco}$. On définit l'opération d'affectation non-inversible $i := j + c$, où $c \in \mathbb{Z}$, par :*

$$\{\!|i := j + c|\!\}^{arco}(\Phi) = (\{\!|i := j + c|\!\}^{\mathbb{Z}}(\eta), \mu, \{\psi_p\}_{p\in P\setminus P_i} \cup \{\psi^{\Phi}(\varphi_p[j + c/i]^{\mathbb{Z}})\}_{p\in P_i}),$$

*où $P_i = \{p \in P \mid i \in Var(\varphi_p)\}$.*

*Démonstration.* Là encore, la preuve de correction est aisée, par correction de l'opérateur d'extraction de propriétés de tableaux. □

On prend pour exemple le programme donné à la Figure 7.4(b) (p. 186) qui recherche la première cellule d'un tableau d'entier dont le contenu est non nul. Si une telle cellule existe, et une fois trouvée, elle est repérée par la variable d'indice $j$, grâce à l'affectation $j := i$ (ligne 5). Au point de contrôle avant cette affectation, on a $i < j = n + 1$ et donc seules trois tranches représentent des ensembles non vides de cellules, illustrées Figure 6.13(a) : $\varphi_1 = (1 \leq \ell < i \leq n \wedge \ell < j \leq n + 1)$, $\varphi_4 = (1 \leq \ell = i \leq n \wedge \ell < j \leq n + 1)$ et $\varphi_7 = (1 \leq i < \ell \leq n \wedge \ell < j \leq n + 1)$. On a l'invariant suivant : toutes les cellules avant $i$ contiennent zéro, $\psi_1 = (a^0 = 0)$, la cellule en $i$ est non nulle, $\psi_2 = (a^0 \neq 0)$, et pour le reste des cellules on a pas d'information, $\psi_3 = \top^{\mathbb{K}}$. Après l'affectation, où $i = j$, là encore seulement trois tranches sont valides : $\varphi_1, \varphi_5 = (1 \leq \ell = i = j \leq n)$ et $\varphi_9 = (1 \leq i < \ell \leq n \wedge j < \ell \wedge j \leq n + 1)$, illustrées Figure 6.13(b).




(a) Les tranches $\varphi_1, \varphi_4$ et $\varphi_7$, lorsque $i < j = n + 1$

(b) Les tranches $\varphi_1, \varphi_5$ et $\varphi_9$, lorsque $j = i \leq n$

Figure 6.13 – Représentation schématique de certaines tranches de la partition du programme donné à la Figure 7.4(b) : celles représentant un ensemble de cellules non-vides selon la propriété vérifiée par les variables d'indice, avant et après l'affectation $j := i$

Si on calcule $\varphi_1[j/i]^{\mathbb{Z}} = (1 \leq \ell < i \leq n)$, $\varphi_5[j/i]^{\mathbb{Z}} = (1 \leq \ell = i \leq n)$ et $\varphi_9[j/i]^{\mathbb{Z}} = (1 \leq i < \ell \leq n)$ on voit immédiatement que ces trois propriétés représentent les mêmes cellules que $\varphi_1, \varphi_4$ et $\varphi_7$, avant l'affectation. En effet $\psi^{\Phi}(\varphi_1[j/i]^{\mathbb{Z}}) = \psi_1$, $\psi^{\Phi}(\varphi_5[j/i]^{\mathbb{Z}}) = \psi_4$ et $\psi^{\Phi}(\varphi_9[j/i]^{\mathbb{Z}}) = \psi_7$. Ainsi les deux propriétés de tableaux associées aux tranches $\varphi_1$ et $\varphi_5$ expriment-elles, après l'affectation, l'invariant qui nous intéresse, $a[j] \neq 0$ et $\forall \ell, 1 \leq \ell < j \Rightarrow a[\ell] = 0$. Il ne reste plus qu'à ce que cet invariant soit maintenu lors des incrémentations de $i$ pour prouver la correction de ce programme.

L'opérateur pour les affections inversibles d'une variable d'indice, donc de la forme $i := i + c$, est notre sujet suivant. On peut définir cet opérateur simplement comme le précédent, en associant cette fois à chaque tranche $\varphi_p$ la propriété $\psi^{\Phi}(\varphi_p[i + c/i]^{\mathbb{Z}})$. L'opération $\varphi[i + c/i]^{\mathbb{Z}}$ s'implante même plus efficacement que précédemment, par $\{\!|i := i - c|\!\}^{\mathbb{Z}}(\varphi)$.

Définition 6.19 (affectation inversible d'une variable d'indice). *Soit $\Phi \in \mathcal{D}_P^{arco}$, $\Phi \neq \bot^{arco}$. On définit l'opération d'affectation $i := i + c$, où $c \in \mathbb{Z}$, par :*

$$\{\!|i := i + c|\!\}^{arco}(\Phi) = (\{\!|i := i + c|\!\}^{\mathbb{Z}}(\eta), \mu, \{\psi_p\}_{p \in P \setminus P_i} \cup \{\psi^{\Phi}(\{\!|i := i - c|\!\}^{\mathbb{Z}}(\varphi_p))\}_{p \in P_i}),$$

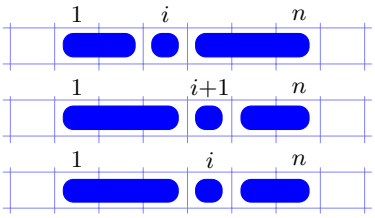
*où $P_i = \{p \in P \mid i \in Var(\varphi_p)\}$.*

*Démonstration.* Là encore, il s'agit simplement de noter que les tranches $\varphi_p[i + c/i]^{\mathbb{Z}}$ représentent bien les mêmes cellules que les tranches $\varphi_p$ de la partition $P$ après l'affectation. Alors la correction de l'opération d'extraction de propriétés de tableaux permet de conclure. □

On prend pour exemple l'affectation $i := i + 1$ sur la valeur abstraite $\Phi$ définie à la Figure 6.14. $\Phi$ s'entend sur la partition $P$ et on a regroupé dans la partition $P'$ l'ensemble des propriétés $\varphi_p[i + 1/i]^{\mathbb{Z}}$, où $\varphi_p \in P$ ; la définition de ces deux partitions est donnée en (a). Les cellules que représentent ces deux partitions avant l'affectation, sont également illustrées en (a), suivi de celles que représente $P$ après l'affectation. D'une part, on voit clairement la correspondance entre la deuxième et la dernière, et d'autre part on voit entre la première et la deuxième l'approximation induite par l'opération : en effet, la propriété de tableau associée à $\varphi_1$ après l'affectation devra vérifier $\varphi_1 \Rrightarrow \psi_1 \vee \psi_2$. Les étapes du calcul, où cette approximation apparaît durant l'opération d'extraction, et le résultat sont donnés en (b).

**Affectations de variables de contenu** L'expression qui va être affectée à la variable de contenu, que l'on notera $\langle expr \rangle$, peut contenir une ou plusieurs cellules de tableau, de la forme $b[j + d]$ (ou $b[d]$), où $b \in A, j \in I$ et $d \in \mathbb{Z}$. Lorsqu'on va décliner l'affectation par diverses affectations dans les propriétés de tableaux, la question de l'expression qui sera utilisée pour ces affectations est celle de la traduction des expressions $b[j + d]$ dans $\langle expr \rangle$ pour construire cette expression. Si on prend le cas de l'affectation d'une cellule de tableau $a[i+c]$, pour laquelle on a identifié que la variable de tranche $a^z$ dans la propriété $\psi_p$ doit être modifiée (parce que la cellule $i + c$ est représentée par $\varphi_p \oplus z$), la fonction $t$ suivante permet de savoir si la cellule $j + d$ peut elle aussi être représentée par une translation de la propriété $\varphi_p$. Si c'est le cas, l'application $t(\varphi_p)(j + d)$ renvoie l'entier

$$\Phi = ((1 \leq i \leq n), \top^{\mathbb{K}}, \{\psi_1 = (0 \leq a^0 \leq 4); \psi_2 = (a^0 = 7); \psi_3 = (a^{-1} \leq a^0)\}) \in \mathcal{D}_P^{arco}$$




$$P = \{\varphi_1 = (1 \leq \ell < i \leq n);\ \varphi_2 = (1 \leq \ell = i \leq n);\ \varphi_3 = (1 \leq i < \ell \leq n)\}$$

$$P' = \{\varphi_p[i+1/i]^{\mathbb{Z}}\} = \{\varphi'_1 = (1 \leq \ell \leq i < n);\ \varphi'_2 = (1 \leq \ell = i+1 \leq n);\ \varphi'_3 = (1 \leq i+1 < \ell \leq n)\}$$

(a) À droite, la définition des partitions $P$ et $P'$, et à gauche leurs représentations selon la valeur de $i$ avant et, pour la dernière ligne, après l'affectation. On a dans l'ordre $P$, $P'$, et de nouveau $P$.

$$\psi'_1 = \psi^{\Phi}(\varphi'_1) = (0 \leq a^0 \leq 4) \sqcup^{\mathbb{K}} (a^0 = 7) = (0 \leq a^0 \leq 7) \qquad \psi'_2 = \psi^{\Phi}(\varphi'_2) = (a^{-1} \leq a^0)$$
$$\psi'_3 = \psi^{\Phi}(\varphi'_3) = (a^{-1} \leq a^0)$$

$$\Phi' = ((2 \leq i \leq n+1), \top^{\mathbb{K}}, \{\psi'_1 = (0 \leq a^0 \leq 7); \psi'_2 = (a^{-1} \leq a^0); \psi'_3 = (a^{-1} \leq a^0)\}) \in \mathcal{D}_P^{arco}$$

(b) Les nouvelles propriétés de tableau $\psi'_p$ calculées par l'opérateur, qui fait appel à l'opération d'extraction, et enfin le résultat final $\Phi'$

Figure 6.14 – Exemple d'affectation inversible d'une variable d'indice (Déf. 6.19), l'affectation $i := i + 1$ sur $\Phi$. L'opération calcule les tranches de la partition $P'$ pour définir les nouvelles propriétés de tableaux de $\Phi'$, son résultat.

$w$ correspondant à cette translation, et par exemple l'affectation $a[i + c] := b[j + d] + 5$ peut se décliner en $a^z := b^w + 5$, sinon il renvoie $\infty$.

$$t(\varphi)(\langle expr_i \rangle) = \begin{cases} w & \text{si } \exists w \in \mathbb{Z} \text{ tel que } (\varphi \sqcap^{\mathbb{Z}} (\ell' = \langle expr_i \rangle)) \sqsubseteq^{\mathbb{Z}} (\ell' = \ell + w) \\ \infty & \text{sinon} \end{cases} \tag{6.4}$$

On notera $t_p = t(\varphi_p)$. Dans le dernier cas ($\infty$), il serait dommage de faire simplement appel à l'opérateur d'oubli pour traiter l'affectation. En effet, on peut utiliser l'opération d'extraction de propriétés de tableaux pour trouver la meilleure information que l'on a sur la cellule $b[j + d]$, dans le contexte de la tranche $\varphi_p$. Une partie de cette information peut être utilisée pour gagner en précision : par exemple si on trouve que $b[j + d]$ est égale à $x$ alors l'affectation de notre exemple pourrait se traduire par $a^z := x + 5$. Il est évident que si $b[j + d]$ ne pouvait être traduite sous la forme d'une variable de tranche dans $\langle expr \rangle$ il en sera de même pour toute cellule de tableau apparaissant dans cette information. Ainsi, de la propriété $\psi' = \psi^{\Phi}((\ell = j + d) \sqcap^{\mathbb{Z}} \overline{\varphi_p})$, qui nous donne pour $b^0$ cette information, seule la partie scalaire peut-être utilisée, et finalement, seule la partie scalaire reliée à $b^0$ nous intéresse. Dans la pratique on utilisera une variable scalaire fraîche $y$ pour véhiculer ces informations scalaires sur $b[j + d]$. Si on note $\overline{\psi}^x$, où $\psi \in \mathcal{D}^{\mathbb{K}}$ et $x \in V$, la propriété $\psi$ où toute variable de tranche et toute variable scalaire n'étant pas en relation avec $x$ a été éliminée, alors ont utilisera la propriété $\psi'' = \overline{\psi'[y/b^0]}^y$.

L'affectation sera alors retranscrite de la manière suivante : $y$ remplace $b[j + d]$ dans $\langle expr \rangle$, et on effectue l'affectation $a^z := y + 5$, non pas sur $\psi_p$ mais sur $\psi_p \sqcap^{\mathbb{K}} \psi''$. Enfin on élimine $y$ du résultat. Toujours sur le même exemple, si on a $\psi_p = \top^{\mathbb{K}}$,

$\psi' = (b^0 = x \in [5,7] \wedge u = v)$, et donc $\psi'' = (y = x \in [5,7])$, le résultat de l'affectation $a^z := y + 5$ sur $\psi_p \sqcap^{\mathbb{K}} \psi''$ est $a^z = y + 5 = x + 5 \in [10,12]$. Après élimination de $y$ on obtient $a^z = x + 5 \in [10,12]$.

On entre à présent dans le détail. On commence par le cas où la variable affectée est une variable scalaire : $x := \langle expr \rangle$ (Déf. 6.20).

- $\mu$ doit être modifiée. Ici, si des cellules de tableaux apparaissent dans $\langle expr \rangle$, on ne peut les retranscrire qu'en faisant appel à la technique que l'on vient de décrire. On note $inexact^{\mu}_x = \{b[j+d] \in \langle expr \rangle\}$ l'ensemble de ces références au contenu des tableaux. Ainsi l'expression affectée sera $expr^{\mu}_x = \langle expr \rangle[y_k / b[j+d]]$, où $y_k$ sont des variables fraîches, et l'information transmise à $\mu$ avant cette affectation sera :

$$info^{\mu}_x = \begin{cases} \top^{\mathbb{K}} & \text{si } inexact^{\mu}_x = \emptyset \\ \bigsqcap_{b[j+d] \in inexact^{\mu}_x} \overline{(\psi^{\Phi}(\ell = j+d))[y_k/b^0]}^{y_k} & \text{sinon} \end{cases}.$$

- certaines propriétés $\psi_p$ doivent être modifiées. Bien entendu, il y a toutes celles où $x$ apparaît, pour assurer la correction de l'opération. Mais on souhaite également que l'affectation ait lieu si $\langle expr \rangle$ contient une cellule de tableau pouvant être représentée par une variable de tranche dans $\psi_p$, afin de ne pas perdre une propriété relationnelle explicite du programme. Dans tous les cas, on identifie les cellules de tableau ne pouvant être représentées par une variable de tranche, par $inexact^{p}_x = \{b[j+d] \in \langle expr \rangle \mid t_p(j+d) = \infty\}$. L'expression affectée sera donc $expr^{p}_x = \langle expr \rangle[b^w / b[j+d]]_{t_p(j+d)=w}[y_k / b[j+d]]_{t_p(j+d)=\infty}$, où $y_k$ sont des variables fraîches, et l'information transmise à $\psi_p$ avant cette affectation sera :

$$info^{p}_x = \begin{cases} \top^{\mathbb{K}} & \text{si } inexact^{p}_x = \emptyset \\ \bigsqcap_{b[j+d] \in inexact^{p}_x} \overline{(\psi^{\Phi}((\ell = j+d) \sqcap^{\mathbb{Z}} \overline{\varphi_p}))[y_k/b^0]}^{y_k} & \text{sinon} \end{cases}.$$

On peut maintenant donner la définition qui utilise ces diverses notations pour implanter le schéma d'affectation présenté au début du paragraphe.

Définition 6.20 (affectation d'une variable scalaire de contenu). *Soit* $\Phi \in \mathcal{D}^{arco}_P$, $\Phi \neq \bot^{arco}$. *On définit l'opération d'affectation* $x := \langle expr \rangle$ *par :*

$$\{\!|x := \langle expr \rangle|\!\}^{arco}(\Phi) = (\eta, \mu', \{\psi'_p\}_{p \in P}), \text{ où}$$

$$\mu' = \exists y_1, \ldots, y_n \{\!|x := expr^{\mu}_x|\!\}^{\mathbb{K}}(\mu \sqcap^{\mathbb{K}} info^{\mu}_x)$$

$$\psi'_p = \begin{cases} \exists y_1, \ldots, y_n \{\!|x := expr^{p}_x|\!\}^{\mathbb{K}}(\mu \sqcap^{\mathbb{K}} info^{p}_x) & \textit{si } x \in Var(\psi_p) \textit{ ou } exact^{p}_x \neq \emptyset \\ \psi_p & \textit{sinon} \end{cases}$$

On prend pour exemple, dans le programme de recherche du maximum donné à la Figure 7.4(a) (p. 186), l'affectation $x := a[i]$ (ligne 5). La partition à ce point de contrôle est représentée et donnée à la Figure 6.8(b) (p. 146, partition du haut). L'invariant recherché et atteint à partir d'une certaine itération avant cette affectation est : $x$ est plus grand que le contenu des cellules 1 à $i-1$ mais strictement plus petit que le contenu de la cellule $i$, autrement dit $\psi_1 = \psi_2 = (a^0 \leq x), \psi_3 = (a^0 > x)$ et $\psi_4 = \top^{\mathbb{K}}$. Si on ne s'intéresse qu'aux propriétés de tableaux – il n'y aucune information scalaire sur $x$ : $\mu = \top^{\mathbb{K}}$ en tous points de contrôle, les propriétés $\psi_1, \psi_2$ et $\psi_3$ doivent être modifiées de

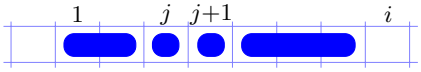
par la présence de $x$ en leurs sein. Naturellement, l'affectation peut-être représentée de manière exacte seulement pour $\psi_3 : x := a^0$. Pour les deux autres on a l'affectation $x := y$ et la même information à propager sur $a[i]$ : $info^1_x = info^2_x = \overline{(a^0 > x)[y/a^0]}^y = (y > x)$. Grâce à cette information, il est direct que les invariants sur $\psi_1$ et $\psi_2$ sont conservés après l'affectation (on a $\psi_1 = \psi_2 = (a^0 < x)$), invariants nécessaires pour la correction du programme ; et que $\psi_3 = (a^0 = x)$.

Il reste le cas où la variable affectée est une cellule de tableau : $a[i + c] := \langle expr \rangle$ (Déf. 6.21, p. suivante). Nous avions identifié, pour la définition de l'opérateur d'oubli (Déf. 6.17, p. 157), les différentes variables de tranche qui devaient être affectées pour que cette opération soit correcte. Ici, l'objectif est aussi que si elle n'est pas déjà présente, une variable de tranche soit identifiée permettant de rendre compte de l'affectation. De par la construction de nos partitions, on sait qu'il existe une tranche $\varphi_p$ représentant exactement la cellule $i + c$, et donc la variable de tranche $a^0$ dans $\psi_p$ peut toujours remplir cet office. Ainsi, on se contente d'ajouter seulement cette variable de tranche à celles devant être affectées : $E = \{(p, z) \mid \varphi_p \in P \wedge (z = 0 \vee a^z \in Var(\psi_p)) \wedge (\ell = \langle expr_i \rangle) \sqcap^{\mathbb{Z}} \eta \sqcap^{\mathbb{Z}} \varphi_p \oplus z \neq \bot^{\mathbb{Z}}\}$.

On prend pour exemple les tranches $\varphi_1 = (1 \leq \ell < j < i)$, $\varphi_2 = (1 \leq \ell = j < i)$ et $\varphi_3 = (2 \leq \ell = j + 1 < i)$, représentées Figure 6.15(a), les propriétés $\psi_1 = (a^1 = 5)$, $\psi_2 = \psi_3 = \top^{\mathbb{K}}$, et l'affectation $a[j] = 7$. On a $E = \{(1, 1); (2, 0)\}$, *i.e.* que $a^0$ apparaîtra dans $\psi_2$, mais pas $a^{-1}$ dans $\psi_3$, propriété qui restera volontairement inchangée – et qu'une normalisation après l'opération d'affectation ne changerait pas plus.

Par ailleurs, on voit dans cet exemple que deux situations sont mélangées dans

$$\Phi = ((1 < j < i - 1), \top^{\mathbb{K}}, \{\psi_1 = (a^{-1} \leq a^0); \psi_2 = (a^{-1} \leq a^0 \wedge x < a^0); \\ \psi_3 = \psi_4 = (x < a^{-1} \leq a^0)\})$$




$$E^s_p = \{(3, 0)\},\ E^w_p = \{(4, -1)\}$$

(a) Représentation schématique de certaines tranches de la partition utilisée dans la boucle interne du tri par insertion (Fig. 7.5(a), p. 189) et représentant au moins une cellule lorsque $1 < j < i - 1$. À droite, les variables de tranches touchées par l'affectation.

$$\begin{aligned}
\psi'_3 &= \{\!|a^0 := a^{-1}|\!\}^{\mathbb{K}}(\psi_3) = (x < a^{-1} = a^0) \\
\psi'_4 &= \psi_4 \sqcup^{\mathbb{K}} \{\!|a^{-1} := a^{-2}|\!\}^{\mathbb{K}}(\psi_4 \sqcap^{\mathbb{K}} (x < a^{-2} \leq a^{-1} \leq a^0)) \\
\psi'_4 &= \psi_4 \sqcup^{\mathbb{K}} \{\!|a^{-1} := a^{-2}|\!\}^{\mathbb{K}}(x < a^{-2} \leq a^{-1} \leq a^0) \\
\psi'_4 &= \psi_4 \sqcup^{\mathbb{K}} (x < a^{-1} \leq a^0) = \psi_4
\end{aligned}$$

$$\Phi' = ((1 < j < i - 1), \top^{\mathbb{K}}, \{\psi'_1 = (a^{-1} \leq a^0); \psi'_2 = (a^{-1} \leq a^0 \wedge x < a^0); \\ \psi'_3 = (x < a^{-1} = a^0); \psi'_4 = (x < a^{-1} \leq a^0)\})$$

(b) Calcul des nouvelles propriétés de tableau modifiées par l'affectation. La première subit une affectation normale, la seconde une affectation faible. Enfin, le résultat final de l'affectation, $\Phi'$.

Figure 6.15 – Exemple d'affectation d'une cellule de tableau (Déf. 6.21), l'affectation $a[j + 1] := a[j]$. L'opération identifie les variables de tranches à modifier, puis selon la nature des tranches auxquelles elles appartiennent, les affecte normalement ou faiblement.

$E$ : certaines variables de tranche représentent une cellule et d'autres représentent potentiellement plusieurs cellules. La variable de tranche $a^0$ dans $\psi_2$ fait partie des premières, $a^1$ dans $\psi_1$ fait partie des secondes (à moins que $j = 2$). Pour celles-ci, on doit mettre en œuvre une affectation faible (*cf.* 4.3.1). Comme précédemment on note $E_p = \{z \in \mathbb{Z} \mid (p, z) \in E\}$. On définit alors l'ensemble des variables de tranche au sein de $\psi_p$ qui seront affectées normalement, $E_p^s = \{z \in E_p \mid \varphi_p \oplus z \sqcap^{\mathbb{Z}} \eta \sqsubseteq^{\mathbb{Z}} (\ell = \langle expr \rangle)\}$ qui est soit vide, soit un singleton, et l'ensemble de celles qui seront affectées faiblement, par complément, $E_p^w = E_p \setminus E_p^s$. La définition des nouvelles propriétés $\psi'$ après l'opération (Déf. 6.21) se sépare donc en deux cas :

- affectation normale, $z \in E_p^s$. Le traitement est équivalent à celui de l'affectation d'une variable scalaire de contenu, mis à part que c'est la variable de tranche $a^z$ qui est affectée. On définit : $inexact_s^p = inexact_x^p$, $expr_s^p = expr_x^p$ et $info_s^p = info_x^p$.
- affectation faible, $z \in E_p^w$. Si on décline le principe de ces affectations faibles, cela consiste à voir l'ensemble de cellules que représente $\varphi_p \oplus z$ en deux parties : la cellule affectée, précisément définie par $\varphi_p \oplus z \sqcap^{\mathbb{Z}} (\ell = i + c)$ et les autres. Pour définir l'expression qui sera affectée à la première, on utilise le même principe que précédemment, sachant ici que les variables de tranches, renvoyées par la fonction $t$, qui pourront représenter des cellules de tableaux $b[j + d]$, devront être translatées de $z$. En effet ici on doit appeler la fonction $t_w = t((\varphi_p \oplus z) \sqcap^{\mathbb{Z}} (\ell = i + c))$ qui a pour référence comme on le voit $\varphi_p \oplus z$ et non $\varphi_p$ comme c'était le cas pour la fonction $t_p$. Ainsi, on a $inexact_w^p = \{b[j + d] \in \langle expr \rangle \mid t_w(j + d) = \infty\}$ et $expr_w^p = \langle expr \rangle [b^{w+z}/b[j + d]]_{t_w(j+d)=w} [y_k/b[j + d]]_{t_w(j+d)=\infty}$. Enfin, afin de gagner en précision, et cela s'avèrera nécessaire pour certains programmes (*cf.* prochain exemple), on ajoute à la définition des informations sur les variables temporaires $y_k$, la propriété la plus précise que l'on puisse obtenir sur la cellule affectée, soit $\psi_w = \psi^{\Phi}(\varphi_p \oplus z \sqcap^{\mathbb{Z}} (\ell = i + c)) \boxplus z$ :

$$info_w^p = \begin{cases} \psi_w & \text{si } inexact_w^p = \emptyset \\ \psi_w \bigsqcap_{b[j+d] \in inexact_w^p} \overline{(\psi^{\Phi}((\ell = j + d) \sqcap^{\mathbb{Z}} \overline{\varphi_p}))[y_k/b^0]}^{y_k} & \text{sinon} \end{cases} .$$

On obtient la définition suivante à partir de ces notations.

Définition 6.21 (affectation d'une cellule de tableau). *Soit $\Phi \in \mathcal{D}_P^{arco}$, $\Phi \neq \bot^{arco}$. On définit l'opération d'affectation $a[i + c] := \langle expr \rangle$, par :*

$$\{\!|a[i + c] := \langle expr \rangle|\!\}^{arco}(\Phi) = (\eta, \mu, \{\psi'_p\}_{p \in P}), \text{ où}$$

$$\psi'_p = \begin{cases} \exists y_1, \ldots, y_n \{\!|a^z := expr_s^p|\!\}^{\mathbb{K}}(\psi_p \sqcap^{\mathbb{K}} info_s^p) & si\ E_p^s = \{z\} \\ \psi_p \sqcup^{\mathbb{K}} \exists y_1, \ldots, y_n \{\!|a^z := expr_w^p|\!\}^{\mathbb{K}}(\psi_p \sqcap^{\mathbb{K}} info_w^p) & si\ E_p^w = \{z, \ldots\} \\ \psi_p & si\ E_p = \emptyset \end{cases} .$$

On prend pour exemple, dans le programme de tri par insertion donné à la Figure 7.5(a) (p. 189), l'affectation $a[j + 1] := a[j]$ (ligne **6**). On considère seulement certaines tranches de la partition à ce point de contrôle, représentées à la Figure 6.15(a). En particulier elles ne relèvent pas d'un cas limite (*e.g.* $j = 0$ ou $j = i - 1$), mais du cas général $1 < j < i - 1$ (Fig. 6.11(b), p. 152). Les trois premières tranches ont été définies précédemment et $\varphi_4 = (1 \leq j + 1 < \ell < i)$. À partir d'une certaine itération, l'invariant suivant est fixé : le tableau est trié en ordre croissant des cellules 1 à $i - 1$ et

à partir de la cellule $j$, toutes les valeurs jusqu'à la cellule $i-1$ sont strictement plus grandes que $x$, la valeur que l'on cherche à insérer en bonne place. Formellement, on a la valeur $\Phi$ donnée Figure 6.15. Ainsi, l'affectation se décline en deux affectations : on a $E = \{(3,0);(4,-1)\}$, la seconde affectation étant faible puisque $\varphi_4 \oplus -1$ ne représente pas nécessairement que la cellule $j+1$. Pour la première affectation, sa traduction est immédiate, la cellule $a[j]$ peut être représentée exactement : le résultat est donné Figure 6.15(b). Pour l'affectation faible, si on comprend aisément la traduction de l'affectation en $a^{-1} := a^{-2}$, c'est plus difficile de comprendre les informations générées. Si on note $\varphi = \varphi_4 \oplus -1 \sqcap^{\mathbb{Z}} (\ell = i+c)$ la cellule que l'on affecte, bien entendu $\varphi$ implique $j+1 < i-1$, *i.e.* une combinaison des variables d'indice telle qu'au moins cette cellule existe. Du coup, on a : $\psi^{\Phi}(\varphi) = (x < a^{-1} \leq a^0) \sqcap (x < a^0 \leq a^1)$, le premier terme étant dû à la propriété sur $\varphi_3$, le second étant dû au fait que $\varphi_3$, dans le contexte de $\overline{\varphi}$ est inclus dans $\varphi_4 \oplus -1$. Donc $\psi_w = (x < a^{-1} \leq a^0 \leq a^1) \boxplus -1 = (x < a^{-2} \leq a^{-1} \leq a^0)$. Grâce à cette dernière information, on voit que le calcul de l'affectation faible (Fig. 6.15(b)) préserve l'invariant, de tri notamment, sur les cellules de $j+2$ à $i-1$.

#### 6.6.2.3 Conditionnelles

Si on reprend la syntaxe de nos programmes (Fig. 4.5, p. 76), une condition, que l'on notera $g$, est soit une comparaison portant sur des variables d'indices, soit une comparaison sur des variables de contenu, soit la négation, la disjonction ou la conjonction de telles comparaisons.

On ne donne ici que le contour du traitement des conditionnelles, les détails de ces traitements étant très similaires à ceux que nous avons exposés pour les affectations. Bien entendu, on aura soin d'appeler l'opération de normalisation, avant ou après selon la précision que l'on souhaite.

Si $g$ ne porte que sur des variables d'indices, alors on l'applique tel qu'elle sur $\eta$ : $\{\!|g|\!\}^{arco}(\Phi) = (\{\!|g|\!\}^{\mathbb{Z}}(\eta), \mu, \{\psi_p\}_{p\in P})$. Sinon, si $g$ ne porte que sur des variables de contenu :

- on applique $g$ sur $\mu$, où chaque cellule de tableau est traitée *via* une variable scalaire fraîche, véhiculant l'information connue sur cette cellule, comme dans le cas de l'affectation à une variable scalaire de contenu (Déf. 6.20, p. 161) ;
- pour ce qui est des propriétés de tableau, on identifie d'abord celles devant être modifiées :
  - si $g$ contient une variable scalaire $x$ alors on modifie chaque propriété $\psi_p$ tel que $x \in \mathit{Var}(\psi_p)$ ;
  - pour chaque cellule de tableau $a[i+c]$ de $g$, s'il existe une variable de tranche dans $\psi_p$ qui la représente, on souhaite modifier la propriété de tableau correspondante. De plus, si on s'appuie sur la particularité de nos partitions, on sait qu'il existe une tranche $\varphi_p$ qui représente exactement la cellule $i+c$ [7]. Si $a^0$ n'existe pas dans $\psi_p$ on souhaite tout de même retranscrire la conditionnelle dans $\psi_p$. Les propriétés de tableau que l'on souhaite modifier, de par la présence de la cellule $i+c$ dans la conditionnelle, sont donc proches de celles définies dans le cas de l'affectation à une cellule de tableau (Sec. 6.6.2.2). Si ce n'est que, si une tranche

7. Cela ne veut pas dire que $g$ peut toujours être représentée exactement. Par exemple si $g = (a[i] < a[j])$ et la propriété $\eta$ n'implique pas que la différence $i-j$ est constante, on ne pourra ni avec la tranche $\varphi_{p_i} = (\ell = i \wedge \ldots)$, ni avec la tranche $\varphi_{p_j} = (\ell = j \wedge \ldots)$ représenter exactement cette comparaison dans $\psi_{p_i}$ ou $\psi_{p_j}$.

$\varphi_p$ représente plus d'une cellule, incluant $i + c$, on ne peut bien entendu tirer aucune information de $g$ sur la propriété $\psi_p$ correspondante (le cas « faible » n'existe pas).

Ainsi, si on regroupe ces choix, on définit l'ensemble des propriétés de tableau à modifier par :

$$\begin{aligned} E = \{p \in P \mid (\exists x \in g \text{ t.q. } x \in \psi_p) \vee \\ (\exists a[\langle expr_i \rangle] \in g \text{ t.q. } \exists z, (z = 0 \vee a^z \in Var(\psi_p)) \\ \wedge\, \eta \sqcap^{\mathbb{Z}} \varphi_p \oplus z \sqsubseteq^{\mathbb{Z}} (\ell = \langle expr_i \rangle))\}. \end{aligned}$$

Il resterait à définir pour chaque $\psi_p$, $p \in E$, la conditionnelle à appliquer. Il est aisé de la définir avec les mêmes techniques qu'utilisées précédemment pour les affectations.

Enfin, dès lors que $g$ contient à la fois des tests sur des variables d'indices et des variables de contenu, on se ramène aux cas précédents en utilisant respectivement, l'opérateur de treillis $\sqcup^{arco}$ et la séquence d'application, pour implanter la disjonction et la conjonction de tests de différentes natures :

$$\begin{aligned} \{\!|g_i \textbf{ or } g_c|\!\}^{arco}(\Phi) &= \{\!|g_i|\!\}^{arco}(\Phi) \sqcup^{arco} \{\!|g_c|\!\}^{arco}(\Phi) \\ \{\!|g_i \textbf{ and } g_c|\!\}^{arco}(\Phi) &= \{\!|g_c|\!\}^{arco}(\{\!|g_i|\!\}^{arco}(\Phi)) \end{aligned} .$$

## 6.7 Conclusion

Dans ce chapitre, nous avons défini dans le détail une analyse du contenu des tableaux. Avant de conclure, on donne un exemple détaillé et complet de son calcul sur un programme simple.

### 6.7.1 Un exemple détaillé

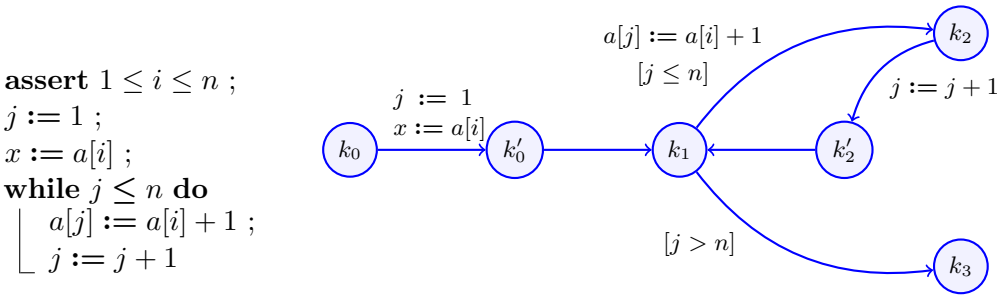
Cet exemple porte sur le programme simple donné à la Figure 6.16(a). C'est un programme *ad hoc* qui permet d'illustrer l'analyse en présence d'une partition selon deux variables d'indice. Il prend en paramètre une valeur $i$ entre 1 et $n$, récupère le contenu $x$ d'un tableau à la $i^{\text{e}}$ cellule et modifie ce tableau tel qu'en sortie il a $x + 1$ dans toutes les cellules à un indice inférieur ou égal à $i$ et $x + 2$ dans toutes les autres.

Dans cet exemple, aucune variable de tranche $a^z$ où $z \neq 0$ n'apparaît et toutes les affectations de variables de tranches sont des affectation fortes. Nous verrons un petit programme où l'on rencontre ces cas mais sur une partition plus simple (Sec. 7.2.2.4). Pour voir de telles variables de tranches et de telles opérations dans un exemple complexe, on pourra se reporter à la présentation de l'analyse du tri par insertion (Sec. 7.2.2.8).

On déroule notre analyse avec $\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{\mathbb{K}} = \mathcal{D}^{zone}$.

**Partitionnement** On considère l'automate interprété donné à la Figure 6.16(b) où les points de contrôle $k_0'$ et $k_2'$ sont confondus avec $k_1$. Les résultats des pré-analyses sont :

- analyse numérique : $x_{k_0}^{zone} = (1 \leq i \leq n)$, $x_{k_1}^{zone} = (1 \leq i \leq n \wedge 1 \leq j \leq n + 1)$, $x_{k_2}^{zone} = (1 \leq i \leq n \wedge 1 \leq j \leq n)$, et $x_{k_3}^{zone} = (1 \leq i \leq n \wedge 2 \leq j = n + 1)$ ;
- analyse de durées de vie : en $k_0$ et en $k_3$ la variable $j$ est morte. Les variables $i$ et $n$ sont considérées comme des paramètres qu'on suppose utilisés après la sortie du programme.




(a) code (b) un automate interprété pour ce programme

Figure 6.16 – Un programme utilisant deux variables d'indice. Il prend un pivot à l'indice $i$, de contenu $x$, et renvoie un tableau contenant $x + 1$ pour les cellules 1 à $i$ et $x + 2$ pour les cellules $i + 1$ à $n$.

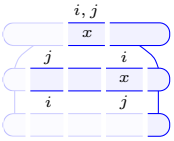
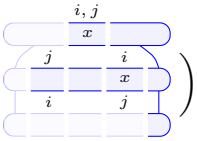
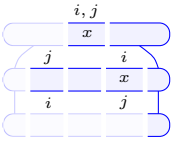
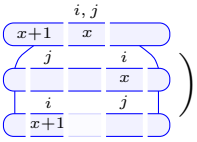
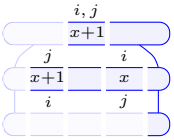
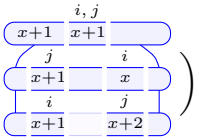
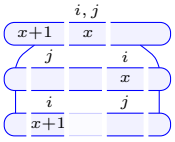
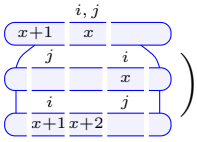
L'analyse de partitionnement sur ce programme est très simple. Son résultat est donné dans la table ci-dessous. Les affectations $x := a[i]$ et $a[j] := a[i] + 1$ conduisent à un partitionnement de l'espace $1 \leq \ell \leq n$ selon $i$ et $j$ pour tous les points de contrôle de la boucle. À la sortie de la boucle, la variable $j$ meurt et elle est donc éliminée de cette partition.

| *itér.* | $P_{k_0}$ | $P_{k_1}$ | $P_{k_2}$ | $P_{k_3}$ |
|---|---|---|---|---|
| *1* | $\{(\ell \geq 1)\}$ | $\{(1 \leq \ell \leq n \wedge \ell < i);$<br>$(1 \leq \ell \leq n \wedge \ell = i);$<br>$(1 \leq \ell \leq n \wedge \ell > i);$<br>$(\ell \geq 1 \wedge \ell > n)\}$ | $\{(1 \leq \ell \leq n \wedge \ell < i \wedge \ell < j);$<br>$(1 \leq \ell \leq n \wedge \ell < i \wedge \ell = j);$<br>$(1 \leq \ell \leq n \wedge \ell < i \wedge \ell > j);$<br>$( \wedge \ell = i \wedge );$<br>$( \ldots );$<br>$( \wedge \ell > i \wedge );$<br>$( \ldots );$<br>$(\ell \geq 1 \wedge \ell > n)\}$ | $\bot^{arpa}$ |
| *2* | inchangée | égale à $P_{k_2}$ | inchangée | $\{(1 \leq \ell \leq n \wedge \ell < i);$<br>$(1 \leq \ell \leq n \wedge \ell = i);$<br>$(1 \leq \ell \leq n \wedge \ell > i);$<br>$(\ell \geq 1 \wedge \ell > n)\}$ |

**Contenu** On passe à l'analyse du contenu que l'on effectue sur l'automate interprété de la Figure 6.16(b). Les points de contrôle $k'_0$ et $k'_2$ ont été ajoutés simplement pour pouvoir reporter les résultats des fonctions de transfert qui avaient pour cibles la tête de boucle $k_1$. Ces deux points de contrôle supplémentaires ont donc la même partition que celle définie pour $k_1$ par l'analyse de partitionnement.

Les résultats de l'analyse du contenu itération par itération sont donnés ci-dessous. On utilise une représentation des partitions que l'on a déjà vue à la Figure 6.11(b) (p. 152), qui est une représentation par cas. Pour la partition principale (celle en $k'_0, k_1, k_2$ et $k'_2$), les différents cas sont $i = j$, $i < j$ et $i > j$. Les tranches $(1 \leq \ell \leq n \wedge \ell < , i, j)$ et $(1 \leq \ell \leq n \wedge \ell > i, j)$, occupant les bords gauche et droit de la représentation, peuvent apparaître dans plusieurs de ces configurations d'où les liens indiquant qu'il s'agit de la même tranche. Lorsque la propriété $\eta$ sur les variables d'indices implique l'insatisfaisabilité d'une des tranches de la partition, celle-ci apparaît en grisé. À l'intérieur d'une tranche, on indique la propriété $\psi_p$ sur le contenu qui lui est associé (la variable de tranche $a^0$ est

implicite : « $x$ » est un raccourci d'écriture pour $a^0 = x$). Si $\psi_p = \top^{\mathbb{K}}$ rien n'est indiqué. Enfin, la tranche où $\ell > n$ dans les différentes partitions n'est pas représentée (et aucune contrainte ne porte sur son contenu).

| | *1* | *2* |
|---|---|---|
| $\Phi_{k_0} =$ | $\left(1 \leq i \leq n, \top^{\mathbb{K}}, \boxed{\top^{\mathbb{K}}}\right)$ | $\left(1 \leq i \leq n, \top^{\mathbb{K}}, \boxed{\top^{\mathbb{K}}}\right)$ |
| $\Phi_{k'_0} =$ | $\left(1 = j \leq i \leq n, \top^{\mathbb{K}}, \right.$  $\left.\right)$ | $\left(1 = j \leq i \leq n, \top^{\mathbb{K}}, \right.$  $\left.\right)$ |
| $\Phi_{k_1} =$ | $\left(1 = j \leq i \leq n, \top^{\mathbb{K}}, \right.$  $\left.\right)$ | $\left(\begin{array}{c}1 \leq i \leq n \\ 1 \leq j \leq i+1\end{array}, \top^{\mathbb{K}}, \right.$  $\left.\right)$ |
| $\Phi_{k_2} =$ | $\left(1 = j \leq i \leq n, \top^{\mathbb{K}}, \right.$  $\left.\right)$ | $\left(\begin{array}{c}1 \leq i,j \leq n \\ j \leq i+1\end{array}, \top^{\mathbb{K}}, \right.$  $\left.\right)$ |
| $\Phi_{k'_2} =$ | $\left(\begin{array}{c}1 \leq i \leq n \\ 2 = j \leq n+1\end{array}, \top^{\mathbb{K}}, \right.$  $\left.\right)$ | $\left(\begin{array}{c}1 \leq i \leq n \\ 2 \leq j \leq n+1 \\ j \leq i+2\end{array}, \top^{\mathbb{K}}, \right.$  $\left.\right)$ |
| $\Phi_{k_3} =$ | $\bot^{arco}$ | $\bot^{arco}$ |

**Première itération** On part en $k_0$ avec aucune connaissance sur le tableau $a$.

Avant d'appliquer les fonctions de transfert de $k_0 \to k'_0$ une opération d'adaptation a lieu. Une valeur abstraite est créée sur la partition de $k'_0$ et à ses dix tranches il est associé la propriété $\psi_p = \top^{\mathbb{K}}$. Suit l'affectation $j := 1$ qui est appliquée à $\eta$, et qui pour chaque tranche tel que $(j = 1 \Rightarrow \varphi_p = \bot^{\mathbb{Z}})$ met $\psi_p$ à la valeur $\bot^{\mathbb{K}}$. L'affection $x := a[i]$ ne modifie pas $\mu$ car il n'y a aucune information sur $a[i]$. Par contre, il existe deux tranches satisfaisables telles que $\varphi_p \Rightarrow \ell = i$, pour lesquelles $a[i]$ est représentable précisément. L'affectation $\{\!|x := a^0|\!\}^{\mathbb{K}}$ est appliquée aux propriétés de ces deux tranches. L'opération de normalisation laisse inchangée la valeur abstraite résultant des opérations précédentes.

Cette valeur abstraite est propagée à $k_1$. La conditionnelle de $k_1 \to k_2$ n'apporte pas d'information. Le traitement de l'affectation $a[j] := a[i] + 1$ est plus complexe. Bien entendu le contenu de la cellule où $\ell = i > j$ est inchangé ($a^0 = x$). Les tranches à affecter sont les deux tranches où $\ell = j$ et qui sont satisfaisables selon $\eta$. Dans le cas de la tranche où $i = j$, l'expression $a[i]$ peut être représentée par la variable de tranche $a^0$ (on a $t_p(i) = 0$, Éq. 6.4, p. 160). Ainsi, l'affectation effectuée est $\{\!|a^0 := a^0 + 1|\!\}^{\mathbb{K}}$ qui a pour résultat $a^0 = x + 1$. Pour l'autre tranche où $i < j$, la cellule $a[j]$ ne peut pas être représentée par une translation de la tranche affectée (on a $t_p(i) = \infty$). On a alors recours à l'opération d'extraction de propriétés de tableaux. On recherche les propriétés sur $(\ell = i)$ dans le contexte des propriétés d'indices qu'impliquent la tranche affectée, *i.e.* $(1 \leq j < i \leq n)$. On a $\psi^{\Phi_{k_1}}(1 \leq j < i = \ell \leq n) = (a^0 = x)$ et donc l'affectation originale est transformée en l'opération suivante, où $y$ est une variable fraîche : $\exists y.\{\!|a^0 := y + 1|\!\}^{\mathbb{K}}(\top^{\mathbb{K}} \sqcap^{\mathbb{K}} (y = x))$. Son résultat est là encore $a^0 = x + 1$.

Suit l'incrémentation de $j$. L'opération est reportée sur $\eta$. Des tranches alors insatis-

faisables ne le sont plus, comme par exemple la tranche où $\ell < i, j$. Pour cette dernière tranche, la nouvelle propriété associée est $a^0 = x+1$, résultat de l'extraction de propriétés de tableaux sur $1 \leq \ell \leq n \wedge l < i \wedge \ell \leq j$ dans $\Phi_{k_2}$. Les propriétés de toutes les autres tranches sont calculées de la même manière (Déf. 6.19, p. 159).

**Deuxième itération** Selon notre stratégie d'itération récursive, on itère jusqu'à stabilisation de la boucle. L'union de $\Phi_{k'_0}$ et $\Phi_{k'_2}$ est directe : on a $\eta = (1 \leq j \leq 2 \wedge 1 \leq i \leq n \wedge j \leq i+1)$, $\mu = \top^{\mathbb{K}}$ et les mêmes propriétés $\psi_p$ que celles de $\Phi_{k'_2}$. Si on élargit $\Phi_{k_1}$ par cette valeur abstraite, on obtient $\eta = (1 \leq j \wedge 1 \leq i \leq n)$ et pour propriétés $\psi_p$ toujours celles de $\Phi_{k'_2}$. Si on a perdu toute relation entre $i$ et $j$ dans $\eta$, la règle 3 de l'opération de normalisation va en ramener une. En effet, la propriété associée à la tranche où $i < \ell < j$ est $\bot^{\mathbb{K}}$ et donc on restreint $\eta$ par la négation de la propriété d'indice impliquée par cette tranche, *i.e.* $(i \geq j-1 \vee n < 1 \vee i > n \vee j \leq 1)$. On trouve alors $\eta = (1 \leq j \wedge 1 \leq i \leq n \wedge j \leq i+1)$ pour la valeur abstraite résultant de l'élargissement.

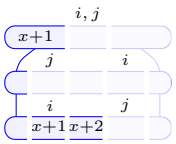
Suit la conditionnelle et l'affectation $a[j] := a[i]+1$ de $k_1 \to k_2$. Par rapport à l'itération précédente, on notera simplement la propriété de tableau $a^0 = x+2$ découverte pour la tranche où $\ell = j > i$. La précision de ce résultat est simplement due au fait que la partition a permis de garder précisément la valeur de $a[i]$ dans le cas où $i < j$.

Enfin, après l'incrémentation de $j$, toutes les tranches de la partition sont satisfaisables. On voit d'ailleurs apparaître l'un des invariants de la boucle, qui est $\forall \ell, i < \ell < j \Rightarrow a[\ell] = x+2$.

| | *3* ↻ | *4* |
|---|---|---|
| $\Phi_{k_0} =$ | $\left(1 \leq i \leq n, \top^{\mathbb{K}}, \boxed{\top^{\mathbb{K}}}\right)$ | $\left(1 \leq i \leq n, \top^{\mathbb{K}}, \boxed{\top^{\mathbb{K}}}\right)$ |
| $\Phi_{k'_0} =$ | $\left(1 = j \leq i \leq n, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} & \overset{i,j}{x} & \\ \overset{j}{\_} & & \overset{i}{x} \\ \underset{i}{\_} & & \underset{j}{\_} \end{array}}\right)$ | $\left(1 = j \leq i \leq n, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} & \overset{i,j}{x} & \\ \overset{j}{\_} & & \overset{i}{x} \\ \underset{i}{\_} & & \underset{j}{\_} \end{array}}\right)$ |
| $\Phi_{k_1} =$ | $\left(\begin{array}{c} 1 \leq i \leq n \\ 1 \leq j \leq n+1 \end{array}, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} \overset{i,j}{x+1} & \overset{}{x} & \\ \overset{j}{\_} & & \overset{i}{x} \\ \underset{i}{\_} & & \underset{j}{\_} \\ \underset{x+1}{\_} & \underset{x+2}{\_} & \end{array}}\right)$ | $\left(\begin{array}{c} 1 \leq i \leq n \\ 1 \leq j \leq n+1 \end{array}, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} \overset{i,j}{x+1} & \overset{}{x} & \\ \overset{j}{\_} & & \overset{i}{x} \\ \underset{i}{\_} & & \underset{j}{\_} \\ \underset{x+1}{\_} & \underset{x+2}{\_} & \end{array}}\right)$ |
| $\Phi_{k_2} =$ | $\left(\begin{array}{c} 1 \leq i \leq n \\ 1 \leq j \leq n \end{array}, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} \overset{i,j}{x+1} & \overset{}{x+1} & \\ \overset{j}{\_} & & \overset{i}{x} \\ \underset{i}{\_} & & \underset{j}{\_} \\ \underset{x+1}{\_} & \underset{x+2}{\_} & \underset{x+2}{\_} \end{array}}\right)$ | $\left(\begin{array}{c} 1 \leq i \leq n \\ 1 \leq j \leq n \end{array}, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} \overset{i,j}{x+1} & \overset{}{x+1} & \\ \overset{j}{\_} & & \overset{i}{x} \\ \underset{i}{\_} & & \underset{j}{\_} \\ \underset{x+1}{\_} & \underset{x+2}{\_} & \underset{x+2}{\_} \end{array}}\right)$ |
| $\Phi_{k'_2} =$ | $\left(\begin{array}{c} 1 \leq i \leq n \\ 2 \leq j \leq n+1 \end{array}, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} \overset{i,j}{x+1} & \overset{}{x} & \\ \overset{j}{\_} & & \overset{i}{x} \\ \underset{i}{\_} & & \underset{j}{\_} \\ \underset{x+1}{\_} & \underset{x+2}{\_} & \end{array}}\right)$ | $\left(\begin{array}{c} 1 \leq i \leq n \\ 2 \leq j \leq n+1 \end{array}, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} \overset{i,j}{x+1} & \overset{}{x} & \\ \overset{j}{\_} & & \overset{i}{x} \\ \underset{i}{\_} & & \underset{j}{\_} \\ \underset{x+1}{\_} & \underset{x+2}{\_} & \end{array}}\right)$ |
| $\Phi_{k_3} =$ | $\bot^{arco}$ | $\left(\begin{array}{c} 1 \leq i \leq n \\ 2 \leq j = n+1 \end{array}, \top^{\mathbb{K}}, \boxed{\begin{array}{ccc} & \overset{i}{} & \\ \underset{x+1}{\_} & \underset{x+1}{\_} & \underset{x+2}{\_} \end{array}}\right)$ |

**Troisième itération** L'union de $\Phi_{k'_0}$ et $\Phi_{k'_2}$ renvoie la valeur abstraite où $\eta = (1 \leq j \leq n+1 \wedge 1 \leq i \leq n \wedge j \leq i+2)$, $\mu = \top^{\mathbb{K}}$ et les propriétés $\psi_p$ sont celles de $\Phi_{k'_2}$. Le résultat de l'élargissement de $k_1$ par cette valeur abstraite est l'invariant de boucle. Après calcul des fonctions de transfert suivantes, on voit que $\Phi_{k_1}$ est stable.

**Quatrième itération** Il ne reste plus qu'à calculer $\Phi_{k_3}$. La conditionnelle de $k_1 \to k_3$ est appliquée avant l'opération d'adaptation puisque la partition en $k_1$ est plus précise (Rem. 6.3, p. 155). On obtient $\eta = (1 \leq i \leq \wedge 2 \leq j = n+1)$ et donc toutes les tranches tel que $\ell \leq n \wedge \ell \geq j$ deviennent insatisfaisables. L'application de la règle 7 de l'opération de normalisation attribue à ces tranches la propriété $\bot^{\mathbb{K}}$. On obtient donc les propriétés schématisées ci-contre. Il ne reste plus qu'à adapter cette valeur abstraite sur la partition de $k_3$, partition sans $j$, dont le résultat est donné au tableau ci-dessus.




L'analyse a découvert l'invariant souhaité : $\forall \ell, 1 \leq \ell \leq i \leq n \Rightarrow a[\ell] = x+1$ et $\forall \ell, 1 \leq i < \ell \leq n \Rightarrow a[\ell] = x+2$.

**Séquence descendante** Une itération est effectuée et montre la stabilité des invariants. L'analyse s'arrête. On remarquera que la contrainte $j \leq i+1$ ajoutée par la normalisation durant la seconde itération a permis de découvrir en $k_1$ que $j \leq n+1$ durant la séquence ascendante. Sans cette information, c'est la première itération de la séquence descendante qui aurait découvert que $j \leq n+1$.

Notre prototype d'analyseur Enkidu (Sec. 7.1) traite cet exemple en 0.42 seconde.

## Conclusion

Dans ce chapitre, nous avons décrit en détail notre analyse du contenu des tableaux. Notamment, nous avons supposé une certaine uniformité des programmes quant aux tranches symboliques pour lesquelles il existe un invariant du contenu des tableaux. Dès lors, nous avons choisi un partitionnement statique et donc séparé le problème du partitionnement de la recherche des propriétés sur le contenu elles-mêmes. Ceci nous a permis de nous concentrer sur ce second problème qui, on l'a vu tout au long du chapitre, demande déjà de définir des opérations complexes. Nous avons d'ailleurs fait un second choix à propos de ces propriétés sur le contenu, qui est d'associer des propriétés aux tranches seulement (et pas entre des tranches, comme $\forall \ell_1, \ell_2,\ (\ldots \Rightarrow a[\ell_1] \leq a[\ell_2])$), en leur donnant une sémantique cellule-par-cellule.

Ces deux choix, partitionnement statique et utilisation d'une variable quantifiée dans toute propriété de tableau, ont certainement permis que l'on conçoive cette analyse, simplifiant les raisonnements et contribuant à éviter que l'analyse diverge. Bien entendu, avec notre compréhension actuelle des analyses du contenu des tableaux, on ne pense qu'à repousser ces limitations. Pourtant celles-ci ont la vertu de rendre cette analyse aisément compréhensible, tant à propos de son fonctionnement que des résultats que l'on peut attendre d'elle. À ce sujet, nous avons été étonnés des remarques de David Wonnacott, nous écrivant que ses travaux sur l'analyse de dépendances des indices de tableaux [PW98] bénéficieraient de propriétés sur le contenu des tableaux que notre analyse pourrait découvrir. Il donnait alors des exemples d'invariants qu'il lui faudrait découvrir pour paralléliser le programme

```
for i = 1 to n do
    for j = 1 to b[i] do
        c[a[i] + j] := ...;
```

Figure 6.17 – Un programme, issu du *benchmark* DYFESM [PW98]

donné à la Figure 6.17, qui peuvent en effet être exprimés par notre domaine abstrait : $\forall \ell, 1 \leq \ell \leq n \Rightarrow a[\ell + 1] \geq a[\ell] + b[\ell]$, de par l'indexation du tableau $c$ par le contenu du tableau $a$, et $\forall \ell, 1 \leq \ell \leq n \Rightarrow b[\ell] > 0$, de par l'utilisation du contenu de $b$ pour borner l'itération interne.

La question qui suit est de savoir si notre analyse répond à ces attentes. L'exemple que nous avons déroulé dans cette section laisse entrevoir les capacités de notre analyse. Les résultats que nous avons obtenus, présentés au chapitre suivant (Sec 7.2), sont très bons. C'est dans ce chapitre que nous comparerons notre analyse aux analyses existantes et discuterons de ses améliorations.

# Chapitre 7

# Implantation et expérimentation des analyses

Nous présentons dans ce chapitre notre prototype d'analyseur Enkidu (Sec. 7.1), dans lequel nous avons implanté les deux analyses définies dans cette thèse. Nous l'utilisons pour valider expérimentalement nos analyses (Sec. 7.2). Dans cette section, on discute les résultats et les performances de l'implantation de notre analyse du contenu des tableaux, en détaillant chaque exemple analysé avec succès. Pour les autres exemples, nous décrivons comment certains peuvent être analysés grâce à de simples modifications de notre analyse. Enfin, nous concluons brièvement ce chapitre (Sec. 7.3), en donnant notamment les résultats des travaux de Manuel Garnacho sur la génération automatique de certificats par Enkidu.

## 7.1 Présentation d'Enkidu

Enkidu est un prototype d'analyseur que nous avons développé pour nous aider à concevoir nos deux analyses et nous permettre de les valider expérimentalement.

Ce prototype, que nous étions seuls à développer, n'a jamais eu vocation à devenir un analyseur abouti. Par conséquent, ses points d'entrée (notamment le langage utilisé pour décrire les programmes) et ses points de sortie ont été négligés. Ses performances n'ont pas été particulièrement travaillées. Par contre le développement de ses parties internes a été pensé pour être générique et modulaire afin de pouvoir prototyper rapidement toute nouvelle idée ou analyse. Grâce à la théorie de l'interprétation abstraite, ce prototype partage des caractéristiques essentielles avec des analyseurs utilisés dans l'industrie, comme Astrée[1] ou PolySpace[2] : Enkidu est paramétrique, modulaire, exhaustif, correct et automatique.

Enkidu est écrit en OCaml à partir d'une base de code écrite par Bertrand Jeannet, incluant principalement un solveur de systèmes d'équations de points fixes, générique pour un domaine abstrait, et nommé Fixpoint [Jea10]. Sur les 17000 lignes que compte le code, la moitié est de notre fait. Ce code est organisé dans quatre sous-répertoires. Un répertoire « langages » où se trouvent les compilateurs entre les langages d'entrée et de sortie et notre représentation interne des programmes en automates interprétés (Déf. 2.5, p. 20). Les

1. `http://www.absint.com/astree/`
2. `http://www.mathworks.com/products/polyspace/`

actions et gardes de ces automates sont celles définies en Préliminaires (Fig. 4.5, p. 76) où $\mathbb{K} = \mathbb{Z}$ car le seul type disponible est celui des entiers [3]. Un répertoire « analyses » où se trouvent les analyses statiques, principalement d'atteignabilité, d'un automate interprété et paramétrées par des domaines abstraits. Un répertoire « domaines abstraits » pour ces derniers et enfin un répertoire « librairies » contenant des structures de données utilisées par plusieurs modules. À la racine on trouve seulement les signatures principales et le pilote de l'analyseur articulant compilations, appels aux analyses et affichage des résultats. Il peut être invoqué avec les options suivantes :

```
 -analysis=ANALYSIS       Run the ANALYSIS on the program
                            ANALYSIS is 'numerical', by default, or 'array'
-underlying=DOMAIN        Use DOMAIN as parameter of the numerical analysis
                            DOMAIN is 'interval', by default, 'dbm', 'ddbm',
                                   or 'ddbm-weak'
           =(DOMAIN_Z, DOMAIN)
                          Use (DOMAIN_Z, DOMAIN) as parameter of the array analysis
                            DOMAIN_Z is 'dbm'
                            DOMAIN is 'interval', 'dbm', 'ddbm-weak', by default,
                                   or 'ddbm'
-widening=NUM             Apply widening after NUM iterations,
                            NUM is non-negative, 1 by default
-narrowing=NUM            Apply NUM descendings iterations
                            NUM is non-negative, 3 by default
-verbose=NUM              Run with verbose level NUM
                            Num ranges in {0,1,2}, 0 by default
```

### 7.1.1 Fonctionnement détaillé

On présente Enkidu dans le détail, par ses répertoires. On s'intéresse principalement à l'implantation des différentes analyses et des domaines abstraits sous-jacents.

#### 7.1.1.1 Langages

Nous nous sommes contentés d'un seul langage d'entrée, le langage d'automates textuel Fast [BLP06]. Il s'agit du langage d'un analyseur d'automates à compteur éloigné de nos travaux. Le choix de Fast est un choix historique, lié au développement d'un autre prototype d'analyseur dans l'équipe, ASPIC [Gon07]. Nous avons simplement étendu la grammaire de Fast avec des expressions portant sur les tableaux, pour pouvoir exprimer l'ensemble du langage présenté en Préliminaires.

Nous n'avons également qu'un langage de sortie : les programmes analysés et leurs invariants sont donnés au format Dot [4], ce qui permet de les visualiser avec `dotty` par exemple.

3. On s'affranchit de tous les problèmes liés au fait que ce sont des entiers machines et non les nombres entiers parfait que nous avons supposés pour la définition de nos domaines abstraits. De la même manière on s'affranchit du fait que les domaines abstraits sont eux-mêmes implantés avec des entiers machines. On trouvera des solutions à ces problèmes dans [Min01] par exemple.

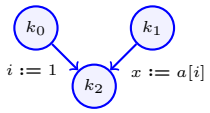
4. `http://www.graphviz.org/`

#### 7.1.1.2 Analyses

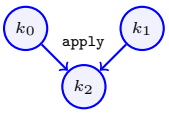
Une analyse prend en entrée et renvoie en sortie un automate interprété auquel est attaché un ensemble d'informations, possiblement vide. Les informations données en entrée ne sont pas forcément nécessaires au fonctionnement de l'analyse, elles peuvent par exemple lui permettre d'améliorer ses performances en temps ou en précision. Les informations données en sortie contiennent bien entendu les résultats de l'analyse. Par exemple, pour une analyse d'atteignabilité l'automate interprété restera inchangé et les informations consisteront en une table associant à chaque place de cet automate une sur-approximation de ses états atteignables sous forme d'un système de contraintes, une représentation indépendante de tout domaine abstrait. L'analyseur, qui connaît les dépendances entre les informations apportées par telle analyse et nécessaires à telle autre, peut composer des analyses.

Les différentes analyses que nous avons implantées utilisent toutes le solveur Fixpoint. On commence par présenter brièvement comment il s'utilise avant de présenter nos analyses.

Fixpoint Ce solveur est équivalent à l'interpréteur abstrait présenté à la Section 2.2.2.2 mis à part qu'il ne requiert que les opérateurs ensemblistes et l'opérateur d'élargissement du domaine abstrait $\mathcal{D}^\sharp$ (Déf. 2.6, p. 22, restreinte aux points 1, 4 et 5). Ainsi il ne travaille pas sur l'automate interprété mais sur un graphe dont les transitions sont déjà les fonctions de transfert abstraites. Ces fonctions sont à définir dans une même fonction `apply` qui, pour une transition et une valeur abstraite renvoie une valeur abstraite. Par exemple, l'automate interprété ci-contre peut-être analysé par Fixpoint en le traduisant dans le graphe ci-contre et la fonction `apply` suivante :




(a) Automate interprété




(b) Graphe

```
let apply transition av =
  match transition with
      (k0, k2) -> D.assign (Scalar "i", Cst 1) av
    | (k1, k2) -> ...
```

où `av` est une valeur abstraite de $\mathcal{D}^\sharp$ et `D.assign` l'opérateur sémantique de $\mathcal{D}^\sharp$ pour les affectations.

Pour invoquer Fixpoint, il faut donc fournir :

- ce graphe, avec ses états initiaux, et la fonction `apply` qui n'est autre que la fonction de l'équation de point fixe à résoudre, partitionnée ;
- une stratégie d'itérations qui sera toujours la stratégie récursive classique présentée à la Section 2.2.2.2 ;
- le nombre d'itérations avant d'appliquer l'opérateur d'élargissement et le nombre d'itérations descendantes, paramètres modifiables par l'utilisateur comme on l'a vu ;
- et enfin un domaine abstrait dont la signature est formée des opérateurs précités seulement.

Alors Fixpoint renvoie une solution associant à chaque nœud du graphe une valeur abstraite qu'il faut faire correspondre aux places de l'automate interprété, ce qui sera toujours direct pour nos utilisations.

**Analyse numérique** Cette analyse n'a besoin en entrée d'aucune autre information que l'automate interprété. Elle est paramétrée par un domaine abstrait numérique $\mathcal{D}^{\mathbb{Z}}$. Un tel domaine ne fournit pas d'opérateurs abstraits pour les fonctions de transfert portant sur des éléments de tableaux. Par conséquent, la traduction de l'automate interprété en graphe doit s'occuper d'approximer toute action ou garde portant sur des éléments de tableaux. Cette approximation conservative est très simple : les affectations à un élément de tableau disparaissent, les affectations à des scalaires et les gardes sont traduites par un appel à l'opérateur d'oubli correspondant. Toujours avec le même exemple, on a :

```
let apply transition av =
  match transition with
      (k0, k2) -> ...
    | (k1, k2) -> D.forget (Scalar "x") av
```

Une fois cette traduction faite, Fixpoint est invoqué avec $\mathcal{D}^{\mathbb{Z}}$. Son résultat est transformé en une table associant à chaque place non pas un élément de $\mathcal{D}^{\mathbb{Z}}$ mais son équivalent en système de contraintes. Cette table est renvoyée avec l'automate interprété tel qu'en entrée.

**Analyse de partitionnement** Cette analyse prend en entrée le résultat d'une analyse numérique et d'une analyse de durées de vie des variables. Elle est paramétrée par un domaine abstrait $\mathcal{D}^{\mathbb{P}}$ de partitionnement symbolique. Un tel domaine a une signature similaire à un domaine abstrait classique, à l'exception de ses opérateurs sémantiques qui prennent un argument supplémentaire, contenant les résultats de ces deux pré-analyses. En pratique, ces opérations sémantiques n'ont besoin que d'une connaissance locale de ces résultats. Pour ce qui est de la pré-analyse numérique, seul l'ensemble atteignable au point de contrôle précédant la fonction de transfert est nécessaire (pour pouvoir calculer $\mathit{ContextSplit}(\langle \mathit{expr}_i \rangle)$, Sec. 6.5.1.1). Pour la pré-analyse de durées de vie, seul l'ensemble des variables « mourant » au point de contrôle suivant la fonction de transfert sont nécessaires (pour pouvoir appeler $\mathit{Forget}(i)(P)$, Sec. 6.5.1.1 toujours). La fonction `apply` est alors définie comme suit pour notre exemple, où l'on voit l'intérêt de la généricité de Fixpoint.

```
let apply transition av =
  match transition with
      (k0, k2) -> ...
    | (k1, k2) ->
        let constraint_set = get_numerical_information k1 in
        let dying_set = get_liveness_information (k1, k2) in
        let av' = D.assign constraint_set
                           (Scalar "x", Cell (Array "a", Scalar "i")) av in
        let av'' = D.simplify dying_set av' in
          match av'' with
              None -> av'
```

```
    | Some av' -> ((add_vertex_between (k1, k2) av') ; av'')
```

L'analyse ne renvoie pas directement le résultat de Fixpoint invoqué avec $\mathcal{D}^{\mathbb{P}}$, qui associe à chaque place de l'automate interprété une partition. Elle effectue une analyse simple sur cet automate (donc sans utiliser Fixpoint) et ces résultats pour collecter deux informations.

- À chaque place de l'automate, cette analyse identifie l'ensemble des variables utilisées par la partition pour représenter les tranches symboliques, *i.e.* les variables d'indices. Toutes les autres variables seront considérées comme des variables de contenu. Ces deux ensembles de variables sont nécessaires à l'analyse du contenu des tableaux pour qu'elle puisse appliquer les fonctions de transfert adéquates (Sec. 6.6.2).
- Puis cette analyse identifie si les partitions à la place précédente et à la place suivante de chaque transition sont différentes. Cette information est utile à l'analyse du contenu des tableaux pour qu'elle sache quand elle doit appliquer l'opération d'adaptation (Déf. 6.14, p. 154).

À partir de ces informations l'analyse de partitionnement renvoie les informations suivantes :

- une table associant à chaque place de l'automate l'ensemble des variables d'indices, l'ensemble des variables de contenu et la partition calculée (sous la forme d'un élément de $\mathcal{D}^{\mathbb{P}}$) ;
- et une table indiquant pour chaque transition si une opération d'adaptation est nécessaire.

Enfin, on note que l'automate interprété renvoyé peut être différent de celui donné en entrée. Cela arrive lorsqu'une fonction de transfert nécessite temporairement une partition plus fine (*cf.* paragraphe précédent la Définition 6.12, p. 150).

**Analyse du contenu des tableaux** Cette analyse prend en entrée le résultat d'une analyse de partitionnement. Elle est paramétrée par un domaine abstrait de contenu des tableaux $\mathcal{D}^{\mathbb{A}}$ qui a accès au domaine abstrait $\mathcal{D}^{\mathbb{P}}$ utilisé par la pré-analyse de partitionnement, puisque c'est des partitions de ce type qui lui seront transmises. $\mathcal{D}^{\mathbb{A}}$ n'a pas exactement la même signature qu'un domaine abstrait classique. D'une part, il inclut une nouvelle opération, qui est l'opération d'adaptation qui prend en argument la valeur abstraite à adapter et les partitions cibles et sources. Si cette opération apparaît dans la signature, c'est parce que c'est l'analyse que l'on décrit qui l'appellera et non les opérations sémantiques de $\mathcal{D}^{\mathbb{A}}$ en interne. Ainsi ces opérations sémantiques travaillent toujours sur une seule partition. D'autre part, les opérations sémantiques de $\mathcal{D}^{\mathbb{A}}$ prennent, comme $\mathcal{D}^{\mathbb{P}}$ précédemment, un argument particulier pour accéder aux résultats locaux de la pré-analyse. Cet argument est un couple composé de la partition sur laquelle s'effectue la fonction de transfert et d'une table indiquant si les variables apparaissant dans la fonction de transfert sont des variables d'indice ou de contenu. Toujours avec le même exemple, si on suppose que la transition `(k1, k2)` doit faire appel à l'opération d'adaptation selon la pré-analyse, on a la fonction `apply` suivante (*cf.* Remarque 6.3, p. 155, sur la gestion de l'adaptation avant ou après l'affectation) :

```
let apply transition av =
  match transition with
      (k0, k2) ->
```

```
      let partition = get_partition k2 in
      let types = get_variables_information ["i"] in
        D.assign (partition, types) (Scalar "i", Cst 1) av
  | (k1, k2) ->
      let source_partition = get_partition k1 in
      let target_partition = get_partition k2 in
      let types = get_variables_information ["x", "i"] in
      let adapt = D.adapt (source_partition, target_partition) in
      let assign partition =
        D.assign (partition, types)
                 (Scalar "x", Cell (Array "a", Scalar "i")) in
      if DP.is_leq (source_partition, target_partition) then
        let av' = adapt av in
          (assign target_partition) av'
      else
        let av' = (assign source_partition) av in
          adapt av'
```

Le résultat de Fixpoint avec $\mathcal{D}^{\mathbb{A}}$ est couplé à la table fournie par la pré-analyse associant à chaque place une partition, pour créer une table associant à chaque place le système de contraintes (quantifiées *a contrario* de l'analyse numérique) représentant l'ensemble atteignable. C'est cette table qui est renvoyée par l'analyse, avec l'automate interprété inchangé.

#### 7.1.1.3 Domaines abstraits

Parmi les domaines abstraits que nous avons implantés, on trouve les domaines classiques des intervalles ($\mathcal{D}^{int}$, Sec. 2.2.3.1) et des zones ($\mathcal{D}^{zone}$, Sec. 3.1.1). Les autres domaines sont ceux nécessaires à nos deux analyses, pour lesquels on ne fait que quelques remarques sur leur implantation.

- Trois domaines abstraits pour les *zones précisant alias* (Ch. 5) sont disponibles. Ils utilisent la même base de code et diffèrent seulement par leurs opérations de normalisation. Un domaine implante l'opération de normalisation du cas dense, un autre l'opération de normalisation exacte du cas discret et enfin, celui que l'on notera $\mathcal{D}^{zoal}$ implante l'opération de normalisation approximée du cas discret. Ces deux derniers domaines sont respectivement ceux nommés `ddbm` et `ddbm-weak` dans les options d'Enkidu.
  Comme on l'avait évoqué à la Section 5.8, l'implantation de ces domaines n'utilise pas de matrice creuse pour représenter les non-égalités.
- Une implantation du domaine abstrait $\mathcal{D}^{arco}$, défini au Chapitre 6, est donnée. Il s'agit d'un foncteur de trois domaines abstraits : $\mathcal{D}^{\mathbb{Z}}, \mathcal{D}^{\mathbb{K}}$ et $\mathcal{D}^{\mathbb{P}}$ ce dernier étant un domaine abstrait de partitionnement. $\mathcal{D}^{\mathbb{K}}$ doit respecter la signature donnée à la Définition 6.3 (p. 129), et $\mathcal{D}^{\mathbb{Z}}$ doit seulement respecter les contraintes supplémentaires 6, 7 et 10 de cette même signature. La raison est que notre implantation diffère de la définition de $\mathcal{D}^{arco}$. Elle est plus générique, d'où la présence supplémentaire de $\mathcal{D}^{\mathbb{P}}$ en paramètre. Dans la définition, les tranches de la partition et la propriété sur les variables d'indices $\eta$ sont des valeurs abstraites de $\mathcal{D}^{\mathbb{Z}}$.

Dans notre implantation l'utilisation de $\mathcal{D}^{\mathbb{Z}}$ est ramenée à la manipulation de la propriété $\eta$ seulement. Par contre une tranche est une structure de données abstraite de $\mathcal{D}^{\mathbb{P}}$, qui fournit un sous-module pour manipuler ces tranches. On retrouve dans la signature de ce sous-module toutes les opérations nécessaires pour effectuer les opérations sémantiques de $\mathcal{D}^{arco}$ : des opérations de treillis sur ces tranches, mais aussi des opérations spécifiques comme la translation (Déf. 6.7, p. 136) ou encore l'opération $t(\varphi_p)(\langle expr_i \rangle)$ définie à la Section 6.6.2.2 nécessaire à l'implantation des affectations de variables de contenu. Pour le reste, l'implantation est très proche de la définition. Aucune structure de données particulière n'est nécessaire.

– Enfin, une implantation du domaine abstrait $\mathcal{D}^{arpa}$ de partitionnement, définie à la Section 6.5, est donnée. Il s'agit là encore d'un foncteur d'un domaine abstrait numérique $\mathcal{D}^{\mathbb{Z}}$. Ce domaine est utilisé pour représenter les tranches symboliques et doit donc respecter la signature donnée à la Définition 6.3.
Aucun travail n'a été fait pour définir une structure de donnée évoluée pour les partitions. Ici une partition est simplement un tableau de tranches.

#### 7.1.1.4 Librairies

On trouve dans ce répertoire plusieurs structures de données utilisées par Fixpoint (par exemple les graphes qu'il analyse) et que nous réutilisons dans notre code. Nous avons simplement ajouté à cette librairie des matrices génériques (proposant l'algorithme de Floyd-Warshall (Fig. 3.5, p. 36), etc) pour l'implantation des différents domaines abstraits basés sur les zones.

#### 7.1.1.5 Pilote

On donne à la Figure 7.1 le flot d'exécution d'Enkidu lorsqu'il lui est demandé une analyse du contenu des tableaux (`-analysis array`) d'un programme Fast. On voit d'une part les instances des foncteurs $\mathcal{D}^{arpa}$ et $\mathcal{D}^{arco}$ qu'il crée à partir des domaines abstraits $\mathcal{D}^{\mathbb{Z}}$ et $\mathcal{D}^{\mathbb{K}}$, et d'autre part la suite d'analyses qu'il instancie et exécute.

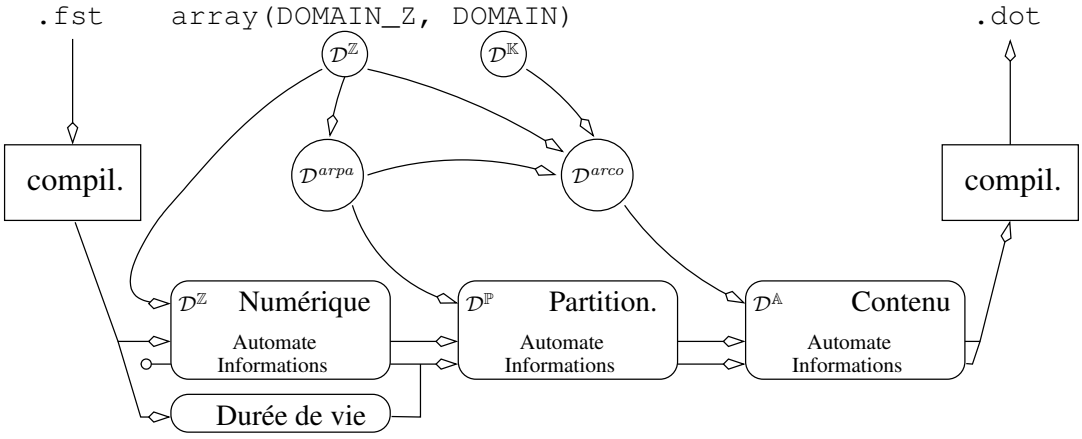



Figure 7.1 – Flot d'exécution d'Enkidu pour une analyse du contenu des tableaux

En plus de la sortie Dot, Enkidu rappelle sur le terminal les options d'analyse choisies, et les invariants découverts en têtes de boucles et au point de contrôle final

(*cf.* exemple à la Section C). Selon le mode `-verbose` choisi, un fichier `.log` contenant les résultats de l'analyse itération par itération est créé (niveau 1) et diverses informations de débogage sont insérées dans ce fichier (niveau 2).

**Conclusion** Cette description de notre prototype permet de comprendre comment les analyses que l'on a définies peuvent s'articuler et être implantées de manière modulaire. Ce que nous apprend aussi Enkidu c'est que l'implantation de ces analyses dans un analyseur existant (donc déjà équipé d'un solveur et d'une architecture pour définir de nouvelles analyses) n'est pas si coûteux. En effet, l'implantation de l'ensemble des domaines abstraits représente un peu moins de cinq mille lignes de code OCaml. Mille lignes ont été nécessaires pour implanter les domaines abstraits des zones précisant alias, huit cents pour le domaine abstrait de partitionnement des tableaux et trois mille pour le domaine abstrait du contenu des tableaux. Il s'agit de développements conséquents mais envisageables pour des équipes développant des analyseurs, même si elles auront à faire face à de nombreux problèmes que nous avons évités ici. Par exemple, on peut s'attendre à ce que l'analyse de partitionnement devienne difficile sur la représentation intermédiaire utilisée par ces analyseurs ; probablement la taille des programmes analysés empêchera de garder en mémoire une partition par point de contrôle, etc.

Enkidu continue à évoluer depuis mon départ de l'équipe. Valentin Perrelle a ajouté un sous-ensemble de C en langage d'entrée, plusieurs nouveaux domaines abstraits, dont le produit réduit et ceux nécessaires à une analyse de permutation du contenu des tableaux [PH10], analyse également disponible.

## 7.2 Validation expérimentale de l'analyse du contenu des tableaux

On rappelle que la validation expérimentale de l'analyse des zones précisant alias étant très faible, elle a été présentée directement à la suite de la description de cette analyse (Sec. 5.8, p. 120). La validation expérimentale de notre seconde analyse est beaucoup plus conséquente. On donne tout de suite une vue générale sur les résultats de cette analyse avant de détailler son fonctionnement sur de nombreux exemples (Sec. 7.2.2 et 7.2.4).

### 7.2.1 Résultats

Dans la table ci-dessous (Fig. 7.2), on liste l'ensemble des programmes qui ont été analysés avec Enkidu. On retrouve dans la partie haute de cette table les programmes proposés dans la littérature (Sec. 4.4, p. 69), au détail près que le programme de segmentation du tri rapide considéré ici est une de ses versions optimisées, plus complexe. Dans la partie basse, on trouve de nouveaux exemples. On s'est intéressé à plusieurs autres programmes (Annexe B). Certains étaient hors de portée de notre analyse (Sec. 7.2.5), mais d'autres ont pu être analysés avec succès (à la main) avec une version légèrement modifiée de l'analyse (Sec. 7.2.4).

Tous ces programmes permettent de valider de nombreuses manipulations différentes de tableaux : on trouve des programmes à une boucle, ou deux boucles imbriquées, incrémentant ou décrémentant les variables d'indices d'un pas de un ou plus, échangeant le

contenu de deux cellules, écrasant le contenu d'une cellule par celui d'une autre, comparant le contenu d'une cellule à une variable scalaire, stockant dans une variable entière l'indice d'une cellule au contenu particulier, etc. Enfin, la plupart des ces programmes ne sont pas des programmes *ad hoc* : on les rencontre en pratique, sous des formes similaires. Par exemple, on trouve un tri par insertion, quasiment équivalent à celui que l'on analyse ici, dans `mscorlib`.

| | *[HP08]* |
|---|---|
| initialisation | 0.02 |
| filtre par indirection | × |
| filtre partitionnant | 0.12 |
| premier non nul | 0.85 |
| copie de tableau | 0.02 |
| tri par insertion | 5.38 |
| insertion seulement | + |
| tri à bulle | × |
| tri par sélection | × |
| minimum seulement | 0.1 |
| segmentation tri rapide | 22.87 |
| initialisation pivot | 0.42 |
| initialisation séquence | 0.05 |
| initialisation *copy prop.* | 0.07 |
| sentinelle | 0.21 |
| *machine (cadence proc.)* | 1.6Ghz |
| *(capacité mém.)* | 2Gb |

Figure 7.2 – Performances

Pour toutes nos expériences, nous avons utilisé la même analyse, où $\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{zone}$ et $\mathcal{D}^{\mathbb{K}} = \mathcal{D}^{zoal}$. Enkidu était exécuté sur une machine avec un processeur *Core Duo 2* cadencé à 1.6Ghz et avec 2Mo de mémoire vive. Comme dans la table à la Figure 4.4, qui donne les résultats des analyses existantes, on donne dans la table ci-dessus nos temps d'analyses en secondes, et lorsque l'on n'a pas ces temps, on indique d'un « + » le fait que l'analyse trouve l'invariant souhaité et d'un « × » le fait qu'elle échoue à trouver cet invariant.

Si on se restreint aux programmes qui ont été utilisés pour valider les analyses existantes, notre analyse est celle qui analyse avec succès le plus de programmes. C'est un résultat important car à l'inverse de la quasi-totalité de ces analyses, les invariants de tous ces programmes sont découverts automatiquement. De plus, cette polyvalence n'empêche pas notre analyse d'être clairement plus efficace. Si l'on regarde pour chaque programme le meilleurs temps d'analyse des analyses existantes, on constate que : sur les cinq premiers exemples, les plus simples, notre analyse s'exécute de 4 à 40 fois plus rapidement ; pour les six derniers exemples, les plus complexes, si on exclut l'analyse exigeant de l'utilisateur le plus d'information ([SG09]), notre analyse s'exécute de 2 à 10 fois plus rapidement.

La capacité de notre analyse à découvir les invariants des nouveaux exemples donnés dans la partie basse, où l'on trouve notamment l'exemple particulier de la sentinelle, confirme son caractère polyvalent et sa précision.

Bien entendu, notre analyse ne réussit pas à découvrir les invariants de tous les exemples ci-dessus. Nous avions déjà vu que le choix de l'expressivité de notre domaine abstrait (Sec. 6.2) nous empêcherait de traiter le tri à bulle et le filtre par indirection (pour lequel il faut pouvoir exprimer que $a[b[\ell]] = 0$). Nous verrons plus loin pourquoi on découvre pour le tri par sélection que $\forall \ell,\ (1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] \leq x)$, mais pas l'invariant de tri. Quant à nos temps d'analyse, s'ils sont satisfaisants, eu égards à notre implantation, pour la majorité des exemples, ils sont trop importants pour les exemples complexes comme le tri par insertion ou la segmentation du tri rapide. C'est surtout le rapport entre les temps d'analyse de ces deux exemples et le rapport avec les temps d'analyses des exemples plus simples qui posent problème.

Au final, ces résultats montrent que notre analyse, automatique, a atteint son objectif : elle est bien plus précise, polyvalente et efficace que les analyses existantes. Alors que ces expérimentations révèlent déjà les capacités importantes de notre analyse, celle-ci recèle un grand potentiel d'amélioration. Nous allons voir qu'avec de simples adaptations, notre analyse peut découvrir les invariants de programmes aux fonctionnalités encore différentes. Enfin, en travaillant sur les structures de données, notamment pour représenter la partition, elle pourrait être beaucoup plus efficace.

### 7.2.2 Les exemples analysés avec succès

On décrit et on commente dans cette section les exemples analysés avec succès. On commence par les algorithmes d'initialisation de tableaux, puis ceux recherchant un élément particulier dans un tableau et enfin ceux de tri. Les performances en temps des analyses sont discutées de manière globale à la section suivante (Sec 7.2.3).

Tous nos programmes sont analysés avec l'état initial $n \geq 1$. Enfin, pour plus de clarté dans les résultats, on omet les propriétés de tableaux portant sur les tranches où $\ell > n$, sauf lorsque l'analyse découvre une information sur ces cellules.

#### 7.2.2.1 Initialisation selon un autre tableau (Fig. 7.3(a))

Ce petit exemple est à rapprocher de l'exemple d'initialisation à zéro ou de la copie de tableau que nous avons déjà vu fonctionner plusieurs fois. Tous ces exemples sont traités avec la même efficacité.

L'exemple très similaire ci-contre effectue une initialisation partielle du tableau. La première cellule est laissée telle qu'elle parce qu'elle porte une information particulière par exemple. Cet exemple *ad hoc* (car on l'aurait probablement écrit avec une boucle allant de 2 à $n$) rappelle simplement le caractère sémantique de notre analyse de partitionnement. En s'appuyant sur une pré-analyse numérique (qui découvre que $2 \leq i+1 \leq n$) et son opérateur *ContextSplit* (Sec. 6.5.1.1), elle propose un partitionnement qui permet à l'analyse de contenu de découvrir une propriété particulière sur la première cellule. En tête de boucle, on a la partition $\{(\ell = 1); (2 \leq \ell \leq n \wedge \ell \leq i); \ldots\}$. Si l'on donne à l'analyse l'exemple d'initialisation de la Figure 7.3(a) (p. 182) suivi de cet exemple, elle découvre au dernier point de contrôle que :

$i := 1$ ;
**while** $i \leq n-1$ **do**
  $a[i+1] := 7$ ;
  $i := i+1$ ;

$$\begin{array}{ll} & 1 \leq i = n \\ \wedge & a[1] = b[1] + 1 \\ \wedge & \forall \ell,\ 2 \leq \ell \leq n \Rightarrow a[\ell] = 7. \end{array}$$

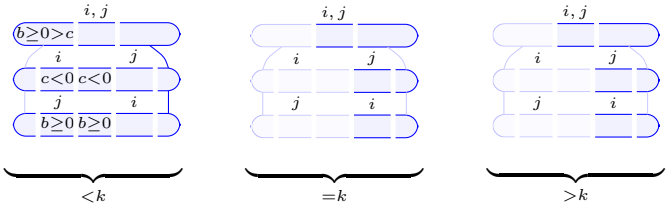
#### 7.2.2.2 Filtre partitionnant (Fig. 7.3(b))

Cet exemple, proposé dans la littérature, est le seul parmi ceux que nous avons testé à utiliser trois variables d'indices. C'est l'exemple qui génère le plus de tranches, les relations entre $i$, $j$ et $k$ étant faibles. À la sortie de la conditionnelle (l. **11**) on a $i \leq k+1$ et $j \leq k+1$. Ainsi, sur les vingt-sept partitions composant un partitionnement complet selon $i$, $j$, $k$ de l'espace $1 \leq \ell \leq n$, seulement cinq sont insatisfaisables en ce point de

contrôle : celles où $\ell > k$ et $\ell < i$ ou $\ell < j$. Malgré ces nombreuses tranches, l'analyse reste efficace. Elle découvre au dernier point de contrôle que :

$$\begin{array}{ll} & 1 \leq i, j \leq k \wedge 1 \leq k = n + 1 \\ \wedge & \forall \ell,\ 1 \leq \ell < i \Rightarrow b[\ell] \geq 0 \\ \wedge & \forall \ell,\ 1 \leq \ell < j \Rightarrow c[\ell] < 0. \end{array}$$

Ce n'est pas sous cette forme qu'apparaît réellement le résultat puisqu'il est éclaté dans des propriétés de tableaux plus nombreuses. Pour se repérer, on donne schématiquement ci-dessous les propriétés de tableaux de la valeur abstraite stable obtenue en tête de boucle :




Il n'y a rien de particulier à signaler sur le calcul effectué par l'analyse. Simplement, on peut se reporter à l'exemple développé à la fin du chapitre précédent (Sec. 6.7.1) pour comprendre comment les invariants sur les cellules $1 \leq \ell < i$ et $1 \leq \ell < j$ ont pu être découverts en parallèle. En effet, dans cet exemple la partition contenait deux indices et l'un était incrémenté alors que l'autre était fixé, une situation similaire aux incrémentations de $i$ et $j$ dans la conditionnelle du programme de filtre partitionnant.

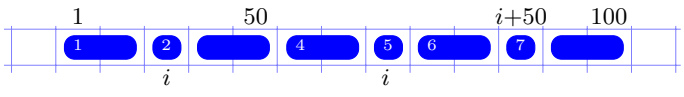
#### 7.2.2.3 Initialisation *copy propagation* (Fig. 7.3(c))

Cet exemple est tiré de [VJB+03] où il est utilisé pour introduire une méthode avancée de propagation des copies d'éléments de tableaux. Notre analyse découvre les invariants permettant de faire les mêmes transformations que celles opérées par cette méthode. Dans cet exemple, on voit que les copies des cinquante premières cellules du tableau $b$ dans le tableau $a$ peuvent être propagées, en transformant le programme en celui ci-contre. Pour pouvoir faire cette transformation, il faut prouver en tête de boucle que :

$$\begin{array}{l} \forall \ell,\ 1 \leq \ell \leq 50 \wedge \ell < i \Rightarrow c[\ell] = a[\ell] \\ \forall \ell,\ 51 \leq \ell \leq 100 \wedge \ell < i \Rightarrow c[\ell] = a[\ell] = b[\ell - 50]. \end{array}$$

```
i := 1 ;
while i ≤ 100 do
    if i ≤ 50 then
        c[i] := a[i]
    else
        c[i] := b[i − 50]
    i := i + 1
x := b[1]
```

De la même manière que dans le premier exemple discuté dans cette section, où la première cellule était différenciée des autres, la partition proposée par notre analyse différencie les cinquante premières cellules des cinquantes suivantes. Ces dernières sont partitionnées selon $i + 50$, alors que toutes les cellules sont partitionnées selon $i$. On a en tête de boucle la partition suivante, donnée schématiquement :



```
1  i := 1 ;
2  while i ≤ n do
3  |  a[i] := b[i] + 1 ;
4  |  i := i + 1
```

(a) Initialisation selon les valeurs d'autres tableaux

```
1  k := 1 ;
2  i := 1 ;
3  j := 1 ;
4  while k ≤ n do
5  |  if a[k] ≥ 0 then
6  |  |  b[i] := a[k] ;
7  |  |  i := i + 1
8  |  else
9  |  |  c[j] := a[k] ;
10 |  |  j := j + 1
11 |  k := k + 1
```

(b) Filtre partitionnant

```
1  i := 1 ;
2  while i ≤ 100 do
3  |  if i ≤ 50 then
4  |  |  a[i+50] := b[i]
5  |  c[i] := a[i] ;
6  |  i := i + 1
7  x := a[51]
```

(c) Initialisation *copy propagation*

```
1  a[1] := 7 ;
2  i := 2 ;
3  while i ≤ n do
4  |  a[i] := a[i − 1] + 1 ;
5  |  i := i + 1
```

(d) Initialisation avec une séquence de nombres

Figure 7.3 – Quatre algorithmes d'initialisation

Si on déroule l'analyse, on retrouve le phénomène que l'on avait vu avec l'analyse du programme d'initialisation pivot (Sec. 6.7.1). Le résultat du premier élargissement est ramené à un palier, ici $i = 50$. En effet, la propriété associée aux tranches $\varphi_4$ et $\varphi_5 = (51 \leq \ell \leq 100 \wedge \ell = i)$ est $\bot^{\mathbb{K}}$ et donc par contraposée $i$ est bornée par 50. À cette itération, on a donc $\eta \Rightarrow 1 \leq i \leq 50$ et l'élargissement a amené la propriété $(c^0 = a^0)$ pour la tranche $\varphi_1$ et $(a^0 = b^{-50})$ pour la tranche $\varphi_6$. Pour comprendre cette dernière propriété, il suffit de se rappeler que dans une tranche où $\ell = i + 50$ (comme la tranche $\varphi_7$, qui a reçu l'information apportée par l'affectation $a[i+50] := b[i]$), la cellule $a[i+50]$ est bien représentée par la variable de tranche $a^0$ et la cellule $b[i]$ par la variable de tranche $b^{-50}$ (Déf. 6.21, p. 163, avec $inexact_s^7 = \emptyset$, *i.e.* affectation de cellule où l'expression affectée peut être représentée exactement). Les itérations suivantes laissent inchangée la propriété $(c^0 = a^0)$ associée à la tranche $\varphi_1$. Elles transmettent la propriété $(a^0 = b^{-50})$ aux tranches $\varphi_4$ et $\varphi_5$. En même temps, l'affectation $c[i] := a[i]$ est prise en compte sur les cellules $\ell \geq 51$, amènant sur ces deux mêmes tranches que $(c^0 = a^0)$, et donc par déduction que $(c^0 = b^{-50})$. L'analyse a alors découvert les deux invariants évoqués ci-dessus.

C'est le premier exemple où l'on voit une variable de tranche $a^z$, avec $z \neq 0$, apparaître. Si on s'intéresse à l'opération de normalisation (Déf. 6.8, p. 142), on remarque que la présence de cette variable de tranche active la règle 2, qui va transférer de l'in-

formation au niveau de la tranche $\varphi_7$. En effet, dès la première itération, la propriété associée à cette tranche porte sur la variable de tranche $b^{-50}$. L'opération de normalisation teste alors, entre autres, si $\varphi_7 \oplus -50 \sqsubseteq^{\mathbb{Z}} \varphi_2$. Or, en suivant la Définition 6.7 (p. 136), on a $(51 \leq \ell = i + 50 \leq 100) \oplus -50 = (1 \leq \ell = i \leq 50) = \varphi_2$. De par cette égalité, la règle 2 est activée dans les deux sens : la propriété $\psi_7$ est modifiée par $\psi_7 \sqcap^{\mathbb{K}} (\psi_2 \boxplus 50) = (a^0 = b^{-50}) \sqcap^{\mathbb{K}} (c^{-50} = a^{-50})$ et la propriété $\psi_2$ est remplacée par $\psi_2 \sqcap^{\mathbb{K}} (\psi_7 \boxplus -50) = (a^0 = c^0) \sqcap^{\mathbb{K}} (a^{50} = b^0)$. Ainsi, l'analyse découvre en réalité les propriétés suivantes, équivalentes aux premières données, en tête de boucle :

$$\forall \ell,\ 1 \leq \ell \leq 50 \wedge \ell < i \Rightarrow c[\ell] = a[\ell] \wedge c[\ell + 50] = a[\ell + 50] = b[\ell]$$
$$\forall \ell,\ 51 \leq \ell \leq 100 \wedge \ell < i \Rightarrow c[\ell] = a[\ell] = b[\ell - 50] \wedge c[\ell - 50] = a[\ell - 50].$$

C'est un exemple où l'opération de normalisation a explicité des informations qui n'ont finalement permises aucune déduction ultérieure.

#### 7.2.2.4 Initialisation séquence (Fig. 7.3(d))

On retrouve pour cet exemple la partition différenciant la première cellule des autres. On rappelle qu'il n'y a pas de partitionnement selon $i - 1$, l'heuristique consistant pour une affectation de tableau à partitionner uniquement pour pouvoir représenter exactement le résultat de l'affectation (Déf. 6.11, p. 149).

Pour ce qui est de l'analyse du contenu de ce programme, elle a déjà été dévoilée lors de la présentation de l'opération de normalisation. L'Exemple 6.1 (p. 140) explique comment, à la première itération, la règle 2 de l'opération de normalisation permet de trouver que $a[2] = 8$. Puis, les autres itérations de l'analyse sont expliquées dans le paragraphe précédent la Définition 6.9 (p. 144) de l'opération de normalisation améliorée. Notamment, il est expliqué la découverte de la propriété $a[i] \geq 8$, qui n'est pas évidente.

L'analyse découvre au point de contrôle final l'invariant suivant :

$$\begin{array}{ll} & 1 \leq n = i - 1 \\ \wedge & a[1] = 7 \\ \wedge & \forall \ell,\ 2 \leq \ell \leq n \Rightarrow 8 \leq a[\ell] = a[\ell - 1] + 1. \end{array}$$

On peut regretter que l'analyse ne découvre pas que $\forall \ell,\ (2 \leq \ell \leq n \Rightarrow a[\ell] \leq n + 6)$. Ceci lui est impossible avec la séparation stricte entre variables d'indices et de contenu. Cette restriction, imposée pour la présentation de l'analyse, peut être levée. On peut alors considérer, lorsque $\mathbb{K} = \mathbb{Z}$, que toute variable est une variable de contenu. Par conséquent, certaines variables sont à la fois des variables d'indices et de contenu, comme ici $i$ et $n$. Pour ces variables, les deux fonctions de transfert correspondantes sont appliquées, d'abord celle pour les variables d'indice, puis celle pour les variables de contenu.

Pour le programme de la Figure 7.7(a) (p. 192), qui initialise simplement toutes les cellules par leur indice, on trouve alors en tête de boucle que $\eta = \mu = (1 \leq i \leq n+1)$ et que $\forall \ell,\ 1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] < i$. On déroule rapidement l'analyse pour se convaincre de ce résultat. À la première itération, on a avant l'incrémentation $(i = 1) \wedge \forall \ell, (1 \leq \ell = i \leq n \Rightarrow a[\ell] = i)$. Après l'incrémentation on a $(i = 2) \wedge \forall \ell, (1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] = i - 1)$. On entre dans la second itération : en tête de boucle on trouve après l'élargissement que $(1 \leq i) \wedge \forall \ell, (1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] = i - 1)$. Après l'incrémentation, on a $(2 \leq i) \wedge \forall \ell, (1 \leq \ell \leq n \wedge \ell < i \Rightarrow i - 2 \leq a[\ell] \leq i - 1)$. La troisième itération est stable : en effet on voit que l'on a en tête de boucle $(1 \leq i) \wedge \forall \ell, (1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] \leq i - 1)$

l'élargissement des propriétés $\psi$ ayant conservé seulement la contrainte $(a^0 \leq i-1)$ pour la tranche où $\ell < i$.

Bien entendu, l'idéal serait de découvrir aussi que $\forall \ell, (1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] = \ell)$. Nous aborderons brièvement cette question à la Section 7.2.4.

Si l'on revient à l'exemple qui nous intéresse, l'initialisation séquence, et que l'on applique l'analyse sans les restrictions précitées, on ne trouve toujours pas la borne $a[\ell] < n+6$. En effet, si on a bien $(\mu \Rightarrow n \geq 1)$, cette information n'est pas transmise aux propriétés de tableaux, ni par les fonctions de transfert, ni par la normalisation. En effet cette dernière opération ne transmet que des informations sur des scalaires déjà présents dans les propriétés $\psi$ (traitement préalable de l'opération de normalisation, Sec. 6.4.2). Évidemment on peut modifier ce choix, et propager $\mu$ systématiquement aux propriétés de tableau. Ce choix enleverait toute chance à l'analyse de passer à l'échelle, mais il permet de trouver l'invariant désiré : on a $a[1] = 7 \wedge 1 \leq n \Rightarrow a[1] \leq n+6$, etc.

Une solution intéressante serait de découvrir que $\forall \ell, (2 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] = \ell + 6)$, qui implique que $a[\ell] \leq n+6$. Les modifications de l'analyse évoquées à la Section 7.2.4, pour pouvoir propager la variable quantifiée $\ell$ dans les propriétés $\psi$, lui permettent de découvrir cet invariant.

On passe maintenant aux programmes recherchant certains éléments dans un tableau (Fig. 7.4, p. 186).

#### 7.2.2.5 Recherche du maximum (Fig. 7.4(a))

Il s'agit d'un exemple simple, où la propriété de tableau à découvrir est une propriété unaire. Encore une fois, ce programme différencie dans ses partitions la première cellule des autres. C'est le premier programme où l'on a l'affectation d'une variable scalaire de contenu, qui est d'ailleurs la seule difficulté posée à l'analyse de programme. La résolution de cette difficulté a été décrite de manière détaillée au paragraphe suivant la Définition 6.20 (p. 161) de cette opération d'affectation. Au final, l'analyse découvre l'invariant suivant au dernier point de contrôle :

$$\begin{array}{ll} & 1 \leq n = i - 1 \\ \wedge & a[1] \leq x \\ \wedge & \forall \ell,\ 2 \leq \ell \leq n \Rightarrow a[\ell] \leq x. \end{array}$$

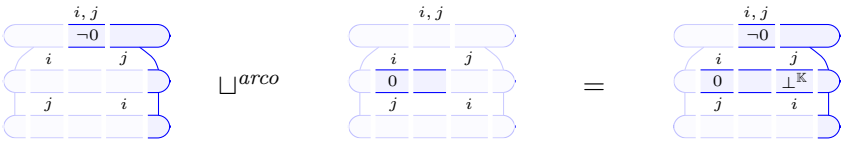
On remarque que cet invariant pourrait être exprimé sur la partition plus simple composée de la seule tranche $(1 \leq \ell \leq n)$. À l'inverse, cette simplification ne serait pas souhaitable pour l'invariant à la fin de l'analyse de l'exemple précédent, puisqu'il en serait affaibli : on conserverait seulement que le contenu de toutes les cellules est supérieur ou égal à 7. Une analyse pouvant modifier dynamiquement les partitions serait la plus à même de faire le choix d'une simplification ou non. Pour le cas présent, une heuristique simple de fusion de deux tranches est, bien entendu, qu'elles aient les mêmes propriétés $\psi$ associées.

#### 7.2.2.6 Premier non nul (Fig. 7.4(b))

Cet exemple, proposé dans [FQ02], est intéressant car il est typique de simples programmes de recherche d'un élément d'un tableau renvoyant l'indice auquel cet élément a été découvert.

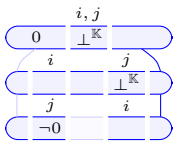
La partition choisie pour la boucle de ce programme est la même que celle choisie pour le programme d'initialisation pivot (Sec. 6.7.1). En effet, après la conditionnelle sur $a[i] \neq 0$ (l. **4**), l'espace $(1 \leq \ell \leq n)$ est partitionné selon $i$. Ainsi, pour l'affectation $j$ **:=** $i$, comme il existe une tranche $\varphi_p$ telle que $\varphi_p \Rightarrow \ell = i$ (l.**5**), l'application de la Définition 6.11 (p. 149) amène un partitionnement selon $j$. On rappelle que ce partitionnement se fait dans l'espace de l'expression affectée à $j$, *i.e.* $(1 \leq \ell \leq n)$ toujours.

Le fonctionnement de l'analyse du contenu est similaire à celle de l'initialisation pivot. On donne tout de même un aperçu de la première itération de l'analyse, avec ci-dessous les valeurs abstraites aux sorties des deux branches de la conditionnelle (dans l'ordre, la branche **then**, puis la branche **else**), et le résultat de leur union, à la première itération :




$(i = j = 1 \leq n)$ $(1 = i < j = n + 1)$ $(1 = i \leq j \leq n + 1)$

On remarque qu'il n'existe pas de contenu pour les cellules représentées par la tranche où $\ell = j > i$, et donc que $\neg(j > i \wedge 1 \leq j \wedge j \leq n)$. Pour autant, cette tranche n'est pas grisée car la conjonction de $\eta$ et de cette dernière propriété ne peut être représentée exactement par une zone, et donc $\eta$ n'implique pas l'insatisfaisabilité de cette tranche. La présence de cette propriété n'a qu'une conséquence anecdotique sur l'analyse.




On donne également ci-contre le résultat de l'opération d'incrémentation de $i$ sur la valeur abstraite résultante de l'union ci-dessus. On voit se dessiner l'invariant du programme. On remarque également que l'incrémentation de $i$ a eu pour effet de propager $\bot^{\mathbb{K}}$ à la tranche où $\ell = j = i$. En fait, et c'est là l'anecdote, l'analyse découvre en ce point de contrôle, qui est le dernier de la boucle, une propriété de non-égalité. Précisément, on a $\neg(1 \leq j = i \leq n)$, dont on peut conclure avec $\eta$ que si $j \leq n$ alors $i \neq j$ [5].

Mais ce qui est à regarder précisément dans l'analyse de ce programme, c'est le traitement de l'affectation $j$ **:=** $i$, qui a eu lieu avant les calculs présentés ci-dessus. Ce traitement a été détaillé dans le paragraphe suivant la Définition 6.18 (p. 158) des affectations non inversibles de variables d'indices. Notamment, on retrouve à la Figure 6.13 (p. 158) cinq des neufs tranches ci-dessus.

Au dernier point de contrôle, l'analyse découvre l'invariant suivant :

$$\begin{array}{ll} & 1 \leq j \leq i = n + 1 \\ \wedge & \forall \ell,\ 1 \leq \ell \leq n \wedge \ell < j \Rightarrow a[\ell] = 0 \\ \wedge & \forall \ell,\ 1 \leq \ell = j \leq n \Rightarrow a[\ell] \neq 0. \end{array}$$

Cet invariant indique à la fois que le tableau contient des zéros jusqu'à la cellule $j - 1$ et que si $j \leq n$ alors $a[j] \neq 0$. Ce qui est exactement ce que l'on souhaitait découvrir.

5. Ce résultat est évidemment au-delà des capacités de l'analyse des zones précisant alias seule (Chapitre 5).

```
1 x := a[1] ;
2 i := 2 ;
3 while i ≤ n do
4 |   if x < a[i] then
5 |   |   x := a[i]
6 |   i := i + 1
```

(a) Recherche du maximum

```
1 j := n + 1 ;
2 i := 1 ;
3 while i ≤ n do
4 |   if j = n + 1 and a[i] ≠ 0 then
5 |   |   j := i
6 |   i := i + 1
```

(b) Recherche de la première valeur non nulle

```
1 a[n] := x ;
2 i := 1 ;
3 while a[i] ≠ x do
4 |   i := i + 1
```

(c) Recherche d'une valeur avec une sentinelle

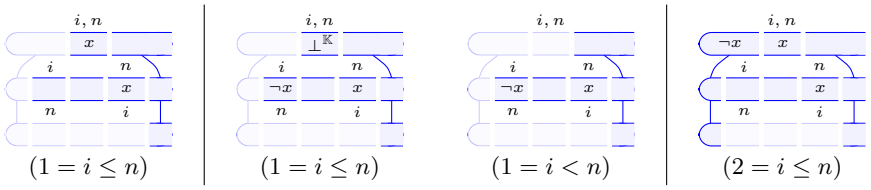
Figure 7.4 – Trois algorithmes de recherche d'information

#### 7.2.2.7 Sentinelle (Fig. 7.4(c))

Notre analyse découvre que ce programme n'accède pas au tableau $a$ hors de ses bornes, *i.e.* que la « sentinelle » $x$ est soit avant ou à la position $n$ dans ce tableau. C'est un exemple intéressant de propriété sur les indices impliquée par une propriété sur le contenu d'un tableau.

L'analyse de partitionnement de ce programme est directe. Après l'affectation de la sentinelle à $a[n]$, on a simplement la partition selon $n$ de l'espace ($\ell \geq 1$), puisque l'on suppose que $n \geq 1$. Dans la boucle, la conditionnelle sur $a[i]$ amène à partitionner selon $i$, et ce, toujours sur le même espace ($\ell \geq 1$) puisque la pré-analyse numérique a seulement découvert que $i \geq 1$. La partition obtenue, de neuf tranches donc, est donnée schématiquement ci-dessous.

Lorsque l'on déroule l'analyse de contenu, on voit dès la première itération comment l'invariant recherché en tête de boucle, *i.e.* $i \leq n$, est découvert. On donne ci-dessous quelques valeurs abstraites de cette itération. La première est celle en tête de boucle. Après l'application de la condition de la boucle, on trouve la seconde valeur abstraite (avant normalisation) et la troisième (après normalisation). Enfin, la troisième valeur abstraite est celle obtenue après l'incrémentation de $i$.




L'obtention des deux premières valeurs abstraites est directe. On note simplement dans la seconde l'apparition de la propriété de tableau $\forall \ell,\ (1 \leq \ell = i = n \Rightarrow \bot^{\mathbb{K}})$.

Comme nous l'avons déjà expliqué dans l'Exemple 6.2 (p. 141), qui porte sur ce même programme, l'opération de normalisation déduit alors que $i < n$. C'est ce que l'on voit dans la troisième valeur abstraite ci-dessus, où la tranche où $i = n$ est d'ailleurs grisée. Suit l'incrémentation de $i$ dont le résultat est la dernière valeur abstraite ci-dessus. On voit que la contrainte $i < n$, malgré l'élargissement futur, sera découverte en tête de boucle. En effet, l'invariant découvert par l'analyse, après quelques itérations supplémentaires, au dernier point de contrôle est le suivant :

$$\begin{array}{ll} & 1 \leq i \leq n \\ \wedge & a[i] = x \\ \wedge & a[n] = x \\ \wedge & \forall \ell,\ 1 \leq \ell < i \Rightarrow a[\ell] \neq x. \end{array}$$

On remarque que la partition n'a pas été simplifiée à la sortie de la boucle, la variable $n$ étant considérée comme vivante puisqu'il s'agit d'un paramètre du programme.

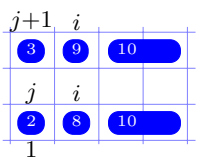
Les deux derniers exemples que nous avons analysé avec succès sont deux algorithmes de tri (Fig. 7.5, p. 189).

#### 7.2.2.8 Tri par insertion (Fig. 7.5(a))

Nous avons déjà discuté à la Section 6.5.1.1 l'analyse de partitionnement de cet exemple complexe. On donnait à la Figure 6.11(a) (p. 152) la partition en tête de boucle interne. Cette partition, formée de onze tranches, est simplifiée après l'opération d'affectation $a[j+1] := x$ (l. **8**), car c'est la dernière lecture de j avant l'écriture de la ligne **4**. Ainsi, la partition en tête de boucle externe est composée seulement de cinq tranches : $\{(\ell = 1 \wedge \ell < i); (2 \leq \ell \leq n \wedge \ell < i); (2 \leq \ell = i \leq n)\,;\ (2 \leq \ell \leq n \wedge \ell > i)\,;$ $(\ell \geq 2 \wedge \ell > i \wedge \ell > n)\}$. Cette partition est encore simplifiée lors de la sortie de la boucle externe, la variable $i$ mourrant. Au dernier point de contrôle, on a la partition $\{(\ell = 1); (2 \leq \ell \leq n); (\ell \geq 2 \wedge \ell > n)\}$.

Le résultat de l'analyse de contenu, non commenté, est donné en Annexe C. On évoque tout de même comment apparaissent des propriétés de tri $a[\ell - 1] \leq a[\ell]$ dans les propriétés de tableau et comment elles sont conservées malgré les modifications en place aux lignes **6** et **8**.

Pour leur apparition, on considère le calcul de l'analyse lors de la première itération sur la suite d'instruction ci-dessous (on ne prend donc pas toutes les traces en compte). Ci-contre, on a extrait de la représentation schématique de la partition donnée à la Figure 6.11(b) les tranches satisfaisables (sachant que $i = 2 \wedge ((j = 1) \vee (j = 0))$) aux différents points de contrôle correspondant à ces instructions.



$\{\psi_2 = (a^0 \geq x); \psi_8 = (a^0 = x); \psi_{10} = \top^{\mathbb{K}}\}$
$\mathbf{a[j+1] := a[j]}$ ;
$\{\psi_2 = (a^0 \geq x); \psi_8 = (a^0 = a^{-1}); \psi_{10} = \top^{\mathbb{K}}\}$
$\{\psi_2 = (a^0 = a^1 \geq x); \psi_8 = (a^0 = a^{-1} \geq x); \psi_{10} = \top^{\mathbb{K}}\}$ (normalisée)
$\mathbf{j := j - 1}$ ;
$\{\psi_3 = (a^0 = a^1 \geq x); \psi_9 = (a^0 = a^{-1} \geq x); \psi_{10} = \top^{\mathbb{K}}\}$
$\mathbf{a[j+1] := x}$ ;
$\{\psi_3 = (a^1 \geq x = a^0); \psi_9 = (a^0 \geq x = a^{-1}); \psi_{10} = \top^{\mathbb{K}}\}$

La première valeur abstraite est le résultat de l'affectation $x := a[i]$ (l. **3**) et de la conditionnelle $a[j] > x$ (l. **5**). L'affectation $a[j+1] := a[j]$ n'impacte que la tranche $\varphi_8$ et elle est traduite en l'affectation $a^0 := a^{-1}$ (Déf. 6.21, p. 163). On trouve donc la valeur abstraite donnée à la suite, et comme $\eta \sqcap^{\mathbb{Z}} (\varphi_8 \oplus -1) = \eta \sqcap^{\mathbb{Z}} \varphi_2$, on trouve après normalisation la valeur abstraite signalée comme normalisée. Avec la décrémentation de $j$, les propriétés sont propagées aux tranches $\varphi_3$ et $\varphi_9$ sans perte d'information. C'est avec l'affectation $a[j+1] := x$ que la propriété de tri apparaît. L'affectation est appliquée aux deux propriétés : $a^0 := x$ pour $\psi_3$ et $a^{-1} := x$ pour $\psi_9$. Ce sont les propriétés découvertes par la normalisation que l'on a mis en évidence qui permettent ici la déduction de la propriété $a^{-1} \leq a^0$ dans $\psi_9$, *i.e.* de la propriété $a[i-1] \leq a[i]$.

Quant à la conservation de ces propriétés de tri, nous l'avons déjà expliquée à la suite de la Définition 6.21 (p. 163), pour l'affectation $a[j+1] := a[j]$ (l. **6**) dans le cas général de l'algorithme (*i.e.* où $1 < j < i - 1$). Notamment, on a donné à la Figure 6.15 (p. 162), le calcul détaillé correspondant à ce cas. On donne ci-dessous le résultat final (donc avec tous les cas : $1 \leq j < i$) de l'analyse au même point de contrôle ①.

$$\begin{array}{rl} & 1 \leq j \leq i-1 \leq n-1 \\ \wedge & \text{si } j = 1 \text{ alors } (a[1] = a[2] > x) \wedge \forall \ell,\ (3 \leq \ell \leq i \Rightarrow x < a[\ell-1] \leq a[\ell]) \\ & \text{sinon } \forall \ell,\ (2 \leq \ell < j \Rightarrow a[\ell-1] \leq a[\ell]) \wedge (a[j] > x) \wedge (a[j+1] = a[j] \geq a[j-1]) \\ & \qquad \wedge \forall \ell,\ (j+2 \leq \ell \leq i \Rightarrow x < a[\ell-1] \leq a[\ell]). \end{array}$$

On termine en donnant l'invariant découvert automatiquement par notre analyse au dernier point de contrôle, qui est exactement l'invariant souhaité :

$$\begin{array}{rl} & 2 \leq i = n+1 \\ \wedge & \forall \ell,\ 2 \leq \ell \leq n \Rightarrow a[\ell-1] \leq a[\ell]. \end{array}$$

#### 7.2.2.9 Segmentation du tri rapide (Fig. 7.5(b))

Ce programme est une version du programme *partition*, qui est une sous-routine du tri rapide [Hoa61]. Il prend pour pivot $x$ du tableau $a$ le contenu de la première cellule et réorganise le contenu de ce tableau de telle manière que d'un côté soient tous les éléments plus petits que $x$ et de l'autre côté soient tous ceux plus grands ou égaux à $x$. La frontière entre ces deux parties est à l'indice $i - 1$, renvoyé par le programme.

Ce programme est le plus complexe que l'on ait analysé avec succès. On ne le détaille pas : les exemples précédemment présentés couvrent les difficultés que l'on rencontre durant son analyse. L'invariant découvert par notre analyse au dernier point de contrôle

```
1 i := 2 ;
2 while i ≤ n do
3 |   x := a[i] ;
4 |   j := i − 1 ;
5 |   while j ≥ 1 and a[j] > x do
6 |   |   a[j + 1] := a[j] ; ①
7 |   |   j := j − 1
8 |   a[j + 1] := x ;
9 |   i := i + 1
```

(a) Tri par insertion

```
1  x := a[1] ;
2  i := 2 ;
3  j := n ;
4  while i ≤ j do
5  |   if a[i] < x then
6  |   |   a[i − 1] := a[i] ;
7  |   |   i := i + 1
8  |   else
9  |   |   while j ≥ i and a[j] ≥ x do
10 |   |   |   j := j − 1
11 |   |   if j > i then
12 |   |   |   a[i − 1] := a[j] ;
13 |   |   |   a[j] := a[i] ;
14 |   |   |   i := i + 1 ;
15 |   |   |   j := j − 1
16 a[i − 1] := x ;
```

(b) Phase de segmentation du tri rapide

Figure 7.5 – Deux algorithmes de tri

est exactement celui que l'on attendait :

$$\begin{array}{ll} & 2 \leq i \leq n+1, j = i-1 \\ \wedge & \text{si } n = 1 \text{ alors } a[1] = x. \\ & \text{sinon } a[i-1] = x \wedge \forall \ell,\ (1 \leq \ell < i-1 \Rightarrow a[\ell] < x) \wedge \forall \ell,\ (i \leq \ell \leq n \Rightarrow x \leq a[\ell]) \end{array}$$

Si on s'intéresse au tri rapide (ci-contre), on peut simplement remarquer les déductions qui seraient nécessaires pour qu'une analyse interprocédurale, capable de travailler sur des procédures récursives (comme [Bou90, JS04]), trouve l'invariant de tri. Il faudrait pouvoir déduire que : $\forall \ell,\ (lb+1 \leq \ell \leq ub \Rightarrow a[\ell-1] \leq a[\ell])$, à partir des propriétés de tableaux suivantes :

$\forall \ell,\ (lb+1 \leq \ell \leq p-1 \Rightarrow a[\ell-1] \leq a[\ell] \leq x)$
$\forall \ell,\ (\ell = p \Rightarrow a[\ell] = x)$
$\forall \ell,\ (p+1 \leq \ell \leq ub-1 \Rightarrow x \leq a[\ell] \leq a[\ell+1])$

.

```
TriRapide(a, lb, ub) :
if lb < ub then
|   p = Segmentation(a, lb, ub) ;
|   TriRapide(a, lb, p − 1) ;
|   TriRapide(a, p + 1, ub) ;
```

### 7.2.3 Performances en temps

Si la plupart des exemples prennent moins d'une seconde pour être analysés, les temps d'analyse des exemples de tri sont insatisfaisants. De plus, la différence entre ces deux exemples de tri interroge. En fait, si on regarde les programmes listés par temps d'analyse croissant à la Figure 7.6(a), on voit, même parmi les exemples simples, de brusques

augmentations des temps d'analyse entre des exemples similaires, par exemple entre la copie de tableau et l'initialisation séquence, ou encore entre la recherche du maximum et le premier non nul.

| | $\lvert K\rvert \times \lvert\delta\rvert$ | $\lvert P\rvert$ | $\lvert\{a^z \mid z \neq 0\}\rvert$ *moy. (max.)* | *itér.* | *temps* |
|---|---|---|---|---|---|
| copie de tableau | $3 \times 3$ | 4 | 0 (0) | 5 | 0.02 |
| initialisation séquence | $3 \times 3$ | 5 | 0.8 (2) | 5 | 0.05 |
| recherche du maximum | $5 \times 6$ | 5 | 0 (0) | 5 | 0.10 |
| premier non nul | $6 \times 8$ | 10 | 0 (0) | 6 | 0.85 |
| tri par insertion | $9 \times 11$ | 5–11 | 4.6 (11) | 7 | 5.38 |
| segmentation tri rapide | $9 \times 13$ | 15 | 6.7 (14) | 6 | 22.87 |

(a) Taille de l'automate interprété $(K, \delta, k_0)$ représentant le programme, nombre de tranches dans les partitions en tête de boucle, nombres de « tranches translatées » rencontrées en moyenne et au maximum dans les valeurs abstraites en tête de boucle, nombre d'itérations (dont les itérations descendantes) effectuées par l'analyse, et enfin, le temps d'analyse

| | *normal.* | *temps (%)* | *temps moyen (normal. / (*$\lvert P\rvert + \lvert\{a^z \mid z \neq 0\}\rvert$*))* |
|---|---|---|---|
| copie de tableau | 30 | - | - |
| initialisation séquence | 32 | 54 | 0.00016 |
| recherche du maximum | 58 | 50 | 0.00017 |
| premier non nul | 89 | 50 | 0.00072 |
| tri par insertion | 169 | 85 | 0.00153 |
| segmentation tri rapide | 171 | 74 | 0.00453 |

(b) Opération de normalisation : nombre de normalisations effectuées durant l'analyse du programme, pourcentage du temps de l'analyse passé à normaliser, et moyenne des temps de calcul de l'opération de normalisation divisés par le nombre de tranches et de tranches translatées de la valeur abstraite normalisée

Figure 7.6 – Différentes métriques, sur une partie de nos exemples, pour comprendre les performances en temps de notre analyse

Pour expliquer ce comportement, on se tourne naturellement vers l'étude de l'opération de normalisation qui est la plus coûteuse. On a donc calculé quelques métriques sur cette opération pendant l'exécution des analyses, que l'on donne à la Figure 7.6(b). On apprend notamment que la plupart des analyses passent la moitié de leurs temps à normaliser une valeur abstraite. Ce ratio monte à 80% en moyenne pour les exemples complexes. On s'intéresse donc plus en détail à cette opération de normalisation.

Il est clair que la complexité de cette opération (Déf. 6.8, p. 142) dépend du nombre de tranches de la partition $P$ et du nombre de variables de tranches $a^z$ où $z \neq 0$ dans les propriétés $\psi_p$. On peut voir ces variables de tranches comme des références à des tranches que l'on appellera les « tranches translatées ». On note $T$ l'ensemble de ces tranches. Précisément la complexité de l'opération est dominée par les tests recherchant si les règles 1 et 2 sont activées. Pour la première règle, il est recherché l'ensemble des

couples $(p, p')$ tels que $\eta \sqcap^{\mathbb{Z}} \overline{\varphi_p} \sqsubseteq^{\mathbb{Z}} \overline{\varphi_{p'}}$. Si on note $\mathsf{C}(\sqsubseteq^{\mathbb{Z}})$ la complexité de l'opération $\sqsubseteq^{\mathbb{Z}}$, cette recherche a une complexité en $\mathcal{O}(|P|^2\mathsf{C}(\sqsubseteq^{\mathbb{Z}}))$. Pour la seconde règle, la recherche des couples $((p, z), (p', z'))$, où $z \in \mathit{Var}(\psi_p)$ et $z' \in \mathit{Var}(\psi'_p)$ tels que $\eta \sqcap^{\mathbb{Z}} \varphi_p \oplus z \sqsubseteq^{\mathbb{Z}} \varphi_{p'} \oplus z'$, a une complexité en $\mathcal{O}((|P||T| + |T|^2)\mathsf{C}(\sqsubseteq^{\mathbb{Z}}))$, les tranches $\varphi_p$ de la partition ne pouvant être incluses l'une dans l'autre (Déf. 6.2, p. 127). On a donc pour ces deux recherches réunies une complexité en $\mathcal{O}((|P| + |T|)^2\mathsf{C}(\sqsubseteq^{\mathbb{Z}}))$.

Si on regarde alors expérimentalement la durée d'une normalisation par tranche (*i.e.* la durée d'une normalisation d'une valeur abstraite divisée par $|P| + |T|$), donnée à la dernière colonne de la Figure 7.6(b), et le nombre de tranches et de tranches translatées donnés à la Figure 7.6(a), on remarque que l'on a pas la croissance linéaire à laquelle on pourrait s'attendre au vu des complexités que l'on vient de donner. Par exemple, pour le tri par insertion et la segmentation du tri rapide, on a respectivement 15.6 et 21.7 tranches $(|P| + |T|)$ en moyenne dans les valeurs abstraites, alors que l'on a un coût moyen de normalisation par tranche de 0.00153 et 0.00453 seconde respectivement.

Ce comportement est à imputer à l'implantation d'Enkidu. Plusieurs tables sont utilisées lors du traitement de l'opération de normalisation, de taille $|P|$ ou $|T|$. Ces tables sont implantées par des arbres binaires équilibrés, pour lesquels la recherche et l'insertion d'un élément ont une complexité logarithmique dans le nombre d'éléments. En fait, l'implantation de nos règles 1 et 2 est basée sur une opération ayant une complexité en $\mathcal{O}((|P| + |T|)^2(4\ \log_2(|P|) + 2\ \log_2(|T|))\mathsf{C}(\sqsubseteq^{\mathbb{Z}}))$, qui explique les temps de normalisation croissants brusquement avec le nombre de tranches et de tranches translatées.

On constate ici ce que nous avions dit lors de la présentation d'Enkidu : nous n'avons pas assez travaillé sur ses performances. En fait, il suffirait d'utiliser des structures de données adéquates, dont des tables de hachage, pour améliorer très significativement les performances d'Enkidu. Cela dit, on a vu que la complexité de l'opération de normalisation avait un facteur quadratique dans la taille de la partition et dans le nombre de variables de tranches $a^z$ où $z \neq 0$. On doit donc s'attendre à ce que des écarts, importants par conséquent, perdurent entre les temps d'analyses de programmes ayant un nombre de tranches sensiblement différent.

Dans l'optique d'améliorer le passage à l'échelle de l'analyse, on notera qu'une représentation arborescente des partitions serait la bienvenue. Rien que pour l'opération de normalisation, elle permettrait d'éviter de nombreux tests.

### 7.2.4 À la lisière des capacités de l'analyse

On discute ici trois exemples qui peuvent être analysés en modifiant simplement certaines opérations de notre analyse, ou en en ajoutant.

#### 7.2.4.1 Initialisation à l'indice (Fig. 7.7(a))

L'analyse proposée dans [SPW09] (Sec. 4.2.2.3) « découvre » (l'objectif de preuve est fourni) l'invariant de boucle exact de ce programme en un peu moins d'une seconde.

Comme nous l'avons vu lorsque nous avons discuté l'exemple de l'initialisation séquence, notre analyse est moins précise (même en ayant considéré $i$ comme une variable d'indice et de contenu). C'est dommage, car on sent qu'il ne s'agit que d'un problème d'expressivité simple et que notre analyse serait tout aussi efficace sur ce programme que sur une simple copie de tableau.

En fait, lorsque $\mathbb{K} = \mathbb{Z}$, rien n'empêche d'utiliser la variable quantifiée $\ell$ dans les propriétés $\psi_p$. Bien sûr, cela demande des adaptations. Les opérations ensemblistes restent telles quelles mais certaines opérations sémantiques doivent être modifiées. Par exemple, pour les règles 1 et 4 de l'opération de normalisation (Déf. 6.8, p. 142), qui propagent des informations sur les variables scalaires de contenu entre les $\psi_p$ et $\mu$, aucune information sur $\ell$ ne doit être propagée. En fait, il suffit de redéfinir $\overline{\psi_p}$ par $\exists(S(A) \cup \{\ell\}).\psi_p$. Pour que la règle 2 de l'opération de normalisation soit correcte, l'opérateur $\boxplus$ de translation sur les propriétés $\psi_p$ (Déf. 6.7, p. 136) doit être remplacé par $\psi \boxplus z = \psi[a^{y-z}/a^y][\ell - z/\ell]$. En effet, si $1 \leq \ell \leq 2 \Rightarrow a[\ell] = \ell$ alors $5 \leq \ell \leq 6 \Rightarrow a[\ell - 4] = \ell - 4$. Enfin, la dernière opération qui doit être modifiée est celle d'affectation d'une cellule de tableau. Ici, ce n'est pas la correction mais la précision que l'on doit considérer. Par exemple, est-ce que les affectations comme $a[i] := i + 1$, doivent être simplement traduites par $a^0 := \ell + 1$ ? Si tel est le cas, on trouve l'invariant du programme que l'on considère ici, mais pas celui que l'on souhaitait découvrir pour l'initialisation séquence ($a[\ell] = \ell + 6$). Il faudrait qu'après l'instruction $a[1] = 7$, l'analyse déduise que $1 = \ell \Rightarrow a[\ell] = 7 = \ell + 6$. En fait, une solution simple consiste à propager la propriété $\varphi_p$ à $\psi_p$ pour toute tranche modifiée par une opération d'affectation de cellule de tableau. On trouve alors les invariants les plus précis de ces deux programmes.

```
i := 1 ;
while i ≤ n do
    a[i] := i ;
    i := i + 1
```

(a) Initialisation à l'indice

```
i := 1 ;
while i < n do
    if ... then return ;
    a[i] := j + 3 ;
    a[i + 1] := j + 2 ;
    i := i + 2
```

(b) Extrait (adapté) de la fonction `ixj_daa_cid_read` (linux driver)

```
i := 1 ;
x := 5 ;
while i ≤ n do
    if i = 6 then x := 10 ;
    a[i] := x ;
    i := i + 1
```

(c) Initialisation deux blocs

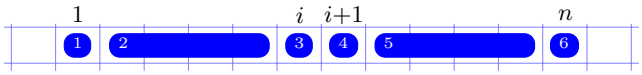
Figure 7.7 – Trois algorithmes dont l'analyse serait concluante

#### 7.2.4.2 Initialisation modulo deux (Fig. 7.7(b))

Cet exemple a été proposé dans [SPW09] également. Leur analyse trouve, en quelques secondes, que toutes les cellules paires sont égales à $j + 2$ et toutes les cellules impaires sont égales à $j + 3$.

Notre analyse n'est pas totalement imprécise, malgré l'incrémentation d'un pas de deux utilisée. En effet, elle découvre au dernier point de contrôle que $\forall \ell,\ (2 \leq \ell < n \Rightarrow j + 2 \leq a[\ell] \leq j + 3)$ et que si $i \geq 2$ alors $a[1] = j + 3$.

Si on regarde l'analyse de partitionnement, la partition obtenue pour la boucle est le résultat des partitionnements de l'espace $(1 \leq \ell < n)$ selon $i$ et de l'espace $(2 \leq \ell \leq n)$ selon $i+1$. Ainsi, cette partition différencie les cellules 1 et $n$. On a précisément la partition $\{(1 = \ell < n \wedge \ell < i); (1 = \ell < n \wedge \ell = i); (2 \leq \ell < n \wedge \ell < i)\ ; (2 \leq \ell < n \wedge \ell = i)\ ; (2 \leq \ell < n \wedge \ell > i); (n = \ell \geq 1 \wedge \ell < i + 1); (n = \ell \geq 1 \wedge \ell = i + 1)\ ; (n = \ell \geq 1 \wedge \ell > i + 1); (\ell \geq 2 \wedge \ell > n)\}$. Cette partition est schématisée ci-dessous pour le cas où $1 < i < i + 1 < n$.



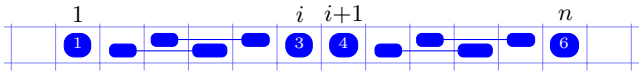

On ne déroule pas l'analyse de contenu. Remarquons que les cellules que représente $\varphi_2$ après l'affectation de $i$ (Déf. 6.19, p.159) sont $\{\!|i := i-2|\!\}^{\mathbb{Z}}(\varphi_2) = (2 \leq \ell \leq n \wedge \ell \leq i+1)$. Or, les affectations de la boucle couvrant à la fois la cellule $i$ et $i+1$, il n'y a pas de cellule pour laquelle on ait aucune connaissance avant l'affectation de $i$. On a $\psi_3 = (a^0 = j+3)$ et $\psi_4 = (a^0 = j+2)$. Si on suppose que l'on avait $\psi_2 = (j+2 \leq a^0 \leq j+3)$, l'union de ces trois propriétés laisse $\psi_2$ identique après l'affectation $i := i+2$.

Cependant, si on considère le même programme mais avec les affectations $a[i] := x\,;$ $a[i+1] := y$, notre analyse n'aurait découvert aucune propriété de tableau. En fait, notre analyse peut tout à fait être précise. Il suffit de prendre $\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{zone} \times \mathcal{D}^{mod}$, où $\mathcal{D}^{mod}$ est le domaine abstrait des congruences simples (Fig. 3.2(c), p. 31), et de définir pour ce produit réduit la fonction *ContextSplit* adéquate pour l'analyse de partitionnement.

La pré-analyse numérique découvre que $i = 1[2]$ dans la boucle. Pour l'affectation à $a[i]$ l'espace qui doit être partitionné selon $i$ est donc $(1 \leq \ell < n \wedge \ell = 1[2])$. Ainsi, pour renvoyer une partition couvrante, on définit la fonction *ContextSplit* de telle manière qu'elle propose dans l'ensemble *out* (Sec. 6.5.1.1) la tranche $(1 \leq \ell < n \wedge \ell = 0[2])$. Cette fonction applique le même mécanisme pour l'expression $i+1$. On a :

$$\begin{array}{lcl}
\mathit{ContextSplit}(i)^{zone\times mod} & = & (\{(1 \leq \ell < n \wedge \ell = 1[2])\}^{in}, \\
 & & \{(1 \leq \ell < n \wedge \ell = 0[2]); (\ell < 1); \\
 & & (\ell \geq n \wedge \ell \geq 1\}^{out}). \\
\mathit{ContextSplit}(i+1)^{zone\times mod} & = & (\{(2 \leq \ell \leq n \wedge \ell = 0[2])\}^{in}, \\
 & & \{(2 \leq \ell \leq n \wedge \ell = 1[2]); (\ell \leq 1); \\
 & & (\ell > n \wedge \ell \geq 2\}^{out}).
\end{array}$$

Ainsi, l'analyse de partitionnement propose la partition suivante pour la boucle de ce programme.




Quant à l'analyse de contenu, aucune modification des opérations sémantiques n'est nécessaire. L'analyse découvre alors, au dernier point de contrôle, l'invariant suivant :

$$\begin{array}{ll}
 & 1 \leq i \leq n+1 \\
\wedge & \forall \ell,\ (2 \leq \ell < n \wedge \ell = 0[2] \Rightarrow a[\ell] = j+2) \\
\wedge & \forall \ell,\ (2 \leq \ell < n \wedge \ell = 1[2] \Rightarrow a[\ell] = j+3) \\
\wedge & \text{si } i \geq 2 \text{ alors } a[1] = j+3 \text{et si } n = 0[2] \text{ alors } a[n] = j+2.
\end{array}$$

En pratique, l'utilisation du domaine des congruences simples n'est pas souhaitable : elle amène l'analyse de partitionnement à générer des partitions dont la taille dépend directement d'une constante du programme. En effet, si un programme initialise uniquement les cellules multiples de 20, la fonction *ContextSplit* renverra une partition avec plus d'une vingtaine de tranches ($\ell = 0[20], \ell = 1[20]$, etc). Ainsi, il faut préférer l'utilisation d'un domaine abstrait capable d'exprimer que $\ell \in [1, 19][20]$, comme par exemple le domaine des congruences trapézoïdales (Fig. 3.2(c)).

#### 7.2.4.3 Initialisation deux blocs (Fig. 7.7(c))

Il s'agit d'un exemple très simple pour rappeler que les choix de partitionnement sont toujours imparfaits.

Ici, le programme que l'on prend en exemple et celui ci-contre initialisent identiquement le tableau $a$. Cependant, si l'analyse de partitionnement propose pour le programme ci-contre une partition distinguant les cellules 1 à 5 des cellules 6 à $n$, ce n'est pas le cas pour le programme qui nous intéresse. Notre analyse découvre seulement au dernier point de contrôle que $\forall \ell,\ (1 \leq \ell \leq n \Rightarrow 5 \leq a[\ell] \leq 10)$.

$i := 1$ ;
$x := 5$ ;
**while** $i \leq n$ **do**
  **if** $i \leq 5$ **then** $a[i] := 5$ ;
  **else** $a[i] := 10$ ;
  $i := i + 1$

Une solution simple pour avoir une précision identique sur les valeurs initiales de $a$ pour ces deux programmes consiste à ajouter le cas suivant à la Définition 6.11 (p. 149). Si une conditionnelle contraint une variable scalaire utilisée pour définir certaines tranches de la partition, alors on partitionne selon cette contrainte. On remarque que sur ce programme, l'analyse de partitionnement ne convergerait pas à la première itération. En effet, lors de la première itération, le partitionnement selon $i$ n'a pas encore eu lieu lorsque la conditionnelle sur $i$ est traitée. Ce n'est donc qu'à la seconde itération que la partition sera intersectée avec la partition $\mathit{Split}(6)$.

Cette simple modification permettrait de découvrir l'invariant souhaité. Cependant, elle augmenterait significativement le coût général de l'analyse de partitionnement. On pense aux conditionnelles $i \leq n$ que l'on trouve dans la plupart de nos programmes qui devraient être traitées (intersection avec la partition $\{(\ell \leq n); (\ell > n)\}$), et qui en pratique modifieraient rarement les partitions.

### 7.2.5 Les limites de l'analyse

Bien sûr, notre analyse échoue à analyser précisément de nombreux programmes. Parmi les programmes proposés dans les travaux existants, nous avons déjà expliqué que l'expressivité de notre domaine abstrait était insuffisante pour les programmes de filtre par indirection (Fig. 7.8(a)) et de tri à bulle (Fig. 7.8(b)).

**Propriété d'ordre : découverte et partitionnement** Pour le tri par sélection (Fig. B.3(b), p. 205), la raison de l'échec de notre analyse à découvrir l'invariant de tri est plus complexe. Le tri par sélection fonctionne par recherche itérative de l'élément minimum de la partie non-triée du tableau, qu'il ajoute à la partie triée. Notre analyse trouve aisément à la sortie de la boucle externe, que $x$ est l'élément minimum des cellules $i+1$ à $n$ et qu'il est égal à $a[k]$. Dès lors, avant l'incrémentation de $i$ (dernière ligne du programme), on sait que $\forall \ell,\ (1 \leq \ell \leq n \wedge i < \ell \Rightarrow a[\ell] \geq x)$ et que $a[i] = x$. Si on regarde la première itération, on a après l'incrémentation que $\forall \ell,\ (1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] = x)$ et $a[i] \geq x$. Autrement dit, comme $i = 2$, on a $a[1] = x \leq a[2]$ si $n \geq 2$. Cependant, notre analyse ne découvre pas que $a[i-1] \leq a[i]$ car elle ne cherche pas à déduire des relations entre deux tranches distantes d'une constante (ici 1) s'il n'existe pas une variable de tranches $a^z$ l'invitant à le faire. C'est ainsi qu'est définie la règle 2 de notre opération de normalisation (Déf. 6.8, p. 142). Il est aisé – mais coûteux, c'est justement ce que nous voulions éviter –

de chercher les $z \in \mathbb{Z}, \varphi_p, \varphi_{p'} \in P$ tels que $\varphi_p \oplus z \sqsubseteq^{\mathbb{Z}} \varphi_{p'}$ et d'appliquer la règle 2. Si on note $\varphi_1 = (1 \leq \ell \leq n \wedge \ell < i)$ et $\varphi_2 = (1 \leq \ell = i \leq n)$, on a $\varphi_2 \oplus -1 \sqsubseteq^{\mathbb{Z}} \varphi_1$ et donc on aurait trouvé que $\forall \ell,\ (1 \leq \ell = i \leq n \Rightarrow a[\ell] \geq a[\ell - 1] = x)$. Par contre, $\varphi_1 \oplus 1 \not\sqsubseteq^{\mathbb{Z}} \varphi_2$. En effet, il aurait été faux de déduire que $\forall \ell,\ (1 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell] \leq a[\ell + 1])$, puisque l'on a pas d'information sur la cellule $a[\ell + 1]$ si $n = 1$, et que $\overline{\varphi_1} \not\Rightarrow n \neq 1$.

On remarque alors le problème qui empêche vraiment notre analyse de trouver l'invariant de tri. La partition ne permet pas d'exprimer cet invariant : il faudrait que la première cellule (ou la $n^{\text{e}}$) soit distinguée des autres pour pouvoir exprimer que $\forall \ell,\ (2 \leq \ell \leq n \wedge \ell < i \Rightarrow a[\ell - 1] \leq a[\ell])$ (ou que $\forall \ell,\ (1 \leq \ell < n \wedge \ell < i \Rightarrow a[\ell] \leq a[\ell + 1])$). Il n'est pas exclu d'adapter l'analyse de partitionnement (notamment on voit que $j \in [2, n+1]$ en tête de boucle interne), mais il vaut mieux admettre que l'on touche ici aux limites de notre analyse.

Les exemples évoqués ci-dessus respectent notre langage d'entrée (présenté en Préliminaires, p. 75). Ce n'est pas le cas des deux programmes suivants, la copie par blocs (Fig. 7.8(c)), qui accède à une cellule à l'indice $i + n$, et la recherche dichotomique (Fig. 7.8(d)), qui affecte l'expression $(i + j) \div 2$ à une variable d'indice. Il n'y a donc rien de surprenant à ce que notre analyse soit imprécise pour ces programmes. Ils restent néanmoins des exemples plutôt simples, délimitant les capacités de notre analyse.

```
i := 1 ;
j := 1 ;
while i ≤ n do
    if a[i] ≥ 0 then
        b[j] := i ;
        j := j + 1
    i := i + 1
```

(a) Filtre par indirection

```
i := 1 ;
while i ≤ n do
    j := n ;
    while j > i do
        if a[j] < a[j + 1] then
            x := a[j] ;
            a[j] := a[j + 1] ;
            a[j + 1] := x
        j := j − 1
    i := i + 1
```

(b) Tri à bulle

```
i := 1 ;
while i ≤ n do
    a[i] := b[i] ;
    a[i + n] := c[i] ;
    i := i + 1
```

(c) Copie par blocs

```
i := 1 ; j := n ;
while i ≤ j do
    k = (i + j) ÷ 2 ;
    if a[k] < x then
        i := k + 1
    else
        if a[k] > x then
            j := k − 1
        else
            return Some(k)
return None
```

(d) Recherche dichotomique

Figure 7.8 – Quatre algorithmes dont l'analyse n'est pas concluante

**Expressivité des propriétés $\psi_p$** Le programme de copie par blocs concatène deux tableaux $b$ et $c$ de taille $n$. L'analyse de partitionnement ne pose pas de problème. En utilisant $\mathcal{D}^{\mathbb{Z}} = \mathcal{D}^{poly}$ le domaine des polyèdres (Fig. 3.2(a)), on aura la partition suivante en tête de boucle : $\{(1 \leq \ell \leq n) \sqcup^{arpa} \mathit{Split}(i)\} \cup \{(n+1 \leq \ell \leq 2n) \sqcup^{arpa} \mathit{Split}(i+n)\} \cup \{\ell > 2n \wedge \ldots\}$. En revanche, notre domaine abstrait ne permet pas d'exprimer la partie droite de la propriété de tableau à découvrir : $\forall \ell,\ (n+1 \leq \ell \leq 2n \wedge \ell < i+n \Rightarrow a[\ell] = c[\ell - n])$.

Pour les mêmes raisons, le renversement de tableau, dont une implantation est donnée ci-contre, est un autre exemple pour lequel notre analyse sera imprécise.

```
i := n ;
while i ≥ 1 do
    b[i] := a[n − i + 1] ;
    i := i − 1
```

**Propriété d'ordre et déduction** L'analyse du programme de recherche dichotomique suppose que l'on a pour état initial le tableau $a$ trié. Ainsi, si on s'intéresse aux trois tranches $\{(i \leq \ell \leq j) \sqcup^{arpa} \mathit{Split}(k)\}$, et si l'on suppose que l'on a trouvé les partition adéquates, on a après la conditionnelle $a[k] < x$, les propriétés de tableaux suivantes :

$$\begin{array}{l} \forall \ell,\ (i \leq \ell \leq j \wedge \ell < k \Rightarrow a[\ell] \leq a[\ell+1]) \\ \forall \ell,\ (i \leq \ell = k \leq j \Rightarrow a[\ell] < x) \\ \forall \ell,\ (i \leq \ell \leq j \wedge \ell > k \Rightarrow a[\ell-1] \leq a[\ell]). \end{array}$$

Pour trouver l'invariant du programme, il est nécessaire de déduire des deux premières propriétés ci-dessus que $\forall \ell,\ (i \leq \ell \leq j \wedge \ell < k \Rightarrow a[\ell] \leq a[\ell+1] < x)$, ce que l'on ne sait pas faire de par la sémantique cellule-par-cellule de nos propriétés $\psi_p$. En effet, ici on sait que le dernier élément de la tranche triée $[i, k]$ est borné par $x$, mais il nous manque l'argument de transitivité pour déduire que tous les éléments de cette tranche sont bornés par $x$.

## 7.3 Conclusion

**Génération de certificats** On donne ici directement les résultats de travaux effectués dans une autre équipe de notre laboratoire, portant sur la génération automatique de certificats pour les résultats d'Enkidu. L'approche suivie par ces travaux [GPP10] [6], est d'instrumenter directement l'analyseur pour produire les certificats.

Cette technique a été validée expérimentalement sur un sous-ensemble de nos exemples (Fig. 7.9). Le temps total d'exécution d'Enkidu (*i.e.*, analyse [7] et génération du certificat) est augmenté en moyenne de 35%. On remarque surtout que des certificats sont générés même pour les programmes complexes de tri.

**Conclusion** Enkidu n'est véritablement qu'un prototype, mais il aura permis d'expérimenter plusieurs travaux de recherche au sein du laboratoire Verimag. Ses faiblesses nous ont empêchés de faire une validation expérimentale complète de notre analyse du contenu des tableaux. En effet, il aurait été très intéressant d'évaluer le comportement de

6. Cet article, que nous soumettrons au courant du mois d'août, est accessible à l'adresse Internet suivante : `http://www-verimag.imag.fr/~peron/docs/research/papers/iwil10.pdf`.

7. Les temps d'analyse donnés ici sont bien plus longs que ceux donnés à la Figure 7.2 (p. 179). En fait, le prototype utilisé par nos collègues est une version ancienne d'Enkidu, dont les performances ont été améliorées depuis.

| | *analyse* | ⟵ *preuve* ⟶ | | |
|---|---|---|---|---|
| | | *génération* | *taille (Ko)* | *vérification* |
| copie de tableau | 0.042 | 0.014 | 9.2 | 0.97 |
| recherche du maximum | 0.284 | 0.028 | 15 | 1.03 |
| sentinelle | 0.461 | 0.21 | 61 | 2.67 |
| premier non nul | 4.782 | 2.58 | 100 | 6.54 |
| tri par insertion | 6.047 | 2.75 | 145 | 13.54 |
| segmentation tri rapide | 31.697 | 5.92 | 548 | 41.22 |

Figure 7.9 – Génération automatique de certificats : temps, en secondes, nécessaires à Enkidu pour générer les certificats et à Coq pour les vérifier

notre analyse sur de vraies études de cas, où des tableaux seraient utilisés plus ou moins intensivement, en des endroits épars. Si c'est ce que j'ai commencé lors de mon stage à *Microsoft Research*, je n'ai pas eu le temps d'étudier la précision des résultats obtenus.

Si on en reste aux résultats des programmes unitaires présentés dans ce chapitre, il était évident pour la communauté scientifique qui s'était intéressée à l'analyse du contenu des tableaux, que de découvrir précisément les invariants de tous ces programmes automatiquement était un vrai défi. Notre analyse semble l'avoir relevé.

# Chapitre 8

# Conclusion

Nous avons déjà présenté de manière factuelle notre contribution à l'analyse statique de programmes manipulant des tableaux en introduction (Sec. 1.4). On revient plutôt sur notre démarche face aux difficultés posées par l'essence de cette structure de données. De par l'adressage numérique des éléments qu'elle contient, le phénomène d'alias associé, et la possibilité de modifier directement tout élément en place, les invariants qui apparaissent sur des parties de son contenu dépendent de propriétés numériques complexes.

Pour découvrir de tels invariants, nous avons naturellement utilisé la théorie de l'interprétation abstraite, qui propose une méthode générale pour l'analyse approchée de programmes. En effet, elle permet de surmonter l'indécidabilité du problème correspondant et surtout de maîtriser la complexité de la solution que l'on cherche à apporter.

Nous avons pensé que pour atteindre notre objectif nous devions étudier deux sujets.

1. Il nous semblait nécessaire, pour qu'une analyse du contenu du tableau soit précise, d'avoir une analyse numérique capable de découvrir des propriétés de non-alias entre variables d'indices. Dès lors que l'on cherche à exprimer dans un domaine abstrait des inégalités et même les non-égalités relationnelles les plus simples, il est très difficile de discerner les propriétés qui peuvent être déduites de la conjonction de ces non-égalités et inégalités. Nous avons donc concentré notre effort sur l'opération de normalisation. En soi, les résultats obtenus sur la forme normale du cas dense sont intéressants, tout comme l'approximation du cas discret qu'ils permettent de construire. Mais ces résultats n'ont pas de conséquences pratiques pour notre objectif, parce que les programmes où un invariant du contenu des tableaux repose sur la non-égalité de deux variables d'indice, sont simplement très rares. En pratique, ce sont des invariants numériques plus forts qui apparaissent entre les variables d'indice des programmes manipulant des tableaux.
2. Nous sommes donc revenus à notre objectif. Lorsqu'on cherche des invariants du contenu des tableaux, deux tâches se distinguent clairement : construire les sous-parties symboliques des tableaux où il existe des invariants et construire les propriétés invariantes vérifiées par ces sous-parties. On peut imaginer effectuer ces tâches de manière concomitante, l'une influençant l'autre. Cette construction dynamique de la partition symbolique nous paraissait risquée au vu des résultats obtenus par la communauté scientifique jusqu'alors, montrant des analyses imprécises et inefficaces lorsqu'elles visaient la découverte d'une large classe de propriétés de tableaux. Nous avons donc choisi un partitionnement statique qui, en restreignant

l'expressivité de l'analyse du contenu proprement dite, permet de construire plus aisément une analyse efficace et précise. Un autre atout du partitionnement statique est de faciliter les raisonnements nécessaires à la conception de l'analyse du contenu. Ce n'est d'ailleurs que bien après avoir défini cette dernière que nous avons défini notre analyse de partitionnement. Dans [GRS05], cette analyse n'était présentée qu'à l'état de concept. Ici, nous l'avons formalisée, définie complètement, et enfin rendue plus sémantique en utilisant les résultats d'une pré-analyse numérique. Bien entendu, cette solution n'est pas idéale. Notamment, il s'agit d'une approximation importante hors du domaine abstrait, qui dépend fortement de la représentation intermédiaire des programmes utilisée par l'analyseur. Ceci implique un problème de maintenance et le plus souvent des difficultés importantes : par exemple, si une variable temporaire est systématiquement introduite pour les affectations inversibles présentes dans les boucles, l'incrémentation $i := i + 1$ du code source étant transformée en l'affectation $j := i + 1$ et la temporaire $j$ étant affectée à $i$ plus loin. Une analyse de partitionnement statique pourrait être fortement perturbée par de telles modifications.

Que l'on utilise un partitionnement statique ou non, une abstraction doit être choisie pour les propriétés du contenu associées aux tranches et donc les questions classiques d'expressivité et de pouvoir de déduction se posent. Ces questions sont d'autant plus difficiles qu'il s'agit ici de propriétés quantifiées pour lesquelles on a des interactions non triviales entre les propriétés sur les variables d'indices et les propriétés sur les variables de contenu (scalaires ou tableaux). Nous nous sommes ici inspirés des tableaux du langage de flot de données synchrone Lustre [CPHP87], où l'on peut déclarer des équations « bien typées » sur des tranches de tableaux. Par exemple, on peut écrire l'équation `a[1..5] = b[3..7] + 1^5` indiquant pour tout $\ell \in [1, 5]$ que $a[\ell] = b[\ell + 2] + 1$. C'est ainsi que nous avons restreint notre expressivité en imposant une seule variable quantifiée par propriété de tableaux et en permettant d'exprimer seulement des relations cellules-par-cellules entre des tranches translatées d'une constante. Cette idée est à rapprocher de certains prédicats que l'on trouve dans les travaux de [LRSW00, Cou03], mais elle est ici généralisée.

Nous avons alors mené à bien le travail conséquent pour définir l'opération de normalisation et les opérations sémantiques. Au final, l'analyse du contenu des tableaux que nous proposons est la plus aboutie des analyses proposées jusqu'à présent et obtient de loin les meilleurs résultats.

Dans cette thèse, nous avons clarifié l'état de l'art des analyses statiques du contenu des tableaux. Notre analyse apporte une avancée significative pour ce sujet et ses principes peuvent être étendus pour définir d'autres analyses automatiques, encore plus puissantes et efficaces. Ces travaux ont d'ailleurs ouvert de nombreuses perspectives que nous allons brièvement mentionner.

Ils ont aussi motivé des travaux complémentaires au sein de notre laboratoire. Nous avons déjà évoqué ces derniers : il s'agit d'une analyse automatique de permutation du contenu d'un tableau [PH10] et de la génération automatique de certificats des résultats d'Enkidu [GPP10]. Simplement, on remarque que l'alliance de ces travaux de recherche permet par exemple à Enkidu de découvrir automatiquement l'invariant exact du tri par insertion, et d'en donner une preuve, le tout en quelques secondes.

## 8.1 Perspectives

On donne brièvement quelques pistes d'études ou de recherches. On ne s'attarde pas sur les perspectives évidentes visant à améliorer l'expressivité de l'analyse, elles sont nombreuses : pouvoir exprimer des propriétés de tableaux avec deux quantificateurs, des propriétés sur les multi-ensembles de valeurs, partitionner dynamiquement, analyser le contenu de matrices, etc.

**Des partitions**

- Les analyses symboliques ne passent pas à l'échelle en général : par exemple, les analyses de pointeurs ont beaucoup gagné en expressivité depuis celle proposée par Anderson, mais n'ont pas conservé la capacité à passer à l'échelle. Notre analyse n'échappe pas à cet écueil. Pour améliorer le passage à l'échelle, nous évoquions l'usage d'une structure de données arborescente pour représenter nos partitions. Une telle structure de données est proposée dans [CCM10]. Conçue pour représenter certaines disjonctions, elle a apparemment été utilisée dans le but de permettre le passage à l'échelle d'une analyse du contenu des tableaux.
- Si le programme est annoté par des pré-conditions ou des post-conditions portant sur le contenu des tableaux (par exemple, Code Contracts de *Microsoft Research* permet de spécifier aisément dans un programme .Net de telles post-conditions : `Contract.Ensures(Contract.ForAll(0, i, l => a[l] != null))`), l'analyse de partitionnement peut se servir de ces informations pour affiner ses partitions (en les rendant plus précises ou en les simplifiant), notamment par une analyse arrière.
- Enfin, l'impact d'une analyse des parties vivantes des tableaux, permettant comme pour les variables d'indices de simplifier la partition, peut être étudiée.

**De la généricité**

- Si notre analyse est générique dans les domaines abstraits utilisés pour représenter les propriétés de tableaux, elle ne peut être utilisée en présence de tableaux de différents types. Dans le cas d'un programme manipulant des tableaux de différents types, on peut imaginer de transmettre le type de contenu de chaque tableau à l'analyse de contenu, qui pourrait choisir un domaine abstrait $\mathcal{D}^{\mathbb{K}}$ adéquat pour chaque tableau. Cette modification de l'analyse n'est pas directe car elle implique de gérer une partition par type de contenu.
- Sur ce sujet, on peut s'intéresser à des langages comme Python, où les tableaux peuvent contenir des éléments de différents types. Il faut alors étudier si lors d'une telle utilisation des tableaux, des tranches « typées » apparaissent ou pas du tout.

**De la visualisation**

- Une question importante que nous avons éludée jusqu'à présent est la présentation des résultats de l'analyse. En effet, même en présence de quelques tranches, ces résultats peuvent être très difficiles à relire sous la forme de contraintes. Pour ce problème, des méthodes de visualisation mises au point pour l'explication d'algorithmes [WMS01] pourraient nous aider. En fait, nous pourrions aller plus loin avec ce domaine de recherche, car certaines approches ont besoin que soient calculés les invariants du programme à expliquer. Notre analyse a les capacités nécessaires pour répondre à ce besoin, pour les programmes manipulant des tableaux.

# Annexes

## A Définitions de notions classiques utilisées

### Ordres partiels

Définition A.1 (continuité sur les ordres partiels). *Soit* $(\mathcal{D}^1, \sqsubseteq^1, \sqcup^1, \bot^1)$ *et* $(\mathcal{D}^2, \sqsubseteq^2, \sqcup^2, \bot^2)$ *deux ordres partiels complets. Alors la fonction* $f : \mathcal{D}^1 \to \mathcal{D}^2$ *est continue si et seulement si :*

$$\forall D \subseteq \mathcal{D}^1, \quad f(\sqcup^1\{x \in D\}) = \sqcup^2\{f(x) \mid x \in D\}.$$

Définition A.2 (profondeur d'un ordre partiel). *Soit* $(\mathcal{D}, \sqsubseteq)$ *un ordre partiel. La profondeur de* $\mathcal{D}$ *est la longueur maximale des suites strictement croissantes d'éléments de* $\mathcal{D}$*. Si la profondeur de* $\mathcal{D}$ *est finie, on dit qu'il satisfait la propriété de « chaînes croissantes ».*

### Graphes

Définition A.3 (graphe réductible). *Soit* $\mathcal{G} = (V, E)$ *un graphe orienté tel que tout sommet* $v$ *est accessible depuis* $v_0$*, son sommet racine. Un sommet* $v$ *domine un sommet* $v'$ *si tout chemin de* $v_0$ *à* $v'$ *passe par* $v$*. Alors* $\mathcal{G}$ *est reductible si l'on peut partitionner l'ensemble des arrêtes selon :*

- *l'ensemble des arrêtes avant, formant un graphe acyclique et dont chaque sommet est accessible depuis* $v_0$ *;*
- *l'ensemble des arrêtes arrières, telles que leur sommet de tête domine leur sommet de queue.*

Définition A.4 (graphe mixte). *Un graphe mixte est la réunion sur un même ensemble de sommet d'un graphe orienté et d'un graphe non orienté. Un fonction de valuation des arrêtes peut être attachée à chacun de ces graphes.*

Définition A.5 (chemin simple). *Un chemin simple est un chemin où chaque nœud interne est différent.*

### Géométrie affine

Lemme A.1 (lemme de Farkas [Far02]). *Soit* $S$ *un système de contraintes linéaires* $Ax + b \geq 0$*, où* $x \in \mathbb{R}^n$*. Si* $S$ *est satisfaisable alors :*

$$(S \Rightarrow c^t x + d \geq 0) \Leftrightarrow \exists \lambda \geq 0 \text{ tel que } A^t\lambda = c \text{ et } b^t\lambda \leq d.$$

Définition A.6 (enveloppe affine). *L'enveloppe affine d'un ensemble de points* $X = \{x_1, \ldots x_n\}$ *est le plus petit ensemble affine contenant* $X$ *:*

$$\mathit{Affine}(X) = \{\alpha_1 x_1 + \ldots + \alpha_n x_n \mid \sum_{i=1}^{n} \alpha_i = 1\}.$$

# B Une collection de programmes à analyser

Cette collection de vingt-six programmes, qui inclue le *benchmark* proposé dans la littérature, est composée d'une part des différents programmes présentés dans les chapitres précédents :

- algorithmes d'initilialisation (10 programmes) : Fig. 7.3 (p. 182), Fig. 7.7 (p. 192), Fig. 7.8(a)–7.8(c) (p. 195) et Fig. 6.16(a) (p. 166).
- algorithmes de recherche d'information (4 programmes) : Fig. 7.4 (p. 186) et Fig. 7.8(d) (p. 195).
- algorithmes de tri (3 programmes) : Fig. 7.5 (p. 189) et Fig. 7.8(b) (p. 195).

D'autre part, cette collection est formée des neuf programmes suivants, que nous n'avons pas étudié précisemment. Parfois il s'agit d'algorithmes très proches de programmes déjà présentés, mais dont les variations déplacent une difficulté ou en introduisent une nouvelle.

```
i := 1 ;
x := 0 ;
while i ≤ n do
    x := x + a[i] ;
    i := i + 1
```

- Algorithmes d'initialisation (Fig. B.1, 2 programmes). Notamment on a déjà évoqué le programme en (b) pour lequel il faut découvrir une propriété de tableau portant sur le muti-ensemble de valeurs de $a$ (Sec. 4.1). C'est aussi le cas pour le programme ci-contre.
- Algorithmes de recherche (Fig. B.2, 1 programme).
- Algorithmes de réorganisation du contenu (Fig. B.3 et Fig. B.4, 5 programmes).

```
i := 1 ;
while i ≤ n do
    a[i] := 0 ;
    i := i + 1
```

(a) Initialisation à zéro

```
j := 0 ; i := 1 ;
while i ≤ n do
    if a[i] > 0 then
        j := j + 1
    i := i + 1
b := array[j] ;
j := 1 ; i := 1 ;
while i ≤ n do
    if a[i] > 0 then
        b[j] := a[i] ;
        j := j + 1
    i := i + 1
```

(b) Filtre puis copie

Figure B.1 – Programmes d'initialisation

```
i := 1 ; j := 1 ;
while i ≤ n do
    if a[i] > a[j] then
        j := i
    i := i + 1
```

(a) Recherche du maximum sans scalaire

Figure B.2 – Programmes recherchant un élément

```
i := 1 ; j := 1 ; k := n ;
while j ≤ k do
    switch a[j] do
        case white
            j := j + 1
        case red
            a[j] :=: a[k] ;
            k := k − 1
        case blue
            a[i] :=: a[j] ;
            i := i + 1 ;
            j := j + 1
```

(a) Drapeau hollandais

```
i := 1 ;
while i ≤ n do
    k := i ;
    x := a[i] ;
    j := i + 1 ;
    while j ≤ n do
        if a[j] < x then
            k := j ;
            x := a[j]
        j := j + 1
    a[k] := a[i] ;
    a[i] := x ;
    i := i + 1
```

(b) Tri par sélection

```
x := a[1] ;
i := 2 ;
j := 2 ;
while j ≤ n do
    if a[j] > x then
        a[j] :=: a[i] ;
        i := i + 1
    j := j + 1
```

(c) Phase de segmentation du tri rapide non optimisée (une boucle)

```
i := 1 ;
j := 1 ;
k := 1 ;
x := a[1] ;
while i ≤ n and j ≤ m do
    if a[i] ≤ b[i] then
        c[k] := a[i] ;
        i := i + 1
    c[k] := b[j] ;
    j := j + 1
while i ≤ n do
    c[k] := a[i] ;
    i := i + 1
while j ≤ m do
    c[k] := b[j] ;
    j := j + 1
```

(d) Fusion d'un tri fusion ($a$ et $b$ sont triés)

Figure B.3 – Programmes de tri

$i := 1$ ;
$x := a[1]$ ;
**while** $i \leq n - 1$ **do**
  $a[i] := a[i + 1]$ ;
  $i := i + 1$
$a[n] := x$

(a) Rotation de tableau

Figure B.4 – Programmes réorganisant le contenu (hors tri)

# C Analyse du tri par insertion : sortie sur le terminal

```
---- enkidu ----
compiling
  file = insertion.sort.fst ... OK
internal structures generation ... OK
analysis
  kind = array (dbm x ddbm)
  size = 9 x 11
  widening start = 1
  descendings = 3
  strategy = default ... OK
results
  loop = {
    2<=i,1<=n,i<=n+1
    ~
    2<='l,3<=i,'l<=i-1,'l<=n => a_-1<=a_0
  }
  loop_sort = {
    2<=i,0<=j,2<=n,j<=i-1,i<=n,j<=n-1
    ~
    'l=1,2<=i,j<=0,1<=n,'l<=i-1,j<='l-1,j<=i-2,j<=n-1,'l<=n
                                                  => k<=a_1-1,k<=a_0-1,a_0=a_1
    2<='l,3<=i,3<=j,2<=n,'l<=i-1,'l<=j-1,j<=i-1,'l<=n => a_-1<=a_0
    2<='l,3<=i,2<=j,2<=n,'l<=i-1,j='l,j<=i-1,j<=n,'l<=n => a_-1<=a_0
    2<='l,3<=i,1<=j,2<=n,'l<=i-1,j='l-1,j<=i-2,j<=n-1,'l<=n
                                                        => k<=a_0-1,a_-1<=a_0
    2<='l,3<=i,2<=n,'l<=i-1,j<='l-2,j<=i-3,j<=n-2,'l<=n
                    => k<=a_1-1,k<=a_0-1,a_0<=a_1,k<=a_-1-1,a_-1<=a_1,a_-1<=a_0
    2<='l,2<=i,2<=n,1<=j,i='l,i<=n,j='l-1,j=i-1,j<=n-1,'l<=n => a_0=k
    2<='l,2<=i,2<=n,0<=j,i='l,i<=n,j<='l-2,j<=i-2,j<=n-2,'l<=n
                                                  => k<=a_0-1,k<=a_-1-1,a_-1<=a_0
  }
```

```
  final = {
    2<=i,1<=n,n=i-1
    ~
    2<='l,'l<=n => a_-1<=a_0
  }
statistics
  time = 5.384344
  iterations = 6
  descendings = 1
-----------------
```

# Bibliographie


[Abr96] Jean Raymond Abrial: *The B book - Assigning Programs to Meanings*. Cambridge University Press, 1996. (cité page 2)

[All08] Xavier Allamigeon: *Non-disjunctive Numerical Domain for Array Predicate Abstraction*. Dans *ESOP'08 : 17th European Symposium on Programming*, tome 4960 de *Lecture Notes in Computer Science*, pages 163–177. Springer, avril 2008. (cité pages 56, 69 et 70)

[BBFM99] Patrick Behm, Paul Benoit, Alain Faivre et Jean Marc Meynadier: *Météor : A Successful Application of B in a Large Project*. Dans *FM'99 : World Congress on Formal Methods in the Development of Computing Systems*, tome 1708 de *Lecture Notes in Computer Science*, pages 369–387. Springer, septembre 1999. (cité page 2)

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux et Xavier Rival: *Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software*. Dans *The Essence of Computation : Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, Lecture Notes in Computer Science, pages 85–108. Springer, octobre 2002. (cité page 113)

[BCC+03] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux et Xavier Rival: *A Static Analyzer for Large Safety-Critical Software*. Dans *PLDI'03 : 2003 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–207. ACM, juin 2003. (cité pages 3, 5, 56, 57, 58 et 113)

[BCC+07] Josh Berdine, Aziem Chawdhary, Byron Cook, Dino Distefano et Peter O'Hearn: *Variance Analyses from Invariance Analyses*. Dans *POPL'07 : 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 211–224. ACM, janvier 2007. (cité page 20)

[BDM08] Flavia Bonomo, Guillermo Durán et Javier Marenco: *Exploring the Complexity Boundary between Coloring and List-Coloring*. Annals of Operations Research, 2008. Sous presse. (cité page 102)

[Ber08] Gérard Berry: *Pourquoi et comment le monde devient numérique*. Leçons inaugurales du Collège de France. Fayard, 2008. (cité page 1)

[BHI+09] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konečný et Tomáš Vojnar: *Automatic Verification of Integer Array Programs*. Dans *CAV'09 : 21st International Conference on Computer Aided Verification*, tome 5643 de

*Lecture Notes in Computer Science*, pages 157–172. Springer, juin 2009. (cité pages 48, 68 et 70)

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar et Andrey Rybalchenko: *Path Invariants.* Dans *PLDI'07 : 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–309. ACM, juin 2007. (cité pages 53 et 68)

[BLP06] Sébastien Bardin, Jérôme Leroux et Gérald Point: *Tool Presentation : FAST Extended Release.* Dans *CAV'06 : 18th International Conference on Computer Aided Verification*, tome 4144 de *Lecture Notes in Computer Science*, pages 63–66. Springer, août 2006. (cité page 172)

[BMS06] Aaron R. Bradley, Zohar Manna et Henny B. Sipma: *What's Decidable About Arrays ?* Dans *VMCAI'06 : 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, tome 3855 de *Lecture Notes in Computer Science*, pages 427–442. Springer, janvier 2006. (cité pages 48 et 53)

[Bou90] François Bourdoncle: *Interprocedural Abstract Interpretation of Block Structured Languages with Nested Procedures, Aliasing and Recursivity.* Dans *PLILP'90 : 2nd International Workshop on Programming Language Implementation and Logic Programming*, tome 456 de *Lecture Notes in Computer Science*, pages 307–323. Springer, août 1990. (cité page 189)

[Bou93] François Bourdoncle: *Efficient Chaotic Iteration Strategies with Widenings.* Dans *International Conference on Formal Methods in Programming and their Applications*, tome 735 de *Lecture Notes in Computer Science*, pages 128–141. Springer, juin 1993. (cité page 24)

[BPR02] Thomas Ball, Andreas Podelski et Sriram K. Rajamani: *Relative Completeness of Abstraction Refinement for Software Model Checking.* Dans *TACAS'00 : 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, tome 2280 de *Lecture Notes in Computer Science*, pages 158–172. Springer, avril 2002. (cité page 52)

[CC76] Patrick Cousot et Radhia Cousot: *Static Determination of Dynamic Properties of Programs.* Dans *ISOP'76 : 2nd International Symposium on Programming*, pages 106–130. Dunod, avril 1976. (cité pages 16, 17, 20, 25, 29, 31 et 123)

[CC77] Patrick Cousot et Radhia Cousot: *Abstract Interpretation : a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints.* Dans *POPL'77 : 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM, janvier 1977. (cité pages 5, 13, 14 et 18)

[CC79a] Patrick Cousot et Radhia Cousot: *Constructive Versions of Tarski's Fixed Point Theorems.* Pacific Journal of Mathematics, 82(1) :43–57, mai 1979. (cité page 16)

[CC79b] Patrick Cousot et Radhia Cousot: *Systematic Design of Program Analysis Frameworks.* Dans *POPL'79 : 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 269–282. ACM, janvier 1979. (cité pages 14 et 28)

[CC92a] Patrick Cousot et Radhia Cousot: *Abstract Interpretation Frameworks.* Journal of Logic and Computation, 2(4) :511–547, août 1992. (cité pages 15 et 28)

[CC92b] Patrick Cousot et Radhia Cousot: *Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation.* Dans *PLILP'92 : 4th International Symposium on Programming Language Implementation and Logic Programming*, tome 631 de *Lecture Notes in Computer Science*, pages 269–295. Springer, août 1992. (cité pages 17 et 113)

[CC04] Robert Clarisó et Jordi Cortadella: *The Octahedron Abstract Domain.* Dans *SAS'04 : 11th International Symposium on Static Analysis*, tome 3148 de *Lecture Notes in Computer Science*, pages 312–327. Springer, août 2004. (cité page 31)

[CCM10] Patrick Cousot, Radhia Cousot et Laurent Mauborgne: *A Scalable Segmented Decision Tree Abstract Domain.* Dans *Time for Verification, Essays in Memory of Amir Pnueli*, tome 6200 de *Lecture Notes in Computer Science*, pages 72–95. Springer, juillet 2010. (cité page 201)

[Čer03] Pavol Černý: *Verification by Abstract Interpretation of Parameterized Predicates*, 2003. (cité page 69)

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu et Helmut Veith: *Counterexample-Guided Abstraction Refinement.* Dans *CAV'00 : 12th International Conference on Computer Aided Verification*, tome 1855 de *Lecture Notes in Computer Science*, pages 154–169. Springer, juillet 2000. (cité page 52)

[CGP99] Edmund M. Clarke, Orna Grumberg et Doron A. Peled: *Model Checking.* MIT, janvier 1999. (cité page 28)

[CH78] Patrick Cousot et Nicolas Halbwachs: *Automatic Discovery of Linear Restraints among Variables of a Program.* Dans *POPL'78 : 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 84–96. ACM, janvier 1978. (cité pages 29 et 31)

[CL05] Agostino Cortesi et Francesco Logozzo: *Abstract Interpretation-Based Verification of Non-functional Requirements.* Dans *COORDINATION'05 : 7th International Conference on Coordination Models and Languages*, tome 3454 de *Lecture Notes in Computer Science*, pages 49–62. Springer, avril 2005. (cité page 20)

[Cou81] Patrick Cousot: *Semantic Foundations of Program Analysis.* Dans *Program Flow Analysis : Theory and Applications*, chapitre 10, pages 303–342. Prentice-Hall, 1981. (cité page 23)

[Cou99] Patrick Cousot: *The Calculational Design of a Generic Abstract Interpreter.* Dans *Calculational System Design*, tome 173 de *NATO Advanced Study Institutes : Computer & Systems Sciences*. IOS Press, 1999. (cité page 20)

[Cou02] Patrick Cousot: *Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation.* Theoretical Computer Science, 277(1–2) :47–103, avril 2002. (cité page 20)

[Cou03] Patrick Cousot: *Verification by Abstract Interpretation.* Dans *International Symposium on Verification : Theory and Practice – Honoring Zohar Manna's 64th Birthday*, tome 2772 de *Lecture Notes in Computer Science*, pages 243–268. Springer, juin 2003. (cité pages 54, 55, 69, 70, 71, 125 et 200)

[Cou05] Patrick Cousot: *Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming.* Dans *VMCAI'05 : 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, tome 3385 de *Lecture Notes in Computer Science*, pages 1–24. Springer, janvier 2005. (cité page 20)

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs et John A. Plaice: *Lustre : A Declarative Language for Programming Synchronous Systems.* Dans *POPL'87 : 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 178–188. ACM, janvier 1987. (cité page 200)

[CS01] Michael A. Colón et Henny B. Sipma: *Synthesis of Linear Ranking Functions.* Dans *TACAS'01 : 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, tome 2031 de *Lecture Notes in Computer Science*, pages 67–81. Springer, avril 2001. (cité page 20)

[DDP99] Satyaki Das, David L. Dill et Seungjoon Park: *Experience with Predicate Abstraction.* Dans *CAV'99 : 11th International Conference on Computer Aided Verification*, tome 1633 de *Lecture Notes in Computer Science*, pages 160–171. Springer, juillet 1999. (cité page 49)

[Deu94] Alain Deutsch: *Interprocedural May-Alias Analysis for Pointers : Beyond k-limiting.* Dans *PLDI'94 : ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, pages 230–241. ACM, juin 1994. (cité page 70)

[Dil89] David L. Dill: *Timing Assumptions and Verification of Finite-State Concurrent Systems.* Dans *International Workshop on Automatic Verification Methods for Finite State Systems*, tome 407 de *Lecture Notes in Computer Science*, pages 197–212. Springer, juin 1989. (cité pages 7, 31, 33 et 34)

[DSW88] Perino M. Dearing, Douglas R. Shier et Daniel D. Warner: *Maximal Chordal Subgraphs.* Discrete Applied Mathematics, 20(3) :181–190, juillet 1988. (cité page 103)

[Far02] Julius Farkas: *Über die Theorie der Einfachen Ungleichungen.* Journal für die Reine und Angewandte Mathematik, 124 :1–27, 1902. (cité pages 90 et 203)

[Fer04] Jérôme Feret: *Static Analysis of Digital Filters.* Dans *ESOP'04 : 13th European Symposium on Programming*, tome 2986 de *Lecture Notes in Computer Science*. Springer, mars 2004. (cité page 31)

[FQ02] Cormac Flanagan et Shaz Qadeer: *Predicate Abstraction for Software Verification.* Dans *POPL'02 : 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 191–202. ACM, janvier 2002. (cité pages 50, 51, 54, 68 et 184)

[GDD+04] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps et Mooly Sagiv: *Numeric Domains with Summarized Dimensions.* Dans *TACAS'04 : 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, tome 2988 de *Lecture Notes in Computer Science*, pages 512–529. Springer, mars 2004. (cité pages 5, 56, 57, 58 et 59)

[GK79] Alla Goralčíková et Václav Koubek: *A Reduct-and-Closure Algorithm for Graphs.* Dans *MFCS'79 : 8th International Symposium on Mathematical*

*Foundations of Computer Science*, tome 74 de *Lecture Notes in Computer Science*, pages 301–307. Springer, septembre 1979. (cité page 101)

[GLS09] Sumit Gulwani, Tal Lev-Ami et Mooly Sagiv: *A Combination Framework for Tracking Partition Sizes*. Dans *POPL'09 : 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 239–251. ACM, janvier 2009. (cité page 47)

[GM09] Yeting Ge et Leonardo de Moura: *Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories*. Dans *CAV'09 : 21st International Conference on Computer Aided Verification*, tome 5643 de *Lecture Notes in Computer Science*, pages 306–320. Springer, juin 2009. (cité page 50)

[GMT08] Sumit Gulwani, Bill McCloskey et Ashish Tiwari: *Lifting Abstract Interpreters to Quantified Logical Domains*. Dans *POPL'08 : 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 235–246. ACM, janvier 2008. (cité pages 4, 64, 67, 68, 69, 70, 82 et 133)

[Gon07] Laure Gonnord: *Accélération abstraite pour l'amélioration de la précision en Analyse des Relations Linéaires*. Thèse de doctorat, Université Joseph Fourier, Grenoble, France, octobre 2007. (cité page 172)

[GPP10] Manuel Garnacho, Mathias Péron et Michaël Périn: *Automatic Generation of Coq Certificates from Static Analyzers by Lightweight Instrumentation*. Soumis à IWIL'10 : 8th International Workshop on the Implementation of Logics, août 2010. (cité pages 7, 196 et 200)

[Gra89] Philippe Granger: *Static Analysis of Arithmetical Congruences*. International Journal of Computer Mathematics, 30(3-4) :165–190, 1989. (cité pages 28 et 31)

[Gra91] Philippe Granger: *Static Analysis of Linear Congruence Equalities among Variables of a Program*. Dans *TAPSOFT'91 : International Joint Conference on Theory and Practice of Software Development on Colloquium on trees in algebra and programming (CAAP'91)*, pages 169–192. Springer, avril 1991. (cité page 30)

[GRS05] Denis Gopan, Thomas Reps et Mooly Sagiv: *A Framework for Numeric Analysis of Array Operations*. Dans *POPL'05 : 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 338–350. ACM, janvier 2005. (cité pages 4, 5, 8, 59, 60, 63, 64, 65, 66, 67, 68, 69, 70, 71, 124, 125, 145, 147, 148 et 200)

[GS97] Susanne Graf et Hassen Saïdi: *Construction of Abstract State Graphs with PVS*. Dans *CAV'97 : 9th International Conference on Computer Aided Verification*, tome 1254 de *Lecture Notes in Computer Science*, pages 72–83. Springer, juin 1997. (cité page 49)

[GT06] Sumit Gulwani et Ashish Tiwari: *Combining Abstract Interpreters*. Dans *PLDI'06 : 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 376–386. ACM, juin 2006. (cité page 64)

[HHKV10] Thomas Henziger, Thibaud Hottelier, Laura Kovács et Andrei Voronkov: *Invariant and Type Inference for Matrices*. Dans *VMCAI'10 : 11th International Conference on Verification, Model Checking, and Abstract Interpretation*,

tome 5944 de *Lecture Notes in Computer Science*, pages 163–179. Springer, janvier 2010. (cité pages 46 et 82)

[HIV08] Peter Habermehl, Radu Iosif et Tomáš Vojnar: *What Else Is Decidable about Integer Arrays ?* Dans *FoSSaCS'08 : 11th International Conference on Foundations of Software Science and Computational Structures*, tome 4962 de *Lecture Notes in Computer Science*, pages 474–489. Springer, mars 2008. (cité page 48)

[HJMM04] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar et Kenneth L. McMillan: *Abstractions From Proofs*. Dans *POPL'04 : 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–244. ACM, janvier 2004. (cité page 50)

[HM07] Thierry Hubert et Claude Marché: *Separation Analysis for Deductive Verification*. Dans *HAV'07 : Workshop on Heap Analysis and Verification*, mars 2007. (cité page 47)

[Hoa61] C. A. R. Hoare: *Algorithm 63 : Partition and Algorithm 64 : Quicksort*. Communications of the ACM, 4(7) :321, juillet 1961. (cité page 188)

[HP08] Nicolas Halbwachs et Mathias Péron: *Discovering Properties about Arrays in Simple Programs*. Dans *PLDI'08 : 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 339–348. ACM, juin 2008. (cité pages 7, 67, 76, 126 et 179)

[HS97] Warwick Harvey et Peter J. Stuckey: *A Unit Two Variable Per Inequality Integer Constraint Solver for Constraint Logic Programming*. Dans *ACSC'97 : 20th Australasian Computer Science Conference*, tome 19, pages 102–111, février 1997. (cité page 42)

[Imb93] Jean Louis Imbert: *Variable Elimination for Generalized Linear Constraints*. Dans *ICLP'93 : 10th International Conference on Logic Programming*, pages 499–516. MIT, juin 1993. (cité page 41)

[Jac09] Daniel Jackson: *A Direct Path to Dependable Software*. Communications of the ACM, 52(4) :78–88, avril 2009. (cité page 2)

[JB96] Peter Jonsson et Christer Bäckström: *A Linear-Programming Approach to Temporal Reasoning*. Dans *AAAI'96 : 13th National Conference on Artificial Intelligence*, tome 2, pages 1235–1240. MIT, août 1996. (cité page 40)

[Jea00] Bertrand Jeannet: *Partitionnement dynamique dans l'analyse de relations linéraires et application à la vérification de programmes synchrones*. Thèse de doctorat, Institut National Polytechnique de Grenoble, Grenoble, France, septembre 2000. (cité pages 33 et 37)

[Jea10] Bertrand Jeannet: *Fixpoint Solver Library 3.0*. INRIA, février 2010. `http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/fixpoint/fixpoint.pdf`. (cité pages 9 et 171)

[JGR05] Bertrand Jeannet, Denis Gopan et Thomas Reps: *A Relational Abstraction for Functions*. Dans *SAS'05 : 12th International Symposium on Static Analysis*, tome 3672 de *Lecture Notes in Computer Science*, pages 186–202. Springer, septembre 2005. (cité page 57)

[JM07] Ranjit Jhala et Kenneth L. McMillan: *Array Abstractions from Proofs*. Dans *CAV'07 : 19th International Conference on Computer Aided Verification*, tome 4590 de *Lecture Notes in Computer Science*, pages 193–206. Springer, juillet 2007. (cité pages 4, 53, 54, 64, 66, 69 et 70)

[JS04] Bertrand Jeannet et Wendelin Serwe: *Abstracting Call-Stacks for Interprocedural Verification of Imperative Programs*. Dans *AMAST'04 : 10th International Conference on Algebraic Methodology and Software Technology*, tome 3116 de *Lecture Notes in Computer Science*, pages 258–273. Springer, juillet 2004. (cité page 189)

[Kar76] Michael Karr: *Affine Relationships among Variables of a Program*. Acta Informatica, 6(2) :133–151, juin 1976. (cité page 30)

[Kil73] Gary A. Kildall: *A Unified Approach to Global Program Optimization*. Dans *POPL'73 : 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 194–206. ACM, octobre 1973. (cité pages 16 et 30)

[KJS07] Hyondeuk Kim, Hoonsang Jin et Fabio Somenzi: *Disequality Management in Integer Difference Logic via Finite Instantiations*. Journal on Satisfiability, Boolean Modeling and Computation, 3 :47–66, juillet 2007. (cité pages 42 et 103)

[Kle52] Stephen C. Kleene: *Introduction to Metamathematics*, tome 1 de *Bibliotheca Mathematica*. North-Holland, 1952. (cité page 15)

[Kou92] Manolis Koubarakis: *Dense Time and Temporal Constraints With* $\neq$. Dans *KR'92 : 3rd International Conference on Principles of Knowledge Representation and Reasoning*, pages 24–35. Morgan Kaufmann, octobre 1992. (cité page 41)

[Kou01] Manolis Koubarakis: *Tractable Disjunctions of Linear Constraints : Basic Results and Applications to Temporal Reasoning*. Theoretical Computer Science, 266(1-2) :311–339, septembre 2001. (cité pages 40 et 41)

[LB04a] Shuvendu K. Lahiri et Randal E. Bryant: *Constructing Quantified Invariants via Predicate Abstraction*. Dans *VMCAI'04 : 5th International Conference on Verification, Model Checking, and Abstract Interpretation*, tome 2937 de *Lecture Notes in Computer Science*, pages 267–281. Springer, janvier 2004. (cité page 51)

[LB04b] Shuvendu K. Lahiri et Randal E. Bryant: *Indexed Predicate Discovery for Unbounded System Verification*. Dans *CAV'04 : 16th International Conference on Computer Aided Verification*, tome 3114 de *Lecture Notes in Computer Science*, pages 135–147. Springer, juillet 2004. (cité page 50)

[LBC03] Shuvendu K. Lahiri, Randal E. Bryant et Byron Cook: *A Symbolic Approach to Predicate Abstraction*. Dans *CAV'03 : 15th International Conference on Computer Aided Verification*, tome 2725 de *Lecture Notes in Computer Science*, pages 141–153. Springer, juillet 2003. (cité page 51)

[LL09] Vincent Laviron et Francesco Logozzo: *SubPolyhedra : A (More) Scalable Approach to Infer Linear Inequalities*. Dans *VMCAI'09 : 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, tome 5403 de *Lecture Notes in Computer Science*, pages 229–244. Springer, janvier 2009. (cité page 32)

[LM88] Jean-Louis Lassez et Ken McAloon: *Applications of a Canonical Form of Gerneralized Linear Constraints*. Dans *FGCS'88 : International Conference on Fifth Generation Computer Systems*, pages 703–710. Ohmsha & Springer, novembre 1988. (cité page 40)

[LM89] Jean-Louis Lassez et Ken McAloon: *Independence of Negative Constraints*. Dans *TAPSOFT'89 : International Joint Conference on Theory and Practice of Software Development*, tome 351 de *Lecture Notes in Computer Science*, pages 19–27. Springer, mars 1989. (cité page 40)

[LM92] Jean-Louis Lassez et Ken McAloon: *A Canonical Form for Generalized Linear Constraints*. Journal of Symbolic Computation, 13(1) :1–24, janvier 1992. (cité pages 40, 41 et 89)

[LPWY99] Kim G. Larsen, Justin Pearson, Carsten Weise et Wang Yi: *Clock Difference Diagrams*. Nordic Journal of Computing, 6(3) :271–298, 1999. (cité page 39)

[LRSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv et Reinhard Wilhelm: *Putting Static Analysis to Work for Verification : A Case Study*. Dans *ISSTA'00 : 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 26–38. ACM, août 2000. (cité pages 64, 70 et 200)

[LS00] Tal Lev-Ami et Shmuel Sagiv: *TVLA : A System for Implementing Static Analyses*. Dans *SAS'00 : 7th International Symposium on Static Analysis*, tome 1824 de *Lecture Notes in Computer Science*, pages 280–301. Springer, juin 2000. (cité page 60)

[Mas92] François Masdupuy: *Array Abstractions using Semantic Analysis of Trapezoid Congruences*. Dans *ICS'92 : 6th International Conference on Supercomputing*, pages 226–235. ACM, juillet 1992. (cité page 31)

[Mas93] François Masdupuy: *Array Indices Relational Semantic Analysis using Rational Cosets and Trapezoids*. Thèse de doctorat, École Polytechnique, Palaiseau, France, décembre 1993. (cité pages 46 et 121)

[Min01] Antoine Miné: *The Octagon Abstract Domain*. Dans *AST'01 : Workshop on Analysis, Slicing, and Transformation*, pages 310–319. IEEE, octobre 2001. (cité pages 31 et 172)

[Min02] Antoine Miné: *A Few Graph-Based Relational Numerical Abstract Domains*. Dans *SAS'02 : 9th International Symposium on Static Analysis*, tome 2477 de *Lecture Notes in Computer Science*, pages 117–132. Springer, septembre 2002. (cité pages 30 et 31)

[Min04] Antoine Miné: *Weakly Relational Numerical Abstract Domains*. Thèse de doctorat, École Polytechnique, Palaiseau, France, décembre 2004. (cité pages 26, 32, 36, 37, 88, 101 et 119)

[Min06] Antoine Miné: *The Octagon Abstract Domain*. Higher-Order and Symbolic Computation, 19(1) :31–100, mars 2006. (cité page 114)

[MLAH99] Jesper B. Møller, Jakob Lichtenberg, Henrik R. Andersen et Henrik Hulgaard: *Difference Decision Diagrams*. Dans *CSL'99 : 13th International Workshop on Computer Science Logic*, tome 1683 de *Lecture Notes in Computer Science*, pages 111–125. Springer, septembre 1999. (cité page 39)

[MM07] Yannick Moy et Claude Marché: *Inferring Local (Non-)Aliasing and Strings for Memory Safety*. Dans *HAV'07 : Workshop on Heap Analysis and Verification*, pages 35–51, mars 2007. (cité page 131)

[MP95] Zohar Manna et Amir Pnueli: *Temporal Verification of Reactive Systems : Safety*. Springer, août 1995. (cité pages 119 et 120)

[MR05] Laurent Mauborgne et Xavier Rival: *Trace Partitioning in Abstract Interpretation Based Static Analyzers*. Dans *ESOP'05 : 14th European Symposium on Programming*, tome 3444 de *Lecture Notes in Computer Science*, pages 5–20. Springer, avril 2005. (cité pages 28, 58 et 146)

[NB95] Bernhard Nebel et Hans Jürgen Bürckert: *Reasoning about Temporal Relations : A Maximal Tractable Subclass of Allen's Interval Algebra*. Journal of the ACM, 42(1) :43–66, janvier 1995. (cité page 41)

[PH07] Mathias Péron et Nicolas Halbwachs: *An Abstract Domain Extending Difference-Bound Matrices with Disequality Constraints*. Dans *VMCAI'07 : 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, tome 4349 de *Lecture Notes in Computer Science*, pages 268–282. Springer, janvier 2007. (cité pages 7, 82, 83 et 101)

[PH10] Valentin Perrelle et Nicolas Halbwachs: *An Analysis of Permutations in Arrays*. Dans *VMCAI'10 : 11th International Conference on Verification, Model Checking, and Abstract Interpretation*, tome 5944 de *Lecture Notes in Computer Science*, pages 279–294. Springer, janvier 2010. (cité pages 9, 47, 82, 178 et 200)

[Pie02] Benjamin C. Pierce: *Types and programming languages*. MIT, février 2002. (cité page 28)

[PP07] Heggernes Pinar et Charis Papadopoulos: *Single-Edge Monotonic Sequences of Graphs and Linear-Time Algorithms for Minimal Completions and Deletions*. Dans *COCOON'07 : 13th Annual International Computing and Combinatorics Conference*, tome 4598 de *Lecture Notes in Computer Science*, pages 406–416. Springer, juillet 2007. (cité page 103)

[Pra77] Vaughan R. Pratt: *Two Easy Theories whose Combination is Hard*. rapport technique, Massachusetts Institute of Technology, Cambridge, septembre 1977. (cité page 34)

[PW98] William Pugh et David Wonnacott: *Constraint-Based Array Dependence Analysis*. TOPLAS : Transactions on Programming Languages and Systems, 20(3) :635–678, mai 1998. (cité pages 46 et 169)

[RH80] Daniel J. Rosenkrantz et Harry B. Hunt III: *Processing Conjunctive Predicates and Queries*. Dans *VLDB'80 : 6th International Conference on Very Large Data Bases*, pages 64–72. IEEE Computer Society, octobre 1980. (cité pages 7 et 42)

[Ric53] Henry G. Rice: *Classes of Recursively Enumerable Sets and Their Decision Problems*. Transactions of the American Mathematical Society, 74(2) :358–366, mars 1953. (cité pages 2 et 76)

[RT75] Donald J. Rose et Robert E. Tarjan: *Algorithmic Aspects of Vertex Elimination*. Dans *STOC'75 : 7th Annual ACM Symposium on Theory of Computing*, pages 245–254. ACM, mai 1975. (cité page 103)

[SD07] Jean Souyris et David Delmas: *Experimental Assessment of Astrée on Safety-Critical Avionics Software*. Dans *SAFECOMP'07 : 26th International Conference on Computer Safety, Reliability, and Security*, tome 4680 de *Lecture Notes in Computer Science*, pages 479–490. Springer, septembre 2007. (cité page 58)

[SG09] Saurabh Srivastava et Sumit Gulwani: *Program Verification using Templates over Predicate Abstraction*. Dans *PLDI'09 : 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234. ACM, juin 2009. (cité pages 47, 51, 68, 70 et 179)

[Sif82] Joseph Sifakis: *A unified approach for studying the properties of transition systems*. Theoretical Computer Science, 18(3) :227–258, 1982. (cité page 28)

[SKH02] Axel Simon, Andy King et Jacob M. Howe: *Two Variables per Linear Inequality as an Abstract Domain*. Dans *LOPSTR'02 : 12th International Workshop on Logic Based Program Synthesis and Tranformation*, tome 2664 de *Lecture Notes in Computer Science*, pages 71–89. Springer, septembre 2002. (cité page 31)

[SPW09] Mohamed Nassim Seghir, Andreas Podelski et Thomas Wies: *Abstraction Refinement for Quantified Array Assertions*. Dans *SAS'09 : 16th International Symposium on Static Analysis*, tome 5673 de *Lecture Notes in Computer Science*, pages 3–18. Springer, août 2009. (cité pages 53, 69, 70, 191 et 192)

[SS99] Hassen Saïdi et Natarajan Shankar: *Abstract and Model Check While You Prove*. Dans *CAV'99 : 11th International Conference on Computer Aided Verification*, tome 1633 de *Lecture Notes in Computer Science*, pages 443–454. Springer, juillet 1999. (cité page 49)

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma et Zohar Manna: *Scalable Analysis of Linear Systems Using Mathematical Programming*. Dans *VMCAI'05 : 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, tome 3385 de *Lecture Notes in Computer Science*, pages 25–41. Springer, janvier 2005. (cité page 32)

[SW02] Robert Seater et David Wonnacott: *Efficient Manipulation of Disequalities During Dependence Analysis*. Dans *LCPC'02 : 15th Workshop on Languages and Compilers for Parallel Computing*, tome 2481 de *Lecture Notes in Computer Science*, pages 295–308. Springer, juillet 2002. (cité page 103)

[SW04] Zhendong Su et David Wagner: *A Class of Polynomially Solvable Range Constraints for Interval Analysis without Widenings*. Dans *TACAS'04 : 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, tome 2988 de *Lecture Notes in Computer Science*, pages 280–295. Springer, mars 2004. (cité page 16)

[Tar55] Alfred Tarski: *A Lattice-Theoretical Fixpoint Theorem and Its Applications*. Pacific Journal of Mathematics, 5(2) :285–309, juin 1955. (cité page 13)

[Tar72] Robert E. Tarjan: *Depth-First Search and Linear Graph Algorithms*. SIAM Journal on Computing, 1(2) :146–160, juin 1972. (cité page 24)

[Ven02] Arnaud Venet: *Nonuniform Alias Analysis of Recursive Data Structures and Arrays*. Dans *SAS'02 : 9th International Symposium on Static Analysis*, tome

2477 de *Lecture Notes in Computer Science*, pages 36–51. Springer, septembre 2002. (cité page 70)

[Ven05] Arnaud Venet: *Towards the Integration of Symbolic and Numerical Static Analysis*. Dans *VSTTE'05 : First IFIP TC 2/WG 2.3 Conference on Verified Software : Theories, Tools, Experiments*, tome 4171 de *Lecture Notes in Computer Science*, pages 227–236. Springer, octobre 2005. (cité page 70)

[vH01] Willem Jan van Hoeve: *The alldifferent Constraint : A Survey*. Dans *6th Annual Workshop of the ERCIM Working Group on Constraints (The Computing Research Repository)*, tome cs/0110012, page cs.PL/0105015, juin 2001. (cité page 103)

[Vil82] Marc B. Vilain: *A System for Reasoning About Time*. Dans *AAAI'82 : National Conference on Artificial Intelligence*, pages 197–201. AAAI, août 1982. (cité page 41)

[VJB+03] Peter Vanbroekhoven, Gerda Janssens, Maurice Bruynooghe, Henk Corporaal et Francky Catthoor: *Advanced Copy Propagation for Arrays*. Dans *LCTES'03 : 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, pages 24–33. ACM, juin 2003. (cité page 181)

[WMS01] Reinhard Wilhelm, Tomasz Müldner et Raimund Seidel: *Algorithm Explanation : Visualizing Abstract States and Invariants*. Dans *Software Visualization : International Seminar*, tome 2269 de *Lecture Notes in Computer Science*, pages 381–394. Springer, mai 2001. (cité page 201)

[Yov98] Sergio Yovine: *Model Checking Timed Automata*. Dans *Lectures on Embedded Systems, School on Embedded Systems*, tome 1494 de *Lecture Notes in Computer Science*, pages 114–152. Springer-Verlag, octobre 1998. (cité page 38)

**Résumé**

Si l'analyse automatique des accès aux tableaux a été largement étudiée, on trouve très peu de résultats convaincants sur l'analyse du contenu des tableaux.

Pour une telle analyse, les analyses numériques sont centrales. Notamment, si l'on découvre l'invariant $i \neq j$, on évite d'affaiblir la connaissance sur $a[j]$ lors d'une affectation à $a[i]$. Nous proposons une nouvelle analyse numérique faiblement relationnelle, combinant des contraintes de zones ($x - y \leq c$ ou $\pm x \leq c$) à des contraintes de non-égalités ($x \neq y$ ou $x \neq 0$). Cette analyse a une complexité en $\mathcal{O}(n^4)$, si les variables prennent leur valeurs dans un ensemble dense. Dans le cas arithmétique, décider de la satisfaisabilité d'une conjonction de telles contraintes est un problème NP-complet. Nous proposons une analyse en $\mathcal{O}(n^4)$ également pour ce cas.

Au cœur des analyses du contenu des tableaux, on trouve aussi des analyses de partitionnement symbolique. Pour une boucle « **for** $i = 1$ **to** $n$ » où un tableau est accédé à la cellule $i$, il est nécessaire de considérer le contenu des tableaux sur les tranches $[1, i-1], [i, i]$ et $[i+1, n]$ pour être précis. Nous définissons une analyse de partitionnement sémantique, puis une analyse du contenu des tableaux basée sur ses résultats. Cette dernière associe à chaque tranche $\varphi$ une propriété $\psi$ dont les variables représentent le contenu des tableaux sur cette tranche. La propriété $\psi$ est interprétée cellule-par-cellule, ainsi pour $\varphi = [1, i-1]$ et $\psi = (a = b + 1)$ il est exprimé que $\forall \ell \in [1, i-1], a[\ell] = b[\ell] + 1$.

Les résultats expérimentaux montrent que notre analyse automatique est efficace et précise, sur une classe de programmes simples : tableaux unidimensionnels, indexés par une variable au plus ($x + c$ ou $c$), traversés par des boucles, imbriquées ou non, avec des compteurs suivant une progression arithmétique. Elle découvre par exemple que le résultat d'un tri par insertion est un tableau trié, ou que durant le parcours d'un tableau gardé par une « sentinelle », tous les accès à ce tableau sont corrects.

*Mots clefs :* analyse statique, interprétation abstraite, non-égalités numériques, contenu des tableaux.

# Contributions to the Static Analysis of Programs Handling Arrays

**Abstract**

Array bound checking has been widely studied. However, there are very few convincing results about array contents analysis.

For such an analysis, numerical analyses are fundamental. In particular, when assigning $a[i]$, knowledge about $a[j]$ is kept unchanged if the invariant $i \neq j$ is discovered. We propose a new weakly relational numerical analysis, combining potential constraints ($x - y \leq c$ or $\pm x \leq c$) with disequalities ($x \neq y$ or $x \neq 0$). If the variables are valued in a dense set, the analysis runs in $\mathcal{O}(n^4)$. In the arithmetic case, the satisfiability problem of the conjunction of such constraints is NP-complete. We propose an analysis with complexity $\mathcal{O}(n^4)$ for this case too.

In the core of array contents analyses, we also find symbolic partitioning analyses. To precisely analyze a loop "**for** $i = 1$ **to** $n$", in which an array is accessed at index $i$, slices $[1, i-1]$, $[i, i]$ and $[i+1, n]$ of arrays contents must be considered. We define a semantic partitioning analysis and then an array contents analysis based on its results. This later analysis binds to each slice $\varphi$ a property $\psi$ whose variables stand for the arrays contents of the slice. The properties $\psi$ are interpreted pointwise on the slice. So for $\varphi = [1, i-1]$ and $\psi = (a = b + 1)$, it means that $\forall \ell \in [1, i-1], a[\ell] = b[\ell] + 1$.

Experimental results show that our automatic analysis is efficient and precise on simple programs: one-dimensional arrays, indexed by one variable at most ($x + c$ or $c$), traversed by possibly nested "**for**" loops, the counters of which follow an arithmetic progression. The analysis is able, for instance, to discover that the result of an insertion sort is a sorted array, or that, in an array traversal guarded by a "sentinel", the index stays within the bounds.

*Key words:* static analysis, abstract interpretation, numerical disequalities, array contents.