Analyse statique de programmes manipulant des tableaux

Valentin Perrelle

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Valentin Perrelle

Thèse dirigée par **Nicolas Halbwachs**

préparée au sein du laboratoire **Verimag**
et de l'école doctorale **Mathématiques, Sciences et Technologies de l'Information, Informatique**

Analyse statique de programmes manipulant des tableaux

Thèse soutenue publiquement le **21 Février 2013**,
devant le jury composé de :

M. Jean Claude Fernandez
Professeur Université Joseph Fourier, Président
M. Ahmed Bouajjani
Professeur Université Paris Diderot, Rapporteur
M. Xavier Rival
Chargé de Recherche INRIA Paris-Rocquencourt, Rapporteur
M. Bertrand Jeannet
Chargé de Recherche INRIA Rhône-Alpes, Examinateur
M. Francesco Logozzo
Senior scientist Microsoft Research, Examinateur
M. Nicolas Halbwachs
Directeur de recherche Verimag, Directeur de thèse

Remerciements

Je remercie l'ensemble des membres de mon jury de thèse pour m'avoir fait l'honneur d'accepter d'évaluer mes travaux. Je les remercie de l'intérêt qu'ils y ont porté et pour les discussions que nous avons pu avoir à ce sujet. Je remercie Xavier Rival de m'avoir envoyé ses notes détaillées à propos de mon manuscrit et d'avoir accepté de me recevoir pour en discuter.

Je remercie chaleureusement mon Directeur de Thèse, Nicolas Halbwachs, pour l'opportunité qu'il m'a offerte, pour ce merveilleux sujet de recherche, et pour les années où nous avons travaillé ensemble. Je suis très reconnaissant de la confiance et de la très grande liberté qui m'ont été données. Je suis très reconnaissant aussi pour le soutien inconditionnel qui m'a été accordé, et sans lequel ces travaux n'auraient jamais pu être menés à terme.

Je remercie Mathias Péron pour la qualité de ses travaux de thèse, son prototype d'analyseur et pour le manuscrit très complet qu'il a rédigé. Ses travaux sont à l'origine de ma thèse et j'ai pu à de nombreuses reprise en constater la grande valeur. Je le remercie également pour son agréable complicité.

Je remercie les membres de l'équipe Synchrone pour l'ambiance de travail qu'ils maintiennent, et pour les séminaires agités qu'ils animent et qui laissent un souvenir intarissable. Je remercie Florence Maraninchi de me l'avoir fait découvrir en m'offrant le stage qui a été ma première expérience dans l'univers de la recherche académique.

Je remercie les membres du Bureau 39, Giovanni, Nicolas, Romain et Tayeb pour les moments passés ensemble et pour les discussions enrichissantes sur des sujets variés. Je les remercie pour ce qui a fait la vie de ce bureau, le partage, la découverte et l'entraide.

Enfin je remercie ma famille d'avoir été présente pendant ces années, et d'avoir fait le déplacement pour s'assurer du bon déroulement de ma soutenance et que personne n'y manquât de camembert de Normandie ni d'andouille de Vire.

Table des matières

1 Introduction

Concevoir des logiciels sans erreur a toujours été un défi ardu. Cela est vrai même de programmes simples et la complexité des logiciels ne cesse de croître. Il n'est pas nécessairement juste de blâmer le programmeur. Si c'est bien le produit de son travail qui contient les « bugs » et si ses compétences ont une influence directe sur leur quantité, l'expérience montre que cela est inéluctable : un programmeur, fût t-il reconnu parmi les meilleurs ne saura développer un logiciel simplement dépourvu d'erreur. Le génie logiciel n'est pas parvenu à décrire une méthode de développement qui puisse assurer l'absence d'erreur dans un logiciel. Tout au plus permet-il à travers des méthodes et des préceptes d'organisation d'en réduire la quantité. A défaut de pouvoir garantir l'absence de bugs, on essaiera de les trouver pour pouvoir les éliminer avant le déploiement du logiciel. On soumettra les logiciels à des batteries de tests dont la conception elle-même nécessite une méthodologie adaptée. Le logiciel peut avoir une spécification écrite, qu'il faudra vérifier à travers tout un processus de validation.

Ce n'est pas suffisant pour le *logiciel critique*. Pour ces logiciels, les conséquences d'un dysfonctionnement peuvent être économiquement lourdes, l'accident le plus connu du grand public étant celui de la fusée Ariane 5 [La96]. Les répercussions d'une erreur logicielle ne se limitent pas seulement à des conséquences financières et peuvent mettre en danger la vie des individus. [1] Pour ces systèmes critiques, il faut des garanties très fortes de la *correction* du système.

Ce problème a fait l'objet du développement de méthodes formelles. Ces méthodes s'appuient sur la définition de modèles mathématiques permettant définir le comportement des programmes afin de prouver que leur exécution ne peut jamais être erronée et est conforme à la spécification. Cette preuve ne peut être obtenue par les méthodes de tests, qui ne peuvent conclure qu'à l'absence d'erreurs dans les cas particuliers testés. À l'inverse, les méthodes formelles garantissent la correction du logiciel en toute situation.

Il y a d'abord toute une classe de méthodes formelles *déductives*. Il s'agit d'écrire une preuve mathématique de la correction du programme. Le travail à fournir peut vite devenir excessif et on doit utiliser des techniques permettant de générer automatiquement des parties de cette preuve ou encore faire appel à des démonstrateurs automatiques pour décharger certaines *obligations de preuves*. Ces approches restent généralement très coûteuses et nécessitent l'intervention d'un utilisateur qualifié.

Une autre approche consiste à aborder le problème de manière algorithmique et de proposer des méthodes automatiques permettant de prouver la correction des programmes. Ces méthodes se heurtent à l'indécidabilité de cette correction, impliquée par le théorème de Rice [Ric53] : il est impossible de concevoir un algorithme qui, en un temps fini, puisse décider si un programme est correct ou non. En revanche, il est possible de concevoir un algorithme qui tente de répondre à la question pour un sous-ensemble aussi grand que possible de programmes. Même si ces méthodes échouent pour un programme particulier, elles peuvent en donner une preuve partielle qui pourra être complétée par des méthodes déductives.

1. [Mon09a] cite notamment une série d'accidents dus à des dysfonctionnements de systèmes médicaux ou militaires.

Parmi ces approches, on trouve notamment le *model-checking* [QS82, CES86], *l'interprétation abstraite* [CC77, CC92] ainsi que des procédures de décision pour des fragments de *logiques décidables*. Il s'agit de méthodes d'*analyse statique* qui ne requièrent pas l'exécution du programme contrairement aux méthodes de tests et aux autres méthodes *d'analyse dynamique* qui observent l'exécution des programmes. En dehors des enjeux de la vérification de programme, un avantage de ces méthodes automatiques est qu'elles permettent de trouver les erreurs très tôt dans le processus de développement. Trouver les erreurs tôt permet d'identifier plus rapidement d'éventuels problèmes de conception, de réduire la durée de développement en diminuant le temps qui sépare l'introduction du bug et sa correction et de limiter le coût du test.

L'interprétation abstraite permet comme son nom l'indique d'analyser un programme en l'interprétant à l'aide d'une approximation correcte de sa sémantique. Une telle interprétation repose sur la définition d'un *domaine abstrait* qui délimite une classe de propriétés à rechercher. Il existe dans la littérature tout un ensemble de domaines abstraits qu'il est possible de combiner pour embrasser une large classe de propriétés. Le temps nécessaire à une analyse par interprétation abstraite dépend en grande partie du choix du domaine abstrait.

Une des caractéristiques de l'interprétation abstraite, c'est qu'elle permet de *découvrir* des propriétés sur les programmes et pas seulement de les prouver. En pratique, cela signifie qu'il n'est pas nécessaire qu'on lui donne un objectif de preuve. Elle se contente de chercher les propriétés les plus fortes qu'elle peut découvrir d'un programme. Cela en fait une méthode automatique nécessitant peu d'intervention de l'utilisateur, adéquate pour d'autres applications. On peut notamment citer son application à la recherche d'optimisations à réaliser dans la compilation d'un programme et la génération automatique de tests.

1.1 L'interprétation abstraite de programmes manipulant des tableaux

Cette thèse s'intéresse à l'analyse de programmes manipulant des tableaux. Le problème de la vérification des indices d'accès aux tableaux a déjà été largement traité, étant notamment l'un des objectifs premiers de l'interprétation abstraite. On ne s'inquiètera donc pas de savoir si les expressions d'indices utilisées dans les accès aux tableaux sont bien dans le domaine de définition du tableau. En revanche, on se focalisera sur la découverte de propriétés à propos des contenus des tableaux.

Si les langages de haut niveau ont tendance à délaisser les tableaux au profit d'abstractions de plus haut niveau, l'intérêt d'une telle analyse reste grand. Outre la base importante de logiciels déjà écrits, on continue d'écrire des programmes manipulant des tableaux. D'abord pour l'efficacité des accès indexés, utiles à la manipulation des vecteurs et des matrices, ou encore mis à profit pour le stockage des solutions des sous-problèmes en programmation dynamique [Bel52]. Cette qualité et les bonnes propriétés de *localité* qu'ils offrent en font des éléments majeurs de l'implémentation efficace des types abstraits de données. Par ailleurs, une collection permettant l'accès indexé, quand bien même n'est elle pas implémentée avec des tableaux, en partagera la complexité. Il est probable qu'on puisse réutiliser les techniques d'analyse de programme manipulant des tableaux pour ceux utilisant de telles collections. Enfin, citons le cas des tableurs, logiciels permettant de manipuler des feuilles de calcul, et apportant parfois des langages de programmation pour automatiser certains traitement ou réaliser des calculs complexes. Ces langages de programmation manipulent les feuilles de calcul comme des tableaux bi-dimensionnels

et ont déjà été l'objet de travaux en interprétation abstraite [CR12].

Nous abordons le sujet de l'analyse statique de programmes manipulant des tableaux en nous reposant sur les travaux déjà réalisés autour de cette question dans le domaine de l'interprétation abstraite [GDD+04, GRS05, JGR05, HP08, GMT08, CCL11]. Nous rappelons quatre aspects des méthodes d'analyse développées dans ces travaux.

Le partitionnement des tableaux. Il n'est pas rare qu'une preuve « à la main » d'un programme manipulant des tableaux requière que l'on considère différents *fragments* d'un tableau afin d'en distinguer des caractéristiques. Considérons par exemple le programme ci-contre, calculant le maximum d'un tableau de nombres. Pour exprimer l'invariant de la boucle, on partitionnera le tableau en deux fragments : les cellules d'indice strictement inférieur à i et celles d'indice supérieur ou égal à i. L'invariant dira que les premières ont une valeur inférieure ou égale à celle de la variable *max* mais ne dira rien des secondes.

```
max ← A[1]
Pour i de 2 à n faire
    Si A[i] > max alors
        max ← A[i]
```

La transposition d'une preuve d'un tel invariant en une preuve par interprétation abstraite va également nécessiter qu'on ait choisi une collection adéquate de fragments desquels on va chercher les propriétés. Le plus souvent, cette collection est un partitionnement des tableaux. Les méthodes utilisées pour les choix de ces fragments sont variées. Il peut s'agir d'un paramètre de l'analyse donné par l'utilisateur ou bien elles peuvent être calculées avant l'interprétation abstraite d'un programme [GRS05, HP08] ou encore être adaptées au fil de l'analyse [GMT08, CCL11].

Les propriétés relationnelles. L'invariant du programme précédent peut être exprimé par la formule

$$\forall \ell,\ 1 \leq \ell < i \Rightarrow A[\ell] \leq max$$

La propriété « le tableau est trié » quant à elle peut s'énoncer par une formule

$$\forall \ell_1, \ell_2,\ 1 \leq \ell_1 \leq \ell_2 < n \Rightarrow A[\ell_1] \leq A[\ell_2]$$

La plupart des méthodes d'analyse citées manipulent des formules semblables aux deux précédentes et dont la forme générale est la suivante.

$$\forall \ell_1, \ldots, \ell_n,\ \varphi(\mathcal{V}, \ell_1, \ldots, \ell_n) \Rightarrow \psi(\mathcal{V}, A_1[\ell_1], \ldots, A_n[\ell_n])$$

où $\mathcal{V}$ est l'ensemble des variables du programme, φ et ψ deux formules, et $A_1, \ldots, A_n$ des symboles de tableaux. Lorsque la sous formule ψ est unaire, on dira qu'il s'agit d'une *propriété non-relationnelle*. A l'inverse, quand ψ fait intervenir plusieurs variables ou cellules de tableaux, on parle de *propriété relationnelle*.

La réutilisation des domaines abstraits. L'analyse par interprétation abstraite des programmes manipulant des tableaux a tout intérêt à tirer profit de tout le travail déjà accompli sur les domaines abstraits. Si l'interprétation abstraite « classique » est capable de découvrir une certaine classe de propriétés impliquant des variables scalaires, nous aimerions pouvoir découvrir les mêmes propriétés sur des cellules de tableau. On cherche donc à réutiliser les domaines abstraits et à les étendre pour qu'ils puissent aussi décrire des propriétés de contenu de tableaux.

Là encore, il y a diverses méthodes. Certains travaux se contentent de ne considérer que des domaines abstraits non relationnels [CCL11]. D'autres s'emploient à rechercher toutes les relations qu'un domaine abstrait peut exprimer sur l'ensemble des fragments de tableaux [GDD+04]. Malheureusement, leur méthode ne peut fonctionner que si les fragments sont non-vides quelle que soit la situation. Pour reprendre l'exemple du programme calculant l'indice du maximum, l'ensemble des cellules d'indice supérieur ou égal à 1 mais inférieur strictement à i est vide lorsque i est inférieur ou égal à 1. Le cas échéant, il sera nécessaire de conduire deux analyses en parallèle l'une pour les cas où le fragment est vide, l'autre pour le cas où il contient toujours au moins une cellule. La distinction devant être faite pour chaque fragment, le nombre de cas à considérer est exponentiel. Enfin, la méthode proposée dans [HP08] peut travailler avec des fragments dont la vacuité dépend de la situation, mais ne traite que des relations entre des fragments de même tailles [2].

Phénomène d'alias Dans un programme manipulant des tableaux, deux expressions pourtant différentes d'un accès à un tableau peuvent désigner la même cellule. Les expressions $A[i]$ et $A[j]$ peuvent faire référence à la même cellule si $i = j$. On dit que ces expressions sont des *alias* l'une de l'autre. Ce phénomène complique notablement l'analyse de ces programmes. Les solutions apportées à ce problème sont similaires à celles apportées aux problèmes de fragments vides : on conduira des analyses parallèles pour chacun des cas d'alias, leur nombre pouvant encore être exponentiel. Pour réduire le nombre de cas potentiels, on pourra tirer profit d'une analyse préalable ou simultanée portant sur les variables d'indices. Il existe d'ailleurs des domaines abstraits dédiés à ce problème, comme le domaine des zones précisant alias [PH07].

1.2 Contribution

Nos travaux développent la théorie de l'interprétation abstraite de programmes manipulant des tableaux suivant trois axes.

- **Le traitement des phénomènes d'alias.** Nous proposons une méthode permettant traiter les programmes dans lesquels il y a des alias et qui ne nécessite pas pour autant l'énumération systématique des différents cas. Cette méthode repose sur la définition d'une nouvelle abstraction des collections de fragments.
- **Le choix des fragments de tableaux.** Nous présentons une nouvelle manière de choisir les fragments de tableaux. L'objectif est de combiner les qualités des deux classes de critères : ceux qui définissent les fragments avant l'analyse et ceux qui les définissent pendant.
- **L'extension des domaines abstraits.** Notre théorie constitue une troisième alternative pour la réutilisation des domaines abstraits dans l'analyse de programmes manipulant des tableaux. Cette solution permet l'expression de propriétés entre des fragments de tailles différentes et ne nécessite pas de distinguer les cas de vacuité mais implique un travail supplémentaire d'adaptation du domaine abstrait.

En outre, nous avons implémenté les méthodes présentées dans cette thèse dans un prototype d'analyseur statique.

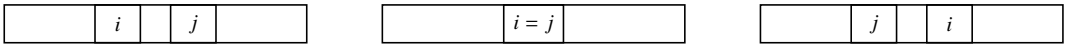
2. Ici, de même taille signifie que le nombre de cellules dans ces fragments est le même quelles que soient les valeurs des variables du programme. Par exemple, le fragment constitué de l'ensemble des cellules d'un tableau dont l'indice est compris entre 1 et i et celui constitué des cellules d'indice compris entre 2 et $i + 1$ contiennent toujours le même nombre de cellules quelle que soit la valeur de i.

1.2.1 Le traitement des phénomènes d'alias

Le traitement classique des phénomènes alias revient à énumérer les configurations d'alias. Cette solution a l'avantage de la simplicité, mais introduit un coût exponentiel dans l'analyse.

Considérons l'exemple ci-contre d'une fonction vérifiant s'il existe dans un tableau deux cellules de même valeur. Cet exemple n'est certes pas optimisé et la seconde boucle pourrait se contenter de ne parcourir que les cellules d'indice compris entre 1 et i. L'auteur du programme a néanmoins tout à fait le droit de proposer cette solution. Pour représenter la partition du tableau et pour distinguer les cas d'alias, on considèrera séparément les situations $i < j$, $i = j$ et $i > j$. Pour chacune de ces situations, on peut considérer plusieurs fragments de tableau : les fragments singletons contenant $A[i]$ et $A[j]$, le fragment des cellules entre ces deux singletons et les fragments des cellules positionnées avant et après les singletons. On pourra ensuite parler de la ou des tranches situées avant l'indice i et dire que toutes leurs valeurs sont distinctes, ou encore parler de la ou des tranches situées avant l'indice j et dire que leurs valeurs diffèrent de celle de $A[i]$.

```
Pour i de 1 à n faire
    Pour j de 1 à n faire
        Si A[i] = A[j] ∧ i ≠ j alors
            Retourner Vrai
Retourner Faux
```



Pour prouver ces propriétés, il n'est néanmoins pas nécessaire de distinguer autant de fragments et autant de situations. Il suffirait que l'on considère quatre fragments : celui des cellules d'indice inférieur à i, celui des cellules d'indice inférieur à j et pour interpréter le test les fragments singletons $A[i]$ et $A[j]$. Finalement, la distinction des cas d'alias n'est pas nécessaire et quatre fragments f_1, f_2, f_3, f_4 suffisent :

$$f_1 = \{A[\ell] \mid 1 \leq \ell \leq i\} \qquad f_2 = \{A[\ell] \mid 1 \leq \ell \leq j\}$$
$$f_3 = \{A[i]\} \qquad f_4 = \{A[j]\}$$

On pourra dire des cellules de f_1 que leurs valeurs sont différentes de celles des cellules f_2 et si la fonction retourne *Vrai* on pourra également dire que la cellule de f_3 a la même valeur que la cellule de f_4.

Le pas à franchir pour diminuer ainsi le nombre de fragments est d'abandonner le découpage des tableaux en partitions. On a simplement une collection de fragments qui d'une part ne recouvrent pas nécessairement l'ensemble du tableau, et d'autre part peuvent se chevaucher. Dans l'exemple précédent, les fragments f_1 et f_2 peuvent contenir les mêmes cellules.

Dans la littérature, les fragments sont souvent représentés par des valeurs abstraites. Pour des propriétés de la forme

$$\forall \ell_1, \ldots, \ell_n,\ \varphi(\mathcal{V}, \ell_1, \ldots, \ell_n) \Rightarrow \psi(\mathcal{V}, A_1[\ell_1], \ldots, A_n[\ell_n])$$

il est classique d'utiliser une valeur abstraite pour représenter φ. Il est alors possible d'utiliser les opérateurs des domaines abstraits pour calculer des intersections ou des unions de fragments. Néanmoins, comme le font remarquer les auteurs de [GMT08], les domaines abstraits ont été développés pour des calculs de sur-approximation. Or pour que la formule précédente soit vraie, il faut que φ soit effectivement une sous-approximation des fragments vérifiant la propriété ψ.

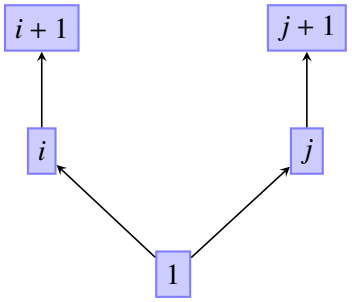
Ils proposent donc une méthode systématique permettant d'adapter les opérateurs des domaines abstraits pour pouvoir produire des sous-approximations.

Notre première approche a suivi cette idée, mais n'a pas été fructueuse. Nous avons d'abord spécialisé l'algorithme proposé dans [GMT08] au domaine abstrait des zones [Dil89] dans le but d'améliorer la précision. Lorsque cette sous-approximation est effectivement nécessaire, un autre problème se pose : il n'y a pas en général de meilleure sous-approximation des opérateurs des domaines abstraits, mais il y en a plusieurs maximales et aucune raison a priori d'en choisir une particulière. Une seconde approche a été d'abandonner l'usage des domaines abstraits pour concevoir des abstractions de fragments pour lesquels la meilleure sous-approximation des opérateurs existait. Il s'agissait d'abstractions de fragments de la forme

$$\{\, A[\ell] \mid \min\{\, x_i + c_i \mid x_i \in \mathcal{V}, c_i \in \mathbb{Z} \,\} \leq \ell \leq \max\{\, y_i + d_i \mid y_i \in \mathcal{V}, c_i \in \mathbb{Z} \,\} \,\}$$

Ces deux approches n'ont pas été concluantes et ne seront plus évoquées dans le reste de cette thèse. Néanmoins, nous fournirons les éléments nécessaires à la compréhension de leur échec.

Notre troisième tentative peut être vue comme une généralisation des approches de la littératures basées sur un partitionnement des tableaux. Nous représentons nos collections de fragments de tableaux par des graphes dont les arcs désignent les fragments et les sommets des jalons délimitant ces fragments et nous appelons ces graphes des *diagrammes de tranches*. Ainsi, le graphe suivant représenterait l'ensemble des quatre fragments évoqués pour l'exemple précédent. Chaque arc représente l'ensemble des cellules d'indice compris entre la valeur associée au sommet de départ et celle associée au sommet de destination.



Nous proposons pour cette représentation des ensembles de fragments les algorithmes qui permettent leur manipulation. Cette solution permet de se rapprocher de la précision qu'on obtient par le partitionnement des tableaux tout en assurant une complexité polynomiale. Elle n'atteint toutefois pas toujours cette précision et la distinction des cas d'alias reste parfois nécessaire.

Pour pouvoir être appliquée, elle requiert qu'il soit possible d'extraire de la valeur abstraite les éventuelles inégalités entre les jalons. En pratique, on utilisera donc un domaine abstrait produit

- d'un domaine pouvant exprimer les propriétés que l'on cherche à découvrir sur les contenus et
- d'un domaine pouvant exprimer les inégalités entre les jalons.

1.2.2 Le choix des fragments de tableaux.

Avoir une représentation pour les collections de fragments ne dispense pas de développer la question du choix de ces fragments. Nous avons donc tenté d'aborder le sujet sans mélanger les deux problèmes. Notre solution s'inspire essentiellement des propositions de [HP08] et

[GMT08]. Les premiers proposent de partitionner le tableau selon un critère syntaxique. Une série de règles définit comment, à partir du code source du programme, on doit choisir les fragments. Il s'agit d'heuristiques produisant des choix de fragments adaptés pour beaucoup de programmes et algorithmes classiques sur les tableaux. La nature syntaxique du critère le rend néanmoins rigide et deux programmes ayant la même sémantique mais pas la même syntaxe ne donneront pas nécessairement les même fragments.

A l'inverse, les méthodes qui consistent à choisir les fragments tout au long de l'analyse [GMT08, CCL11] tirent profit de l'information acquise pendant l'analyse pour réaliser ce choix. Leur idée générale est de collecter les cellules à mesure qu'elles sont manipulées et de les regrouper. Elles peuvent être regroupées parce qu'elles ont des propriétés similaires ou simplement parce qu'elles sont adjacentes. Quelle que soit la méthode, il s'agit toujours d'heuristiques.

Nous proposons un nouveau critère sémantique constituant une troisième alternative pour le regroupement des cellules. L'idée est de regrouper les cellules lorsqu'elles sont manipulées par la même instruction. Elle repose sur le pari que les différentes cellules affectées par une même instructions le sont de la même manière et qu'on peut donc essayer de chercher ce que ces cellules ont en commun. Ainsi, nos heuristiques produisent des choix de fragments similaires à ceux de [HP08] mais sans les défauts d'un critère syntaxique.

Nous construisons ces choix de fragments par une instrumentation du programme à analyser. De cette manière, il est possible d'utiliser notre méthode quelle que soit la représentation des fragments, qu'il s'agisse d'une partition ou de nos diagrammes de tranches. Toutefois, dans sa version la plus précise, notre méthode peut amener à considérer de nouveaux cas d'alias et pourra tirer profit de la représentation offerte par les diagrammes.

1.2.3 L'extension des domaines abstraits.

Nous proposons une nouvelle formalisation des correspondances de Galois entre les valeurs abstraites et concrètes permettant d'étendre les domaines abstraits à l'analyse de programmes manipulant des tableaux. Il s'agit d'une généralisation de la formalisation déjà proposée par [GDD+04, JGR05]. Son objectif est de permettre à la fois

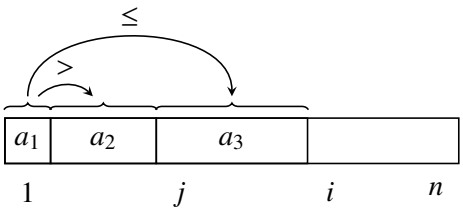
- la manipulation de fragments potentiellement vides,
- l'expression de relations entre des fragments de tailles différentes et
- une définition des fragments dépendante de la valeur des variables du programme.

Pour comprendre le problème que posent les fragments potentiellement vides, observons l'exemple ci-contre de la segmentation, étape nécessaire du tri par segmentation.

```
i ← 2
j ← 2
Tant que i < n faire
    Si A[i] < A[1] alors
        A[i] ↔ A[j]
        j ← j + 1
    i ← i + 1
A[j − 1] ↔ A[1]
```

Pour prouver la correction de cet exemple, il suffit de considérer trois fragments : la cellule $A[1]$, les cellules d'indice compris entre 2 et $j-1$ et les cellules d'indice compris entre j et $i-1$. Le premier est un singleton et les deux autres peuvent être vides : le second est vide lorsque $j = 2$ et le troisième quand $i = j$. L'invariant de la boucle de ce programme implique en particulier que les cellules du second fragment ont une valeur inférieure au pivot $A[1]$ et que celles du troisième fragment en ont une valeur supérieure. Si on associe des variables a_1, a_2 et a_3 à ces fragments, l'invariant peut s'exprimer de la manière suivante :

$$a_2 < a_1 \wedge a_1 \leq a_3$$



Il s'agit d'une inégalité qui peut être représentée par de nombreux domaines abstraits numériques. Notre méthode permet de transformer une valeur abstraite exprimant cette inégalité en une valeur abstraite l'exprimant sur des cellules de tableaux :

$$\forall \ell_1, \ell_2,\ (\ell_1 = 1 \wedge 2 \leq \ell_2 < j) \Rightarrow (A[\ell_2] < A[\ell_1])$$
$$\wedge \forall \ell_1, \ell_3,\ (\ell_1 = 1 \wedge j \leq \ell_3 < i) \Rightarrow (A[\ell_1] \leq A[\ell_3])$$

La nouveauté par rapport aux méthodes précédentes est que les implications viennent s'insérer pour chaque atome de la formule. En effet, si le résultat de la transformation avait été la formule

$$\forall \ell_1, \ell_2, \ell_3,\ (\ell_1 = 1 \wedge 2 \leq \ell_2 < j \wedge j \leq \ell_3 < i) \Rightarrow (A[\ell_2] < A[\ell_1] \wedge A[\ell_1] \leq A[\ell_3])$$

on aurait obtenu une propriété beaucoup moins forte. Cette dernière propriété ne nous apprend rien lorsque l'un des deux fragments représentés par a_2 ou a_3 est vide : la partie gauche de l'implication devient fausse et la formule n'apporte aucune information. Au contraire, notre transformation permet de conserver les propriétés entre a_1 et a_2 (respectivement a_3) quand le fragment représenté par a_3 (respectivement a_2) est vide.

En contrepartie, notre méthode ne peut utiliser directement les fonctions d'abstraction et de concrétisation des domaines abstraits. Elle nécessite qu'elles soient adaptées. Concrètement, cela signifie essentiellement que dans ces domaines abstraits adaptés, les variables représentant des fragments potentiellement vides soient distinguées des autres.

1.3 Plan du mémoire

Le chapitre 2 rappelle les principes de l'interprétation abstraites utiles à la compréhension de ce manuscrit. La définition de quelques domaines abstraits utilisés par la suite y est donnée.

Les chapitres 3 à 5 couvrent l'essentiel de la contribution théorique. Le chapitre 3 introduit notre formalisation de l'extension des domaines abstraits aux propriétés de tableaux. Le chapitre 4 décrit notre critère pour le choix des fragments de tableau ainsi qu'un cadre pour la définition des abstractions des collections de fragments. Enfin, le chapitre 5 définit les diagrammes de tranches, les algorithmes permettant de les manipuler. En outre, une section y est consacrée à l'application aux diagrammes du critère défini dans le chapitre précédent.

Les trois chapitres suivants traitent de l'intégration de notre contribution dans le domaine de l'interprétation abstraite. Le chapitre 6 adapte la méthode proposée dans [HP08] pour la recherche de relations entre des fragments qui sont des translations l'un de l'autre. Les chapitres 7 et 8 discutent des adaptations nécessaires à apporter aux domaines abstraits pour qu'ils puissent respectivement synthétiser des propriétés sur les contenus des cellules des fragments et des propriétés sur les ensembles des cellules des fragments.

Pour finir, le chapitre 9 est consacré aux travaux connexes et le chapitre 10 présente le prototype et les expérimentations réalisés.

2 L'interprétation abstraite

L'*interprétation abstraite* est un domaine de recherche crée par Patrick Cousot et Radhia Cousot [CC77] dont l'application pratique concerne l'analyse statique de programmes. Nous rappelons dans ce chapitre quelques bases de la théorie nécessaires à la compréhension de cette thèse.

2.1 Modèles de programmes

Pour décrire les méthodes d'analyse des programmes, nous aurons d'abord besoin d'une représentation abstraite de ces programmes. Nous utiliserons ici un modèle classique en interprétation abstraite reposant sur les automates interprétés. Il s'agit d'un modèle général, indépendant du langage de programmation utilisé pour l'écriture du programme.

Nos automates interprétés seront des graphes dont les sommets sont les points de contrôle du programme et les arcs sont les transitions entre ces différents points de contrôle. Les arcs seront étiquetés avec des commandes gardées qui sont la donnée

- d'une garde, c'est à dire d'une formule qui doit être vérifiée pour que la transition soit franchissable ainsi que
- d'une action, indiquant l'effet de la transition.

On distingue en outre un sommet représentant le point de contrôle initial.

Avant de définir ces concepts plus formellement, nous introduisons quelques notations. Notons $\mathcal{V}$ l'ensemble des variables du programme. On appelle *état* une fonction σ des variables du programme vers l'ensemble $\mathbb{V}$ des valeurs qu'elles peuvent prendre. On note Σ l'ensemble des états.

$$\Sigma = \mathcal{V} \to \mathbb{V}$$

Définition 1 (Commande gardée)**.**
Soit K un ensemble de points de contrôle. Une commande gardée est la donnée d'un quadruplet (k, g, a, k') où

- $k, k' \in K^2$ sont les points de contrôle respectivement origine et destination de la commande ;
- $g : \Sigma \to \{Vrai, Faux\}$ est une garde qui à tout état $\sigma \in \Sigma$ associe *Vrai* si σ la vérifie, *Faux* sinon ;
- $a : \Sigma \to \Sigma$ est une action qui à tout état $\sigma \in \Sigma$ associe son état successeur.

Définition 2 (Automate interprété)**.**
Un automate interprété est la donnée d'un triplet (K, τ, k_{init}) :

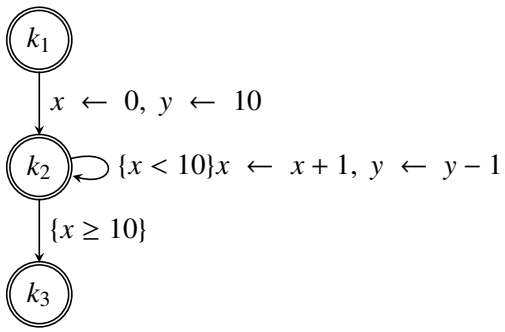
- K est l'ensemble des points de contrôle,
- τ est un ensemble de commandes gardées sur K,
- $k_{init} \in K$ est le point de contrôle initial.

Exemple 1 (Automate interprété)**.**
Considérons le programme suivant.

$x \leftarrow 0$
$y \leftarrow 10$
Tant que $x < 10$ **faire**
  $x \leftarrow x + 1$
  $y \leftarrow y - 1$

On construit pour ce programme un automate interprété. On distingue trois points de contrôles $k_1 = k_{init}$, k_2 et k_3 : le début du programme, la tête de boucle et la fin du programme respectivement. Entre ces points de contrôle, on a trois transitions :

1. la transition de k_1 à k_2 sans garde mais pour action l'affectation de 0 à x et 10 à y,
2. la transition de k_2 à lui même, avec la garde $x < 10$ et pour action l'incrémentation de x et y et
3. la transition de k_2 vers k_3 avec la garde $x \geq 10$ et aucune action.



L'exécution d'un programme peut être simulée sur l'automate interprété correspondant. Une exécution correspond à une succession de couples de la forme (k, σ) signifiant que l'exécution du programme en est au point de contrôle k dans l'état σ. Un pas de simulation correspondra à une transition dans l'automate interprété. On introduit la relation $\rightarrow$ permettant de signifier qu'on peut passer d'un couple à un autre en un pas de simulation.

$$(k, \sigma) \rightarrow (k', \sigma') \Leftrightarrow \exists g, a, \begin{cases} & (k, g, a, k') \in \tau \\ \wedge & g(\sigma) = Vrai \\ \wedge & \sigma' = a(\sigma) \end{cases}$$

Toute exécution commence au point de contrôle k_{init} avec n'importe quel état de $\sigma_{init} \in \Sigma$. On dira qu'un état σ est *atteignable* en k si $(k_{init}, \sigma_{init})$ est relié à (k, σ) par la clôture transitive et réflexive de $\rightarrow$, c'est à dire si et seulement si

$$\exists \sigma_{init} \in \Sigma,\ (k_{init}, \sigma_{init}) \rightarrow^* (k, \sigma)$$

Les ensembles d'états atteignables vérifient des équations qu'il est intéressant de poser. Les états accessibles à un point de contrôle k_i (autre que k_{init}) viennent des transitions dont la destination est k_i. Il n'y en a ni plus ni moins. Autrement dit, l'ensemble des états X_i au point de

contrôle k_i est l'union des états venant des transitions. Les états qui viennent d'une transition (k_j, g, a, k_i) quant à eux sont ceux de l'ensemble X_j des états atteignables en k_j, vérifiant la garde g et modifiés par a. On notera

$$g(X) = \{ \sigma \in X \mid g(\sigma) = Vrai \}$$

l'ensemble des états de X vérifiant g et

$$a(X) = \{ a(\sigma) \mid \sigma \in X \}$$

l'ensemble des images par a des états de X. Les ensembles d'états atteignables vérifient

$$X_{init} = \Sigma \ \wedge \bigwedge_{k_i \neq k_{init}} \left(X_i = \bigcup_{(k_j,g,a,k_i)\in\tau} (a \circ g)(X_j) \right)$$

Il s'agit d'un système d'équations de point fixe : en posant $X = (X_i)_i$, la formule précédente peut s'écrire sous la forme $X = f(X)$. Cette fonction f est monotone pour l'ordre d'inclusion et le théorème de point fixe de Tarski [Tar55] nous assure de l'existence d'un plus petit point fixe pour cette équation.

2.2 Principes de l'Interprétation abstraite

L'objectif principal de l'interprétation abstraite est de calculer les ensembles d'états atteignables, c'est à dire de résoudre l'équation de point fixe. Pour comprendre les problèmes que suscite cette résolution, considérons d'abord une première approche simple mais vouée à l'échec. On peut tenter de calculer les ensembles d'états atteignables par *interprétation* du programme. On construit itérativement les ensembles des états atteignables en chaque point de contrôle. On part de l'ensemble d'états Σ au point de contrôle initial, puis chaque fois qu'un état peut être obtenu en suivant une transition, on ajoute le nouvel état à l'ensemble d'états atteignables au bout de la transition. On continue d'ajouter les états atteignables, jusqu'à ce qu'ils ne soit plus possible d'en rajouter de nouveaux. On aura alors atteint un *plus petit point fixe*.

Cette approche est à peu de chose près l'application du théorème de Kleene [Kle52]. Si on part d'ensembles atteignables X vides, ce que nous noterons $X = \bot$, le théorème de Kleene nous dit que le plus petit point fixe $\mathrm{lfp}(f)$ de f s'obtient en appliquant f, éventuellement une infinité de fois :

$$\mathrm{lfp}(f) = \bigcup_{n\in\mathbb{N}} f^n(\bot)$$

En résumé, la *méthode d'itération de Kleene* consiste à partir d'un ensemble d'états vides en chaque point de contrôle puis à chaque itération d'appliquer f à ces ensembles d'états. On peut arrêter d'itérer une fois qu'on a atteint un point fixe.

Cette méthode souffre de deux problèmes.

- Les ensembles d'états atteignables sont en général infinis et même indénombrables. Il n'est donc pas toujours possible de tous les représenter. Même si les variables numériques d'un programmes sont souvent bornées, le nombre d'ensembles d'états reste bien trop grand, y compris sur des programmes très simples.

- Le point fixe de f peut ne pas être atteint en un nombre fini d'itérations. Il est possible que l'on puisse toujours ajouter de nouveaux états sans jamais s'arrêter, c'est à dire que la suite $(f^n(\bot))_n$ ne soit pas stationnaire. Par exemple quand le programme qu'on interprète possède une boucle infinie incrémentant une variable.

L'interprétation abstraite résout ces deux problèmes. Elle résout le premier en utilisant des abstractions des ensembles d'états atteignables et le second en utilisant des techniques d'extrapolation pour assurer que le nombre d'itérations soit toujours fini.

2.2.1 Domaines abstraits

Un domaine abstrait définit un ensemble d'abstractions d'ensembles d'états. Nous appellerons ces abstractions des *valeurs abstraites* et par symétrie, nous appellerons les ensembles d'états des *valeurs concrètes*.

Les valeurs abstraites doivent être plus simples que les valeurs concrètes, et en particulier avoir une représentation machine raisonnable. En contrepartie, toutes les valeurs concrètes n'auront pas toujours de valeur abstraite les représentant. A défaut, on devra se contenter d'une valeur abstraite approximant une valeur concrète. On ne voudra en général pas de n'importe quelle approximation, mais on voudra une sur-approximation : une abstraction représentant un ensemble d'états plus grand. C'est une condition capitale dans le domaine de la vérification, car si on peut prouver que tous les états d'une sur-approximation vérifient une propriété, alors le résultat vaudra pour tout ensemble plus petit, en particulier pour le véritable ensemble d'états atteignables.

Notons γ la fonction qui à une valeur abstraite $\widetilde{X}$ associe la valeur concrète $X = \gamma(\widetilde{X})$ qu'elle abstrait. On appelle cette fonction la *concrétisation*. Nous pouvons construire une fonction réalisant l'opération opposée, que nous appellerons l'*abstraction*, que nous noterons α et qui à toute valeur concrète X associera une valeur abstraite $\widetilde{X}$ qui la sur-approxime. Pour être correcte, cette fonction devra donc donner un état abstrait dont la concrétisation ne peut être que plus grande. Autrement dit, pour tout ensemble d'états X,

$$\gamma(\alpha(X)) \supseteq X$$

On aura besoin sur ces valeurs abstraites d'une relation d'ordre partielle $\sqsubseteq$. On requiert que cette relation sur-approxime la relation d'ordre entre les valeurs concrètes. Formellement, on veut

$$\widetilde{X} \sqsubseteq \widetilde{Y} \Rightarrow \gamma(\widetilde{X}) \subseteq \gamma\widetilde{Y}$$

Autrement dit, on veut que la fonction γ soit monotone.

Correspondances de Galois

La relation définie par les fonctions α et γ entre les valeurs concrètes ordonnées par $\subseteq$ et les valeurs abstraites ordonnées par $\sqsubseteq$ peut être modélisée par une *correspondance de Galois*.

Définition 3 (Correspondance de Galois)**.**
Une correspodance de Galois entre deux ensembles L et $\widetilde{L}$ ordonnés respectivement par $\subseteq$ et $\sqsubseteq$

est la donnée de deux fonctions monotones α et γ telles que pour tout X et tout $\widetilde{Y}$

$$\gamma(\alpha(X)) \supseteq X$$
$$\alpha(\gamma(\widetilde{Y})) \sqsubseteq \widetilde{Y}$$

ou de manière équivalente [1]

$$\alpha(X) \sqsubseteq \widetilde{Y} \Leftrightarrow X \subseteq \gamma(\widetilde{Y})$$

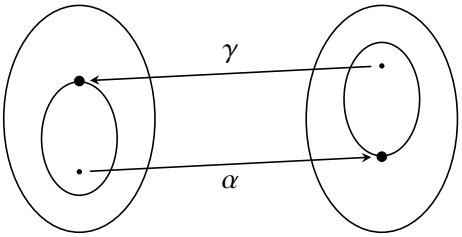
Pour que nos fonctions α et γ forment une correspondance de Galois, il faudra donc également exiger la monotonie de α, ainsi qu'une autre condition. Il peut arriver que deux valeurs abstraites $\widetilde{X}$ et $\widetilde{Y}$ aient la même concrétisation, c'est à dire que $\gamma(\widetilde{X}) = \gamma(\widetilde{Y})$. Pour avoir une correspondance de Galois, la fonction α doit toujours donner la plus petite des valeurs abstraites ayant la même concrétisation. Ceci sera vrai si et seulement si, chaque fois que l'on prend la concrétisation $\gamma(\widetilde{Y})$ d'une valeur abstraite $\widetilde{Y}$, alors la fonction d'abstraction α donnera forcément une valeur abstraite inférieure ou égale :

$$\alpha(\gamma(\widetilde{Y})) \sqsubseteq \widetilde{Y}$$

En ajoutant ces conditions aux conditions de la section précédente, on obtient une correspondance de Galois. Ces nouvelles conditions n'étant en général pas utiles, il existe d'autres modélisations plus légères, en particulier des formalisations n'utilisant que γ et ne faisant pas intervenir α. [CC92]

La correspondance de Galois fait toujours correspondre à un état concret la plus petite valeur abstraite qui le sur-approxime et fait toujours correspondre à une valeur abstraite la plus grande valeur concrète qu'il sur-approxime. Si on note L l'ensemble des valeurs concrètes et $\widetilde{L}$ l'ensemble des valeurs abstraites, ces plus grands états concrets et ces plus petits états abstraits sont respectivement $\alpha(L)$ et $\gamma(\widetilde{L})$. α et γ décrivent un isomorphisme d'ordre entre $\alpha(L)$ et $\gamma(\widetilde{L})$.

Nous pouvons résumer ces caractéristiques sur une figure simple. Dans la figure suivante, nous représentons L et $\widetilde{L}$ par les grands ovales. Les ovales plus petits représentent deux classes d'équivalences pour α et γ respectivement. Si on prend n'importe quel élément d'une de ces classes, représenté sur la figure par un petit point, son image par α et γ respectivement sera toujours le plus petit et le plus grand élément de la classe correspondante, représenté par un gros points. Si on considère l'ensemble des gros points pour toutes les classes d'équivalence, ils sont en bijection par la correspondance de Galois.



1. L'équivalence des deux définitions découle de la monotonie de α et γ.

Calcul de point fixe dans un domaine abstrait

Rappelons l'équation de point fixe qu'il faut résoudre :

$$X_{init} = \Sigma \wedge \bigwedge_{k_i \neq k_{init}} \left(X_i = \bigcup_{(k_j,g,a,k_i)\in\tau} (a \circ g)(X_j) \right)$$

L'usage d'abstraction permet de résoudre le premier problème rencontré à l'application des itérations de Kleene. A chaque point de contrôle k_i, on associe une valeur abstraite $\widetilde{X_i}$. A chaque itération, on doit calculer la nouvelle valeur des $\widetilde{X_i}$ en fonction des valeurs précédentes. Cela nécessite de pouvoir donner une abstraction de

$$\bigcup_{(k_j,g,a,k_i)\in\tau} (a \circ g)(\gamma(\widetilde{X_j}))$$

Pour réaliser ce calcul, on le décompose, quitte à perdre un peu en précision. On distingue l'effet de la garde, l'effet de l'action, et le calcul de l'union. On utilisera une fonction $\widetilde{g}$ qui à toute valeur abstraite $\widetilde{X_j}$ associe une valeur abstraite sur-approximant le résultat de l'application de la garde sur la valeur concrète correspondante. C'est à dire une fonction qui vérifie :

$$\widetilde{g}(\widetilde{X}) \sqsupseteq \alpha(g(\gamma(\widetilde{X})))$$

De la même manière on utilisera une fonction $\widetilde{a}$ sur-approximant l'effet de l'action et vérifiant

$$\widetilde{a}(\widetilde{X}) \sqsupseteq \alpha(a(\gamma(\widetilde{X})))$$

Enfin, il nous faut une sur-approximation $\sqcup$ de l'union, vérifiant donc

$$\widetilde{X} \sqcup \widetilde{Y} \sqsupseteq \alpha(\gamma(\widetilde{X}) \cup \gamma(\widetilde{Y}))$$

Notons que cette dernière inégalité est vérifiée si on prend $\sqcup$ comme la borne supérieure, si elle existe, pour l'ordre $\sqsubseteq$. C'est pourquoi, il sera commode d'organiser les valeurs abstraites en treillis ou demi-treillis. Lorsqu'il s'agit d'un treillis, on notera $\sqcap$ sa borne inférieure.

Enfin, il nous faut deux abstractions particulières, l'une représentant l'ensemble Σ de tous les états, et l'autre représentant l'ensemble vide d'état. Nous les noterons $\top$ et $\bot$ respectivement.

$$\alpha(\Sigma) = \top \qquad \gamma(\top) = \Sigma$$
$$\alpha(\emptyset) = \bot \qquad \gamma(\bot) = \emptyset$$

Il n'est pas toujours commode d'intégrer $\top$ et $\bot$ dans un domaine abstrait. On pourra se contenter d'étendre un domaine abstrait en lui ajoutant $\top$ et $\bot$ vérifiant les égalités ci-dessus et en étendant naturellement les opérateurs, pour tout $\widetilde{X}$ du domaine :

$$\bot \sqsubseteq \widetilde{X} \sqsubseteq \top$$
$$\widetilde{X} \sqcup \top = \top \qquad \widetilde{X} \sqcap \top = \widetilde{X}$$
$$\widetilde{X} \sqcup \bot = \widetilde{X} \qquad \widetilde{X} \sqcap \bot = \bot$$

Finalement, le système d'équations de point fixe devient

$$\widetilde{X}_{init} = \top \wedge \bigwedge_{k_i \neq k_{init}} \left(\widetilde{X_i} = \bigsqcup_{(k_j,g,a,k_i)\in\tau} (\widetilde{a} \circ \widetilde{g})(\widetilde{X_j}) \right)$$

En notant $\widetilde{X} = (\widetilde{X_i})_i$ c'est un système d'équations de point fixe de la forme $\widetilde{X} = \widetilde{f}(\widetilde{X})$.

2.2.2 Extrapolation

Le second problème auquel s'attaque l'interprétation abstraite est celui de la terminaison des itérations de Kleene. Rien n'implique qu'à partir d'un certain nombre d'itérations, les valeurs abstraites calculées à chaque point de contrôle cesseront d'évoluer.

Dans le cas où la fonction $\widetilde{f}$ est monotone, la suite de valeurs abstraites obtenue en chaque point de contrôle est croissante. Or, certains domaines abstraits ont une profondeur finie, c'est à dire qu'on ne peut pas y trouver de séquence infinie strictement croissante

$$\widetilde{X}_0 \sqsubset \widetilde{X}_1 \sqsubset \widetilde{X}_2 \sqsubset \dots$$

Par suite, dans ces domaines, la suite de valeurs abstraites finit par être constante après un nombre fini d'itérations.

Pour les autres, il faut trouver une solution. L'idée de l'extrapolation est de « deviner » la limite d'une suite infinie de valeurs abstraites en regardant ses premiers termes. Ainsi, on utilisera un *opérateur d'élargissement* noté $\triangledown$ réalisant cette extrapolation à chaque itération de sorte qu'après un nombre fini d'itérations on ait atteint un point fixe.

Définition 4 (Opérateur d'élargissement)**.**
Un opérateur $\triangledown$ sur un domaine abstrait est un opérateur d'élargissement si

1. pour tout $\widetilde{X}$ et $\widetilde{Y}$, on a $\widetilde{X} \sqsubseteq \widetilde{X} \triangledown \widetilde{Y}$ et $\widetilde{Y} \sqsubseteq \widetilde{X} \triangledown \widetilde{Y}$ et
2. toute suite $(\widetilde{X}_i)_i$ de valeurs abstraites de la forme $\widetilde{X}_{i+1} = \widetilde{X}_i \triangledown \widetilde{Y}_i$ est stationnaire.

Ainsi, la suite de valeurs abstraites $\widetilde{X}_0 = \bot$ et $\widetilde{X}_{i+1} = \widetilde{X}_i \triangledown \widetilde{f}(\widetilde{X}_i)$ finit par converger. C'est à dire pour un certain $n \in \mathbb{N}$ on a $\widetilde{X}_{n+1} = \widetilde{X}_n$. Or par définition de l'opérateur d'élargissement, on a aussi

$$\widetilde{f}(\widetilde{X}_n) \sqsubseteq \widetilde{X}_n \triangledown \widetilde{f}(\widetilde{X}_n)$$

et donc si on suit l'application de l'opérateur d'élargissement et que la suite de valeurs abstraites a convergé :

$$\widetilde{f}(\widetilde{X}_n) \sqsubseteq \widetilde{X}_{n+1} = \widetilde{X}_n$$

puis par monotonie de γ

$$\gamma(\widetilde{f}(\widetilde{X}_n)) \subseteq \gamma(\widetilde{X}_n)$$

et par correction de $\widetilde{f}$

$$f(\gamma(\widetilde{X}_n)) \subseteq \gamma(\widetilde{X}_n)$$

et on aura donc trouvé un post-point fixe, soit un majorant du plus petit point fixe.

Il est possible d'obtenir le même résultat en exigeant moins de conditions sur l'opérateur d'élargissement. [Dav05, Mon09b] Il n'est pas nécessaire d'attendre que l'opérateur d'élargissement soit stationnaire pour arrêter d'itérer : dès qu'on a obtenu une valeur abstraite $\widetilde{X}_{n+1}$ inférieure à la précédente le raisonnement précédent nous prouve qu'on a obtenu un post-point fixe.

Il n'est pas nécessaire d'appliquer l'opérateur d'élargissement à tous les points de contrôle. La précision de l'interprétation abstraite dépendra grandement du choix des points de contrôle où appliquer l'élargissement. La méthode proposée dans [Bou93] est basée sur la décomposition de l'automate interprété en composantes fortement connexes [Tar72]. On sélectionne d'abord comme point d'élargissement le premier point de contrôle rencontré dans chaque composante fortement connexe. Puis, pour chaque composante ainsi trouvée, on cherche récursivement les sous composantes fortement connexes : après avoir retiré le point d'élargissement on cherche à nouveau les composantes fortement connexes dans la composante.

2.2.3 Conclusion

Le domaine abstrait est un paramètre de l'interprétation abstraite. Le choix de l'ensemble d'abstractions à utiliser a des conséquences importantes sur la précision et l'efficacité de l'analyse. Une fois choisi cet ensemble d'abstractions, il faudra être capable de calculer ses opérateurs $\sqsubseteq$, $\sqcup$ ainsi que les fonctions $\widetilde{g}$ pour toute garde g et $\widetilde{a}$ pour toute action $\widetilde{a}$. Il faudra également, si nécessaire, proposer un opérateur d'élargissement $\triangledown$ adéquat.

2.3 Quelques domaines abstraits numériques classiques

Nous citons dans cette section quelques principaux domaines abstraits numériques proposés dans la littérature avant de décrire plus en détail trois d'entre eux. Les domaines que nous allons décrire abstraient les ensembles d'états par des systèmes d'inéquations que ces états vérifient. En fixant des contraintes sur la forme des équations, on obtient différents domaines. Plus les contraintes sont fortes, plus l'expressivité est faible, mais plus il est facile et efficace de calculer les opérateurs abstraits.

Le domaine des équations affines [Kar76], comme son nom l'indique n'autorise que des équations affines. Le domaine des intervalles [CC76] abstrait les états en attribuant à chaque variable du programme son intervalle de valeurs, c'est à dire des inéquations de la forme $x \leq c$ ou $x \geq c$ où x est une variable et c une constante.

Le domaine des polyèdres convexes [CH78] autorise n'importe quel système d'inéquations affines. En supposant qu'il y ait n variables dans le programme, les états peuvent être représentés comme des points dans un espace à n dimensions dont les coordonnées sont les valeurs des variables. Un système d'inéquations affines décrit alors un polyèdres convexe, contenant tous les états que l'on veut abstraire. En contraignant plus la forme des systèmes d'inéquations, on obtient des cas particuliers de polyèdres convexes :

- Les zones [Dil89] sont les polyèdres convexes définis par des équations de la forme $x-y < c$ où x et y sont deux variables du programme et c une constante.
- Les octogones [Min01, Min06] sont une généralisation des zones autorisant les équations de la forme $\pm x \pm y < c$. Le nom du domaine vient de ce que les projections sur deux dimensions des polyèdres ainsi définis donne un octogone.
- Les octaèdres [CC07] sont définis par les systèmes d'inéquations affines dont les coefficients sont 0, 1 ou -1.
- Les pentagones [LF10] sont une combinaison des intervalles et de contraintes de la forme $x < y$ où x et y sont deux variables.
- Les systèmes d'inéquations à deux variables [SKH03] sont une combinaison de contraintes de la forme $\alpha x + \beta y \leq c$.

2.3.1 Treillis des équations affines

En 1976, Michael Karr publie une méthode permettant de découvrir les relations affines entre les variables d'un programme [Kar76]. Ces travaux sont antérieurs à l'interprétation abstraite mais on peut en extraire un domaine abstrait utilisable en interprétation abstraite.

Dans ce domaine, on abstrait les valeurs concrètes par un système d'équations affines que chacun des états vérifie. L'ensemble des états ainsi abstrait est une variété affine. Les valeurs abstraites sont ces systèmes d'équations, qu'on représentera par une matrice échelonnée réduite. Ce choix permet d'assurer l'unicité de la représentation et la simplicité des opérateurs.

Michael Karr propose un algorithme d'*union affine* permettant de calculer la meilleure variété affine sur-approximant l'union de deux formes affines. On utilise cet algorithme pour calculer l'opérateur $\sqcup$. Quant à l'opérateur $\sqsubseteq$, on le calcule directement à partir de $\sqcup$:

$$\widetilde{X} \sqsubseteq \widetilde{Y} \;\Leftrightarrow\; \widetilde{X} \sqcup \widetilde{Y} = \widetilde{Y}$$

l'égalité de droite pouvant être directement testée par l'égalité des matrices échelonnées réduites.

Lorsqu'une variété affine est strictement incluse dans une autre, cela se traduit sur les matrices échelonnées réduites par le fait que celle de la première a strictement plus de lignes que la seconde. Par conséquent, il n'existe pas de suite infinie croissante de variétés affines. L'opérateur $\triangledown$ est donc inutile, et on pourra poser

$$\widetilde{X} \triangledown \widetilde{Y} = \widetilde{X} \sqcup \widetilde{Y}$$

On peut définir un opérateur $\sqcap$ concaténant deux systèmes d'équations, puis en calculant la forme échelonnée réduite de cette concaténation.

2.3.2 Treillis des intervalles

Dans le domaine abstrait des intervalles, on approxime l'ensemble des valeurs atteignables de chaque variable par un intervalle de valeurs classiquement dans $\mathbb{Q}$ ou $\mathbb{Z}$. Pour simplifier, nous nous restreindrons ici à $\mathbb{Z}$. Ces intervalles peuvent être infinis, ouverts à gauche vers $-\infty$, ouverts à droite vers $+\infty$ ou les deux. Lorsqu'ils sont finis, on les considèrera toujours fermés par convention. Ainsi, un intervalle est défini par un couple de bornes :

$$\mathbb{Z} \cup \{-\infty\} \times \mathbb{Z} \cup \{+\infty\}$$

L'inclusion, l'union, l'intersection et l'élargissement d'intervalles se définissent par comparaison, maximum et minimum des bornes :

$$\begin{aligned}
(a,b) \sqsubseteq (c,d) &\Leftrightarrow a \geq c \wedge b \leq d \\
(a,b) \sqcup (c,d) &= (\min\{a,c\}, \max\{b,d\}) \\
(a,b) \sqcap (c,d) &= (\max\{a,c\}, \min\{b,d\}) \\
(a,b) \triangledown (c,d) &= \left(\begin{cases} -\infty \text{ si } c < a \\ a \text{ sinon} \end{cases}, \begin{cases} +\infty \text{ si } d > b \\ b \text{ sinon} \end{cases} \right)
\end{aligned}$$

où $\leq$, min et max sont étendus sur $\mathbb{Z} \cup \{+\infty, -\infty\}$ de manière classique.

Chacune de ces abstractions représente l'ensemble des valeurs atteignables par la variable associée. On passe de l'abstraction à l'ensemble de valeurs par les fonctions d'abstraction et de concrétisation suivantes.

$$\alpha(X) = (\min X, \max X)$$
$$\gamma(a, b) = \{\, x \mid a \leq x \leq b \,\}$$

Une valeur abstraite est la donnée non pas d'un seul intervalle, mais d'un intervalle pour chaque variable du programme. Les opérateurs $\sqsubseteq$, $\sqcup$, $\sqcap$ et $\triangledown$ et les fonctions α et γ sont étendus en les appliquant intervalle par intervalle.

2.3.3 Treillis des zones

Les zones sont des polyèdres particuliers, caractérisés par un ensemble de contraintes affines de la forme :

$$x - y \leq c$$

où x et y sont deux variables et c une constante. Quitte à introduire une variable toujours égale à 0 on conviendra que ceci inclut également les contraintes de la forme :

$$c \leq x \leq d$$

où x est encore une variable et c et d deux constantes.

Supposons qu'on travaille sur un ensemble de n variables $\mathcal{V} = \{v_1, \ldots, v_n\}$. Une zone peut être définie par l'ensemble de constantes $(c_{i,j})_{0 \leq i,j \leq n}$ représentant l'ensemble de contraintes

$$\begin{aligned} v_i - v_j &\leq c_{i,j} \\ v_i &\leq c_{i,0} \\ v_j &\leq -c_{0,j} \end{aligned}$$

Les constantes $c_{i,j}$ peuvent être choisies dans différents domaines et ici, pour des raisons de simplicité, nous prendrons les entiers relatifs $\mathbb{Z}$. Lorsqu'on ne veut pas contraindre des variables par une telle relation, on prendra $c_{i,j} = +\infty$ qui rend la contrainte associée inoffensive. Nos valeurs abstraites pour l'ensemble de variable seront des ensembles de $n + 1 \times n + 1$ constantes dans le domaine

$$(\mathbb{Z} \cup \{+\infty\})^{(n+1)^2}$$

La fonction d'abstraction α doit produire la plus petite valeur abstraite sur-approximant un ensemble d'états et va donc chercher à minimiser les constantes $c_{i,j}$.

$$\alpha(X) = \left(\min \left\{\, c \in \mathbb{Z} \cup \{+\infty\} \;\middle|\; \forall \sigma \in X,\ \sigma(v_i) - \sigma(v_j) \leq c \,\right\}\right)_{i,j}$$

Lorsqu'aucune constante finie ne convient, il ne reste plus que $+\infty$ qui se trouve donc être la borne minimum.

La concrétisation produit l'ensemble des états vérifiant les contraintes de la valeur abstraite.

$$\gamma(\widetilde{X}) = \left\{ \sigma \;\middle|\; \forall v_i, v_j \in \mathcal{V}\ \sigma(v_i) - \sigma(v_j) \leq c_{i,j} \right\}$$

L'inclusion, l'union, l'intersection et l'élargissement se calculent constantes par constantes :

$$(c_{i,j})_{i,j} \sqsubseteq (d_{i,j})_{i,j} \Leftrightarrow \forall i, j\ c_{i,j} \leq d_{i,j}$$
$$(c_{i,j})_{i,j} \sqcup (d_{i,j})_{i,j} = \Big(max\{c_{i,j}, d_{i,j}\}\Big)_{i,j}$$
$$(c_{i,j})_{i,j} \sqcap (d_{i,j})_{i,j} = \Big(min\{c_{i,j}, d_{i,j}\}\Big)_{i,j}$$
$$(c_{i,j})_{i,j} \triangledown (d_{i,j})_{i,j} = \left(\begin{cases} +\infty \text{ si } d_{i,j} > c_{i,j} \\ c_{i,j} \text{ sinon} \end{cases}\right)_{i,j}$$

Deux valeurs abstraites peuvent avoir la même concrétisation et être incluse strictement l'une dans l'autre, ou même n'être pas comparables. Par exemple, l'ensemble de contraintes

$$v_1 \leq v_2 \wedge v_2 \leq v_3$$

est logiquement équivalent à

$$v_1 \leq v_2 \wedge v_2 \leq v_3 \wedge v_1 \leq v_3$$

Le premier est représenté par la valeur abstraite

$$c_{1,2} = 0, c_{2,3} = 0, c_{1,3} = +\infty$$

tandis que pour le second on aurait

$$c_{1,2} = 0, c_{2,3} = 0, c_{1,3} = 0$$

Pour la définition de $\sqsubseteq$ cela signifie que le second est inclus dans le premier et pourtant, comme nous venons de le remarquer, ils ont la même concrétisation. L'opération de réduction de la valeur abstraite consiste à minimiser les bornes. Cette minimisation repose sur l'implication suivante.

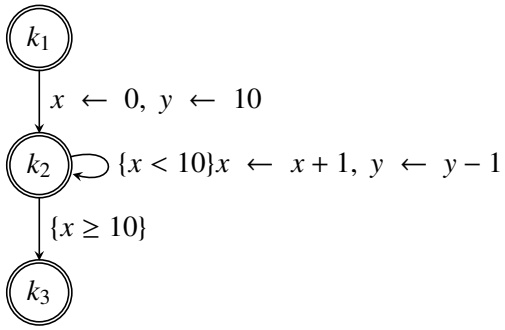
$$v_i - v_j \leq c_{i,j} \wedge v_j - v_k \leq c_{j,k} \;\Rightarrow\; v_i - v_k \leq c_{i,j} + c_{j,k}$$

Cette implication nous dit que pour tout couple de bornes $c_{i,j}, c_{j,k}$ on peut renforcer le borne $c_{i,k}$ en prenant le minimum de cette borne et de la somme des deux autres. Le renforcement de toutes les bornes se réduit à un problème de calcul de clôture transitive dans un graphe dont les arrêtes ont pour coût les bornes.

Il est important de noter que pour conserver les propriétés de l'opérateur d'élargissement $\triangledown$, la réduction ne doit pas être appliquée entre deux élargissement. La valeur à élargir doit toujours être directement le résultat du dernier élargissement et non sa réduction.

2.4 Exemple

Reprenons l'exemple introduit au début de ce chapitre. L'automate interprété était le suivant.



Notons X_1, X_2 et X_3 l'ensemble des états atteignables en k_1, k_2 et k_3 respectivement. Ces ensembles vérifient le système d'équations

$$
\begin{aligned}
X_1 &= \Sigma \\
X_2 &= [x \leftarrow 0,\ y \leftarrow 10](X_1) \cup [x \leftarrow x+1,\ y \leftarrow y-1] \circ \{x < 10\}(X_2) \\
X_3 &= \{x \geq 10\}(X_2)
\end{aligned}
$$

Pour l'interprétation abstraite de ce programme, on associe aux points de contrôle des valeurs abstraites que l'on notera $\widetilde{X_1}$, $\widetilde{X_2}$ et $\widetilde{X_3}$ respectivement. Le système d'équations appliqué aux valeurs abstraites devient

$$
\begin{aligned}
\widetilde{X_1} &= \top \\
\widetilde{X_2} &= [x \leftarrow 0,\ y \leftarrow 10](\widetilde{X_1}) \sqcup [x \leftarrow x+1,\ y \leftarrow y-1] \circ \{x < 10\}(\widetilde{X_2}) \\
\widetilde{X_3} &= \{x \geq 10\}(\widetilde{X_2})
\end{aligned}
$$

On commence avec $\widetilde{X_1} = \widetilde{X_2} = \widetilde{X_3} = \bot$. Puis on itère en calculant à chaque fois une nouvelle valeur des $\widetilde{X_i}$ en remplaçant dans la partie droite du système les variables $\widetilde{X_1}, \widetilde{X_2}, \widetilde{X_3}$ par leur valeur précédente. Lors de la première itération, on obtient donc

$$
\begin{aligned}
\widetilde{X_1} &= \top \\
\widetilde{X_2} &= [x \leftarrow 0, y \leftarrow 10](\bot) \sqcup [x \leftarrow x+1, y \leftarrow y-1] \circ \{x < 10\}(\bot) = \bot \\
\widetilde{X_3} &= \{x \geq 10\}(\bot) = \bot
\end{aligned}
$$

$\widetilde{X_2}$ et $\widetilde{X_3}$ conservent leur valeur $\bot$ et seule $\widetilde{X_1}$ a changé. Lors de la seconde itération, ce changement n'influe que sur le calcul de $\widetilde{X_2}$.

$$
\begin{aligned}
\widetilde{X_2} &= [x \leftarrow 0, y \leftarrow 10](\top) \sqcup [x \leftarrow x+1, y \leftarrow y-1] \circ \{x < 10\}(\bot) \\
&= [x \leftarrow 0, y \leftarrow 10](\top) \sqcup \bot \\
&= [x \leftarrow 0, y \leftarrow 10](\top)
\end{aligned}
$$

Pour les intervalles, on aura la valeur abstraite

$$\widetilde{X_2} \ = \ (x \in [0,0], y \in [10,10])$$

Lors de la troisième itération, $\widetilde{X_2}$ n'étant désormais plus $\top$ on peut calculer l'effet de la seconde transition (celle de la boucle) sur cette nouvelle valeur. x est incrémenté et y décrémenté, ce qui nous donne donc la valeur abstraite

$$(x \in [1,1], y \in [9,9])$$

Le calcul de $\widetilde{X_2}$ devient alors

$$\widetilde{X_2} = (x \in [0,0], y \in [10,10]) \sqcup (x \in [1,1], y \in [9,9]) = (x \in [0,1], y \in [9,10])$$

On pourrait également calculer $\widetilde{X_3}$ mais cela ne serait pas astucieux. Oublions la pour le moment. Jusqu'ici, on a obtenu que les valeurs de x et y dans la boucle sont comprises dans les intervales $[0,1]$ et $[9,10]$ respectivement. Ce n'est pas un point fixe. On pourrait itérer neuf fois encore avant d'atteindre ce point fixe. Pour accélérer la convergence, nous appliquons l'opérateur d'élargissement. Il extrapole à partir des intervalles obtenus lors de la première itération et lors de la seconde itération pour devnier quels seront les intervalles possibles après une infinité d'itérations. L'application de l'opérateur d'élargissement nous donne

$$(x \in [0,0], y \in [10,10]) \triangledown (x \in [0,1], y \in [9,10]) = (x \in [0,+\infty[, y \in]-\infty, 10])$$

Il s'agit d'un post-point fixe, mais très imprécis. A partir de cette valeur de $\widetilde{X_2}$, on peut calculer

$$\widetilde{X_3} = \{x \geq 10\}(\widetilde{X_2}) = (x \in [10,+\infty[, y \in]-\infty, 10])$$

Ce résultat, bien que correct, est peu satisfaisant. Cependant, en continuant d'itérer, on peut l'améliorer. À partir de la dernière valeur de $\widetilde{X_2}$, on calcule une nouvelle fois le résultat d'une itération. L'application de la garde, réduit l'intervalle de x de $[0,+\infty[$ à $[0,9]$. Puis l'application de l'action nous donne l'intervalle $[1,10]$. Par conséquent, la nouvelle valeur devient

$$\widetilde{X_2} = (x \in [0,0], y \in [10,10]) \sqcup (x \in [1,10], y \in [-\infty, 9]) = (x \in [0,10], y \in [-\infty, 10])$$

Elle est toujours imprécise pour la variable y, mais on a enfin un intervalle satisfaisant pour x. Si on calcule maintenant $\widetilde{X_3}$, on aura

$$\widetilde{X_3} = x \in [10,10], y \in [-\infty, 10]$$

et donc la bonne valeur de x en fin de programme. Lors de cette itération, on aura obtenu des valeurs abstraites plus petites que celles de la précédente. Cela signifie qu'on a atteint un post-point fixe et qu'on peut arrêter d'itérer.

En utilisant un domaine abstrait plus précis que les intervalles, et en particulier capable d'exprimer l'invariant $x + y = 10$, alors à partir de la valeur de x, on retrouvera la valeur de y et le programme aura été analysé précisément.

3 Synthèse de propriétés sur les structures de données

3.1 Introduction

3.1.1 Objectif

Les développements de l'interprétation abstraite ont donné naissance à de nombreux domaines abstraits représentants des propriétés variées ainsi qu'à de nombreux perfectionnements de la technique d'analyse. L'analyse de programmes manipulant des structures de données quant à elle est beaucoup plus immature. Il est raisonnable de chercher à porter l'interprétation abstraite sur l'analyse des structures de données tout en préservant les progrès acquis dans l'analyse des programmes plus simples.

Une première conséquence de cet objectif est que l'analyse de structures de données devrait respecter une certaine indépendance vis à vis des méthodes de calcul du point fixe. Il faut autant que possible que le problème de l'analyse puisse, comme c'est le cas en interprétation abstraite, s'exprimer comme la résolution d'un système d'équations. Ceci permettrait en effet d'y appliquer les différentes méthodes de résolution, que ce soit par les méthodes de calcul de point fixe basées sur le théorème de Kleene et/ou par itération de politiques. [CGG+05]

Une idée naïve. Une seconde conséquence est la nécessité de préserver au maximum toute la théorie acquise sur les domaines abstraits. Ceci amène à l'idée suivante, illustrée par la figure 3.1. Nous voudrions pouvoir analyser ces deux programmes avec le même domaine abstrait, de sorte que celui-ci ne fasse pas la différence entre l'un et l'autre. Le premier programme ne devrait présenter aucune difficulté, tandis que le second faisant appel à des accès au tableau ne pourra probablement pas être analysé. Mais ces deux programmes sont très similaires. Il suffirait alors qu'en présence du second nous le transformions en le premier avant de le donner en analyse au domaine abstrait. De manière générale, on prend un programme avec des accès mémoire quelconques et on le transforme en un programme ne manipulant que des variables locales simples. Ainsi, on pourra utiliser n'importe quel domaine abstrait pour analyser le programme résultat.

Décrivons une telle transformation par un algorithme naïf. On remplace chaque accès à une cellule mémoire par un accès à une variable. En particulier, pour chaque élément de chaque

x = x - y	A[1] = A[1] - A[2]
y = x + y	A[2] = A[1] + A[2]
x = y - x	A[1] = A[2] - A[1]
(a) Programme simple	(b) Programme modifié

Figure 3.1 : Ces deux programmes échangent respectivement les valeurs de x et y et les valeurs de $A[0]$ et $A[1]$.

Pour *i de* 1 *à n* **faire**
  $A[i] \leftarrow 0$

FIGURE 3.2 : Ce programme initialise les cellules d'un tableau à la valeur 0.

tableau, on introduit une nouvelle variable. Puis on remplace les accès à ces cellules par des accès aux variables correspondantes.

Pour l'exemple de la figure 3.1(b), supposons que le tableau A soit composé de dix cellules. Introduisons dix nouvelles variables $a_1, \ldots, a_{10}$ pour chacune d'elles. Remplaçons ensuite les accès à $A[1]$ par un accès à a_1 et les accès à $A[2]$ par un accès à a_2. On obtient aux noms près le programme de la figure 3.1(a).

Cette transformation naïve ne fonctionnera pas dans tous les cas : comment faire lorsque l'indice de l'accès au tableau n'est pas constant ? Et comment faire lorsque la taille du tableau est non constante ? Si l'indice d'accès à une cellule varie, il n'est pas possible statiquement de remplacer l'accès à cette cellule par une variable. En outre, la plupart des langages de programmations permettent de définir des tableaux dont la taille dépend d'une variable. Si cette taille n'est pas bornée alors le nombre de variables à introduire n'est pas fini. Cet obstacle rend l'idée inapplicable pour tous les programmes utilisant des tableaux non bornés. En outre, même si la taille des tableaux est bornée, il suffit qu'elle soit grande pour être un problème. Puisque l'efficacité de l'analyse dépend en grande partie du nombre de variables, il n'est pas raisonnable de chercher à en introduire trop.

Une seconde approche Ne jetons pourtant pas complètement l'idée précédente. Mettons la de coté et observons un autre exemple. Considérons le programme figure 3.2 qui initialise l'ensemble des n cellules d'un tableau en leur donnant la valeur 0. Quand bien même on n'aurait pas les problèmes précédemment décrits, il ne serait pas très astucieux d'associer une variable différente à chaque cellule. Chacune d'entre elles est initialisée à la même valeur. Il semble plus approprié de décrire les cellules dans leur ensemble plutôt que de le faire individuellement.

Sur ce principe, on peut faire varier notre première idée. Au lieu d'introduire autant de nouvelles variables que de cellules on ajoute une seule variable pour l'ensemble du tableau. Pour le programme figure 3.2, on ajoute par exemple une variable a représentant le tableau A. Supposons alors qu'après analyse on obtient en sortie de programme cette valeur abstraite :

$$i = n \wedge a = 0$$

On pourra l'interpréter de la manière suivante :

$$\forall \ell, 1 \leq \ell \leq n \Rightarrow (i = n \wedge A[\ell] = 0)$$

On peut opérer de la même manière sur n'importe quel programme. On ajoute une nouvelle variable pour chaque tableau et on interprète les valeurs abstraites en disant que toute propriété vraie de cette variable l'est de tout élément du tableau associé.

Ces deux premières idées sont introduites dans l'analyseur statique ASTRÉE. [BCC+02]

Problèmes de sémantique. Ainsi on résout l'un des problèmes puisqu'on peut désormais traiter des tableaux non bornés. Mais le second problème reste toujours non résolu : on ne sait

toujours pas interpréter les affectations aux cellules dans le cas général. Ce n'est toutefois pas un obstacle insurmontable. Supposons qu'après l'initialisation du tableau on ait une affectation à l'une des cellules :

$$A[1] \leftarrow 1$$

On a choisi d'interpréter les propriétés de notre nouvelle variable a comme les propriétés vraies de toutes les cellules. Donc on peut dire de a qu'elle est comprise entre 0 et 1 puisque c'est le cas de toutes les cellules. On obtiendrait alors après cette affectation :

$$i = n \wedge 0 \leq a \leq 1$$

Le cas général n'est pas beaucoup plus difficile. Considérons que nous avons une propriété φ vraie de toutes les cellules. En particulier φ est vrai de toutes les cellules non touchées par l'affectation. Après l'affectation, ce qu'on peut dire de chaque cellule du tableau c'est

- ou bien qu'elle n'a pas été affectée et qu'elle vérifie toujours φ,
- ou bien qu'elle a été affectée et qu'elle vaut désormais la valeur affectée.

Les propriétés vraies de toutes cellules après l'affectation sont donc celles impliquées par la disjonction des propriétés vraies avant l'affectation et des propriétés vraies de la cellule affectée. La valeur abstraite obtenue peut être calculée comme une union abstraite :

$$[A[i] \leftarrow x](\varphi) = \varphi \sqcup [a \leftarrow x](\varphi)$$

On désigne ce type d'affectation sous le nom d'*affectations faibles*, [BCC+02] car elles ne font qu'affaiblir la valeur abstraite. Nous les noterons ici $\leftsquigarrow$:

$$[A[i] \leftsquigarrow x](\varphi) = \varphi \sqcup [a \leftarrow x](\varphi)$$

Chaque affectation à une cellule doit donc être traitée comme une affectation faible à la variable qui la représente. En conséquence, chaque fois qu'on traite une affectation, on obtient une propriété plus faible. On perd de l'information sur les structures de données au cours de l'interprétation.

Une troisième idée. Ce dernier point a son importance. Dans l'exemple, nous avons parlé de la propriété qu'il faut découvrir mais pas de comment on la découvre. On ne fait que perdre de l'information durant l'interprétation et on ne sait rien initialement du tableau. Le domaine abstrait ne peut produire aucune information sur notre variable a que ce soit au début du programme ou à l'intérieur de la boucle. Et pourtant on aimerait découvrir une nouvelle propriété acquise à la fin du programme : que toutes les cellules valent 0.

Pour comprendre ce problème revenons à une preuve *à la main* du programme. Ici, la propriété invariante de la boucle est que les cellules de tableau d'indice compris entre 1 et i ont toutes pour valeur 0. En sortant de la boucle, on a $i = n$ et donc l'invariant nous dit que toutes les cellules sont initialisées.

Notre espoir est de réussir à transposer l'invariant de boucle. On ne peut plus se contenter d'ajouter une variable symbolisant le tableau en totalité, on a besoin de pouvoir ajouter des variables pour des sous-ensembles de cellules du tableau. On gardera la même interprétation que précédemment, c'est à dire que toute propriété vraie de cette variable l'est de toute cellule de ce sous-ensemble. Pour l'exemple, voici ce que nous pouvons faire :

- ajouter une variable a_1 pour les cellules d'indice compris entre 1 et $i-1$ dont on pourra dire qu'elles ont pour contenu 0,
- ajouter une variable a_2 pour la cellule d'indice i dont le contenu sera modifié dans la boucle et
- ajouter une variable a_3 pour les cellules d'indice compris entre $i+1$ et n dont on ne sait rien.

Ainsi au début d'une itération on espère avoir la valeur abstraite suivante

$$1 \leq i \leq n \wedge a_1 = 0$$

qui signifierait

$$1 \leq i \leq n \wedge \forall \ell, (1 \leq \ell \leq i-1) \Rightarrow A[\ell] = 0$$

On est ensuite en mesure d'interpréter l'affectation du corps de la boucle. Elle ne touche que la cellule $A[i]$ qui est l'unique cellule représentée par a_2. Il est donc correct de dire que toutes les cellules représentées par a_2 ont pour contenu 0 après l'affectation. Ce qui nous donne la valeur abstraite :

$$1 \leq i \leq n \wedge a_1 = a_2 = 0$$

signifiant

$$1 \leq i \leq n \wedge \forall \ell, (1 \leq \ell \leq i) \Rightarrow A[\ell] = 0 \wedge A[i] = 0$$

A la fin de l'itération, i est incrémenté. Les sous-ensembles de cellules changent en conséquence. La cellule associée à a_2 va rejoindre les cellules associées à a_1. Mais puisque a_1 et a_2 vérifient la même propriété, cette propriété reste vraie du nouvel ensemble de cellules associé à a_1. On ne sait toujours rien de a_3 et de la nouvelle cellule associée à a_2, mais on a déjà assez de propriétés pour prouver l'invariant.

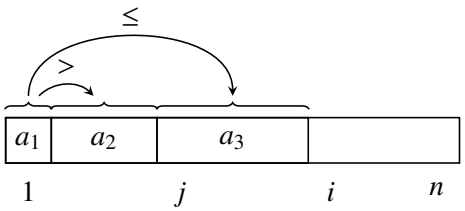
Des ensembles vides de cellules. Pour être un véritable invariant de boucle, il ne manque qu'une seule chose : il doit être vrai à l'entrée de la boucle. Or à l'entrée de la boucle l'ensemble des cellules d'indice compris entre 1 et $i-1$ est un ensemble vide puisque $i = 1$. Et tous les éléments de l'ensemble vide vérifient n'importe quelle propriété en particulier l'invariant. On serait donc en mesure de prouver complètement l'invariant... pour peu qu'on s'autorise à manipuler des variables représentant des ensembles de cellules pouvant être vides.

Il y a bien une autre alternative. On pourrait décrire l'état du programme d'exemple par deux valeurs abstraites : l'une décrivant le programme quand $i = 1$ et n'ayant pas de variable a_1, l'autre décrivant le programme quand $i > 1$ et ayant une variable a_1 représentant un ensemble non vide de cellules. Le même principe peut être appliqué à n'importe quel programme. Mais dans le cas général, le nombre de valeurs abstraites peut devenir grand : chacun des morceaux de tableaux que l'on décide de décrire peut être vide et l'énumération des cas est exponentielle.

Ce n'est pas la solution que nous choisirons ici. Nous allons nous autoriser à décrire des sous-ensembles de cellules pouvant être vides. Cela va nous forcer à adopter un formalisme un peu moins direct quoique pas nécessairement plus complexe. La solution alternative entraîne elle aussi une complexité puisqu'il faut décrire comment se combinent toutes ces valeurs abstraites. Les conséquences de ce choix seront abordées plus tard.

```
i ← 2
j ← 2
Tant que i < n faire
    Si A[i] < A[1] alors
        A[i] ↔ A[j]
        j ← j + 1
    i ← i + 1
A[j − 1] ↔ A[1]
```

(a) Programme



(b) Découpage du tableau

FIGURE 3.3 : Algorithme de segmentation d'un tableau. Ce programme découpe le tableau, mettant les éléments inférieurs à $A[1]$ entre 2 et $j-1$ et les éléments supérieurs entre j et n.

Des propriétés relationnelles. Jusqu'à présent, nous avons introduit des variables représentant des sous-ensembles de cellules en leur donnant comme interprétation qu'une formule vraie de l'une de ces variables le soit aussi de toutes les cellules représentées par ces variables. Mais qu'en est t-il des propriétés faisant intervenir plusieurs de ces variables ?

La figure 3.3 montre l'algorithme de segmentation utilisé dans le tri par segmentation. Cet algorithme sépare un tableau en deux par rapport à un pivot suivant que les valeurs des cellules sont supérieures ou inférieures à ce pivot. Le pivot est lui même un élément du tableau, ici la cellule $A[1]$. L'indice j sépare les deux autres parties : les cellules de contenu inférieur entre 2 et $j-1$ et les cellules de contenu supérieur entre j et n.

Pour analyser ce programme, introduisons 3 variables a_1, a_2 et a_3 associées aux trois parties décrites plus tôt : a_1 représente le pivot et a_2, a_3 les deux sous-ensembles de cellules issus de la segmentation. On espère pouvoir obtenir une valeur abstraite similaire à celle-ci

$$2 \leq i \leq n \wedge a_2 < a_1 \leq a_3$$

qu'on voudra certainement interpréter de la manière suivante :

$$2 \leq i \leq n \wedge \forall \ell_1, \ell_2, \big(2 \leq \ell_1 < j \wedge j \leq \ell_2 \leq i\big) \Rightarrow \big(A[\ell_1] < A[1] \leq A[\ell_2]\big)$$

En particulier, le sens que l'on donne à la formule $a_2 < a_3$ est que quel que soit le choix d'une cellule représentée par a_2 et quel que soit le choix d'une cellule représentée par a_3, la première est strictement inférieure à la seconde.

Conclusion. Quel qu'en soit le but, il semble intéressant de pouvoir exprimer des propriétés sur des sous-ensembles de cellules. Nous ne savons pas encore comment choisir ces sous-ensembles ni comment l'analyse du programme se déroulera une fois ces sous-ensembles choisis. Mais la nécessité de procéder à de tels découpages semble évidente pour un certain nombre de preuves de programmes. Une preuve d'un algorithme sur une structure de donnée utilisera généralement un découpage de cette structure de donnée, qu'il soit trivial ou complexe, afin d'exprimer des propriétés de sous-ensembles de cellules ou de relations entre des sous-ensembles de cellules.

3.1.2 Abstractions et Concrétisations

Pour exprimer des propriétés sur des sous-ensembles de cellules nous souhaitons introduire de nouvelles variables les représentant afin de permettre au domaine abstrait de dériver des propriétés sur ces sous-ensembles. A chaque fois, nous avons donné une interprétation de ces propriétés. Nous allons maintenant formaliser cette interprétation. Cela consiste à définir les opérations d'abstraction et de concrétisation. La formalisation que nous proposons ici est une généralisation de celle publiée dans [GDD+04].

Nous partons d'un domaine abstrait existant pour lequel sont déjà définies une fonction d'abstraction $\widetilde{\alpha}$ et une fonction de concrétisation $\widetilde{\gamma}$. Notre but est de construire à partir de là un domaine abstrait capable d'analyser des programmes manipulant des structures de données et les fonctions α et γ du domaine. Pour le moment nous allons supposer que nous avons déjà fixé par un moyen quelconque les sous-ensembles de cellules qui nous intéressent.

Supposons par exemple que nous ayons obtenu la valeur abstraite $\widetilde{X}$ suivante :

$$0 \leq a_1 < a_2 = 2 < a_3 \leq 4$$

où les variables a_1, a_2, a_3 représentent les ensembles de cellules $A[1..3]$, $A[4]$ et $A[5..6]$ respectivement. La concrétisation $\widetilde{\gamma}(\widetilde{X})$ de cette valeur abstraite doit produire tous les états vérifiant la propriété ci dessus. C'est à dire l'ensemble des 4 fonctions $\overline{\sigma}$ vérifiant

$$0 \leq \overline{\sigma}(a_1) < \overline{\sigma}(a_2) = 2 < \overline{\sigma}(a_3) \leq 4$$

Mais cela ne nous donne pas une description concrète de l'état du programme. Un état concret doit décrire les valeurs possibles pour chacune des cellules du tableau A. Et dans cet état, chaque cellule représentée par a_1 doit vérifier les propriétés de a_1. Autrement dit les états concrets sont les fonctions σ vérifiant

$$\begin{aligned} &\forall \ell, 1 \leq \ell \leq 3 \Rightarrow 0 \leq \sigma(A[\ell]) < 2 \\ \wedge\ &\forall \ell, 5 \leq \ell \leq 6 \Rightarrow 2 < \sigma(A[\ell]) \leq 4 \\ \wedge\ &\sigma(A[4]) = 2 \end{aligned}$$

Chacune des cellules $A[1], A[2], A[3]$ et $A[5], A[6]$ peut prendre 2 valeurs, on a donc 32 états concrets σ qui conviennent.

Notre objectif est de passer du premier ensemble d'états au second et inversement. Nous allons tenter de construire deux nouvelles fonctions $\overline{\alpha}$ et $\overline{\gamma}$ qui opèrent cette transformation. On aura ainsi décomposé l'abstraction et la concrétisation en deux étapes :

$$\begin{aligned} \alpha &= \widetilde{\alpha} \circ \overline{\alpha} \\ \gamma &= \overline{\gamma} \circ \widetilde{\gamma} \end{aligned}$$

L'abstraction. Ici, la recherche de $\overline{\alpha}$ va s'avérer plus intuitive que celle de $\overline{\gamma}$. Aussi allons nous commencer par là. Lorsqu'un domaine abstrait affirme qu'une propriété sur des variables est vraie, cela signifie que pour chaque état de la concrétisation on peut la vérifier. Ici, on souhaite que lorsque la valeur abstraite affirme une propriété sur des variables, qu'elle soit vraie pour tout état concret mais aussi pour toutes les cellules représentées par chacune des variables.

Revenons à l'exemple. La valeur abstraite affirme entre autre que $a_1 < a_3$. Le sens que l'on choisit de donner à cette affirmation est que toute cellule représentée par a_1 est inférieure à toute

$$r : \begin{cases} a_1 \mapsto A[2] \\ a_2 \mapsto A[4] \\ a_3 \mapsto A[6] \end{cases}$$

(a) Choix de représentants

$$\sigma : \begin{cases} A[1] \mapsto 0 \\ A[2] \mapsto 1 \\ A[3] \mapsto 0 \\ A[4] \mapsto 2 \\ A[5] \mapsto 4 \\ A[6] \mapsto 4 \end{cases}$$

(b) État concret

$$\overline{\sigma} : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 2 \\ a_3 \mapsto 4 \end{cases}$$

(c) État obtenu

Figure 3.4 : L'état $\overline{\sigma}$ est obtenu à partir d'un choix de représentants r et d'un état concret σ.

cellule représentée par a_3. Une manière équivalente de dire est que pour chaque choix de couple de cellules, l'une dans $A[1..3]$ l'autre dans $A[5..6]$, on a que la valeur de la première cellule est inférieure à celle de la seconde.

De manière générale, si on a n variables, des propriétés vraies de ces n variables, on peut énumérer les n-uplets de choix de cellules pour lesquels ces propriétés seront vérifiées. (Dans notre exemple, on peut énumérer 6 triplets de cellules pour (a_1, a_2, a_3).) Un choix de cellule peut être vu comme un choix de représentants du sous-ensemble symbolisé. Mieux que des n-uplets, ce sont donc des choix de représentants des variables du domaine abstrait, ou autrement dit des fonctions des variables vers les cellules. Comme les sous-ensembles de cellules que l'on symbolise sont fixés, ces choix de représentants sont eux aussi fixés. On a donc un ensemble $\mathcal{R}$ de choix de représentants valides.

En général $\mathcal{R}$ sera défini en fonction d'un état concret : on peut par exemple vouloir comme dans la section précédente symboliser un ensemble de cellules d'index compris entre 1 et une variable i, dont la valeur dépend de l'état concret considéré. On notera alors $\mathcal{R}(\sigma)$ l'ensemble des choix de représentants pour l'état σ.

La construction de $\mathcal{R}$ est un problème à part entière, supposons pour le moment qu'on l'obtient d'une manière ou d'une autre. Dans notre exemple $\mathcal{R}$, ne dépend pas de σ et on a

$$\mathcal{R}(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[4] \\ a_3 \mapsto A[\ell_2] \end{cases} \;\middle|\; 1 \leq \ell_1 \leq 3 \wedge 5 \leq \ell_2 \leq 6 \right\}$$

A partir de chaque état concret σ et pour chaque choix de représentants r nous pouvons construire un état $\overline{\sigma}$. En effet, pour chaque variable v on choisit son représentant par r, puis on va chercher dans l'état σ la valeur contenue dans cette cellule. De manière concise :

$$\overline{\sigma} = \sigma \circ r$$

La figure 3.4 illustre la construction d'un tel état pour l'exemple.

En procédant ainsi pour chaque état concret et chaque choix de représentants on construit un ensemble d'états intermédiaires avec une caractéristique intéressante. Une fois abstrait par $\widetilde{\alpha}$ on obtiendra des propriétés vraies pour chaque état intermédiaire. Donc par construction, les propriétés vraies de ces états intermédiaires le sont pour chaque état concret et chaque choix de représentants.

Cela semble convenir à la définition de notre abstraction. On a réussi à partir d'un ensemble d'états concrets X à construire un ensemble d'états intermédiaires $\overline{X}$ dont l'abstraction par $\widetilde{\alpha}$ a

une interprétation fidèle à notre objectif initial :

$$\overline{\alpha}(X) = \{\, \overline{\sigma} \mid \exists \sigma \in X, \exists r \in \mathcal{R}(\sigma), \overline{\sigma} = \sigma \circ r \,\}$$

La concrétisation. Classiquement en interprétation abstraite nous travaillons sur des correspondances de Galois et par conséquent le choix de $\overline{\alpha}$ fixe automatiquement $\overline{\gamma}$. On sait qu'à partir d'un état concret on peut générer plusieurs états intermédiaires. Pour qu'un état concret soit dans la concrétisation $\overline{\gamma}(\overline{X})$ il faut donc que tous les états intermédiaires soient dans $\overline{X}$. A partir de tous ces états on peut reconstruire un état concret. Dans le processus, chaque état intermédiaire est associé à un choix de représentants valide pour l'état concret cible.

$$\overline{\gamma}(\overline{X}) = \left\{ \sigma \;\middle|\; \forall r \in \mathcal{R}(\sigma), \exists \overline{\sigma} \in \overline{X}, \overline{\sigma} = \sigma \circ r \right\}$$

Les fonctions $\overline{\alpha}$ et $\overline{\gamma}$ seront décrites plus en profondeur dans la section 3.4. D'ici là, nous allons devoir compliquer encore un peu la situation. Néanmoins, le processus d'abstraction restera le même : on permet au domaine abstrait de décrire des propriétés sur des sous-ensembles de cellules, en énumérant tous les n-uplets de cellules sur lesquels on veut exprimer ces propriétés.

3.1.3 Conséquences de la possible vacuité des sous-ensembles de cellules

Pour saisir les problèmes qu'impliquent la manipulation de sous-ensembles revenons à notre exemple initial de l'initialisation de tableaux. Prenons la valeur abstraite exprimant l'invariant de boucle après l'affectation, en ne considérant que les propriétés sur les cellules du tableau :

$$a_1 = a_2 = 0$$

ce qui signifierait que toutes les cellules d'indice compris entre 1 et $i - 1$ ainsi que la cellule d'indice i ont pour contenu 0. Ce qui pourrait s'écrire :

$$\forall \ell_1, \ell_2, 1 \leq \ell_1 < i \wedge \ell_2 = i \Rightarrow (A[\ell_1] = A[\ell_2] = 0)$$

Dans la formalisation de la section précédente, cela donnerait pour $\mathcal{R}$:

$$\mathcal{R}(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[\ell_2] \end{cases} \;\middle|\; 1 \leq \ell_1 \leq \sigma(i) \wedge \ell_2 = \sigma(i) \right\}$$

Malheureusement lorsque $i = 1$ l'interprétation de la valeur abstraite ne nous apprend rien : la partie gauche de l'implication est impossible et donc l'implication est toujours vraie. Or, on a besoin de la propriété $A[1] = 0$ même lorsque $i = 1$.

Le problème de notre formalisation c'est qu'on doit toujours avoir un choix de représentants pour chaque sous-ensemble de cellules symbolisé. Or quand ce sous-ensemble est vide, il est effectivement impossible d'en choisir un élément.

Comme noté précédemment, on pourrait être tenté de décomposer la valeur abstraite en deux : une partie qui décrit a_1 et une partie qui décrit a_2. Ici, on ne perdrait rien à faire cette décomposition, si ce n'est la relation d'égalité entre a_1 et a_2 qui ne nous sert pas pour la preuve. Mais observons un exemple où l'on a besoin de propriétés relationnelles : la segmentation de tableau figure 3.3. On devrait pouvoir obtenir une valeur abstraite décrivant l'état du programme à la fin de l'exécution similaire à celle-ci :

$$a_2 < a_1 \wedge a1 \leq a_3$$

Qu'on voudra certainement interpréter de la manière suivante afin d'en tirer le maximum d'information :

$$\begin{aligned} &\forall \ell,\ 2 \leq \ell < i \ \Rightarrow\ A[\ell] < A[1] \\ \wedge\ &\forall \ell,\ i \leq \ell \leq n \ \Rightarrow\ A[1] \leq A[\ell] \end{aligned}$$

Bien sûr de cette formule on peut déduire aussi

$$\forall \ell_1, \ell_2, 1 \leq \ell_1 < i \leq \ell_2 \leq n \ \Rightarrow\ A[\ell_1] < A[\ell_2]$$

ce qui s'écrirait dans le domaine abstrait

$$a_2 < a_3$$

Or cette dernière propriété sera en principe déduite automatiquement par le domaine abstrait. Mais cela uniquement si les variables a_1, a_2 et a_3 sont décrites par la même valeur abstraite. Si on décomposait en deux valeurs abstraites, l'une décrivant les relations entre les variables a_1 et a_2 l'autre décrivant les relations entre les variables a_1 et a_3 il faudrait trouver le moyen de combiner ces valeurs abstraites afin de dériver les relations entre a_2 et a_3 qui peuvent être utiles à la preuve du programme.

Pour obtenir le maximum de précision de l'analyse d'un programme, il est nécessaire de regrouper dans une même valeur abstraite toutes les variables ayant une relation pour ce domaine abstrait. Ainsi toute relation valide entre ces variables pourra potentiellement être déduite.

Évolution de la formalisation. Comme nous l'avons vu plus haut, nous ne pouvons pas garder exactement le même formalisme pour les choix de représentants. Lorsqu'un ensemble de cellules est vide, il est impossible d'en choisir un représentant. Modifions alors la définition d'un choix de représentants en n'obligeant plus à ce que ce soient des fonctions totales. Voyons les choix de représentants comme des fonctions partielles qui ne sont pas définies lorsque l'ensemble de cellules symbolisé par une variable est vide. Ainsi, pour l'initialisation de tableau, lorsque $\sigma(i) = 1$ et pour $r \in \mathcal{R}(\sigma)$ on aura que $r(a_1)$ est indéfini.

En conséquence, la construction des intermédiaires est aussi modifiée. La composition

$$\overline{\sigma} = \sigma \circ r$$

n'est plus définie que sur le domaine de définition de r. Ainsi, un état intermédiaire ne sera pas défini sur une variable symbolisant un sous-ensemble de cellules vide.

Conséquences sur le domaine abstrait. Ce choix a des conséquences importantes. Il nous force à abandonner l'une des contraintes que l'on s'était fixé : la préservation des domaines abstraits. On va maintenant demander au domaine abstrait d'être capable d'abstraire des états qui ne sont plus des fonctions totales mais des fonctions partielles. Il va falloir donner du sens à ces états partiels, et ce, pour chaque domaine abstrait. En fait, notre changement de formalisation n'a pas résolu de problème. Notre problème qui était de réussir à exprimer des propriétés entre des sous-ensembles potentiellement vides de cellules n'a été que repoussé jusque dans le domaine abstrait.

Souvent, une valeur peut être vue comme une formule sur des variables logiques, c'est ce que nous avons fait jusqu'ici. Une idée assez générale que nous développerons dans le chapitre 7

$i \leftarrow 1$ ou 2 aléatoirement
Si $i = 1$ **alors**
  $x \leftarrow 0$
  $A[i] \leftarrow x$
sinon
  $x \leftarrow 1$
  $A[i-1] \leftarrow x$

FIGURE 3.5 : Exemple de programme sur lequel une analyse maladroite peut dériver des propriétés fausses.

pour modifier le domaine abstrait est que chaque formule atomique peut être testée sur un état partiel. Si les variables de cette formule atomique ne sont pas définies dans l'état partiel, c'est que l'un des ensembles de cellules symbolisé est vide et on pourra donc considérer la formule vraie. A partir de là, il est possible de tester des conjonctions, des disjonctions et des négations de formules atomiques.

Le danger. Si on a choisi de ne pas décomposer les valeurs abstraites c'est essentiellement pour des questions de précision de l'analyse. En n'utilisant qu'une seule valeur abstraite pour décrire chacun des sous-ensembles de cellules qu'on souhaite décrire, on sera en mesure de dériver des relations entre eux. Malheureusement, ce n'est pas sans risque.

Supposons qu'on ait obtenu lors d'une analyse la valeur abstraite

$$x < a_1 \wedge a_1 < y$$

où x et y sont des variables du programme et a_1 une variable représentant un ensemble de cellules. S'il est correct de dire que toutes les cellules de cet ensemble ont un contenu supérieur à x et inférieur à y, cela n'implique pas nécessairement $x < y$. En toute rigueur, on aura $x < y$ si et seulement si l'ensemble de cellules est non vide. Dans le cas contraire, on ne saura rien de x ni de y.

Voici le genre de danger auquel nous serons confrontés. L'exemple figure 3.5 montre comment le problème peut devenir dramatique. Il initialise une variable i avec pour valeur 1 ou 2 et suivant cette valeur affecte une variable x et la cellule $A[1]$ avec la valeur 0 ou 1. On peut décrire précisément l'état du tableau en utilisant, ce qui ne semble pas nécessairement naturel, deux ensembles de cellules : l'ensemble $\{ A[\ell] \mid 1 \leq \ell \leq i-1 \}$ et $\{ A[\ell] \mid i \leq \ell \leq 1 \}$. Lorsque $i = 1$ le premier est vide, tandis que le second l'est quand $i = 2$. De fait le premier n'est affecté que par la seconde branche de l'instruction conditionnelle tandis que le second ne sera affecté que par la première. Si on associe ces deux ensembles respectivement aux variables a_1 et a_2 on sera en mesure d'écrire :

$$1 \leq i \leq 2 \wedge a_1 = 1 \wedge a_1 = x \wedge a_2 = x \wedge a_2 = 0$$

dont l'interprétation serait

$$\begin{aligned} & 1 \leq i \leq 2 \\ \wedge\ & \forall \ell, 1 \leq \ell \leq i-1 \ \Rightarrow\ A[\ell] = 1 \\ \wedge\ & \forall \ell, 1 \leq \ell \leq i-1 \ \Rightarrow\ A[\ell] = x \\ \wedge\ & \forall \ell, i \leq \ell \leq 1 \ \Rightarrow\ A[\ell] = 1 \\ \wedge\ & \forall \ell, i \leq \ell \leq 1 \ \Rightarrow\ A[\ell] = 0 \end{aligned}$$

Des deux sous-ensembles de cellules, il y en a toujours un qui est vide pour un état donné. On peut donc très bien écrire

$$\forall \ell_1, \ell_2, 1 \leq \ell_1 \leq i-1 \wedge i \leq \ell_2 \leq 1 \ \Rightarrow\ A[\ell_1] = A[\ell_2]$$

qui est une tautologie : il est impossible d'avoir à la fois un ℓ_1 et un ℓ_2 vérifiant la partie gauche de l'implication. Il n'y a donc pas de mal à ce que la propriété

$$a_1 = a_2$$

soit dérivable de la valeur abstraite. Mais la propriété

$$1 = a_1 = x = a_2 = 0$$

est plus gênante car elle conclut à une contradiction. De cette contradiction on peut déduire toute chose fausse du programme.

La transformation des domaines abstraits va donc nécessiter un travail certain, et va en particulier demander à ce qu'une partie des preuves de corrections soient reconstruites.

3.1.4 Expressivité des choix de représentants

On peut encore augmenter l'expressivité des valeurs abstraites, mais cette fois sans changer le formalisme. Supposons que nous voulions prouver la correction d'un algorithme de copie de tableau. Si l'algorithme copie le contenu d'un tableau A de taille n dans un tableau B, la propriété désirée serait

$$\forall \ell, 1 \leq \ell \leq n \ \Rightarrow\ A[\ell] = B[\ell]$$

Cette propriété est différente de celles qu'on a vu jusqu'à présent. On peut bien identifier deux ensembles de cellules, celles de A et celles de B. Mais nous n'avons pas de propriété entre toutes les premières et toutes les secondes.

Fort heureusement, nous pouvons choisir $\mathcal{R}$ comme nous l'entendons. Ainsi plutôt que de prendre comme choix de représentants n'importe quelle cellule de A et n'importe quelle cellule de B, nous pouvons nous restreindre aux seuls choix de représentants où les deux cellules ont le même indice. Si a et b sont les variables symbolisant respectivement A et B, on peut choisir

$$\mathcal{R}(\sigma) = \left\{ r : \begin{cases} a \mapsto A[\ell] \\ b \mapsto A[\ell] \end{cases} \;\middle|\; 1 \leq \ell \leq \sigma(n) \right\}$$

Ainsi, le domaine abstrait sera en mesure d'affirmer l'équation $a = b$ puisqu'elle est valide pour tout état accessible et pour tout choix de représentants de $\mathcal{R}$. De manière générale, dans

un choix de représentants, chaque représentant peut être choisi dépendamment des autres. Il est ainsi possible de ne considérer des propriétés que sur des relations n-aires sur l'ensemble de cellules à condition d'avoir bien choisi ces relations. En particulier sur les tableaux, par des relations entre les indices.

3.1.5 Conclusion.

Le problème que nous avons à résoudre se décompose en deux sous problèmes. Le premier est de déterminer quels sont les ensembles de cellules à considérer pour l'analyse, ou plus formellement, comment choisir $\mathcal{R}$. Le second est celui de l'adaptation des domaines abstraits. Le formalisme que nous avons choisi impose de redéfinir partiellement les domaines abstraits.

3.2 Modèles concrets de la mémoire

Nous avons utilisé dans l'introduction le concept de cellule mémoire. Cette notion peut être interprétée de différentes manières. Cette interprétation dépend du langage d'entrée de l'analyse. Aussi allons nous préciser l'idée générale que l'on cherche à capturer. Il s'agit de se convaincre qu'il est possible d'avoir un cadre général pour l'analyse des structures de données qui dépend peu du langage d'entrée. On donnera à la fin de cette section deux exemples d'interprétations de la notion de cellule.

En d'autres termes, cette section doit permettre la construction de modèles concrets de mémoire [SJR10] adaptés aux différents langages de programmation. On ne s'intéresse pas encore à l'abstraction de ces modèles qui en seront une représentation approchée, finie et concise. On cherche à avoir des modèles capables de décrire de manière exhaustive l'état d'un programme.

De manière informelle, une *cellule* mémoire peut être considérée comme l'emplacement en mémoire d'une variable locale, globale, d'une cellule de tableau, d'un champ de structure de données, etc. On ne donnera pas ici de définition plus précise : d'un langage de programmation à un autre on aura une compréhension différente du concept.

Classiquement, en interprétation abstraite, les accès aux variables ne sont pas ambigus. Une fois résolus les problèmes de portée des variables, on sait exactement quelle variable est accédée par quelle expression. À l'inverse, les accès aux cellules ne peuvent être résolus statiquement. Une déréférenciation de pointeur ou un accès à un tableau peut être calculé à l'exécution. En cela, la cellule est donc une généralisation de la variable classique.

Nous ne pousserons pas plus loin la généralisation. En particulier, nous allons imposer une restriction. Les cellules doivent être atomique. Chaque écriture en mémoire doit modifier une cellule et une seule, dans sa totalité. Il ne doit pas être possible qu'une écriture modifie partiellement une cellule ou qu'elle altère du même coup une autre cellule.

C'est une propriété qui n'est pas nécessairement garantie dans l'exécution des programmes, notamment ceux en langage C non-standard : le programmeur peut par manipulation de pointeur et de types accéder à des fractions de cellules.

Cette restriction peut s'avérer problématique, bien qu'il soit courant de considérer que sa violation est ou bien une mauvaise pratique de programmation ou bien une erreur. Si le pas à franchir pour supprimer cette contrainte ne semble pas insurmontable, il introduirait néanmoins une complexité supplémentaire notable.

L'état d'un programme peut être décrit par la donnée des valeurs stockées dans chacune des cellules mémoire. Autrement dit, c'est une fonction de l'ensemble des cellules vers l'ensemble des valeurs possibles, ce que nous avons utilisé dans l'introduction.

Il est naturel de supposer que l'ensemble de cellules considéré varie durant l'exécution du programme. Lorsqu'on exécute une fonction, on ajoute les variables locales à cet ensemble de cellules, puis on les retire à la sortie. De la même manière, une allocation dynamique ajoutera les cellules nouvellement allouées, tandis qu'une désallocation les retirera.

Pour fixer les choses, considérons que toutes les cellules déjà allouées ou qui seront allouées dans l'avenir appartiennent à un ensemble universel, infini de cellules que nous noterons $\mathbb{C}$. C'est un stock de cellules dans lequel le programme va se servir chaque fois qu'il a besoin de nouvelles cellules. Le domaine des états est un sous-ensemble de $\mathbb{C}$ et les états sont donc des fonctions partielles. Nous noterons Σ l'ensemble des états concrets, fonctions partielles de $\mathbb{C}$ vers l'ensemble des valeurs $\mathbb{V}$:

$$\Sigma = \mathbb{C} \dashrightarrow \mathbb{V}$$

Notons que pour un langage à pointeurs ou à références, il sera commode que l'ensemble des valeurs contienne l'ensemble des cellules.

Ceci nous donne un contexte général qui doit permettre d'analyser des programmes écrits dans la plupart des langages de programmation. Il doit y avoir en fait un travail à effectuer en amont de l'analyse statique pour adapter la sémantique du langage de programmation. Ceci afin que les lectures et écritures de données ne soient décrites qu'en terme d'accès aux cellules.

En général, la sémantique des langages s'appuiera sur la définition d'opérateurs $\mathcal{C}[exp](\sigma)$ et $\mathcal{E}[exp](\sigma)$ permettant d'évaluer dans l'état $\sigma \in \Sigma$ l'expression *exp* respectivement en une cellule et en sa valeur. Ainsi l'affectation

$$destexp \leftarrow srcexp$$

modifierait un état σ en l'état

$$\sigma' : s \mapsto \begin{array}{ll} \mathcal{E}[srcexp](\sigma) & \text{si } s = \mathcal{C}[destexp](\sigma) \\ \sigma(s) & \text{sinon} \end{array}$$

Exemple 2 (Modèle mémoire simplifié pour un langage à tableaux)**.**
Il est souvent utile d'avoir un langage réduit et simplifié pour expérimenter des analyses. Imaginons un langage dans lequel les tableaux sont des objets du premier ordre. On aurait deux types d'identificateurs : un ensemble d'identificateurs de variables scalaires S et un ensemble d'identificateurs de tableaux unidimensionnels T. Voici à quoi pourrait ressembler un programme dans ce langage :

```
array A
scalar i, x = 0

for i = 1 to n
  A[i] = x
```

pour lequel on aurait $\mathcal{S} = \{i, x\}$ et $T = \{A\}$.

Pour décrire la sémantique du programme en terme de cellules il faut décrire ce qui s'appelle classiquement un environnement : il faut associer une cellule à chaque identificateur de variable scalaire et une liste de cellules à chaque identificateur de tableau. C'est à dire une fonction

$$\rho : (\mathcal{S} \to \mathbb{C}) \cup (T \to (\mathbb{N}^* \to \mathbb{C}))$$

vérifiant pour tout $x, y \in \mathcal{S}$ et tout $A, B \in T$

- $x \neq y \Rightarrow \rho(x) \neq \rho(y)$
- $\rho(x) \notin \rho(A)(\mathbb{N}^*)$
- $A \neq B \Rightarrow \rho(A)(\mathbb{N}^*) \cap \rho(B)(\mathbb{N}^*) = \emptyset$
- $i \neq j \Rightarrow \rho(A)(i) \neq \rho(A)(j)$

ou autrement dit que chaque variable et chaque couple formé d'un tableau et d'un indice décrit une cellule distincte.

On peut ici confondre x et $\rho(x)$ et A et $\rho(A)$ et écrire $A(i)$ au lieu de $\rho(A)(i)$. On aurait pour ce programme

$$\mathcal{S} = \{i, x\} \cup \{A(1), A(2), \ldots\}$$

Exemple 3 (Modèle mémoire concret pour le langage C)**.**
Afin de décrire toute une sémantique du langage C (ce que nous ne ferons pas) il est nécessaire que le modèle mémoire reflète la vision à bas niveau du langage. On peut construire $\mathbb{C}$ comme un stock de variables des différents segments : des cellules $g_1, g_2, \ldots$ pour les variables globales, $p_1, p_2, \ldots$ pour la pile et $t_1, t_2, \ldots$ pour le tas.

Supposons que nous devions analyser le programme suivant.

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  unsigned int i, n = atoi(argv[1]);
  int** A = malloc(n*sizeof(int*));

  for (i = 0 ; i < n ; i++)
  {
    A[i] = malloc(n*sizeof(int));
  }
}
```

Ce programme alloue dynamiquement un tableau à deux dimension de taille $n \times n$ où n est un paramètre entré par l'utilisateur en ligne de commande. Le tableau A peut être arbitrairement grand.

Il y a 5 variables sur la pile : *argc*, *argv*, i, n et A. On peut leur attribuer les cellules p_1 à p_5 respectivement, sans problème particulier. Pour des programmes plus compliqués, il faudrait certainement considérer un pointeur de pile qui serait lui même une cellule.

Chaque utilisation de la fonction *malloc* permet d'accéder à un nouvel ensemble de cellules. Il est probablement plus pertinent de considérer que malloc est une construction de haut niveau et de lui donner une sémantique. Nous pouvons lui faire réserver n'importe quelle plage des cellules $t_i, \ldots, t_{i+j}$ disponible. Les cellules disponibles sont celles qui ne sont pas dans le domaine de définition de l'état concret tandis que *malloc* étend le domaine de définition de ce même état pour y ajouter les nouvelles cellules.

La variable A est un pointeur. On modélise ce pointeur en donnant à la cellule de A un contenu qui est lui même une cellule : la cellule pointée. *malloc* retourne la première cellule de la plage allouée qui est ensuite affectée à A. En terme d'opérateurs sémantiques, cela donne

$$\begin{aligned} C[*A](\sigma) &= \mathcal{E}[A](\sigma) \\ \mathcal{E}[\&A](\sigma) &= C[A](\sigma) \end{aligned}$$

Enfin, pour l'arithmétique de pointeur ou l'indexation, on peut utiliser l'indice des cellules allouées. Ainsi on aura

$$\mathcal{E}[A](\sigma) = t_\iota \Rightarrow \mathcal{E}[A+i](\sigma) = C[A[i]](\sigma) = t_{\iota+\mathcal{E}[i](\sigma)}$$

Avec ces choix, le premier appel à *malloc* va allouer les cellules $t_1, \ldots, t_\jmath$ où $\jmath = \mathcal{E}[n](\sigma)$. Puis le second allouera les cellules $t_{\jmath+1}, \ldots, t_{2\times\jmath}$, et ainsi de suite. Ainsi à la fin du programme on pourra décrire l'ensemble des états σ atteignables

$$\left\{ \sigma : \begin{cases} p_3 & \mapsto \jmath \\ p_4 & \mapsto \jmath \\ p_5 & \mapsto t_1 \\ t_1 & \mapsto t_{\jmath+1} \\ t_2 & \mapsto t_{2\times\jmath+1} \\ & \cdots \\ t_\jmath & \mapsto t_{\jmath\times\jmath+1} \end{cases} \middle| \; \jmath \in \mathbb{N}, \sigma \in \{s_1, \ldots, s_5\} \cup \{h_1, \ldots, h_{\jmath\times(\jmath+1)}\} \right\}$$

Remarque (Mémoire non initialisée)**.**
Il n'est pas nécessaire d'introduire dans l'ensemble $\mathbb{V}$ des valeurs possibles une valeur spéciale exprimant qu'une cellule n'a jamais été initialisée. Lorsqu'une cellule n'est pas initialisée, elle peut avoir n'importe quelle valeur : l'ensemble des états atteignables permettra d'énumérer toutes les combinaisons possibles de toutes les valeurs possibles pour chaque cellule non initialisée.

3.3 Fragmentations

Comme nous l'avons vu dans l'introduction, nous allons faire reposer l'analyse des structures de données sur le choix de sous-ensembles de cellules privilégiés. Nous le ferons dans l'espoir de prouver des propriétés intéressantes sur ces sous-ensembles. On appellera chacun de ces sous-ensembles de cellules des *fragments*

Définition 5 (Fragment)**.**
Un fragment est un sous-ensemble de l'ensemble des cellules $\mathbb{C}$.

Pour les propriétés non-relationnelles, l'analyse des programmes reposera sur la définition de plusieurs fragments. En général, ces fragments seront définis en fonction d'un état. Par exemple,

l'ensemble

$$\{ A[\ell] \mid 1 \leq \ell \leq \sigma(i) \}$$

des cellules du tableau A d'indice compris entre 1 et i où i est défini par rapport à l'état σ.

Pour les propriétés relationnelles il faut recourir à un autre objet. On énumère des n-uplets de cellules pour lesquelles on souhaite trouver des propriétés relationnelles. Un tel ensemble de n-uplets forme une relation n-aire entre les cellules. Une relation unaire est alors un fragment au sens de la définition précédente.

Une valeur abstraite peut être vue comme la donnée

- d'une formule sur un ensemble de variables et
- d'une relation décrivant les cellules pour lesquelles cette formule est vraie (ses modèles).

Une formule serait alors valide si elle l'est pour chaque n-uplet de la relation. Les variables de la formule symbolisent les cellules par lesquelles elles doivent être substituées. On les appellera donc des *variables de synthèse* et on notera $\mathcal{S}$ l'ensemble des variables de synthèse.

Plutôt que des n-uplets de cellules, on peut prendre des fonctions des variables de synthèse vers les cellules. On prendra des ensembles de telles fonctions au lieu des relations. On appellera ces ensembles des *fragmentations*.

Définition 6.
Une fragmentation d'une structure de données est un ensemble de fonctions partielles des variables de synthèse vers les cellules. Autrement dit, une fragmentation est un élément de l'ensemble

$$\mathcal{P}(\mathcal{S} \dashrightarrow \mathbb{C})$$

Comme pour les fragments, les fragmentations seront en général symboliques, définies par rapport à un état concret.

Exemple 4 (Fragmentation de deux tableaux égaux)**.**
Revenons à l'exemple de l'introduction, section 3.1.4 : la copie de tableau. Nous étions arrivés à décomposer la valeur abstraite en la donnée d'une formule :

$$\psi(a, b) \equiv a = b$$

et d'une fragmentation dépendant de l'état concret σ

$$F(\sigma) = \left\{ r : \begin{cases} a \mapsto A[\ell] \\ b \mapsto A[\ell] \end{cases} \;\middle|\; 1 \leq \ell \leq \sigma(n) \right\}$$

L'ensemble des variables de synthèse est $\mathcal{S} = \{a, b\}$. Si l'algorithme de copie de tableau est correct alors les états atteignables à la fin du programme doivent vérifier $\psi(r(a), r(b))$ pour tout $r \in F$ de la fragmentation. C'est à dire que les états doivent vérifier :

$$\forall \ell, 1 \leq \ell \leq n \Rightarrow A[\ell] = B[\ell]$$

Pendant l'analyse du programme, on voudra exprimer l'invariant de la boucle de copie, et probablement séparer la cellule en cours de copie et les autres. On utilisera alors la fragmentation suivante.

$$F(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell] \\ b_1 \mapsto A[\ell] \end{cases} \;\middle|\; 1 \leq \ell < \sigma(i) \right\} \cup \left\{ r : \begin{cases} a_2 \mapsto A[\sigma(i)] \\ b_2 \mapsto A[\sigma(i)] \end{cases} \right\}$$

Dans cette fragmentation, les choix de représentants r sont effectivement partiels puisque lorsque ils sont définis en a_1 et b_1, ils ne le sont pas en a_2 et b_2 et inversement. En outre, chaque fois qu'ils sont définis en a_1 et b_1, ces deux variables représentent des cellules de même indice.

L'usage des fonctions partielles permet d'éviter l'énumération des cas où il est impossible d'associer une cellule à une variable de synthèse. Par exemple lorsque l'on souhaite décrire un fragment qui peut être vide dans certaines configurations d'état. Lorsqu'un fragment est vide toute propriété est vraie à propos de ses cellules. Dans ces configurations, on considérera alors vraie toute propriété à propos des variables pour lesquelles la fonction partielle est indéfinie.

L'exemple montre un autre usage des fonctions partielles. Puisque les choix de représentants ne sont définis en a_1 et b_1 si et seulement si ils ne sont pas définis en a_2 et en b_2 alors cela signifie simplement qu'on ne cherche aucune relation entre les variables a_1 et b_1 et les variables a_2 et b_2. En effet, pour un algorithme de copie de tableau, on cherche à prouver $A[\ell] = B[\ell]$ et nous n'avons aucun besoin de comparer les cellules de a_1 et celles de a_2 ou b_2. Il n'est pas nécessaire que la fragmentation lie ces cellules.

La *projection*, permet d'obtenir les différents fragments qui composent une fragmentation.

Définition 7 (Projection d'une fragmentation)**.**
La projection π_s de la fragmentation F sur la variable de synthèse s est le fragment constitué de l'ensemble des images de ses choix de représentants pour s.

$$\pi_s(F) = \{\, r(s) \mid r \in F \,\}$$

Exemple 5 (Projection d'une fragmentation)**.**
Reprenons la première fragmentation de l'exemple précédent. La projection de la fragmentation F sur les variables de synthèse a et b donne respectivement les fragments

$$\{\, A[\ell] \mid 1 \leq \ell \leq \sigma(n) \,\}$$

et

$$\{\, B[\ell] \mid 1 \leq \ell \leq \sigma(n) \,\}$$

Inversement, à partir d'un ensemble de fragments, on peut construire une fragmentation en liant les cellules des différents fragments. Il y a toutefois plusieurs manière de le faire. Si on souhaite associer les fragments $f_1, f_2, \ldots, f_n$ aux variables $a_1, a_2, \ldots, a_n$ on pourra construire la

fragmentation F suivante.

$$F = \left\{ r : \left\{ \begin{array}{ll} a_1 & \mapsto c_1 \\ a_2 & \mapsto c_2 \\ & \vdots \\ a_n & \mapsto c_n \end{array} \middle| \; c_1 \in f_1, c_2 \in f_2, \ldots, c_n \in f_n \right. \right\}$$

C'est la plus grande fragmentation (pour l'inclusion) dont les projections sont les fragments utilisés pour la construction.

3.4 Processus de synthèse

A partir d'une fragmentation on peut regrouper les cellules des différents fragments dans le but d'extraire les propriétés communes à ces cellules. On appelle cette opération la *synthèse* des propriétés. La synthèse transforme un ensemble d'états concrets en ensemble d'états de synthèse. Tandis que les états concrets décrivent un nombre non borné de cellules, les états de synthèse décrivent un nombre fini de variables de synthèse, une variable de synthèse par fragment. Ces ensembles d'états de synthèse sont ensuite destinés à être approximés dans n'importe quel domaine abstrait, moyennant éventuellement quelques modifications de la fonction d'abstraction classique du domaine.

On part d'un domaine abstrait classique qu'on modifie de sorte à ce que ses abstractions s'appliquent aussi bien à des ensembles de cellules qu'à des variables classiques. On obtient un domaine abstrait plus puissant, capable d'exprimer des propriétés des cellules de structures de données. On s'intéresse ici à la construction des fonctions d'abstraction et de concrétisation de ce nouveau domaine.

On construit les nouvelles fonctions d'abstraction et de concrétisation en les décomposant en deux phases : une phase de synthèse et une phase d'abstraction classique. Cette seconde phase peut dans certains cas être la fonction d'abstraction du domaine abstrait qu'on veut transformer. Dans d'autres, il sera nécessaire de modifier cette fonction. Les modifications nécessaires sont décrites dans le chapitre 7.

On recrée ainsi une nouvelle correspondance Galois, composée de deux autres correspondances de Galois. La première qu'il nous faut construire décrit la relation entre les états concrets et les états de synthèse. La seconde décrit l'approximation des états de synthèse. C'est la correspondance de Galois d'origine éventuellement modifiée.

Cette décomposition en deux phases permet de mettre en lumière la partie de la transformation qu'il faut opérer pour chaque domaine abstrait, et la partie qui est indépendante du domaine abstrait. On espère ainsi simplifier la tâche de l'adaptation des domaines abstraits en réglant une fois pour toutes un certain nombre de problèmes communs à l'adaptation d'un domaine abstrait à l'analyse des structures de données.

La décomposition de la correspondance de Galois nous amène à construire un treillis intermédiaire entre le treillis abstrait et le treillis concret. Nous l'appellerons le *treillis de synthèse*.

Commençons par définir formellement le treillis de synthèse. Un état concret σ est une fonction des conteneurs $\mathbb{C}$ vers les valeurs $\mathbb{V}$. On veut synthétiser cet état en symbolisant par une seule variable un ensemble de cellules défini par une fragmentation. Pour opérer la synthèse on

se contentera de choisir pour chaque variable le contenu d'une des cellules de l'ensemble qu'elle représente, en accord avec la fragmentation choisie. Un état de synthèse sera donc une fonction qui à une variable de synthèse de $\mathcal{S}$ associera une valeur de $\mathbb{V}$.

Il peut arriver qu'une variable doive représenter un fragment vide. Dans ce cas l'état de synthèse ne sera pas défini pour la variable associée.

Définition 8 (État de synthèse)**.**
Un **état de synthèse** $\overline{\sigma}$ est une fonction partielle des variables de synthèse $\mathcal{S}$ vers le domaine $\mathbb{V}$ des valeurs. Nous noterons $\overline{\Sigma}$ l'ensemble des états de synthèse.

$$\overline{\sigma} \in \mathcal{S} \dashrightarrow \mathbb{V}$$

Remarque.
La synthèse s'applique généralement pour regrouper des ensembles de cellules de tableaux ou de structures de données. Mais elle peut aussi synthétiser des ensembles de variables scalaires locales ou globales. Nous n'aborderons pas ici cette possibilité. Cela ne nécessite pas de changement de définition puisque l'on peut toujours considérer des fragments composés uniquement d'une seule cellule, celle de la variable scalaire. Ainsi pour chaque variable scalaire, on définira une variable de synthèse la représentant, et ne représentant qu'elle. Par abus on notera la variable de synthèse de la même manière que la variable concrète.

On va approximer un ensemble d'états concrets par un ensemble d'états de synthèse. Ces ensembles d'états forment un treillis pour l'inclusion d'ensembles.

Définition 9 (Treillis de synthèse)**.**
Le **treillis de synthèse** $(\overline{L}, \subseteq)$ est l'ensemble des ensembles d'états de synthèse ordonné par l'inclusion.

Ce nouveau treillis nous sert d'intermédiaire dans la correspondance de Galois, voir figure 3.6. Nous allons maintenant décrire la correspondance de Galois que nous allons utiliser entre le treillis concret et le treillis de synthèse.

Une fragmentation décrit l'ensemble des choix de représentants auxquels on souhaite s'intéresser. Chaque choix de représentants donne naissance à un état de synthèse potentiellement distinct à partir du même état concret. Simplement, un état de synthèse associe à une variable de synthèse la valeur contenue dans la cellule désignée par le choix de représentants. Autrement dit, un état de synthèse $\overline{\sigma}$ est la fonction partielle composée de l'état σ et d'un choix de représentants r :

$$\overline{\sigma} = \sigma \circ r$$

Exemple 6 (Choix de représentants)**.**
Considérons un état σ de programme décrivant le contenu d'un tableau A de 7 cellules, pério-

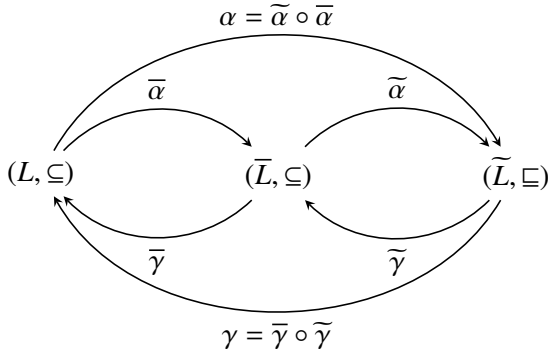


FIGURE 3.6 : Correspondances de Galois entre le treillis concret L, le treillis de synthèse $\overline{L}$ et le treillis abstrait $\widetilde{L}$

dique de période 3 dont les valeurs sont la répétition de la séquence de nombres 1 - 1 - 2 :

$$\sigma : \begin{cases} A[1] \mapsto 1 \\ A[2] \mapsto 1 \\ A[3] \mapsto 2 \\ A[4] \mapsto 1 \\ A[5] \mapsto 1 \\ A[6] \mapsto 2 \\ A[7] \mapsto 1 \end{cases}$$

Soit maintenant le choix de représentants

$$r : \begin{cases} a_1 \mapsto A[3] \\ a_2 \mapsto A[4] \\ a_3 \mapsto A[6] \end{cases}$$

désignant pour les variables de synthèse a_1, a_2 et a_3 trois cellules désignées pour la synthèse. En composant r avec l'état σ on obtient un état $\overline{\sigma}$:

$$\overline{\sigma} : \begin{cases} a_1 \mapsto 2 \\ a_2 \mapsto 1 \\ a_3 \mapsto 2 \end{cases}$$

La construction de $\overline{\sigma}$ est illustrée par la figure 3.7.

Remarque.
La composée peut être indéfinie pour une variable de synthèse s si r n'est pas défini pour s ou si σ n'est pas défini pour $r(s)$. Dans le premier cas, cela signifie simplement que ce choix de représentants ne s'intéresse pas à s, souvent parce que le fragment désigné est vide dans l'état σ. On évitera en revanche le second cas. Que σ ne soit pas défini sur une cellule signifie que la cellule n'a pas encore été allouée. Or on définit des fragmentations en fonction de l'état σ, il n'est pas pertinent de faire un choix de représentants vers une cellule non allouée.

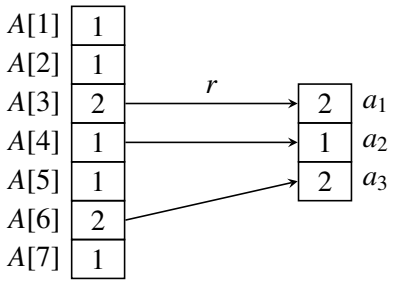


Figure 3.7 : Construction d'un état de synthèse à partir d'un état concret et d'un choix de représentants.

On abstrait un ensemble d'états en générant tous les états de synthèse obtenus à partir d'un état concret et les choix de représentants valides pour cet état. Inversement, on concrétise un ensemble $\overline{X}$ d'états de synthèse en énumérant les états concrets dont toutes les synthèses sont incluses dans $\overline{X}$.

Définition 10 ($\overline{\alpha}$ et $\overline{\gamma}$)**.**
Soient X un ensemble d'états concrets (une valeur concrète) $\overline{X}$ un ensemble d'états de synthèse (une valeur de synthèse) et F une fonction associant une fragmentation à tout état concret. Les fonctions d'abstraction $\overline{\alpha}$ et de concrétisation $\overline{\gamma}$ sont définies par les formules :

$$\overline{\alpha}(X) = \{\, \overline{\sigma} \mid \exists \sigma \in X, \exists r \in F(\sigma), \overline{\sigma} = \sigma \circ r \,\}$$
$$\overline{\gamma}(\overline{X}) = \left\{ \sigma \;\middle|\; \forall r \in F(\sigma), \exists \overline{\sigma} \in \overline{X}, \overline{\sigma} = \sigma \circ r \right\}$$

Exemple 7 (Abstraction et concrétisation)**.**
Reprenons l'exemple précédent, en considérant encore l'état σ décrivant le tableau périodique. Cette fois-ci on utilisera la fragmentation complète suivante :

$$F(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell] \\ a_2 \mapsto A[\ell + 3] \\ a_3 \mapsto A[4] \end{cases} \;\middle|\; 1 \leq \ell \leq 4 \right\}$$

F décrit quatre choix de représentants. Dans chacun des choix de représentants, la variable de synthèse a_3 est toujours associée à la cellule $A[4]$ et le domaine abstrait pourra donc affirmer $a_3 = 1$. On a aussi que les cellules représentées par a_1 et a_2 ont leur indices distant de 3. Puisque le tableau est périodique de période 3 le domaine abstrait pourra donc aussi affirmer l'égalité de a_1 et de a_2. Il peut arriver que a_1 et a_2 soient des *alias* de a_3, c'est à dire qu'ils représentent la même cellule.

Potentiellement, chacun de ces choix de représentants peut produire un état de synthèse différent à partir de σ. Mais trois de ces quatre choix donneront le même état de synthèse. Ce fait est illustré figure 3.8 (a). L'application de $\overline{\alpha}$ à l'ensemble singleton contenant uniquement σ nous

donne donc un ensemble de deux états de synthèse :

$$\overline{\alpha}(\{\sigma\}) = \{\, \sigma \circ r \mid r \in F(\sigma) \,\} = \left\{ \overline{\sigma}_1 : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 1 \\ a_3 \mapsto 1 \end{cases} , \overline{\sigma}_2 : \begin{cases} a_1 \mapsto 2 \\ a_2 \mapsto 2 \\ a_3 \mapsto 1 \end{cases} \right\}$$

On peut sur cet ensemble d'états de synthèse appliquer une fonction d'abstraction classique d'un domaine abstrait. Celui-ci pourrait alors exprimer les égalités vérifiables sur les deux états de synthèse $\overline{\sigma}_1$ et $\overline{\sigma}_2$:

$$a_1 = a_2 \wedge a_3 = 1$$

On peut également appliquer l'opération inverse, la concrétisation, sur l'ensemble d'états de synthèse $\overline{\Sigma} = \{\overline{\sigma}_1, \overline{\sigma}_2\}$. Il s'agit donc de retrouver tous les états σ dont l'abstraction est contenue dans $\overline{\Sigma}$. On peut trouver de tels états en appliquant le processus de construction suivant. On énumère les choix de représentants. Pour chacun d'entre eux, on sélectionne un état de synthèse susceptible d'être la synthèse de l'état que l'on construit. On peut ainsi fixer la valeur associée aux cellules représentantes. On a ainsi partiellement construit l'état. On réitère le processus avec un autre choix de représentants, complétant progressivement l'état concret. Des choix de représentants distincts peuvent sélectionner plusieurs fois la même cellule. Dans ce cas, on sera restreint dans la sélection de l'état de synthèse qui devra être compatible avec ce qu'on a déjà construit. Un tel processus est illustré par la figure 3.8 (b). Sur cette figure, le choix de l'état de synthèse à la première et à la dernière étape est restreint au seul état $\overline{\sigma}_1$.

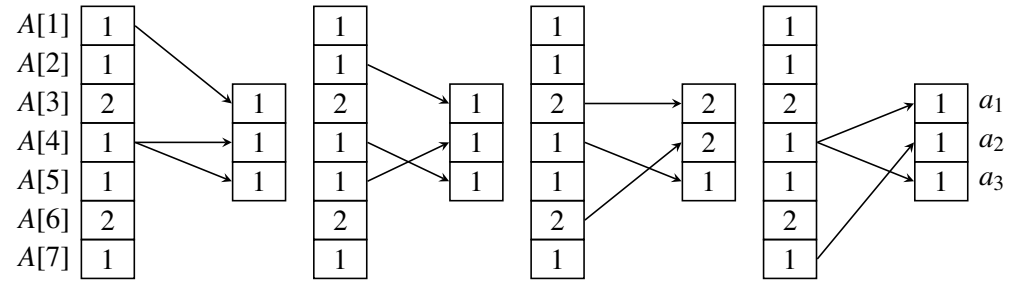
Si on applique la fonction $\overline{\gamma}$ à $\overline{\Sigma}$ on obtiendra quatre états concrets :

$$\overline{\gamma}(\{\overline{\sigma}_1, \overline{\sigma}_2\}) = \left\{ \sigma_1 : \begin{cases} A[1] \mapsto 1 \\ A[2] \mapsto 1 \\ A[3] \mapsto 1 \\ A[4] \mapsto 1 \\ A[5] \mapsto 1 \\ A[6] \mapsto 1 \\ A[7] \mapsto 1 \end{cases} , \sigma_2 : \begin{cases} A[1] \mapsto 1 \\ A[2] \mapsto 2 \\ A[3] \mapsto 1 \\ A[4] \mapsto 1 \\ A[5] \mapsto 2 \\ A[6] \mapsto 1 \\ A[7] \mapsto 1 \end{cases} , \sigma_3 : \begin{cases} A[1] \mapsto 1 \\ A[2] \mapsto 2 \\ A[3] \mapsto 2 \\ A[4] \mapsto 1 \\ A[5] \mapsto 2 \\ A[6] \mapsto 2 \\ A[7] \mapsto 1 \end{cases} , \sigma : \begin{cases} A[1] \mapsto 1 \\ A[2] \mapsto 1 \\ A[3] \mapsto 2 \\ A[4] \mapsto 1 \\ A[5] \mapsto 1 \\ A[6] \mapsto 2 \\ A[7] \mapsto 1 \end{cases} \right\}$$

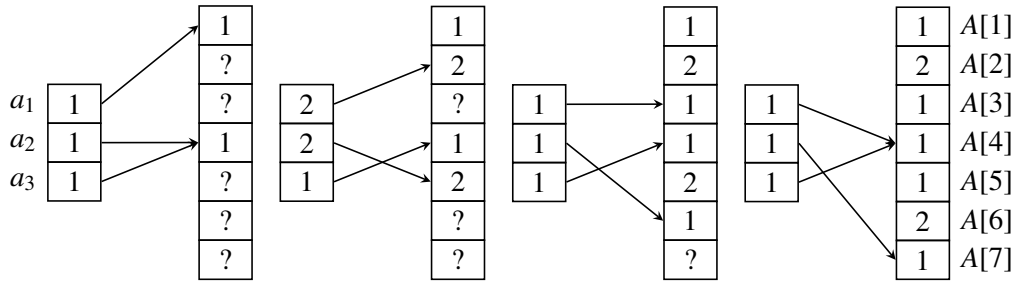
dont l'état σ initial. On remarque la présence de l'état σ_1 décrivant un tableau constant. En effet, un tableau constant égal à 1 vérifie à la fois le critère de périodicité et le fait que le quatrième élément est égal à 1. L'ensemble $\{\sigma_1, \sigma_2, \sigma_3, \sigma\}$ est la sur-approximation par la correspondance de Galois de l'ensemble initial $\{\sigma\}$. Dans cette approximation, on a perdu plusieurs informations :

- L'existence d'une cellule de contenu 2. On sait tout au plus qu'il peut y en avoir une dans le tableau. Plus généralement, on ne connaît plus la multiplicité de chaque valeur.
- La place respective des éléments. S'il y a une cellule de contenu 2 on ne sait plus si c'est la seconde ou la troisième.

Remarque.
Un état concret va être synthétisé en plusieurs états de synthèse. Différents états concrets peuvent donner le même état de synthèse. On n'a pas a priori de relation entre le nombre d'états concrets et le nombre d'états de synthèse.



(a) Synthèse d'un état concret



(b) Reconstruction d'un état concret à partir de ses synthèses

Figure 3.8 : Construction des abstractions et des concrétisation d'états concrets et de synthèse.

Les fonctions d'abstraction $\overline{\alpha}$ et $\overline{\gamma}$ décrivent judicieusement une correspondance de Galois pour tout choix de fragmentation.

Théorème 1.
$(\overline{\alpha}, \overline{\gamma})$ est une correspondance de Galois entre $(L, \subseteq)$ et $(\overline{L}, \subseteq)$.

Démonstration. Soient $\Sigma \in L$ et $\overline{\Sigma} \in \overline{L}$.

- Supposons $\overline{\alpha}(\boldsymbol{\Sigma}) \subseteq \overline{\boldsymbol{\Sigma}}$ et montrons $\Sigma \subseteq \overline{\gamma}(\overline{\Sigma})$.
  Soient $\sigma \in \Sigma$ un état concret et $r \in F(\sigma)$ un choix de représentants.
  On pose $\overline{\sigma} = \sigma \circ r$. On a alors $\overline{\sigma} \in \overline{\alpha}(\Sigma) \subseteq \overline{\Sigma}$ donc $\overline{\sigma} \in \overline{\Sigma}$.
  Par conséquent et par définition de $\overline{\gamma}$ on a que $\sigma \in \overline{\gamma}(\overline{\Sigma})$.
  On a donc prouvé que tout état $\sigma \in \Sigma$ était aussi dans $\overline{\gamma}(\overline{\Sigma})$ et donc que $\Sigma \subseteq \overline{\gamma}(\overline{\Sigma})$.
- Supposons $\boldsymbol{\Sigma} \subseteq \overline{\gamma}(\overline{\boldsymbol{\Sigma}})$ et montrons $\overline{\alpha}(\Sigma) \subseteq \overline{\Sigma}$.
  Soit $\overline{\sigma} \in \overline{\alpha}(\Sigma)$, il existe alors $\sigma \in \Sigma$ et $r \in F(\sigma)$ tels que $\overline{\sigma} = \sigma \circ r$.
  Or $\sigma \in \Sigma \subseteq \overline{\gamma}(\overline{\Sigma})$ donc $\sigma \in \overline{\gamma}(\overline{\Sigma})$.
  D'après la définition de $\overline{\gamma}(\overline{\Sigma})$ cela implique que pour r il existe un état dans $\overline{\Sigma}$ égal à $\overline{\sigma}$ puisque vérifiant l'égalité $\overline{\sigma} = \sigma \circ r$.
  Par conséquent, $\overline{\sigma} \in \overline{\Sigma}$.
  On a donc prouvé que tout état de synthèse $\overline{\sigma} \in \overline{\alpha}(\Sigma)$ était aussi dans $\overline{\Sigma}$ et donc que $\overline{\sigma} \in \overline{\alpha}(\Sigma) \subseteq \overline{\Sigma}$.

Finalement, on a donc $\overline{\alpha}(\Sigma) \subseteq \overline{\Sigma} \;\Leftrightarrow\; \Sigma \subseteq \overline{\gamma}(\overline{\Sigma})$. □

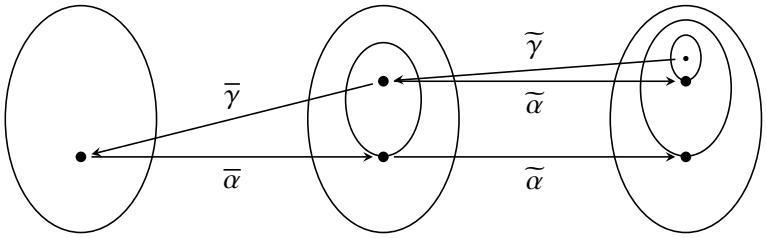


FIGURE 3.9 : Combinaison de la réduction des ensembles d'états de synthèse et de la réduction du domaine abstrait. Les ovales représentent les classes d'équivalence pour les réductions, les points représentent les images.

3.5 Réduction

Chaque ensemble d'états de synthèse a une unique concrétisation pour une fragmentation donnée, qui peut être calculée par la fonction $\overline{\gamma}$. Mais différents ensembles d'états de synthèse peuvent avoir la même concrétisation. La correspondance de Galois assure néanmoins que si on prend tous les ensembles d'états ayant une même concrétisation alors il existe parmi eux un ensemble plus petit que tous les autres. Cet ensemble d'états est d'ailleurs l'image par la fonction d'abstraction $\overline{\alpha}$ de cette concrétisation. Il est intéressant, chaque fois que c'est possible, d'utiliser ce plus petit ensemble d'états de synthèse à la place de tout autre pour assurer une précision maximum des opérateurs abstraits. Dans ce but, nous devons le calculer, c'est à dire calculer la réduction de l'ensemble d'états de synthèse original.

Formellement, la réduction se calcule comme la composée $\overline{\alpha} \circ \overline{\gamma}$. Cette réduction des ensembles d'états de synthèse se combine avec celle du domaine abstrait. Si on applique successivement les fonctions $\widetilde{\gamma}$ et $\overline{\gamma}$ on obtiendra l'ensemble d'états concrets correspondant à une valeur abstraite. Puis si on réapplique $\overline{\alpha}$ et $\widetilde{\alpha}$ on obtiendra la plus petite valeur abstraite ayant cette concrétisation. (Cf. figure 3.9)

Notre objectif en fin de compte sera donc de calculer

$$\widetilde{\alpha} \circ \overline{\alpha} \circ \overline{\gamma} \circ \widetilde{\gamma}$$

Sans la donnée de $\widetilde{\alpha}$ et $\widetilde{\gamma}$ il y a peu de choses que nous soyons en mesure de faire. Le but de ce chapitre étant de déterminer ce que l'on peut faire sans connaissance du domaine abstrait utilisé, nous allons nous concentrer sur la partie connue de la réduction :

$$\overline{\alpha} \circ \overline{\gamma}$$

Nous ne chercherons pas à calculer exactement cette réduction. Même si ce serait intéressant pour la précision de l'analyse, ce n'est pas indispensable et le faire nous causerait d'importants problèmes. On se contentera d'une *réduction partielle*. C'est à dire que l'on va chercher un ensemble d'états de synthèse qui ait la même concrétisation que l'ensemble d'états original, qui soit plus petit que lui, mais pas nécessairement le plus petit.

Calculer la réduction c'est répondre à la question suivante : « quels sont les états de synthèse que nous pouvons retirer de notre ensemble sans changer sa concrétisation ? » Pour répondre à

cette question, nous allons chercher à caractériser quelques états de synthèse qui ne « survivent » pas à la réduction. Il nous suffira alors de prouver en appliquant $\overline{\alpha} \circ \overline{\gamma}$ que ces états ne sont pas dans l'image.

Supposons que l'on parte d'un ensemble d'états de synthèse $\overline{X}$. Reprenons les définitions de la section précédente.

$$\overline{\alpha}(X) = \{\, \overline{\sigma} \mid \exists \sigma \in X,\ \exists r \in F(\sigma),\ \overline{\sigma} = \sigma \circ r \,\}$$
$$\overline{\gamma}(\overline{X}) = \left\{ \sigma \;\middle|\; \forall r \in F(\sigma),\ \exists \overline{\sigma} \in \overline{X},\ \overline{\sigma} = \sigma \circ r \right\}$$

Un état de synthèse survivra à la réduction si, par définition de $\overline{\alpha}$ il peut être produit par la synthèse d'un état de la concrétisation. Un état sera lui même dans la concrétisation, si tous les états de synthèse qui peuvent être produits à partir de lui sont dans $\overline{X}$. Par suite, tous les états de synthèse servant à la construction d'un état concret se retrouveront dans la réduction. Nous allons donc chercher quels sont les états de synthèse qui ne peuvent pas servir à la construction d'un état concret.

Exemple 8 (Réduction d'un ensemble d'états de synthèse)**.**
Supposons que nous avons un tableau A de quatre cellules que nous avons découpé en trois fragments $f_1 = \{A[1], A[2]\}$, $f_2 = \{A[2], A[3]\}$ et $f_3 = \{A[3], A[4]\}$, c'est à dire avec la fragmentation

$$F = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[\ell_2] \\ a_3 \mapsto A[\ell_3] \end{cases} \;\middle|\; \begin{array}{lll} 1 \leq & \ell_1 & \leq 2 \wedge \\ 2 \leq & \ell_2 & \leq 3 \wedge \\ 3 \leq & \ell_3 & \leq 4 \end{array} \right\}$$

Supposons également que nous ayons à réduire l'ensemble d'états de synthèse $\overline{X}$ suivant.

$$\left\{ \overline{\sigma}_1 : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 1 \\ a_3 \mapsto 1 \end{cases}, \overline{\sigma}_2 : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 1 \\ a_3 \mapsto 2 \end{cases}, \overline{\sigma}_3 : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 2 \\ a_3 \mapsto 2 \end{cases}, \overline{\sigma}_4 : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 3 \\ a_3 \mapsto 2 \end{cases} \right\}$$

La figure 3.10 illustre notre exemple. On commence par appliquer $\overline{\gamma}$ à $\overline{X}$ et on obtient l'ensemble d'états concrets suivant.

$$\overline{\gamma}(\overline{X}) = \left\{ \sigma_1 : \begin{cases} A[1] \mapsto 1 \\ A[2] \mapsto 1 \\ A[3] \mapsto 1 \\ A[4] \mapsto 1 \end{cases}, \sigma_2 : \begin{cases} A[1] \mapsto 1 \\ A[2] \mapsto 1 \\ A[3] \mapsto 1 \\ A[4] \mapsto 2 \end{cases}, \sigma_3 : \begin{cases} A[1] \mapsto 1 \\ A[2] \mapsto 1 \\ A[3] \mapsto 2 \\ A[4] \mapsto 2 \end{cases} \right\}$$

Dans cet ensemble, σ_1 est construit en utilisant le même état de synthèse $\overline{\sigma}_1$ pour tous les choix de représentants. σ_2 est une combinaison de $\overline{\sigma_1}$ et $\overline{\sigma_2}$ tandis que σ_3 est une combinaison de $\overline{\sigma_2}$ et de $\overline{\sigma_3}$. Si on applique $\overline{\alpha}$ à cet ensemble, on retrouvera les $\overline{\sigma_1}$, $\overline{\sigma_2}$ et $\overline{\sigma_3}$ que nous venons d'utiliser pour la construction des états concrets.

$$\overline{\alpha} \circ \overline{\gamma}(\overline{X}) = \left\{ \overline{\sigma}_1 : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 1 \\ a_3 \mapsto 1 \end{cases}, \overline{\sigma}_2 : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 1 \\ a_3 \mapsto 2 \end{cases}, \overline{\sigma}_3 : \begin{cases} a_1 \mapsto 1 \\ a_2 \mapsto 2 \\ a_3 \mapsto 2 \end{cases} \right\}$$

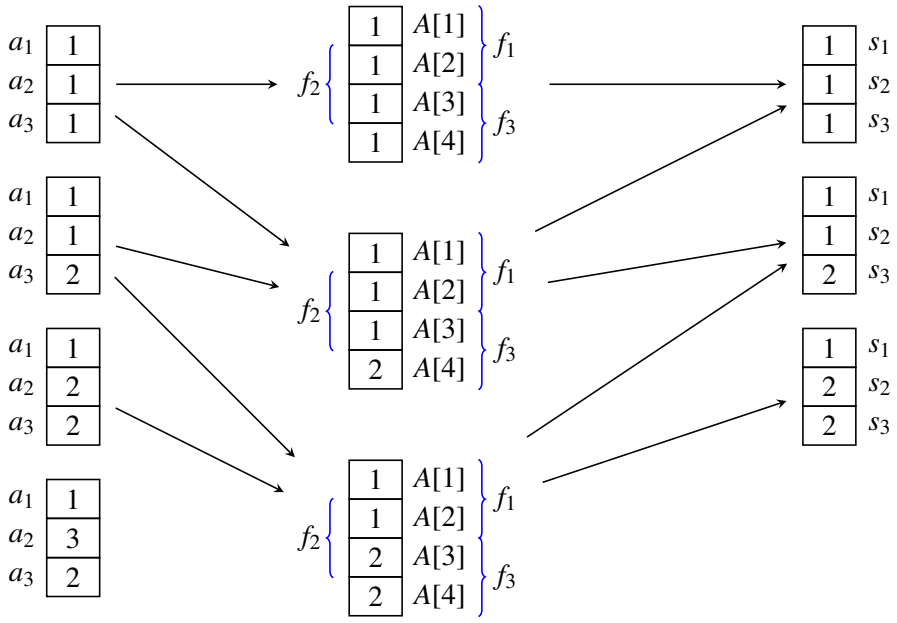


FIGURE 3.10 : Cette figure illustre l'opération de réduction. On part d'un ensemble d'état de synthèse (à gauche) qu'on combine pour former des états concrets (au milieu) avant de les redécomposer en les états de synthèse du résultat. (à droite) Le dernier état de synthèse en bas à gauche ne trouve sa place dans aucune combinaison. Il ne peut donc être reconstruit lors de l'abstraction et il est perdu dans le processus de réduction.

L'état $\overline{\sigma}_4$ n'a pas survecu à la réduction. En effet, il est impossible de construire un état concret avec $\overline{\sigma}_4$. Un tel état concret devrait avoir ou bien $A[2] = \overline{\sigma}_4(a_2) = 3$ ou bien $A[3] = \overline{\sigma}_4(a_2) = 3$. Nous pouvons montrer que ceci est impossible. Il existe des choix de représentants pour notre fragmentation envoyant a_1 sur $A[2]$ et a_3 sur $A[3]$. Pour construire notre état concret, il faudrait qu'on ait des états de synthèse pour de tels choix de représentants. Un tel état de synthèse $\overline{\sigma}'$ devraient alors vérifier selon le cas $\sigma'(a_1) = 3$ ou $\sigma'(a_3) = 3$. Nous n'avons pas de tels états, il est donc impossible de construire cet état concret, $\overline{\sigma}_4$ est inutile et devra disparaitre.

Pour terminer cet exemple, notons quelques éléments qui nous permettent de dire qu'on n'a oublié aucun état dans la concrétisation de l'ensemble initial. D'abord, quel que soit l'état de synthèse $\overline{\sigma} \in \overline{X}$, on a toujours $\overline{\sigma}(a_1) = 1$. Par conséquent, il n'y a aucun autre choix possible que d'avoir $A[1] = A[2] = 1$. Ensuite, notons que si $A[4] = 1$, alors le tableau est forcément constant. En effet dans ce cas, pour les choix de représentants envoyant a_3 sur $A[4]$, seul l'état $\overline{\sigma}_1$ convient et doit être utilisé pour tous les choix de représentant. Or cet état vaut 1 sur toutes les variables de synthèse. L'état concret est donc forcé de prendre cette valeur pour toutes les cellules du tableau. Enfin, si au contraire $A[4] \neq 1$ on doit avoir $A[4] = 2$ puisque c'est l'unique valeur restante. Il ne reste plus de choix que pour $A[3]$ qui pourra être fixé à 1 ou à 2, mais pas à 3 comme nous l'avons remarqué précédemment.

Cet exemple montre comment on peut calculer « à la main » la réduction. L'exercice est toutefois fastidieux et ne peut être automatisé : l'ensemble d'états de synthèse et l'ensemble d'états concrets peuvent être infinis. Nous allons maintenant chercher des règles générales nous permettant de caractériser des sous-ensembles des états de synthèses qui ne survivent pas à la réduction. L'objectif est de trouver des règles qui ne nécessitent pas que l'on calcule la concrétisation. Mais avant de chercher les cas où des états de synthèse peuvent être retirés, commençons par chercher les cas dans lesquels on ne peut en retirer aucun.

Pour des fragmentations bien particulières, il n'y a pas besoin de réduction : il s'agit du cas où la fragmentation décrit une partition de l'ensemble des cellules. Fixons une fragmentation non symbolique F et prenons pour hypothèse que ses choix de représentants sont totaux. Ainsi, les états de synthèse seront toujours définis sur toutes les variables de synthèse et aucun fragment n'est jamais vide. Prenons également pour hypothèse que deux variables différentes ne sont jamais associées à une même cellule dans l'ensemble de la fragmentation. Autrement dit que deux fragments ne se chevauchent jamais, leurs ensembles de cellules sont toujours disjoints. Sous ces conditions, on peut toujours à partir d'un seul état de synthèse total construire au moins un état concret. Cet état est celui dans lequel le tableau est constant sur chaque fragment et égal à la valeur de l'état de synthèse pour ce fragment. Si on part de l'état $\overline{\sigma}$ défini sur les variables $\mathcal{S} = \{s_1, \ldots, s_n\}$ on construit l'état :

$$\sigma : \begin{cases} c & \mapsto \overline{\sigma}(a_1) \text{ si } c \in \pi_{a_1}(F) \\ & \vdots \\ c & \mapsto \overline{\sigma}(a_n) \text{ si } c \in \pi_{a_n}(F) \end{cases}$$

où $\pi_{a_i}(F)$ est la projection sur le fragment représenté par a_i pour la fragmentation F. La définition est valide dans la mesure où, puisque les fragments sont disjoints, une cellule c ne peut être que dans une seule des projections. En outre puisque les choix de représentants sont totaux, les $\pi_{a_i}(F)$ ne sont jamais vides. $\overline{\sigma}$ étant lui aussi total, on a bien une valeur pour toutes les cellules de σ.

Sous ces deux hypothèses, qu'il n'y a pas de chevauchement dans les fragmentations et que la vacuité des fragments est constante, nous avons déduit que n'importe quel état de synthèse suffit à lui seul à construire un état concret. Par conséquent, tout état de synthèse est conservé durant la réduction. Autrement dit, la réduction est la fonction identité ou encore, la correspondance de Galois est en fait une insertion de Galois. Ceci est précisément le cas traité dans [GDD+04].

On peut élargir ce résultat à une classe de fragmentations symboliques. La construction précédente serait une définition récursive de σ si on remplaçait les occurrences de F par $F(\sigma)$. En toute généralité, il se pourrait qu'il n'y ait pas de σ vérifiant cette nouvelle définition, quand bien même on aurait la garantie que la fragmentation vérifie les hypothèses quel que soit le choix de ce σ.

Ceci tient au fait que l'état de synthèse $\overline{\sigma}$ peut potentiellement servir à la construction d'un ensemble d'états concrets sur lesquels la fragmentation varie. Mais souvent, et ce sera toujours le cas dans le reste de cette thèse, un état de synthèse contiendra l'information nécessaire à déterminer complètement la fragmentation. Plus formellement, tous les états concrets qui peuvent être construits à partir d'un état de synthèse $\overline{\sigma}$ auront la même fragmentation, qu'on pourra noter $F(\overline{\sigma})$. On dira que la fragmentation *n'est pas abstraite par la synthèse*.

Définition 11 (Fragmentation non abstraite par la synthèse)**.**
On dira qu'une fragmentation n'est pas abstraite par la synthèse s'il existe pour chaque état de synthèse $\overline{\sigma}$ une fragmentation $F(\overline{\sigma})$ telle que :

$$\forall \sigma, \forall r \in F(\sigma) \quad \overline{\sigma} = \sigma \circ r \;\Rightarrow\; F(\sigma) = F(\overline{\sigma})$$

Pour qu'une fragmentation ne soit pas abstraite par la synthèse, il suffit par exemple que les cellules qui déterminent la fragmentation (classiquement les variables d'indice ou les pointeurs) soient toujours présentes dans les états de synthèse, dans un fragment singleton. Par exemple, supposons que nous ayons une variable d'indice nommée i dans un programme dont on se sert pour délimiter des fragments. Cette variable sera présente dans les états concrets, les états de synthèse devront également avoir une variable de synthèse la représentant dans leur domaine, et les choix de représentants enverront toujours la première sur la seconde. La définition est plus générale et permet par exemple que $A[i]$ soit elle-même utilisée pour délimiter un fragment à condition qu'il y ait une variable de synthèse qui lui soit toujours associée.

Ainsi, notre premier résultat s'applique sur les fragmentations symboliques non abstraites par la synthèse. Fixons une fragmentation $F(\sigma)$ non abstraite par la synthèse et vérifiant nos deux hypothèses. A partir d'un seul état de synthèse $\overline{\sigma}$ on peut toujours construire un état concret σ :

$$\sigma : \begin{cases} c & \mapsto \overline{\sigma}(a_1) \text{ si } c \in \pi_{a_1}(F(\overline{\sigma})) \\ & \vdots \\ c & \mapsto \overline{\sigma}(a_n) \text{ si } c \in \pi_{a_n}(F(\overline{\sigma})) \end{cases}$$

Avec une telle fragmentation, la correspondance reste une insertion de Galois.

Grâce à ce fait, on sait que pour chercher ce que la réduction implique, il faut chercher des conséquences des chevauchements et de la présence de fragments vides. Commençons par ce

dernier cas et intéressons nous aux fragmentations symboliques non abstraites par la synthèse où la vacuité d'un fragment dépend de l'état concret dans lequel on l'évalue.

Pour chaque état $\overline{\sigma}$ nous pouvons regarder si un fragment associé à une variable s est vide dans la fragmentation $F(\overline{\sigma})$. Si c'est le cas, alors $\overline{\sigma}$ doit être indéfini en s. Si au contraire s a toujours une image pour tous les choix de représentants de la fragmentation, alors il faudra que $\overline{\sigma}$ soit toujours défini en s. Entre ces deux cas, si s a parfois une image dans la fragmentation, et d'autre fois n'en a pas, on ne peut pas dire grand chose et on ne cherchera pas plus loin les conséquences de la réduction.

En résumé, ce que nous savons dire des conséquences de la présence des parties vide se résume à la formule suivante, qui donne la propriété que doivent vérifier les états de synthèse $\overline{\sigma}$ pour être dans la réduction.

$$
\begin{array}{l}
\forall s \in \mathcal{S}, \\
\quad (\forall r \in F(\overline{\sigma}),\ r(s) \text{ indéfini}) \ \Rightarrow\ \overline{\sigma}(s) \text{ indéfini} \\
\quad (\forall r \in F(\overline{\sigma}),\ r(s) \text{ défini}) \ \Rightarrow\ \overline{\sigma}(s) \text{ défini}
\end{array}
$$

Cette règle peut être appliquée et permet de réduire partiellement l'ensemble d'états de synthèse sans passer par la concrétisation. Elle peut être utilisée pour renforcer la valeur abstraite. Il faut pour cela que l'on puisse exprimer une condition suffisante de vacuité d'un fragment en fonction des variables du programme. Si la condition est vraie, le fragment est vide et on peut affirmer ce que l'on veut du fragment. Inversement, si le domaine abstrait peut déterminer des conditions sous lesquelles les propriétés sur le fragment sont contradictoires, il peut affirmer la vacuité du fragment. On pourra alors conclure que la condition de vacuité est vraie, et renforcer la valeur abstraite avec cette condition.

Exemple 9 (Première règle de réduction)**.**
Supposons que nous ayons à analyser un programme manipulant une variable d'indice i et un tableau A et que nous ayons à réduire l'ensemble d'états de synthèse suivant.

$$
\left\{ \overline{\sigma}_1 : \begin{cases} \bar{i} \mapsto 2 \\ a \mapsto 2 \end{cases} \quad \overline{\sigma}_2 : \begin{cases} \bar{i} \mapsto 2 \\ \text{indéfini en } a \end{cases} \quad \overline{\sigma}_3 : \begin{cases} \bar{i} \mapsto 1 \\ a \mapsto 1 \end{cases} \quad \overline{\sigma}_4 : \begin{cases} \bar{i} \mapsto 1 \\ \text{indéfini en } a \end{cases} \quad \overline{\sigma}_5 : \begin{cases} \bar{i} \mapsto 0 \\ a \mapsto 0 \end{cases} \right\}
$$

Supposons également que nous utilisions la fragmentation symbolique suivante :

$$
F(\sigma) = \begin{cases} \left\{ r : \begin{cases} \bar{i} & \mapsto i \\ a & \mapsto A[\ell] \end{cases} \;\middle|\; 1 \leq \ell < \sigma(i) \right\} & \text{si } \sigma(i) > 1 \\ \left\{ r : \begin{cases} \bar{i} \mapsto i \\ \text{indéfini en } a \end{cases} \right\} & \text{sinon} \end{cases}
$$

C'est une forme de fragmentation très courante. Elle cherche à désigner les cellules d'indices compris entre 1 et i mais ne souhaite pas pour autant ignorer les cas où la variable i n'est pas strictement supérieure à 1, c'est à dire les cas où le fragment de tableau est vide.

D'abord, remarquons que cette fragmentation n'est pas abstraite par la synthèse. En effet, la fragmentation est totalement déterminée par la connaissance de la variable d'indice i que l'on retrouve dans les états de synthèse sous l'écriture $\bar{i}$. Ainsi tout état de synthèse $\overline{\sigma}$ ne peut

correspondre qu'à un état concret σ dans lequel i a pour valeur $\overline{\sigma}(\bar{i})$. Quelque soit cet état σ, la fragmentation est la même.

Appliquons la règle de réduction à chacun des cinq états de synthèse. Puisque la fragmentation n'est pas abstraite par la synthèse, nous pourrons librement noter $F(\overline{\sigma}_i)$ l'unique fragmentation associée à chacun des $\overline{\sigma}_i$.

- L'état $\overline{\sigma}_1$ survit à la réduction. $\overline{\sigma}_1$ est défini en a et dans $F(\overline{\sigma}_1)$, tous les choix de représentants le sont également.
- En revanche, l'état $\overline{\sigma}_2$ ne survit pas à la réduction. Il est indéfini en a or dans $F(\overline{\sigma}_2) = F(\overline{\sigma}_1)$ les choix de représentants sont définis. Pour cette raison, il est impossible d'utiliser $\overline{\sigma}_2$ dans la construction d'un état concret et la seconde implication de notre règle nous le rappelle.
- Les états $\overline{\sigma}_3$ et $\overline{\sigma}_5$ ne survivent pas non plus à la réduction, mais cette fois-ci, à cause de la première implication de la règle. Dans ces états $\bar{i}$ est égal à 1 et 0 respectivement. Les choix de représentants des fragmentations $F(\overline{\sigma}_3)$ et $F(\overline{\sigma}_5)$ sont donc indéfinis en a là où ces deux états de synthèse sont bien définis.
- L'état $\overline{\sigma}_4$ survit à la réduction, étant indéfini en a comme les choix de représentants de $F(\overline{\sigma}_4)$.

Finalement, la réduction produira l'ensemble d'états

$$\left\{ \overline{\sigma}_1 : \begin{cases} \bar{i} & \mapsto 2 \\ a & \mapsto 2 \end{cases}, \overline{\sigma}_4 : \begin{cases} \bar{i} \mapsto 1 \\ \text{indéfini en } a \end{cases} \right\}$$

Si on se restreint aux quatre premiers états $\overline{\sigma}_1, \ldots, \overline{\sigma}_4$, cet ensemble aurait très bien pu venir de la concrétisation d'une valeur abstraite exprimant

$$a = i \wedge i \in [1, 2]$$

Or la connaissance de la fragmentation nous permet de savoir que a ne représente aucune cellule quand $i = 1$. De là, on peut en déduire $a = 2$. C'est ce qui ressort de l'ensemble d'états réduit :

- ou bien $i = 1$ et le fragment est vide,
- ou bien $i = 2$, le fragment n'est pas vide et ses cellules ont pour valeur 2.

Intéressons nous maintenant aux conséquences des chevauchements dans une fragmentation. Il y a chevauchement lorsque deux fragments distincts ont une intersection non-vide. Si s_1 et s_2 sont deux variables de synthèse, F une fragmentation, il y a chevauchement entre les fragments représentés par s_1 et s_2 si

$$\pi_{s_1}(F) \cap \pi_{s_2}(F) \neq \emptyset$$

En termes de choix de représentants, cela signifie qu'il existe deux choix de représentants dans la fragmentation, pas forcément distincts, qui choisissent la même cellule, respectivement pour s_1 et pour s_2.

$$\exists r_1, r_2 \in F,\ r_1(s_1) = r_2(s_2)$$

Si on synthétise un état concret σ avec cette fragmentation, on obtiendra les états de synthèse pour r_1 et r_2. Ces deux états de synthèse produits doivent donc associer la même valeur, respectivement à s_1 et à s_2.

$$\sigma \circ r_1(s_1) = \sigma \circ r_2(s_2)$$

Ceci nous donne un nouveau critère que les états de synthèse doivent vérifier après réduction. S'il y a chevauchement, les deux fragments doivent avoir au moins une valeur en commun. On pourra en particulier tirer profit de la contraposée de cette implication. Si deux fragments n'ont pas au moins une valeur en commun, alors ils ne se chevauchent pas. Si en regardant la valeur abstraite, on peut dire que deux variables ont des propriétés incompatibles, alors il est possible de renforcer cette valeur abstraite avec une condition de non-chevauchement. Deux variables ont des propriétés incompatibles si leur égalité conduit à une valeur abstraite vide.

Il y a un cas particulier de chevauchement pour lequel on peut déduire plus d'information : le recouvrement. Un fragment est recouvert si chacune de ses cellules appartient à au moins un autre fragment. Formellement, le fragment représenté par une variable s est recouvert par ceux représentés par les variables $s_1, \ldots, s_n$ si

$$\pi_s(F) \subseteq \bigcup_i \pi_{s_i}(F)$$

En termes de choix de représentants, cela signifie que pour tout choix de cellule associé à s il existera le même choix pour l'un des s_i.

$$\forall r \in F,\ \exists r' \in F,\ \bigvee_i \left(r'(s_i) = r(s)\right)$$

Par conséquent, si un état de synthèse donne une valeur à s, il faudra par ailleurs qu'il existe un état dans la fragmentation qui donne la même valeur à l'un des s_i. En effet, cet état doit être la synthèse d'un état concret pour un choix de représentant r. Cet état concret associe donc cette valeur à $r(s)$. Mais il existe au moins un r' qui choisit la même cellule pour s_i. Il y a donc un autre état de synthèse fabriqué à partir de r' qui associe la même valeur à s_i.

$$\forall r \in F,\ \exists r' \in F,\ \bigvee_i \left(\sigma \circ r'(s_i) = \sigma \circ r(s)\right)$$

A partir de ce fait, on peut construire une nouvelle règle de réduction. Si on peut trouver un ensemble de fragments recouvrant un fragment cible, alors on sait que les valeurs du second sont incluses dans toutes celles des premiers. Ses propriétés sont alors une disjonction des propriétés des fragments recouvrant. On aura intérêt à prendre un ensemble recouvrant minimal afin que cette disjonction soit la plus forte possible.

Cette troisième règle de réduction est précisément le cas qui a été traité en exemple en début de section. Le fragment représenté par a_2 est recouvert par ceux représentés par a_1 et a_3. En effet, pour les 8 choix de représentants de la fragmentation, a_2 représente toujours ou bien $A[2]$ ou bien $A[3]$ et ces deux cellules sont également associées à a_1 et a_3 respectivement. Par conséquent, les valeurs associées à a_2 ne peuvent être autres que celles qui sont également associées à a_1 ou à a_3. Or la valeur $\overline{\sigma}_4(a_2) = 3$ n'est associée à a_1 ou a_3 pour aucun des $\overline{\sigma}$. $\overline{\sigma}_4$ n'appartient donc pas à la réduction et il peut être retiré.

Conclusion. Nous disposons de trois règles pour la réduction des ensembles d'états de synthèse.

- Lorsqu'un fragment est vide les propriétés les plus fortes possibles doivent lui être attribuées et inversement des propriétés incohérentes sur un fragment implique la vacuité de celui-ci.

- Si deux fragments n'ont pas de valeur en commun et en particulier s'ils ont des propriétés incompatibles ils ne peuvent pas se chevaucher.
- Lorsqu'un ensemble de fragments recouvre un autre fragment, les propriétés du fragment recouvert doivent être plus forte que la disjonction des propriétés des fragments recouvrants.

A l'aide de ces règles, on peut améliorer la réduction des valeurs abstraites en développant les algorithmes adéquats. Pour la troisième règle, le renforcement des fragments recouverts se fera grâce à des opérations que les domaines abstraits devront implémenter (Cf. section 4.2.4). Le calcul des ensembles recouvrant dépend de la forme de la fragmentation et on donnera un tel algorithme en section 5.5.3.

Même en appliquant ces trois règles, on obtient une approximation de la réduction très imprécise. Des états de synthèse ne survivant pas à la réduction peuvent très bien n'enfreindre aucune de ces règles. Pour l'élaboration d'autres règles, il faut néanmoins garder à l'esprit qu'une nouvelle règle n'est utile que si nous savons la répercuter sur les valeurs abstraites.

3.6 Sémantique sur les états de synthèse

Dans cette section, nous chercherons à développer une sémantique sur les états de synthèse, qui soit une approximation de la sémantique sur les états concrets. Nous avons décomposé la correspondance de Galois en deux parties, nous pouvons décomposer l'approximation de la sémantique également. Notre objectif est donc de trouver un moyen de décrire comment un ensemble d'états de synthèse est affecté par une instruction. Commençons par un exemple illustratif.

Exemple 10.
Soient $c_1, c_2, c_3, c_4, \ldots$ des cellules et σ un état concret, et F une fragmentation :

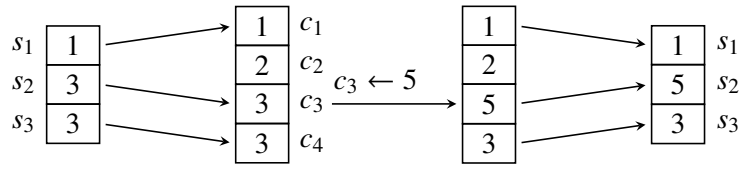
$$\sigma : \begin{cases} c_1 & \mapsto 1 \\ c_2 & \mapsto 2 \\ c_3 & \mapsto 3 \\ c_4 & \mapsto 3 \\ & \vdots \end{cases}, \; F(\sigma) = \left\{ r_1 : \begin{cases} s_1 \mapsto c_1 \\ s_2 \mapsto c_3 \\ s_3 \mapsto c_4 \end{cases}, \; r_2 : \begin{cases} s_1 \mapsto c_1 \\ s_2 \mapsto c_3 \\ s_3 \mapsto c_3 \end{cases} \right\}$$

Considérons une affectation

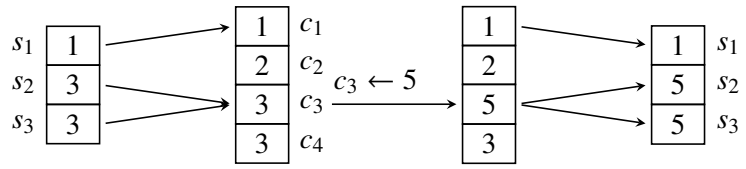
$$c_3 \leftarrow 5$$

Si on applique cette affectation à l'état σ on obtiendra $\sigma' = \sigma[c_3 \mapsto 5]$:

$$\sigma' : \begin{cases} c_1 & \mapsto 1 \\ c_2 & \mapsto 2 \\ c_3 & \mapsto 5 \\ c_4 & \mapsto 3 \\ & \vdots \end{cases}$$



(a) Premier cas



(b) Second cas

Figure 3.11 : Application de la sémantique de $c_3 \leftarrow 5$ sur le même état de synthèse mais pour deux choix de représentants différents. Dans le cas (a) s_3 n'est pas modifée tandis que dans le cas (b) elle est modifiée au même titre que s_2.

Observons les conséquences de cette affectation sur la synthèse des états. Ici, il n'y a qu'un seul état de synthèse $\overline{\sigma}$ qui puisse être obtenu de σ :

$$\overline{\sigma} = \sigma \circ r_1 = \sigma \circ r_2 : \begin{cases} s_1 \mapsto 1 \\ s_2 \mapsto 3 \\ s_3 \mapsto 3 \end{cases}$$

Mais après l'affectation, la synthèse de l'état σ' produit deux états au lieu d'un seul :

$$\overline{\sigma}'_1 = \sigma' \circ r_1 : \begin{cases} s_1 \mapsto 1 \\ s_2 \mapsto 5 \\ s_3 \mapsto 3 \end{cases} , \overline{\sigma}'_2 = \sigma' \circ r_2 : \begin{cases} s_1 \mapsto 1 \\ s_2 \mapsto 5 \\ s_3 \mapsto 5 \end{cases}$$

Changeons de point de vue et supposons que nous soyons partis de l'ensemble d'états de synthèse $\{\overline{\sigma}\}$ et que nous cherchions à approximer l'effet de l'affectation. Nous en aurions conclu que l'affectation a transformé notre état initial $\overline{\sigma}$ en les deux états de synthèse $\overline{\sigma}'_1$ et $\overline{\sigma}'_2$. C'est ce qu'illustre la figure 3.11. En toute rigueur, il nous faudrait considérer tous les états de la concrétisation de $\{\overline{\sigma}\}$. Mais, pour cet exemple, quel que soit l'état que nous prenions, si on lui applique la sémantique de l'affectation et qu'on synthétise le résultat, on obtient les deux mêmes états de synthèse.

On peut déjà dégager quelques remarques de cet exemple simple. Dans la fragmentation F, s_2 est toujours envoyé sur c_3 et représente donc un fragment singleton. Quand c_3 est modifié, s_2 est donc sûrement modifié de la même manière. La variable s_1 représente également un fragment singleton contenant c_1. c_1 n'est pas modifiée par l'affectation et s_1 ne l'est plus non plus. En revanche, la variable s_3 peut représenter ou bien c_3 ou bien c_4. Selon le cas, elle sera ou bien modifiée ou bien non.

Dans le meilleur des cas, nous pouvons directement appliquer la même sémantique aux états concrets et aux états de synthèse. Ce meilleur des cas, c'est lorsque deux conditions sont réunies.

La première, c'est que toutes les cellules concernées par une instruction aient un fragment propre, un fragment singleton ne contenant qu'elles. La seconde, c'est qu'aucun autre fragment ne contienne ces cellules. Dans ces conditions, affecter ou lire une cellule c'est comme affecter ou lire le singleton qui lui correspond.

Reprenons la sémantique de l'affectation donnée dans la section 3.2. L'instruction

$$destexp \leftarrow srcexp$$

modifie un état σ en σ' :

$$(destexp \leftarrow srcexp)(\sigma) = \sigma' : \begin{cases} s \mapsto \mathcal{E}[srcexp](\sigma) \text{ si } s = C[destexp](\sigma) \\ s \mapsto \sigma(s) \text{ sinon} \end{cases}$$

Sous nos deux conditions, on peut l'appliquer directement aux états de synthèse. La première condition nous dit que chaque cellule utilisée dans cette instruction, que ce soit dans *destexp* ou *srcexp*, est toujours représentée par un fragment singleton. Il existe donc une fonction que nous noterons r^{-1}, qui à chaque cellule associe lorsque c'est possible la variable de synthèse correspondant au fragment singleton contenant cette cellule. $\overline{\sigma} \circ r^{-1}$ permet alors de récupérer la valeur de cette cellule dans l'état de synthèse $\overline{\sigma}$. Grâce à r^{-1} on peut définir la sémantique de l'affectation transformant un état $\overline{\sigma}$ en $\overline{\sigma}'$

$$(destexp \leftarrow srcexp)(\overline{\sigma}) = \overline{\sigma}' : \begin{cases} s \mapsto \mathcal{E}[srcexp](\overline{\sigma} \circ r^{-1}) \text{ si } s = C[destexp](\overline{\sigma} \circ r^{-1}) \\ s \mapsto \overline{\sigma}(s) \text{ sinon} \end{cases}$$

Dans le cas général, la présence de fragments qui ne soient pas des singletons nous interdit cette solution. Sans la première condition, nous n'avons pas de singleton pour les cellules intervenant dans les expressions. La fonction r^{-1} n'existe plus, et on ne peut plus évaluer les expressions directement. Si au lieu d'être dans un singleton, les cellules des expressions sont dans des fragments plus grand, l'état de synthèse sur lequel on cherche à évaluer l'expression peut certes contenir la valeur de la cellule, mais il peut aussi contenir la valeur de toute autre cellule du fragment.

Dans cette thèse, la première condition sera toujours vérifiée. On n'appliquera jamais la sémantique d'une instruction avec une fragmentation qui ne réserve pas un fragment singleton pour chaque cellule manipulée par cette instruction, qu'elle soit lue ou écrite. On peut d'ailleurs systématiquement ajouter un fragment singleton avant une instruction, quitte à le faire disparaître juste après.

Par conséquent, nous pouvons définir une traduction des expressions où toute expression d'accès à une cellule est remplacée par la variable symbolique associée au singleton contenant cette cellule. Nous noterons $\overline{exp}$ la traduction de l'expression *exp* pour la fragmentation considérée. Ceci nous permet de simplifier l'évaluation des expressions :

$$\mathcal{E}[exp](\overline{\sigma} \circ r^{-1}) \;=\; \mathcal{E}[\overline{exp}](\overline{\sigma})$$

Pour nous, la traduction de la partie gauche d'une affectation sera toujours une variable.

Intéressons nous à la seconde condition. Qu'advient t-il quand une cellule utilisée dans une instruction n'est pas seule dans un fragment ? Relâcher cette contrainte n'a pas d'influence sur l'évaluation des expressions et a fortiori sur l'évaluation des gardes. En fait, elle n'est utile que

lorsqu'il s'agit de cellules dont le contenu est modifié, c'est à dire de cellules à gauche d'une affectation. Comme nous l'avons vu dans l'exemple, lorsqu'une cellule cible d'une affectation n'est pas seule dans un fragment, on est forcé de raisonner par cas.

Pour chaque état de synthèse présent avant l'affectation et pour chaque variable représentant un fragment pouvant contenir la cellule affectée, on peut potentiellement avoir deux états de synthèse après : un état dans lequel la variable a changé de valeur, et un état dans lequel elle n'a pas été modifiée. Cette manière de présenter l'effet de l'affectation est commode, parce qu'on peut décrire l'ensemble des états résultats comme l'union de deux ensembles : l'ensemble dans lequel rien n'a changé, et l'ensemble dans lequel l'affectation a eu lieu. Il est donc aisé de décrire le résultat de l'affectation à l'aide de l'affectation classique. Pour revenir à l'exemple, l'effet de l'affectation pouvait être ainsi décrit :

$$(c_3 \leftarrow 5)(\{\overline{\sigma}\}) = \overline{\sigma}[s_2 \mapsto 5] \cup \overline{\sigma}[s_2 \mapsto 5, s_3 \mapsto 5]$$

Mieux encore, nous pouvons tenter de décrire l'effet d'une affectation, comme un ensemble d'affectations indépendantes. Ces affectations se divisent en deux groupes. D'une part les affectations aux variables de synthèse représentant des singletons, qu'on appellera *affectations fortes*. D'autre part les affectations qui touchent les fragments contenant plus d'une cellule et qu'on appellera par opposition *affectation faibles*. [BCC+02]

Définition 12 (Affectation forte, Affectation faible)**.**
Soient s une variable de synthèse et $\overline{exp}$ une expression des variables de synthèse. L'**affectation forte** $s \leftarrow \overline{exp}$ de $\overline{exp}$ vers s, est une fonction qui à tout ensemble d'états de synthèse $\overline{X}$ associe le même ensemble d'états dans lesquels on a substitué l'évaluation de $\overline{exp}$ à la valeur associée à s. Autrement dit :

$$(s \leftarrow \overline{exp})(\overline{S}) = \left\{\overline{\sigma}[s \mapsto \mathcal{E}[\overline{exp}](\overline{\sigma})] \mid \overline{\sigma} \in \overline{S}\right\}$$

L'**affectation faible** $s \leftsquigarrow \overline{exp}$ de $\overline{exp}$ vers s est une fonction qui à tout ensemble d'états $\overline{\sigma}$ associe le même ensemble d'états auquel on a ajouté pour chaque état le même état dans lequel on a substitué l'évaluation de $\overline{exp}$ à la valeur de s. Autrement dit :

$$(s \leftsquigarrow \overline{exp})(\overline{S}) = \left\{\overline{\sigma}, \overline{\sigma}[s \mapsto \mathcal{E}[\overline{exp}](\overline{\sigma})] \mid \overline{\sigma} \in \overline{S}\}\right\}$$

ou encore

$$(s \leftsquigarrow \overline{exp})(\overline{S}) = \overline{S} \cup (s \leftarrow \overline{exp})(\overline{S})$$

Avec ces nouveaux opérateurs sémantiques sur les états de synthèse, il est possible d'approximer de manière correcte la sémantique concrète. Chaque fois qu'une cellule apparait dans une expression, on aura un fragment singleton pour cette cellule. On pourra traduire toutes les lectures de cellules en les remplaçant par leurs variables de synthèse associées. Puis, pour les affectations, on générera une séquence d'affectations sur les états de synthèse en énumérant les fragments contenant la cellule destination. Si ce sont des fragments singletons alors on aura des affectations fortes, sinon, des affectations faibles. Pour s'assurer de l'indépendance des affectations, on pourra introduire une variable temporaire pour stocker la valeur à affecter. Notons que cette traduction de la sémantique dépend de la fragmentation : si la fragmentation est calculée dynamiquement, alors la traduction devra l'être tout autant.

Il reste un dernier obstacle à franchir. Dans une fragmentation symbolique, l'appartenance ou non à un fragment de la cellule cible d'une affectation dépend de l'état considéré. Par conséquent, pour chaque état, la séquence d'affectations fortes et faibles à exécuter peut potentiellement être différente. On ne peut pas tout simplement avoir la même séquence d'affectations pour tous les états. En revanche, on peut partitionner l'ensemble des états selon les séquences d'affectations qu'ils entraînent. Cette partition est induite par la fragmentation et il est possible de maintenir cette partition de sorte qu'à tout moment de l'analyse, les affectations puissent être traitées comme des affectations fortes. Cette solution est celle qui est proposée dans [HP08].

Une autre solution est d'utiliser une approximation supplémentaire. Si dans certains états un fragment est affecté et dans d'autres il ne l'est pas, on peut ici aussi approximer l'effet sur ce fragment pour l'ensemble complet des états par une affectation faible. Nous aurions donc deux sens distincts de l'affectation faible :

- une affectation à l'une des cellules d'un fragment qui peut en contenir plusieurs ou
- une affectation qui peut selon les états affecter un fragment ou non.

Pour ces deux sens il s'agit d'une affectation qui peut selon les cas affecter ou non un fragment. Aussi, la même sémantique s'applique. Cette différence se fera néanmoins sentir lorsque nous quitterons le champ de la synthèse de propriétés pour s'intéresser aux propriétés d'agrégation. (Chapitre 8)

Par exemple, supposons qu'on ait deux variables d'indices i et j pouvant être égales ou différentes mais toutes les deux comprises entre 1 et n, c'est à dire $1 \leq i, j \leq n$. Une affectation à $A[i]$ modifierait faiblement le fragment des cellules d'indice compris entre 1 et n tout comme le fragment singleton $\{A[j]\}$. Pour le premier, i est bien compris entre 1 et n on est sûr que le fragment contient la cellule affectée. L'affectation est faible parce qu'il y a plusieurs cellules dans le fragment. Les propriétés qu'on peut exprimer sur ce fragment sont donc la disjonction des propriétés de la cellule affectée et celles des cellules non affectées. Pour le second en revanche, la raison est différente. Il s'agit d'un singleton, il n'est pas nécessaire de distinguer dans le fragment deux ensembles de cellules. Seulement, suivant les états, la cellule pourra être affectée ou non et on retrouve la même disjonction.

Ceci conclut notre recherche d'une approximation de la sémantique concrète. Nous traduisons celle-ci sur les états de synthèse en ajoutant un nouveau type d'instruction, l'affectation faible. Les domaines abstraits, implémentant déjà une abstraction de l'affectation forte, doivent donc interpréter cette nouvelle affectation. Bien sûr, l'affectation faible peut se calculer à partir de la forte, quitte à utiliser l'opérateur d'union $\sqcup$. On voudra peut être toutefois implémenter directement l'affectation faible pour des questions d'efficacité : le traitement des affectations est généralement moins coûteux que le calcul d'une union.

3.7 Terminaison et correction

Dans cette section nous abordons brièvement les conséquences sur la terminaison et la correction. Du point de vue des valeurs abstraites, la principale différence est la variation du nombre de variables d'une itération sur l'autre. La fragmentation peut se complexifier et définir de nouveaux fragments. Il faut prendre en compte ce fait lorsqu'on définit les opérateurs d'élargissement dans les domaines abstraits.

Dans le cas classique, lorsqu'un fragment apparaît dans la valeur abstraite élargissant et n'est pas présent dans la valeur abstraite à élargir, c'est que ce fragment était vide dans la seconde.

Nous avons déjà donné l'exemple d'un fragment $\{ A[\ell] \mid 1 \leq \ell < \sigma(i) \}$ qui apparaitrait lors de la première itération d'une boucle mais serait vide avant la première itération de cette boucle lorsque $i = 1$. Il faut donc accorder une attention particulière à l'élargissement dans ce cas. Le choix classique lorsque l'on parle d'interprétation abstraite de programme manipulant des structures de données, c'est de ne pas élargir les propriétés acquises sur ce nouveau fragment. C'est heureusement ce qui se produira avec la plupart des opérateurs d'élargissement des domaines abstraits classiques.

Par exemple, si on utilise le domaine abstrait des intervalles, on attribuera l'intervalle vide $\bot$ au fragment vide dans la valeur abstraite à élargir. Dans l'élargissement on donnera à ce fragment l'intervalle calculé dans la valeur abstraite élargissant. Autrement dit, on impose la règle

$$\bot \triangledown \widetilde{X} = \widetilde{X}$$

Par conséquent, tant que le nombre de variables augmente, l'élargissement ne nous garantit pas la terminaison. Il faudra donc impérativement s'assurer que le nombre de variables de synthèse dans la fragmentation reste borné. Ainsi, il y aura un moment dans l'analyse où le nombre de variables atteint son maximum, et à partir de là, les propriétés de l'élargissement nous assureront que la séquence de valeurs abstraites ne pourra croître indéfiniment.

En interprétation abstraite, la correction de l'analyse repose sur le fait que, au moins à la dernière itération de l'interprète, la sémantique abstraite et les unions aient été correctement surapproximés. La difficulté qu'introduit la fragmentation des tableaux est que les ensembles de variables de synthèse sont amenés à changer aux différents points de contrôle. Il va donc être nécessaire de prouver que chaque fois qu'on transforme une fragmentation, la valeur abstraite obtenue après la transformation est au pire plus faible que celle obtenue avant la transformation. Pour nous, ceci ajoute des preuves supplémentaires : il faut que les transformations de la fragmentations soient correctement répercutées sur les valeurs abstraites.

4 Critère de fragmentation et abstraction des fragmentations

Chercher une fragmentation adéquate, c'est chercher quelles sont les cellules qui partagent des propriétés similaires. Si on trouve deux cellules qui partagent les mêmes propriétés, alors on peut les regrouper et les dénoter toutes les deux par une même variable. Il suffira ensuite de parler de la variable pour parler de n'importe laquelle de ces deux cellules. Ces groupes de cellules constituent des fragments des structures de données. En général, une structure de donnée n'est pas de taille bornée. En regroupant les cellules d'une structure en un nombre borné de fragments, on est capable de décrire un nombre non borné de cellules grâce à un nombre borné de variables.

Le choix d'une fragmentation adéquate est un problème difficile. Il existe plusieurs méthodes et la pertinence de celle-ci dépend de l'objectif de l'analyse ainsi que des domaines abstraits utilisés. On commencera en général par le choix du domaine abstrait. Celui-ci doit permettre l'expression aussi bien des propriétés ciblées que l'expression des propriétés intermédiaires, tels les invariants de boucle. La preuve de propriétés sur les contenus de tableaux nécessitera également l'usage d'un domaine abstrait numérique capable de dériver des propriétés sur les variables d'indice. De la même manière, la preuve de propriétés sur les structures chaînées nécessitera un domaine abstrait capable de dériver des propriétés d'accessibilité. On voudra donc avoir un produit de domaines abstraits permettant chacun de traiter l'une de ces classes de propriétés. Une fois qu'on a choisi le ou les domaines abstraits, on tente de définir au mieux une fragmentation adaptée à ce ou ces domaines.

La pertinence d'un découpage est difficile à quantifier. On cherchera un compromis entre le nombre de fragments et la précision de l'analyse. Il est toujours possible de raffiner une fragmentation et d'obtenir des propriétés plus fortes. A l'inverse, on peut simplifier un découpage quitte à affaiblir les propriétés des fragments. Mais si un raffinement n'apporte pas d'information intéressante alors on considérera que ce raffinement n'était pas pertinent dans ce cas précis.

Il faut tout d'abord remarquer que la fragmentation parfaite n'existe pas en général. Une fragmentation parfaite serait une fragmentation qui permette de décrire parfaitement une structure de données, sans perte d'information et pour un domaine abstrait donné.

Considérons l'exemple de la figure 4.1. Ce programme initialise un tableau avec des entiers consécutifs. Supposons que notre objectif soit de prouver l'absence de débordements arithmétiques. Un domaine abstrait qui semble adapté à cette vérification est celui des intervalles. Avec ce domaine, la classe de propriétés ciblée est donc celle des formules encadrant les variables par des constantes. Il nous faut aussi choisir une fragmentation pertinente du tableau pour ce domaine.

Or pour chaque cellule du tableau, on peut avoir une propriété d'intervalle différente. Il faudrait alors idéalement avoir une variable distincte pour chaque cellule, une fragmentation désignant dans un fragment différent chaque cellule. Ce n'est qu'ainsi qu'on pourrait affecter un intervalle différent à chacune d'elles. Or, le tableau n'a pas de taille fixée, le nombre de cellules

Pour *i de* 1 *à n* **faire**
  A[i] = i ;

FIGURE 4.1 : Construction d'un tableau d'entiers consécutifs.

peut être arbitrairement grand. Or une fragmentation décrit un nombre fixé de fragments, et il n'y a pas de « meilleure » fragmentation pour cet exemple.

Pour résoudre ce problème, nous avons deux solutions. Ou bien nous renonçons au domaine des intervalles et utilisons un domaine plus complexe permettant d'exprimer des relations entre les cellules et leur indice. Ou bien nous usons d'approximations pour analyser le programme. Un intervalle de validité pour n nous donnerait le même intervalle de validité pour i et par conséquent il nous est possible d'affirmer que les cellules du tableau affectées ont une valeur comprise entre 1 et n.

Il nous suffirait donc de définir un seul fragment simple de tableau : l'ensemble des cellules d'indice compris entre 1 et i. De ce fragment on peut encadrer les valeurs. Cette information serait suffisante pour conclure en l'absence de débordement arithmétique. Mais il nous a fallu choisir de l'information à sacrifier, ici, l'égalité entre l'indice des cellules et leur valeur.

Dans ce chapitre nous introduisons une nouvelle méthode de fragmentation. Celle-ci tente de combiner les méthodes de partitionnement dynamique existantes [CCL11, GMT08] avec des méthodes plus fines mais statiques mais basées sur la syntaxe [HP08]. Ce faisant, nous donnerons une manière de construire dynamiquement la fragmentation par instrumentation des programmes. Ceci nous permettra ensuite dans une deuxième partie d'introduire les domaines de fragmentations, analogues des domaines abstraits pour les fragmentations symboliques. Comme les domaines abstraits doivent se munir d'une sémantique abstraite pour approximer la sémantique concrète, les domaines de fragmentation devront approximer l'instrumentation des programmes pour calculer la fragmentation tout au long de l'analyse. On terminera ce chapitre par une courte discussion sur le choix de sous-approximation et de sur-approximation pour l'abstraction des fragmentations.

4.1 Un nouveau critère de fragmentation sémantique

L'objectif ici est de décrire une nouvelle méthode de fragmentation qui puisse être appliquée sur tout graphe de flot de contrôle et qui soit capable de désigner aussi bien des fragments que des relations entre des cellules. Nous commencerons d'abord par décrire une version simplifiée de la méthode avant de la développer afin de raffiner les fragmentations dans le cas de boucles imbriquées.

4.1.1 Fragmentation simplifiée

Dans cette section nous décrirons un critère de fragmentation. Ce critère se composera de règles. De ces règles nous déduirons logiquement la fragmentation fixée par le critère. Nous ne nous intéresserons que plus tard à l'abstraction et au calcul de cette fragmentation tout au long de l'analyse.

Pour i *de* 1 *à* j **faire**
  $A[i] \leftarrow 1$

Pour i *de* $j+1$ *à* n **faire**
  $A[i] \leftarrow 0$

Pour i *de* 1 *à* m **faire**
  $A[i] \leftarrow 2$

(a) Trois boucles avec chevauchement

Pour i *de* 1 *à* $\min\{j,m\}$ **faire**
  $A[i] \leftarrow 2$

Pour i *de* $\min\{j,m\}+1$ *à* j **faire**
  $A[i] \leftarrow 1$

Pour i *de* $j+1$ *à* $\max\{j,m\}$ **faire**
  $A[i] \leftarrow 2$

Pour i *de* $\max\{j,m\}+1$ *à* n **faire**
  $A[i] \leftarrow 0$

(b) Quatre boucles sans chevauchement

Pour i *de* 1 *à* n **faire**
  Si $i \leq j$ **alors**
    $A[i] \leftarrow 1$
  sinon
    $A[i] \leftarrow 0$

Pour i *de* 1 *à* m **faire**
  $A[i] \leftarrow 2$

(c) Deux boucles avec chevauchement

Figure 4.2 : Ces trois programmes modifient le tableau A de la même manière. Le critère de fragmentation donnera les mêmes fragmentations pour les programmes (a) et (c), mais donnera une fragmentation plus fine pour le programme (b).

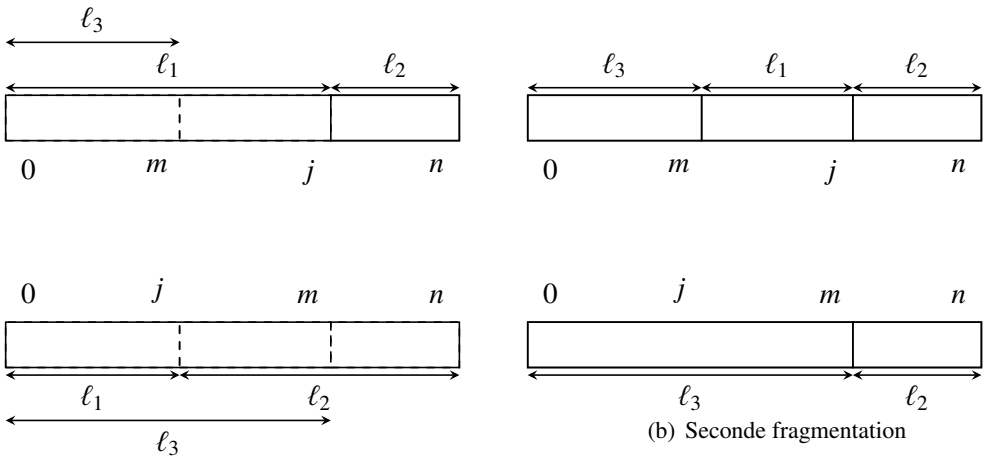
La méthode de fragmentation que nous allons décrire ici suit les principes d'une fragmentation sémantique. Chaque affectation et chaque test sur une cellule va apporter de l'information sur la cellule en question. Si c'est une affectation, on sait que la cellule hérite des propriétés de la source de l'affectation. Si c'est un test, suivant la branche du programme que l'on suit on saura que la cellule vérifie le test ou ne le vérifie pas. On peut, au moins temporairement, introduire un fragment qui contient la cellule et uniquement cette cellule. Ainsi, pour le domaine abstrait, la cellule sera semblable à une variable locale et toute propriété de la cellule pourra être déduite normalement.

L'essence du problème c'est de savoir quelles sont les raisons qui font que l'on peut regrouper des cellules dans un fragment. L'objectif est de désigner un nombre fini de fragments tout en obtenant une précision maximale. L'idée principale du critère repose sur un pari. On parie sur le fait que la même instruction exécutée plusieurs fois (boucle, récursivité) produira les mêmes effets sur les cellules. Ainsi, on regroupera dans le même fragment, les cellules accédées par la même instruction.

Considérons l'exemple de la figure 4.2(a). Trois boucles parcourent des sections du tableau A et trois instructions le modifient. Puisque nous avons trois instructions, le critère nous impose trois fragments. Pour chacun de ces trois fragments, introduisons une variable de synthèse respectivement a_1, a_2 et a_3 pour la première, seconde et troisième boucle. À ces variables on associe respectivement les trois fragments $A[1..j]$, $A[j..n]$ et $A[1..m]$ parcourus par les boucles. On aurait donc en fin de programme la fragmentation illustrée figure 4.3(a) :

$$F(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[\ell_2] \\ a_3 \mapsto A[\ell_3] \end{cases} \;\middle|\; \begin{array}{rcll} 1 \leq & \ell_1 & < \sigma(j) & \wedge \\ \sigma(j) < & \ell_2 & \leq \sigma(n) & \wedge \\ 1 \leq & \ell_3 & \leq \sigma(m) & \end{array} \right\}$$

Cette fragmentation peut être améliorée. Car si on peut dire des cellules représentées par a_3 qu'elles ont pour contenu la valeur 2, on a une information moins précise pour a_1 et a_2. La



(a) Première fragmentation

(b) Seconde fragmentation

FIGURE 4.3 : À gauche, la première fragmentation proposée, à droite la seconde. En haut sont représentés les fragmentations pour le cas $\sigma(m) \leq \sigma(j)$, en bas les fragmentations pour le cas $\sigma(m) \geq \sigma(j)$. Les lignes discontinues indiquent des frontières entre des fragments qui ne sont pas des frontières pour tous les fragments.

troisième boucle écrase le contenu de cellules déjà affectées représentées par a_1 et a_2. On sait donc de celles-ci que leurs valeurs sont dans les ensembles $\{1, 2\}$ et $\{0, 1\}$ respectivement.

Or si on pousse la logique du critère un peu plus loin, il est incohérent de conserver dans les deux premiers fragments les cellules qui ont été modifiées par la troisième boucle. Les propriétés acquises par les première et seconde affectations va être perdue si les cellules sont écrasées par la troisième affectation. Il serait donc logique de retirer des fragments déjà construits toute cellule qui serait modifiée par une affectation ultérieure. Si on retire les cellules représentées par a_3 aux deux premiers fragments, on obtient la nouvelle fragmentation illustrée figure 4.3(b) :

$$F(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[\ell_2] \\ a_3 \mapsto A[\ell_3] \end{cases} \;\middle|\; \begin{array}{rlll} 1 \leq & \ell_1 & \leq \sigma(j) \wedge \sigma(m) < \ell_1 & \wedge \\ \sigma(j) < & \ell_2 & \leq \sigma(n) \wedge \sigma(m) < \ell_2 & \wedge \\ 1 \leq & \ell_3 & \leq \sigma(m) & \end{array} \right\}$$

On peut alors utiliser cette fragmentation pour abstraire l'ensemble des états accessibles à la fin du programme avec un domaine abstrait permettant d'exprimer l'égalité entre variable et constante. On obtiendrait alors la valeur abstraite :

$$a_1 = 1 \wedge a_2 = 2 \wedge a_3 = 3$$

On peut appliquer la même logique aux programmes des figures 4.2(b) et 4.2(c). La fragmentation finale pour le premier différera un peu de celle qu'on a précédemment obtenu. Quatre affectations définiront quatre fragments constituant la fragmentation

$$F(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[\ell_2] \\ a_3 \mapsto A[\ell_3] \\ a_4 \mapsto A[\ell_4] \end{cases} \;\middle|\; \begin{array}{rlll} 1 \leq & \ell_1 & \leq \sigma(m) \wedge \ell_1 \leq \sigma(j) & \wedge \\ \sigma(m) < & \ell_2 & \leq \sigma(j) & \wedge \\ \sigma(j) < & \ell_3 & \leq \sigma(m) & \wedge \\ \sigma(m) < & \ell_4 & \leq \sigma(n) \wedge \sigma(j) < \ell_4 & \end{array} \right\}$$

Pour le dernier exemple en revanche, la fragmentation sera identique à la première. La première boucle distingue selon les cas deux affectations, et chacune d'elle redéfinira alors les deux fragments initiaux.

Ces deux exemples mettent en lumière l'impact que peut avoir une transformation de programmes. Si une transformation d'un programme en programme équivalent modifie la fragmentation de sorte qu'il ne soit plus possible de prouver les mêmes propriétés, alors c'est probablement que le critère de fragmentation utilisé n'est pas assez bon. Il est légitime de penser qu'un tel critère ne sera pas adapté pour traiter une plus large gamme de programmes.

Nous avons vu ce que la fragmentation devait être à la fin du programme. Intéressons nous maintenant à ce qu'elle est à l'intérieur des boucles. Même si les cellules sont regroupées en un fragment à un moment, il est important qu'au moins pendant un temps les cellules accédées par les différentes instructions soient désignées par un fragment propre. Il faut que pour chaque accès à un tableau, une variable puisse être associée à la cellule accédée et uniquement à la cellule accédée. C'est important car ainsi, les instructions peuvent être interprétées comme des instructions sur des variables classiques. D'une part, cela joue sur la précision de l'analyse puisqu'ainsi il n'est pas nécessaire de recourir à une approximation plus grossière de la sémantique concrète. D'autre part, cela implique que l'effort à fournir pour adapter le domaine abstrait sera minime.

Il faut donc aux moments opportuns définir des fragments pour les cellules accédées. Dans les exemples précédents, il suffira d'introduire un nouveau fragment le temps d'interpréter chaque affectation. C'est à dire qu'au point de contrôle situé avant l'affectation on introduira un nouveau fragment et sa variable associée. On traitera l'affectation comme une affectation à cette variable.

Une fois l'affectation passée, on se souvient qu'il faut regrouper ensemble les cellules traitées par la même affectation. On prend donc la cellule du nouveau fragment singleton, et on l'ajoute au fragment regroupant toutes les cellules déjà modifiées par cette affectation. On peut alors oublier le fragment singleton jusqu'à la prochaine affectation.

En suivant ces règles, on aurait respectivement avant et après la première affectation de la figure 4.2(a) les fragmentations F_1 et F_2 :

$$F_1(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a'_1 \mapsto A[\ell_2] \end{cases} \;\middle|\; 1 \leq \ell_1 \leq \sigma(i) - 1 \wedge \ell_2 = i \right\}$$

$$F_2(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a'_1 \text{ indéfini} \end{cases} \;\middle|\; 1 \leq \ell_1 \leq \sigma(i) \right\}$$

Qu'en est t-il des relations entre cellules ? L'idée développée jusque là nous donne un moyen de désigner des fragments. Mais elle peut être réutilisée pour préciser les relations qui nous intéressent entre ces fragments. Notre idée était qu'une même affectation manipule de la même manière les différentes cellules accédées par cette instruction. On peut aussi juger qu'une même instruction ou un même test concernant plusieurs cellules induit une relation entre ces cellules et les lie par une même propriété.

Observons l'exemple de la figure 4.4. Si on utilise le critère précédent sans modification, chacune des deux affectations décrit deux fragments. On devrait donc définir quatre variables de synthèse, chacune décrivant l'un des fragments $A[1..j]$, $B[1..j]$, $A]j..n]$ et $C[1..n - j]$. Contrairement à l'exemple précédent, on n'a pas de propriété liant toutes les cellules d'un fragment à toutes les cellules d'un autre. En revanche, on a des propriétés liant des couples particuliers de cellules de ces fragments : les couples de cellules de même indice.

Pour *i de* 1 *à j* **faire**
  $B[i] \leftarrow A[i]$

Pour *i de j à n* **faire**
  $C[i-j] \leftarrow A[i]$

FIGURE 4.4 : Ce programme copie deux fragments du tableau A dans les tableaux B et C. Il construit ainsi deux relations, l'une entre le premier fragment du tableau A et le tableau B, l'autre entre le seconde fragment du tableau A et le tableau C.

Les affectations du programme construisent les propriétés $A[\ell] = B[\ell]$ pour la première et $A[\ell] = C[\ell - j]$ pour la seconde. On cherchera donc à capturer les couples de cellules $(A[\ell], B[\ell])$ et $(A[\ell], C[\ell - j])$. A la fin du programme il semble raisonnable de souhaiter avoir une fragmentation similaire à celle-ci :

$$F(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[\ell_2] \\ b \mapsto B[\ell_1] \\ c \mapsto C[\ell_2 - \sigma(j)] \end{cases} \;\middle|\; \begin{array}{rcll} 1 \leq & \ell_1 & \leq \sigma(j) & \wedge \\ \sigma(j) \leq & \ell_2 & \leq \sigma(n) & \end{array} \right\}$$

De la même manière il faudra pouvoir donner une fragmentation en tout point de programme. Supposons qu'avant la première affectation on ait la fragmentation

$$F_1(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ b \mapsto B[\ell_1] \end{cases} \;\middle|\; 1 \leq \ell_1 \leq \sigma(i) - 1 \right\}$$

Comme précédemment, pour traiter l'affectation, on doit ajouter deux variables de synthèse a'_1 et b' pour les singletons $A[i]$ et $B[i]$ respectivement :

$$F_2(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ b \mapsto B[\ell_1] \\ a'_1 \mapsto A[\sigma(i)] \\ b' \mapsto B[\sigma(i)] \end{cases} \;\middle|\; 1 \leq \ell_1 \leq \sigma(i) - 1 \right\}$$

Une fois l'affectation traitée, on pourra oublier les variables a'_1 et b' mais pas avant d'avoir ajouté ces cellules à leurs fragments respectifs. C'est ici qu'on introduit une nouveauté. On n'ajoute pas les cellules à leurs fragments indépendamment. On ajoute le couple de cellules représentées par (a'_i, b') aux couples de cellules (a_i, b) précédemment considérés. Formellement, on ajoute une nouvelle relation à la fragmentation, la relation r' dans laquelle le représentant de a_1 est le même que celui de a'_1 et le représentant de a_2 est le même que celui de a'_2.

$$r' : \begin{cases} a_1 \mapsto A[\sigma(i)] \\ b \mapsto B[\sigma(i)] \\ a'_1 \mapsto A[\sigma(i)] \\ b' \mapsto B[\sigma(i)] \end{cases}$$

```
i ← 0, j ← −1
Répéter
    Si A[i] < 0 alors
        j ← i
    i ← i + 1
jusqu'à j ≠ −1
```

FIGURE 4.5 : Version d'un programme cherchant le premier élément strictement négatif d'un tableau.

De manière générale, après une affectation, on peut ajouter à la fragmentation une nouvelle relation dans laquelle les variables associées aux accès ont le même représentant que le singleton introduit. Ceci aura pour effet de collectionner les relations de cellules traitées par chaque affectation.

Enfin, on peut finalement oublier les variables introduites temporairement pour représenter les singletons accédés.

Dans le cas de tests, la propriété obtenue sur les cellules testées est différente selon que le test est satisfait ou non. Il sera donc pertinent de répartir les cellules dans deux fragments : le fragment des cellules qui vérifient le test, et le fragment des cellules qui ne le vérifient pas.

Considérons l'exemple de la figure 4.5. Il s'agit d'un programme qui parcourt un tableau jusqu'à ce qu'il rencontre une cellule de valeur strictement négative. On voudra naturellement dans ce programme séparer les cellules $A[i]$ qui vérifient le test et celles qui ne le vérifient pas. Toutes les premières sont positives ou nulles, et l'ensemble des secondes est réduit à une seule cellule négative.

4.1.2 Fragmentation par instrumentation des programmes

La section précédente décrit plusieurs idées à appliquer pour choisir une fragmentation. Il nous faut aller plus loin formellement pour définir sans ambiguïté le calcul de cette fragmentation. Il va falloir être capable de calculer en chaque point du programme la fragmentation à associer à chacun des états concrets accessibles.

Le critère que nous avons introduit dans la section précédente décrit l'évolution de la fragmentation à travers les instructions du programme. Chaque instruction modifie la fragmentation. De là on peut interpréter le programme comme étant un programme manipulant une fragmentation. On oublie la sémantique des diverses instructions et on leur donne une sémantique propre au calcul des fragmentations.

Dans cette section nous partons de cette idée afin de rendre aussi explicite que possible l'évolution des fragmentations durant l'exécution d'un programme. Pour cela, nous allons instrumenter le programme de sorte à y ajouter des instructions manipulant la fragmentation. On pourra alors voir le calcul de la fragmentation comme l'exécution du programme.

Cela nous donne une manière commode de décrire les fragmentations que l'on souhaite utiliser. On pourra en effet décrire précisément une fragmentation « idéale » au sens du critère. Une exécution du programme et de ses instructions d'instrumentation permet de calculer cette

fragmentation. Il sera ensuite nécessaire d'abstraire cette fragmentation tout comme on abstrait les ensembles d'états accessibles, en chaque point de contrôle du programme. Les raisons sont les mêmes : l'ensemble des fragmentations est indénombrable. Le parallèle continue un peu plus loin. De la même manière qu'on utilise une approximation de la sémantique du programme pour abstraire son comportement, on utilisera une approximation de la sémantique des instructions d'instrumentation que l'on va ajouter.

Pour le moment, il faut définir les instructions d'instrumentation que nous devons rajouter. En chaque point de contrôle, l'état du programme peut être décrit comme un couple (σ, F) d'un état concret au sens classique, et d'une fragmentation. Initialement, on considère que l'on ne sait rien du programme. On doit donc partir d'une fragmentation ne contenant qu'une seule relation, la relation définie nulle part : on ne définit aucune variable de synthèse, et aucun fragment.

Les instructions d'instrumentation dont nous aurons besoin sont au nombre de quatre. La singularisation faible, la singularisation forte, la composition et la réinitialisation.

La singularisation faible Chaque fois qu'une cellule est lue, le critère impose qu'une variable de synthèse lui soit exclusivement associée. Cette variable décrit un fragment singleton contenant la cellule et seulement la cellule. Pour signifier que la variable de synthèse s représente une cellule désignée par l'expression de conteneur exp on utilisera l'instruction d'instrumentation qu'on ajoutera avant l'accès à cette cellule :

$$s : \{exp\}$$

On peut décrire sa sémantique $\mathcal{I}[s : \{exp\}](.)$ de cette instruction en se basant sur l'existence de l'opérateur $C[.](.)$. (cf. Section 3.2)

$$\mathcal{I}[s : \{exp\}] : \sigma, F \mapsto \sigma, \left\{ r : \begin{cases} s' \mapsto C[exp](\sigma) \text{ si } s' = s \\ s' \mapsto r'(s') \text{ sinon} \end{cases} \;\middle|\; r' \in F \right\}$$

La singularisation forte Chaque fois qu'une cellule est écrite, non seulement il faut qu'une variable lui soit exclusivement associée, mais il faut aussi réciproquement que cette cellule soit retirée de tout autre fragment. Ainsi l'affectation n'aura d'effet que sur la variable associée et sur aucune autre. On ajoutera avant toute écriture à une cellule désignée par exp l'instruction d'instrumentation :

$$s :: \{exp\}$$

La sémantique est similaire à celle de la singularisation faible à ceci près qu'on retire dans la fragmentation toute autre association à la cellule affectée :

$$\mathcal{I}[s :: \{exp\}] : \sigma, F \mapsto \sigma, \left\{ r : \begin{cases} s' \mapsto C[exp](\sigma) \text{ si } s' = s \\ s' \mapsto r'(s') \text{ si } r'(s') \neq C[exp](\sigma) \\ \text{indéfini en } s' \text{ sinon} \end{cases} \;\middle|\; r' \in F \right\}$$

La composition Une instruction ou un test impliquant des cellules peut éventuellement créer une propriété liant ces cellules. On doit alors ajouter la relation entre ces cellules à la fragmentation. Les fragments décrits par les variables de synthèse sont augmentés de nouveaux éléments. On dira que ces nouveaux fragments sont la composition des anciens et des nouveaux fragments singletons ajoutés à la relation. Si on souhaite ajouter aux n-uplets de cellules représentées par les variables $s'_1, s'_2, \ldots, s'_n$ à la relation entre les cellules représentées par les variables $s_1, s_2, \ldots, s_n$ on utilisera l'instruction d'instrumentation :

$$s_1, s_2, \ldots, s_n \cup\!: s'_1, s'_2, \ldots, s'_n$$

Cette instruction va ajouter à la fragmentation l'ensemble des n-uplets qui lui manquent : les n-uplets dans lesquels les variables s_1, …, s_n sont respectivement égales aux variables $s'_1, \ldots s'_n$ que l'on rajoute aux fragments.

$$\mathcal{I}[s_1, \ldots, s_n \cup\!: s'_1, \ldots, s'_n] : \sigma, F \mapsto \sigma, F \cup \left\{ r : \begin{cases} s_1 \mapsto r'(s'_1) \\ \vdots \\ s_n \mapsto r'(s'_n) \\ s \mapsto r'(s) \text{ pour } s \neq s_1, \ldots, s_n \end{cases} \;\middle|\; r' \in F \right\}$$

La réinitialisation Une fois qu'on a fini d'utiliser une variable de synthèse, il faut la retirer de la fragmentation. On utilisera alors l'instruction

$$s : \emptyset$$

dont l'effet sera de retirer toute association entre la variable s et les cellules qu'elle représentait auparavant :

$$\mathcal{I}[s : \emptyset] : \sigma, f \mapsto \sigma, \left\{ r : \begin{cases} s' \mapsto r'(s') \text{ si } s' \neq s \\ \text{indéfini en } s \end{cases} \;\middle|\; r' \in F \right\}$$

Exemple 11.
Reprenons l'exemple précédent reproduit figure 4.6(a). Chacune des deux affectations lit une cellule et écrit son contenu dans une autre. Il faudra donc insérer avant chacune d'elles une singularisation forte et une singularisation faible. Une fois l'affectation passée, il faudra composer le couple de cellules accédées avec les couples précédemment calculés. Enfin, après les compositions, on pourra réinitialiser les variables temporaires introduites pour traiter l'affectation. Cette instrumentation donnera le programme de la figure 4.6(b).

Observons le début de l'exécution du programme. On débute avec un état σ_0 dont on supposera $2 < \sigma_0(j)$ et une fragmentation F_0 contenant un unique choix de représentants r_0 défini nulle part. En entrant dans la boucle, on transforme σ_0 en σ_1 dont la valeur de i a été changée pour 1 :

$$\sigma_1(c) = \begin{cases} 1 & \text{si } c = i \\ \sigma_0(c) & \text{sinon} \end{cases}$$

Ensuite, on exécutera les deux instructions d'instrumentation en entrée de boucle. La singularisation faible nécessite l'évaluation de $A[i]$ dans σ_1, ce qui nous donne $A[1]$. Elle change les

Pour i *de* 1 *à* j **faire**
  $B[i] \leftarrow A[i]$

Pour i *de* $j+1$ *à* n **faire**
  $C[i-j] \leftarrow A[i]$

(a) Programme original

Pour i *de* 1 *à* j **faire**
  $a'_1 : \{A[i]\}, b' :: \{B[i]\}$
  $B[i] \leftarrow A[i]$
  $a_1, b \cup: a'_1, b'$
  $a'_1 : \emptyset, b' : \emptyset$

Pour i *de* j *à* n **faire**
  $a'_2 : \{A[i]\},$
  $c' :: \{C[i-j]\}$
  $C[i-j] \leftarrow A[i]$
  $a_2, c \cup: a'_2, c'$
  $a'_2 : \emptyset, c' : \emptyset$

(b) Programme instrumenté

FIGURE 4.6 : Cette figure illustre l'instrumentation (a) du programme (b) de la figure 4.4. Les instructions d'instrumentation ajoutées permettent de calculer la fragmentation désirée en chaque point de contrôle du programme.

cellules associées à a'_1 dans les choix de représentants. Il n'y a que r_0, indéfini en a'_1. La singularisation faible va donc transformer r_0 en r_1 :

$$r_1 : a'_1 \mapsto A[1]$$

La singularisation forte qui suit, va avoir un effet similaire. Elle évalue $B[i]$ en $B[1]$ dans σ_1, puis elle doit supprimer toute association à $B[i]$. r_1 n'a pas de telle association, on ne change donc rien. Puis on associe b' à $B[1]$ dans r_1. On obtient la fragmentation F_1 :

$$F_1 = \left\{ r_2 : \begin{cases} a'_1 \mapsto A[1] \\ b' \mapsto B[1] \end{cases} \right\}$$

On peut ensuite exécuter l'affectation qui modifiera l'état mais pas la fragmentation. La composition qui suit aura pour effet de créer un nouveau choix de représentants à partir de r_2 dans lequel les variables a_1 et b seront associés aux mêms cellules que a'_1 et b' respectivement :

$$F_2 = \left\{ r_2 : \begin{cases} a'_1 \mapsto A[1] \\ b' \mapsto B[1] \end{cases}, r_3 : \begin{cases} a_1 \mapsto A[1] \\ a'_1 \mapsto A[1] \\ b \mapsto B[1] \\ b' \mapsto B[1] \end{cases} \right\}$$

Puis les deux réinitialisations restreindront le domaine de définition de r_2 et r_3 à $\{a_1, b\}$:

$$F_3 = \left\{ r_0, r_4 : \begin{cases} a_1 \mapsto A[1] \\ b \mapsto B[1] \end{cases} \right\}$$

Une deuxième entrée dans la boucle nous donnera un état σ_2 vérifiant $\sigma_2(i) = 2$. Pour les singularisations, l'évaluation de $A[i]$ et $B[i]$ donnera donc $A[2]$ et $B[2]$. Comme lors de la première

itération, $B[2]$ n'apparaît dans aucun des choix de représentants et la singularisation forte n'aura pas plus d'effet que la faible. Celles-ci se contentent donc d'associer a'_1 et b' à $A[2]$ et $B[2]$ dans tous les choix de représentants.

$$F_4 = \left\{ r_5 : \begin{cases} a'_1 \mapsto A[2] \\ b' \mapsto B[2] \end{cases} \quad r_6 : \begin{cases} a'_1 \mapsto A[2] \\ a_1 \mapsto A[1] \\ b' \mapsto B[2] \\ b \mapsto B[1] \end{cases} \right\}$$

Après l'affectation, la composition prendra les deux choix de représentants, les dupliquera et les transformera pour avoir $a'_1 = a_1$ et $b' = b$. r_5 et r_6 donneront le même choix de représentants r_7 dans cette transformation :

$$F_5 = \left\{ r_5 : \begin{cases} a'_1 \mapsto A[2] \\ b' \mapsto B[2] \end{cases} \quad r_6 : \begin{cases} a'_1 \mapsto A[2] \\ a_1 \mapsto A[1] \\ b' \mapsto B[2] \\ b \mapsto B[1] \end{cases} , r_7 : \begin{cases} a'_1 \mapsto A[2] \\ a_1 \mapsto A[2] \\ b' \mapsto B[2] \\ b \mapsto B[2] \end{cases} \right\}$$

Enfin, la réinitialisation supprimera a'_1 et b' des domaines de définition :

$$F_6 = \left\{ r_0, r_7 : \begin{cases} a_1 \mapsto A[1] \\ b \mapsto B[1] \end{cases} , r_8 : \begin{cases} a_1 \mapsto A[2] \\ b \mapsto B[2] \end{cases} \right\}$$

On peut continuer l'exécution de la boucle pour obtenir finalement en sortie de boucle la fragmentation

$$F_7 = \left\{ r_0, r : \begin{cases} a_1 \mapsto A[\ell] \\ b \mapsto B[\ell] \end{cases} \;\middle|\; 1 \leq \ell \leq \sigma(j) \right\}$$

Si on continue cette exécution fastidieuse et qu'on traite la seconde boucle, on obtiendra en fin de programme la fragmentation

$$F_8 = \left\{ r_0, \; r : \begin{cases} a_1 \mapsto A[\ell_1] \\ b \mapsto B[\ell_1] \end{cases} , \; r' : \begin{cases} a_2 \mapsto A[\ell_2] \\ c \mapsto B[\ell_2 - \sigma(j)] \end{cases} , \; r'' : \begin{cases} a_1 \mapsto A[\ell_1] \\ b \mapsto B[\ell_1] \\ a_2 \mapsto A[\ell_2] \\ c \mapsto B[\ell_2 - \sigma(j)] \end{cases} \;\middle|\; \begin{array}{rcll} 1 \leq & \ell_1 & \leq \sigma(j) & \wedge \\ \sigma(j) < & \ell_2 & \leq \sigma(n) & \end{array} \right\}$$

Cette fragmentation est plus compliquée que celle qu'on espérait avoir : elle contient un certain nombre de choix de représentants en trop, dénotés par r_0, r et r'. Toutefois, on remarque qu'ils sont en fait des restrictions des choix r''. Par conséquent, si une propriété est vraie pour les seconds, alors selon notre interprétation elle sera vraie pour les premiers. Ils peuvent donc être ignorés sans risque. On pourrait imaginer une règle dans l'interprétation du programme nous forçant à retirer ces choix de représentants inutiles.

Instrumentation des tests L'instrumentation des tests ne pose pas de problème particulier. Il peut être néanmoins utile de préciser comment on opère. Supposons que nous voulions instrumenter le test suivant que l'on a déjà rencontré dans l'un des exemples précédents.

```
imax ← 1
Pour i de 2 à n faire
    Si A[i] > A[imax] alors
        imax ← i
```

FIGURE 4.7 : Recherche de l'indice de l'élément maximum d'un tableau d'entiers.

```
Si A[i] < 0 alors
    j ← i
```

On va séparer les cellules qui satisfont le test et celles qui ne le satisfont pas. Pour des raisons pratiques, il faut singulariser les cellules avant le test afin que le domaine abstrait puisse le traiter. Mais ce n'est qu'une fois le test passé que l'on sait dans quel fragment on doit ajouter les cellules. On peut commencer par mettre la cellule dans un fragment a_{tmp}. Puis suivant l'issue du test le placer dans le bon fragment a_1 ou a_2. Il faudra développer la branche « sinon » du test qui n'est pas présente dans le programme original.

```
a_tmp : {A[i]}
Si A[i] < 0 alors
    a_1 ∪: a_tmp
    a_tmp : ∅
    j ← i
sinon
    a_2 ∪: a_tmp
    a_tmp : ∅
```

4.1.3 Limite du critère

Le critère précédemment introduit permet de définir une fragmentation qui serait d'une certaine manière « idéale » compte tenu des propriétés que l'on souhaite découvrir. D'autres critères sont possibles qui définiraient alors d'autres fragmentations idéales. Plusieurs raisons font que ce critère ne convient pas toujours en pratique. Elles tiennent au fait que les domaines abstraits opèrent des approximations trop grossières pour que l'on puisse tirer suffisamment d'information des fragments désignés.

Nous allons donner ici quelques exemples de limites du critère précédemment évoqué. La section 4.1.4 proposera des solutions pour dépasser certaines de ces limites.

Prenons comme exemple le programme de la figure 4.7 calculant l'indice de l'élément maximum d'un tableau d'entiers. Seul le test sur la valeur de $A[i]$ nous permet d'obtenir de l'information sur le tableau. En particulier, on peut s'intéresser à l'ensemble des cellules accédées par l'expression $A[imax]$. Il peut être décrit après le test et pour tout état σ comme le fragment

$$\{\, A[\ell] \;\big|\; 1 \leq \ell < \sigma(i) \wedge \forall \ell', 1 \leq \ell' < \ell \Rightarrow A[\ell] < A[\ell'] \,\}$$

En effet, l'ensemble des cellules qui ont à un moment de l'exécution été désignées par l'expression $A[imax]$ est en fait l'ensemble des cellules plus grandes que toutes les cellules qui les précèdent dans le tableau. Cet ensemble est effectivement intéressant. Car à la sortie de la boucle, on sait que toute cellule du tableau ayant cette caractéristique aura effectivement été ajoutée à l'ensemble. Et on sait également que pour la valeur finale de $imax$ la cellule désignée par $A[imax]$ est celle d'indice le plus élevé. Autrement dit, on sait que $A[imax]$ est l'élément de valeur maximum dans le tableau.

Cependant, cet ensemble représente un faible intérêt en pratique parce qu'il est difficile à abstraire. Il faut des abstractions puissantes, ayant suffisamment d'expressivité pour décrire des ensembles de cette forme, définis à la fois par des propriétés sur les indices des cellules dans un tableau et sur leur contenu. Nous ne disposons pas de telles abstractions ici. Aussi faudra t-il se reposer sur des moyens supplémentaires pour parvenir à la preuve de tels programmes.

Essayons d'avoir une vision plus générale du problème précédent. Notre critère propose de singulariser temporairement des cellules dans le but de pouvoir traiter de manière précise les instructions qui les accèdent. Puis, une fois l'instruction terminée, on replace la cellule singularisée dans un fragment plus vaste. Il s'ensuit une perte d'information sur la cellule en question. On ne gardera que les propriétés que la cellule a en commun avec les autres cellules du fragment. Plus le domaine abstrait est puissant, plus on a de chances de conserver de l'information sur cette cellule.

Considérons le programme de peu d'utilité pratique suivant.

Pour i *de* 1 *à* n **faire**
  $x \leftarrow A[i]$
  $y \leftarrow A[i]$
  $B[i] = x - y$

Serions nous capables en ne nous basant que sur le critère de prouver que le tableau B ne contient que des cellules ayant pour valeur 0 à la fin de l'exécution ?

Pour i *de* 1 *à* n **faire**
  $a'_1 : \{A[i]\}$
  $x \leftarrow A[i]$
  $a_1 \cup: a'_1$
  $a'_1 : \emptyset$
  $a'_2 : \{A[i]\}$
  $y \leftarrow A[i]$
  $a_2 \cup: a'_2$
  $a'_2 : \emptyset$
  $b' :: \{B[i]\}$
  $B[i] = x - y$
  $b \cup: b'$
  $b' : \emptyset$

Si on suit la méthode à la lettre, la cellule $A[i]$ est donc regroupée avec les autres cellules du tableau et il n'y a plus de variable qui ne décrive que la cellule $A[i]$. On peut toujours conserver

les propriétés de $A[i]$ mais seulement celles qui sont communes avec les autres cellules du fragment. Or, pour la plupart des domaines abstraits, il n'y a pas de propriété qui soit vraie de toutes les cellules parcourues dans cet exemple.

En d'autres termes, la propriété $a'_1 = x$ est éphémère. Elle est vraie après l'affectation. Mais puisqu'on aura pas en général (c'est à dire pour un tableau A quelconque) la propriété $a_1 = x$ la propriété obtenue sur $A[i]$ ne pourra pas être étendue à l'ensemble du tableau parcouru. Puis dès qu'on traite l'instruction $a'_1 : \emptyset$ on cesse de considérer individuellement la cellule $A[i]$ qui n'est désormais plus qu'une cellule comme les autres dans l'ensemble représenté par a_1. On perd donc l'égalité entre x et $A[i]$ et ainsi on ne parviendra pas à conclure de l'égalité entre x et y.

4.1.4 Fragmentation sémantique développée

Afin de repousser un peu plus les limites de notre critère sémantique, nous proposons ici deux variations. Si le critère a une justification convaincante, le développement proposé ici est plus discutable et tient bien plus de l'heuristique. Il est tout à fait possible de s'en tenir au critère initial, qui a le mérite de produire des fragmentations simples. Certains exemples de programmes ne pourront néanmoins être analysés qu'en adoptant les deux améliorations décrites ci-après.

Augmentation de la durée de vie des singletons. Les problèmes que nous venons d'aborder ont un rapport avec la durée de vie des singletons dans la fragmentation. Lorsqu'on doit traiter une instruction, on extrait les cellules accédées pour les placer dans des fragments singletons. C'est le début de la vie d'un singleton. Plus tard, après avoir traité l'instruction, on réintégrera le fragment singleton dans un autre fragment plus grand. Mais on n'est pas forcé de le faire directement après l'instruction traitée, on peut retarder la réintégration du singleton, augmentant effectivement sa durée de vie.

Si la durée de vie du singleton est trop courte, il y a un risque. On peut un peu plus loin dans le programme réutiliser le même singleton. Le singleton sera une nouvelle fois singularisé mais il aura perdu toutes ses propriétés individuelles entre temps. Il faut autant que possible que le singleton reste en vie tant qu'on continue à le manipuler.

On ne peut évidemment pas augmenter la durée de vie d'un singleton indéfiniment. Plus on augmente sa durée de vie, plus la variable de synthèse associée reste longtemps. Ceci se répercute sur le coût des opérateurs du domaine abstrait. À l'extrême, on garderait indéfiniment chaque singleton mais cela serait en contradiction avec le rôle des critères de fragmentation : composer un nombre borné de fragments.

Afin de trouver un compromis, nous nous proposons de conserver le singleton jusqu'à l'exécution suivante de l'instruction qui a nécessité son introduction. Ainsi, le nombre de fragments reste borné par le nombre d'accès et on maximise sous cette contrainte la durée de vie des singletons.

En terme d'instrumentation, cela revient à retarder la composition de sorte qu'elle ait lieu juste avant la singularisation suivante. La réinitialisation devient inutile et on peut la retirer. Voici le résultat pour l'exemple de la section précédente.

Pour *i de* 1 *à n* **faire**
  $a_1 \cup: a'_1$
  $a'_1 : \{A[i]\}$
  $x \leftarrow A[i]$
  $a_2 \cup: a'_2$
  $a'_2 : \{A[i]\}$
  $y \leftarrow A[i]$
  $b \cup: b'$
  $b' :: \{B[i]\}$
  $B[i] = x - y$

Pour le traitement des tests nous pouvons faire persister deux fragments par accès. Le premier pour les tests réussis, le second pour les tests non réussis. Lorsque le test est satisfait, le premier sera le singleton de l'élément accédé tandis que le second sera vide. Lorsque le test n'est pas satisfait, c'est l'inverse. Avant le test, chacun de ces deux fragment sera composé respectivement avec ceux regroupant toutes les cellules ayant validé le test et toutes celles de ne l'ayant pas validé. Voici ce que donnerait notre exemple de test ainsi instrumenté.

$a_1 \cup: a'_1$
$a_2 \cup: a'_2$
$a_{tmp} : \{A[i]\}$
Si $A[i] < 0$ **alors**
  $a'_1 : \{A[i]\}, a'_2 : \emptyset, a_{tmp} : \emptyset$
  $j \leftarrow i$
sinon
  $a'_1 : \emptyset, a'_2 : \{A[i]\}, a_{tmp} : \emptyset$

Traitement des boucles imbriquées Une seconde idée, inspirée de [HP08] permet d'améliorer la précision des analyses en présence de boucles imbriquées. Il peut être utile au moins dans deux cas d'essayer de distinguer l'effet des différents niveaux de boucles.

Ceci est intéressant lorsqu'un algorithme itère plusieurs fois sur les mêmes cellules. Considérons le programme suivant, peu représentatif mais illustratif.

Pour *j de* 1 *à n* **faire**
  $A[j] \leftarrow 0$

Pour *i de* 1 *à n* **faire**
  Pour *j de* 1 *à n* **faire**
    $A[j] \leftarrow A[j] + 1$

Si on s'arrête au critère simplifié, dès la fin de la première itération de la seconde boucle on aura mis dans le même fragment toutes les cellules du tableau. Au milieu d'une itération de la boucle interne il sera impossible de séparer les cellules déjà incrémentées de celles qui ne le sont

pas encore.

Pour traiter ce problème, nous pouvons développer l'idée à l'origine de notre critère de fragmentation. Cette idée était qu'une instruction produit les même effets d'une itération à l'autre. Mais dans l'exemple précédent, on peut considérer que l'effet de l'affectation varie selon l'itération de la boucle externe. Il est vrai que à chaque itération de la boucle interne, l'effet est le même, la même valeur est associée à chaque cellule. Mais pour une autre itération de la boucle externe, la valeur affectée sera différente. Il ne serait donc pas tout à fait logique de chercher à abstraire de la même manière une exécution de l'affectation pour deux itérations distinctes de la boucle externe. On pourrait donc tenter de distinguer à nouveau les cellules suivant l'itération de la boucle externe à laquelle elles ont été modifiées. Afin de ne pas introduire un nombre non borné de fragments, nous nous contenterons de distinguer les cellules suivant qu'elles ont été modifiée par l'itération courante de la boucle externe ou par n'importe laquelle des itérations précédentes. Dans notre exemple, ceci distinguerait dans une même itération de la boucle externe les cellules déjà incrémentées de celles qui ne le sont pas encore.

Pour réaliser cela, il va nous falloir ajouter de nouveaux fragments. Nous avons déjà un fragment pour le singleton actuellement accédé et un fragment pour toutes les cellules auparavant accédées par l'instruction. On va décomposer le dernier en deux selon que les cellules ont été accédées lors de l'itération courante de la boucle externe ou lors d'itérations précédentes. Ainsi on aura dans le premier les cellules dont la valeur égale celle de i et dans les seconds celles dont la valeur égale celle de $i - 1$.

On généralise cette idée à tout programme et quel que soit le nombre d'imbrications. On réserve une variable pour l'accès courant à une cellule et une variable supplémentaire pour chaque boucle dans laquelle l'accès a lieu. Pour chaque accès et chaque niveau d'imbrication i, on introduirait une variable a_i qui correspondrait aux accès à des itérations précédentes des j boucles les plus imbriquées mais à l'itération courante de toutes les autres. En fin de compte les fragments qui ne sont pas des singletons sont identifiés par le couple composé d'une référence à un accès et d'une référence à une boucle. Ces fragments servent à collecter les cellules touchées par cet accès durant l'exécution de cette boucle pour les itérations courantes des boucles extérieures.

En terme d'instrumentation, on peut tenter de conserver la même logique de durée de vie introduite plus tôt dans cette section. On conservait les singletons jusqu'à l'accès suivant. On peut conserver les fragments collectant les cellules accédée dans une boucle jusqu'à ce qu'on rentre de nouveau dans cette même boucle. Alors, on donne leurs cellules aux fragments de la boucle englobante avant de les réinitialiser. Appliquons cela à l'exemple précédent. On nommera a_1 et a_2 les fragments singletons pour les deux accès du programme. L'indice désignant l'accès on utilisera l'exposant pour différencier les boucles dont on collecte les accès. Ainsi, a_1^1 désigne les cellules accédées par la première affectation dans les itérations précédentes de la première boucle. La boucle interne correspond à l'exposant 2 et la boucle externe à l'exposant 3.

Pour *j de* 1 *à n* **faire**
  $a_1^1 \cup: a_1$
  $a_1 :: \{A[j]\}$
  $A[j] \leftarrow 0$

Pour *i de* 1 *à n* **faire**
  $a_2^3 \cup: a_2^1$
  $a_2^2 : \emptyset$
  Pour *j de* 1 *à n* **faire**
    $a_2^2 \cup: a_2$
    $a_2 :: \{A[i]\}$
    $A[j] \leftarrow A[j] + 1$

Cette manière de traiter les boucles peut servir à décomposer les tableaux multidimensionnels dans un autre objectif. Dans l'exemple précédent, ce qui nous intéressait était la séparation des effets des différents niveaux de boucles. Dans le cas de tableaux multidimensionnels, cette méthode peut aussi servir à simplifier l'abstraction des fragments. Considérons l'initialisation d'un tableau à deux dimensions.

Pour *i de* 1 *à n* **faire**
  Pour *j de* 1 *à m* **faire**
    $A[i][j] \leftarrow 0$

L'invariant à l'entrée de la boucle interne se définit aisément de manière formelle. Il peut être désigné de la manière suivante pour un état concret σ.

$$\left\{ A[\ell_1][\ell_2] \;\middle|\; \begin{array}{rcl} \big(1 \leq \ell_1 < \sigma(i) & \wedge & 1 \leq \ell_2 \leq \sigma(m)\big) \vee \\ \big(\ell_1 = \sigma(i) & \wedge & 1 \leq \ell_2 < \sigma(j)\big) \end{array} \right\}$$

Le problème de cet ensemble, c'est que la formule le décrivant contient une disjonction. Or, s'il existe dans la littérature des abstractions de fragments acceptant une formule disjonctive, elles viennent souvent avec des inconvénients. En revanche, on trouvera couramment des abstractions adéquates pour les deux conjonctions dont elle est composée.

Si on utilise la méthode précédemment introduite pour décomposer ce fragment, on aura d'une part un fragment contenant les lignes du tableau complètement parcourues et un fragment contenant la ligne du tableau en cours de parcours. Ainsi, on peut utiliser cette méthode pour faciliter le travail d'abstraction des fragments.

Dans les deux cas, cette méthode remplit le même rôle : assurer une plus grande régularité des fragments choisis en séparant les effets potentiellement distinct des différentes boucles. Dans le premier cas, on capture la régularité des propriétés de contenu des fragments, dans le second on assure la régularité de la forme des fragments.

Ceci repose néanmoins sur la bonne structuration du programme. On peut transposer la méthode à des programmes qui ne sont pas structurés en boucles. Plutôt que d'associer des variables

aux boucles, on peut les associer aux composantes et sous-composantes fortement connexes du graphe de flot de contrôle. Mais il paraît moins évident que la méthode donnera des résultats convaincants sur des graphes de flot de contrôle quelconques.

Ces deux variations peuvent être utilisées pour raffiner les fragmentations. Dans certains cas, ce raffinement améliorera la précision de l'analyse. Dans d'autres, l'effet du raffinement peut être nul et ne faire qu'augmenter la complexité de l'analyse. La première variation conserve un nombre de variables linéaire en la taille du programme aussi longtemps que la durée de vie des variables n'excède pas un nombre borné d'itérations. Dans la seconde en revanche, l'évolution du nombre de variables peut être quadratique puisque ce nombre est proportionnel au nombre d'accès multiplié par le nombre d'imbrications de boucles.

4.1.5 Conclusion

Le critère de fragmentation développé nous donnera quelques succès (Cf. section 10.2). Cependant, on peut s'interroger sur la généralité de celui-ci. De nombreuses variations du critère sont possibles. Si un exemple ne peut être analysé par une variation du critère, il suffit d'en trouver une autre qui fonctionnera. Ce problème est récurrent dans le développement d'un interprète abstrait. Qu'il s'agisse des méthodes d'itération ou encore des choix d'opérateurs d'élargissement, on cherche toujours à les améliorer et les généraliser. Le processus est long. L'accumulation d'une base de tests toujours plus importante permettra avec un peu de chance de dégager des caractéristiques communes aux échecs du critère de fragmentation et d'en trouver une version plus générale.

Le problème que nous cherchons à résoudre à travers ce critère de fragmentation est semblable à un autre : le partitionnement des variables du programme. Ce problème est capital lorsque l'on analyse des programmes avec plusieurs milliers de variables. On cherchera à constituer des groupes de valeurs abstraites indépendantes. On aura une valeur abstraite pour chaque groupe de valeurs. La complexité des opérations sur les valeurs abstraites étant généralement plus que linéaire, la complexité générale de l'analyse s'en trouve réduite. Or notre critère ne fait pas autre chose que de chercher les relation entre les cellules. Il désigne également par effet de bord les cellules qui n'ont pas de relations et il pourrait être utilisé pour partitionner l'ensemble de variables.

La recherche de relations est néanmoins peu précise. On considère qu'il y a relation lorsque deux cellules apparaissent dans une même instruction ou un même test. D'abord, si le domaine abstrait ne peut exprimer cette relation, alors il n'est pas forcément utile de s'en encombrer. Ensuite, il peut y avoir des relations entre des cellules qui n'apparaissent pas dans la même instruction. Si pour copier un tableau on passe par une variable intermédiaire, on trouvera une relation entre la cellule source et cette variable, puis entre la variable et la cellule destination. Mais notre critère ne trouvera pas la relation entre les cellules sources et destination. Plus sournoisement, si une constante est affectée à une cellule de tableau et que cette même constante est affectée un peu plus loin à une autre cellule, alors il y a une relation d'égalité entre ces deux cellules, qu'il peut être utile de considérer.

Une piste pourrait être de recourir continuellement à des opérations du domaine abstrait pour savoir si des relations peuvent être découvertes. On pourrait alors améliorer notre fragmentation selon que l'on sait qu'une relation a été trouvée ou bien au contraire qu'on n'a pu en exprimer aucune.

Chercher toutes les relations peut nous mener à un autre écueil. L'existence d'une relation entre deux cellules est généralement transitive de nature. Si une cellule a est en relation avec une cellule b et que b est elle même en relation avec c alors on aura probablement une relation entre a et c. Pour voir le problème que cela pose, prenons l'exemple d'une propriété exprimant qu'un tableau est trié :

$$\forall \ell,\ 1 \leq \ell < n \Rightarrow A[\ell] \leq A[\ell + 1]$$

On aura découvert que les cellules $A[\ell]$ sont en relation avec les cellules $A[\ell + 1]$ et on aura intégré cette relation dans la fragmentation. Mais si $A[\ell]$ est en relation avec $A[\ell + 1]$ alors on aura par transitivité $A[\ell]$ en relation avec $A[\ell + 2]$. Par transitivité on peut considérer une infinité de relations que nous ne pourrons pas toutes représenter.

4.2 Domaines de fragmentation

A l'image des domaines abstraits permettant d'abstraire des ensembles d'états, nous allons introduire les domaines de fragmentation permettant l'abstraction des fragmentations. Le parallèle entre les domaines abstraits et les domaines de fragmentation peut être poussé assez loin. Analogue des valeurs abstraites, les fragmentations abstraites approximeront les fragmentations symboliques et on passera des premières aux secondes par des fonctions d'abstraction et de concrétisation. On aura une sémantique abstraite permettant de faire évoluer la fragmentation aux différents points de contrôle. Celle-ci sera une approximation des instructions d'instrumentation. L'analogie s'arrête là et il sera nécessaire de décrire de nouvelles opérations propres aux domaines de fragmentation.

Au delà de la formalisation des fragmentations abstraites et des opérations que les domaines de fragmentation doivent implémenter, il y a un enjeu plus grand : la définition d'une interface entre les domaines de fragmentation et les domaines abstraits. En effet, tandis que l'interprétation du programme entraînera des modifications de la fragmentation, celles-ci devront être répercutées sur les valeurs abstraites. Cette interface permettra au domaine de fragmentation de communiquer ces modifications au domaine abstrait. Ce qu'on espère c'est qu'elle permette suffisamment d'indépendance entre ces deux domaines. Il faudrait pouvoir construire un domaine de fragmentation sans se soucier de quel type de propriétés on exprimera sur le contenu de cellules. Inversement, on aimerait bien que les domaines abstraits fonctionnent quel que soit le type de fragmentation utilisé. L'interface se décomposera en sept transformations élémentaires, chacune décrivant une modification de la partition.

4.2.1 Fragmentations abstraites

L'intérêt d'utiliser une abstraction pour les fragmentations symboliques est le même que celui d'utiliser des abstractions pour les ensembles d'états. Pour peu qu'on considère des programmes dans lesquels la taille des structures de données n'est pas statiquement fixée on devra manipuler un ensemble de cellules non borné. L'ensemble des fragmentations pour cet ensemble de cellules est donc lui même infini et même indénombrable. Quand bien même l'ensemble des cellules ne serait pas infini, le nombre de fragmentations reste grand et il est déraisonnable de chercher à pouvoir toutes les représenter. L'intérêt d'utiliser une abstraction pour les fragmentations symboliques est le même que celui d'utiliser des abstractions pour les ensembles d'états. Pour peu qu'on considère des programmes dans lesquels la taille des structures de données n'est

pas statiquement fixée on devra manipuler un ensemble de cellules non borné. L'ensemble des fragmentations pour cet ensemble de cellules est donc lui même infini et même indénombrable. Quand bien même l'ensemble des cellules ne serait pas infini, le nombre de fragmentations reste grand et il est déraisonnable de chercher à pouvoir toutes les représenter.

Une abstraction courante [GDD+04, GRS05, HP08, CCL11] pour les tableaux est de décrire chaque fragment comme une tranche de tableau, c'est à dire comme un ensemble de cellules d'indices consécutifs que l'on peut désigner par un indice minimum et un indice maximum. Le chapitre 5 est consacré à l'élaboration d'un domaine de fragmentation reposant également sur une telle abstraction.

Rappelons qu'une fragmentation symbolique est une fonction qui à chaque état concret associe une fragmentation et qu'une fragmentation est un ensemble de choix de représentants, fonctions de l'ensemble des cellules vers les variables de synthèse. Si on note $\mathbb{F}$ l'ensemble des fragmentations symboliques, on a

$$\mathbb{F} = \Sigma \to \mathcal{P}(\mathbb{C} \dashrightarrow \mathcal{S})$$

Une fragmentation abstraite est une approximation des fragmentations symboliques. On peut définir une fonction $\widehat{\alpha}$ qui à toute fragmentation symbolique de $\mathbb{F}$ associe une fragmentation abstraite. On notera $\widehat{\mathbb{F}}$ l'ensemble des fragmentations abstraites du domaine. La concrétisation $\widehat{\gamma}$ à l'inverse nous permet de savoir quelle est la fragmentation symbolique représentée par un élément de $\widehat{F}$.

$$\widehat{\alpha} : \mathbb{F} \to \widehat{\mathbb{F}}$$
$$\widehat{\gamma} : \widehat{\mathbb{F}} \to \mathbb{F}$$

Nous n'imposons absolument aucune contrainte sur ces fonctions. Ceci diffère des domaines abstraits où ces deux fonctions doivent constituer une correspondance de Galois. Si une fragmentation abstraite est une approximation d'une fragmentation symbolique, il n'est nécessaire que ce soit ni une sous-approximation, ni une sur-approximation.

La correction de l'analyse ne repose effectivement pas sur une bonne définition de $\widehat{\alpha}$ et $\widehat{\gamma}$. Pour la correction, la seule chose importante c'est que quel que soit le choix de la fragmentation, les propriétés qu'on donne à cette fragmentation soient vraies.

En revanche, la qualité de cette approximation jouera sur la précision. Cette qualité est tout à fait subjective. Si l'approximation est trop grossière, le seul risque est que le critère de fragmentation soit mal interprété. Pour savoir si l'approximation est bonne ou non, le seul moyen est d'observer ce qu'elle donne sur beaucoup d'exemples. La section 4.3 propose une courte discussion sur le choix de sous-approximation ou de sur-approximation et des conséquences sur la qualité de l'approximation.

Il n'y a donc aucune propriété qui lie $\widehat{\alpha}$ et $\widehat{\gamma}$. En fait, dans la suite, nous n'utiliserons que $\widehat{\gamma}$. La fonction $\widehat{\alpha}$ est introduite ici pour établir le parallèle avec les domaines abstraits. Elle peut toutefois être définie dans un domaine de fragmentation dans le but d'éclaircir l'intention de l'approximation. On pourra alors l'utiliser pour calculer l'abstraction des instructions d'instrumentation. Mais puisque cette abstraction n'a pas besoin d'être correcte, ce n'est pas nécessaire. En revanche, $\widehat{\gamma}$ nous servira à établir des preuves de correction pour les transformations de fragmentation.

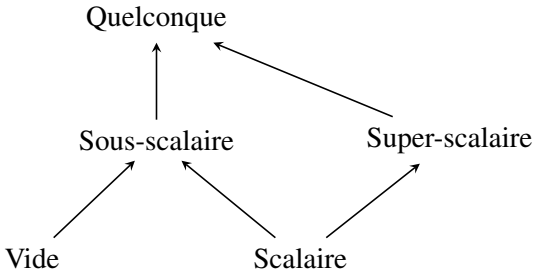


FIGURE 4.8 : Ce diagramme énumère les différents types de fragments, ordonnés des plus spécialisés aux plus généraux.

4.2.2 Gestion des variables de synthèse

Le nombre de fragments variant durant l'analyse, il faudra introduire de nouvelles variables pour les symboliser et en aviser le domaine abstrait. Il sera également nécessaire de fournir une information supplémentaire sur la nature du fragment. En particulier sur le nombre de cellules que ce fragment peut contenir.

- Si le fragment est toujours vide, le domaine abstrait peut y associer l'information maximale.
- Si le fragment a toujours au plus une seule cellule, les affectations au fragment doivent être traitées comme des affectations fortes.
- Si le fragment a toujours au moins une cellule, le domaine abstrait peut dériver des propriétés à partir des informations acquises sur cette cellule, comme s'il s'agissait d'une variable classique.

Lorsqu'un fragment représentera exactement une cellule, au plus une cellule, au moins une cellule, on le qualifiera respectivement de *scalaire*, *sous-scalaire* et *super-scalaire*. Le diagramme figure 4.8 énumère les différents types de fragments.

Il est à la charge du domaine de fragmentation de déterminer et traquer la nature de chaque variable et d'en informer le domaine abstrait.

4.2.3 Opérations des domaines de fragmentation

Comme les domaines abstraits, les domaines de fragmentation doivent implémenter les opérations permettant de faire évoluer la fragmentation tout au long de l'interprétation du programme. Ces opérations sont au nombre de cinq.

Interprétation des instructions et des gardes. Celles-ci ont un effet sur les états concrets et donc sur les fragmentations symboliques. Si un fragment de tableau est défini par rapport à une variable d'indice et que celle-ci est modifiée, il faudra pour conserver le même fragment modifier l'abstraction en conséquence.

Interprétation des instructions d'instrumentation. En général, les instructions d'instrumentation ne pourront pas être interprétées de manière exacte et des approximations seront nécessaires. Pour traduire correctement le critère de fragmentation décrit dans ce chapitre, il faudra implémenter les quatre instructions d'instrumentation dans le domaine de fragmentation.

L'unification des fragmentations. L'unification des fragmentations est une opération qui consiste à transformer indépendamment deux fragmentations de telle manière qu'à la fin de ces transformations ces deux fragmentations soient tout à fait identiques. Cette unification est nécessaire pour que puissent être appliqués les opérateurs binaires du domaine abstraits : comparaison, bornes inférieures et supérieures et élargissement. Ceux-ci n'ont de sens que si les fragmentations des deux opérandes sont les mêmes.

Suivant l'opérateur que l'on souhaite appliquer, les propriétés que l'unification doit vérifier ne sont pas les mêmes. Pour le calcul de la borne inférieure et de la borne supérieure, l'unification peut être tout à fait quelconque. Le domaine de fragmentation est libre de choisir n'importe quelle fragmentation qu'il juge adéquate. Pour les deux autres en revanche, le domaine de fragmentation n'est pas totalement libre.

Pour l'élargissement, l'unification ne doit pas détruire les propriétés de l'élargissement du domaine abstrait. La propriété importante de l'élargissement est que toute suite $(\widetilde{X}_i)_i$ de valeurs abstraites de la forme $\widetilde{X}_{i+1} = \widetilde{X}_i \triangledown \widetilde{Y}_i$ est stationnaire. Mais on doit appliquer l'unification des fragmentations avant de pouvoir appliquer l'élargissement. Cette unification modifie la fragmentation et répercute les changements sur la valeur abstraite. Ce n'est donc plus tout à fait la suite $(\widetilde{X}_i)_i$ que nous construisons. En revanche, il suffirait que la suite des fragmentations considérées à chaque itération soit stationnaire : si après un certain nombre d'unifications la fragmentation ne change plus, alors on peut de nouveau utiliser les propriétés de l'opérateur d'élargissement. Une condition suffisante est donc que pour toute suite $(F_i)_i$ de fragmentations, la suite $(F'_i)_i$ telle que $F'_1 = F_1$ et F'_{i+1} est l'unification de F'_i et de F_i soit stationnaire.

Pour la comparaison, il ne faut pas que l'unification en transformant les fragmentations ne fausse le résultat. Il suffit d'imposer que pour toutes valeurs abstraites $\widetilde{X}$ et $\widetilde{Y}$ telles que $\gamma(\widetilde{X}) \subseteq \gamma(\widetilde{Y})$ alors l'unification des fragmentations associées à chacune de ces valeurs abstraites soit la fragmentation de $\widetilde{Y}$. L'unification peut alors affaiblir $\widetilde{X}$ en $\widetilde{X}'$ mais pas $\widetilde{Y}$ qui reste inchangée. Par conséquent, si après l'affaiblissement on a $\widetilde{X}' \sqsubseteq \widetilde{Y}$ cela implique

$$\gamma(\widetilde{X}) \subseteq \gamma(\widetilde{X}') \subseteq \gamma(\widetilde{Y})$$

et donc il est toujours correct d'utiliser $\sqsubseteq$ pour comparer les deux valeurs abstraites.

La généralisation et la spécialisation. Après l'application d'un élargissement ou l'application d'une garde, les changements de valeur abstraite peuvent faire apparaitre ou disparaitre des singularités dans la fragmentation que l'on peut exploiter pour augmenter la quantité d'information dont on dispose ou au contraire éviter d'en perdre. On appliquera la généralisation et la spécialisation après une perte et un gain d'information respectivement.

Le rôle de la spécialisation se confond avec celui de l'interprétation des gardes. Nous avons néanmoins intérêt à laisser le domaine abstrait traiter la garde pour ensuite en extraire les conséquences de la valeur abstraite. On obtiendra en général plus ainsi que si on interprète directement la garde.

La réduction. On applique la réduction pour répercuter sur la valeur abstraite les conséquences des singularités de la fragmentation. Cette opération consiste à appliquer les règles que nous avons trouvées en section 3.5.

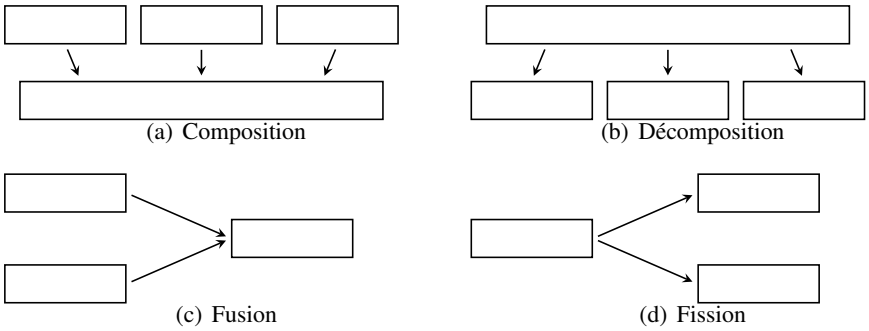


FIGURE 4.9 : Quatre des sept transformations élémentaires. (a) La composition combine des fragments disjoint en un unique fragment union. (b) La décomposition partitionne effectivement un fragment en un ensemble de fragments disjoints dont l'union est le fragment original. (c) La fusion combine deux fragments identiques, c'est à dire représentant les mêmes cellules, de sorte à regrouper les propriétés acquises sur ces deux fragments en un seul. (d) La fission dédouble un fragment en deux fragments identiques, représentant les mêmes cellules, et donc possédant les mêmes propriétés.

4.2.4 Transformations élémentaires

Nous décrivons ici sept transformations élémentaires constituant l'interface entre le domaine de fragmentation et le domaine abstrait. De son coté, le premier est contraint à n'utiliser que ces transformations pour modifier les fragmentations. De l'autre, les domaines abstraits doivent implémenter ces sept transformations afin d'être compatibles.

En pratique, pour chacune des opérations que le domaine de fragmentation doit implémenter et qui sont décrites dans la section précédente, il faudra présenter les transformations qui s'appliquent sur la fragmentation par une suite de transformations élémentaires.

De ces sept transformations, trois d'entre elles sont essentiellement « administratives » et consistent simplement à introduire, retirer ou renommer une variable dans la valeur abstraite. Les quatre autres sont schématisées figure 4.9 et sont la composition, la décomposition, la fusion et la fission. Les deux premières sont des généralisations des fonctions *fold* et *expand* introduites dans [GDD+04] et qui sont implémentées dans la librairie de domaines abstraits APRON [JM09].

Bien que limité, cet ensemble de transformations suffira à la construction du domaine de fragmentation présenté chapitre 5. En outre, la plupart des transformations de fragmentation que l'on peut trouver dans la littérature peuvent s'exprimer grâce à celles-ci.

Introduction, Retrait, Renommage de variables et Changement de type

La fragmentation évoluant, son nombre de fragments est souvent lui même variable. On doit avoir dans le domaine abstrait une opération permettant d'introduire de nouvelles variables dans une valeur abstraite, ou au contraire de les oublier. Ces opérations sont classiques et sont déjà décrites pour la plupart des domaines abstraits.

L'oubli de variable retire cette variable des domaines de définition des choix de représentants.

Oublier la variable t dans la fragmentation symbolique F donne la fragmentation

$$F' : \sigma \mapsto \left\{ r : \begin{cases} v \mapsto r'(v) \text{ si } v \neq t \\ \text{indéfini en } t \end{cases} \;\middle|\; r' \in F(\sigma) \right\}$$

Sur la valeur abstraite, on se contentera de retirer la dimension correspondante par projection.

L'introduction d'une variable permet de modifier librement la fragmentation en associant les cellules que l'on veut à la nouvelle variable. L'application de l'oubli d'une variable après son introduction reviendrait à ne pas changer la fragmentation. Lors de l'introduction d'une variable, le domaine de fragmentation doit informer le domaine abstrait du type de la variable. (Cf. section 4.2.2) Il lui appartient de déterminer si la variable est scalaire, sous-scalaire, super-scalaire ou représente un fragment vide. Dans ce dernier cas, l'introduction de la variable par le domaine abstrait est différente. Si une variable représente un fragment vide, il devra probablement en accord avec sa fonction d'abstraction donner l'information maximum qu'il peut donner à une variable. Pour un domaine non relationnel, cela revient à associer $\bot$ à cette nouvelle variable.

Le renommage de variable permet l'unification de deux fragmentations. Même si deux fragmentations représentent exactement les mêmes fragments avec les mêmes relations, ces fragments peuvent ne pas être associés aux mêmes variables d'un coté et de l'autre. En général, il sera nécessaire d'unifier également les domaines de définition des choix de représentants. Dans ce but, on procèdera aux renommages des variables. Le renommage de la variable s en la variable t dans la fragmentation F donnera

$$F' : \sigma \mapsto \left\{ r : \begin{cases} v \mapsto r'(v) \text{ si } v \neq s, t \\ t \mapsto r'(s) \\ \text{indéfini en } s \end{cases} \;\middle|\; r' \in F(\sigma) \right\}$$

Enfin, le changement de type permet de renforcer ou d'affaiblir le type d'une variable déjà présente dans la fragmentation. Cela n'a pas d'effet sur la fragmentation. Cela survient souvent après des spécialisations ou des généralisations de la fragmentation. Après un élargissement, on peut perdre le type d'un fragment, tandis qu'après le passage d'une garde, on peut en trouver un plus fort.

Composition

La composition permet de regrouper deux fragments disjoints en un unique fragment. Soient s_1, s_2 deux variables de synthèse source et t une variable de synthèse cible. La composition de s_1 et s_2 en t permet d'associer à la nouvelle variable t les cellules associées à s_1 ou s_2. En première approximation, le résultat de cette composition dans la fragmentation F donnerait la fragmentation F' suivante

$$F' : \sigma \mapsto \left\{ r : \begin{cases} v \mapsto r'(v) \text{ si } v \neq t \\ t \mapsto r'(s_1) \end{cases} , r : \begin{cases} v \mapsto r'(v) \text{ si } v \neq t \\ t \mapsto r'(s_2) \end{cases} \;\middle|\; r' \in F(\sigma) \right\}$$

Si cette formalisation est possible, ce n'est pas celle que nous allons garder. Si on s'arrête à cette définition, alors on aura toujours $t = s_1 \vee t = s_2$. Nous aurons besoin d'une composition qui ne créée pas de relation particulière entre s_1 ou s_2 et t. On va donc construire une fragmentation dans laquelle le fragment décrit par la composée t sera en relation avec toutes les cellules de s_1 et avec toutes les cellules de s_2.

Dans ce but, pour chaque choix de représentants r', on cherchera un autre choix de représentants r'' compatible dans le sens où il doit partager les mêmes valeurs sur leur domaine de définition excepté sur t. On utilisera alors leurs différentes valeurs en t pour désigner les valeurs de s_1 et de s_2. Ainsi, les relations entre t et les autres variables sont les mêmes que celles de s_1 et de s_2 mais on n'induit aucune relation entre s_1, s_2 et t.

$$F' : \sigma \mapsto \left\{ r : \begin{cases} v \mapsto r'(v) \text{ si } v \neq t \\ t \mapsto r''(s_1) \end{cases} , r : \begin{cases} v \mapsto r'(v) \text{ si } v \neq t \\ t \mapsto r''(s_2) \end{cases} \;\middle|\; \begin{array}{l} r', r'' \in F(\sigma), \\ \forall v \neq t,\ r'(v) = r''(v) \end{array} \right\}$$

On pourra renforcer la composition en la restreignant aux cas où les deux fragments à composer sont disjoints. Ceci nous sera utile pour les propriétés d'agrégation. (Chapitre 8) On exigera alors que les projections sur les deux variables sources soient disjointes :

$$\pi_{s_1}(F) \cap \pi_{s_2}(F) = \emptyset$$

Décomposition

La décomposition est l'opération inverse de la composition et permet de partitionner un fragment en deux fragments disjoints. Elle est utile, par exemple, pour singulariser une cellule d'un fragment : on décomposera le fragment en cette cellule et le reste du fragment. Le domaine de fragmentation choisit la manière dont il décompose, mais devra vérifier la contrainte suivante : les fragments décomposés doivent être inclus dans le fragment original. Si on décompose dans une fragmentation F une variable source s présente dans la fragmentation en deux variables cible t_1 et t_2 qui ne le sont pas alors la fragmentation résultat F' devra vérifier

$$\forall \sigma,\ \forall r \in F'(\sigma),\ \exists r', r_1, r_2 \in F(\sigma), \begin{cases} \forall v \neq s, t_1, t_2 & r(v) = r'(v) = r_1(v) = r_2(v) \\ & r(s) = r'(s) \\ & r(t_1) = r_1(s) \\ & r(t_2) = r_2(s) \end{cases}$$

Pour la cohérence entre la composition et la décomposition, on pourra exiger que toutes les cellules du fragment source soient présentes dans au moins l'un des deux fragments cibles. Pour les propriétés d'agrégation il faudra en outre exiger la même condition de disjonction que pour la composition.

Fusion

La fusion est une transformation de la fragmentation dans laquelle on prend deux fragments identiques pour les fusionner, combiner leurs propriétés et n'en garder qu'un seul. Si l'effet sur la fragmentation est effectivement de retirer une variable, le but de la fusion est de répercuter l'égalité de deux fragments sur la valeur abstraite avant d'effectuer ce retrait. Par exemple, si nous avons dans la fragmentation deux fragments $\{ A[\ell] \mid 1 \leq \ell \leq \sigma(i) \}$ et $\{ A[\ell] \mid 1 \leq \ell \leq \sigma(j) \}$ et que nous interprétons le test $i = j$ alors ces deux fragments deviennent identiques et nous pouvons les fusionner. Le domaine abstrait gardera la conjonction des propriétés de l'un et de l'autre avant de retirer l'une des variables.

On nomme les deux variables fusionnées source et cible. La source est celle qu'on retire, la cible est celle qui s'enrichit des propriétés de la première. Nous ne décrivons pas l'effet sur la

fragmentation car il s'agit d'une simple projection. En revanche, nous devons décrire la condition nécessaire pour que la fusion puisse s'accomplir. La fusion d'une variable source s en une variable cible t n'est possible dans la fragmentation F que si pour tout choix de représentants, on peut trouver un choix de représentants compatible où s est envoyé sur la cellule associée à t dans le premier choix de représentants :

$$\forall \sigma,\ \forall r \in F(\sigma),\ \exists r' \in F(\sigma), \begin{cases} \forall v \neq s,t & r'(v) = r(v) \\ & r'(s) = r(t) \end{cases}$$

Fission

La fission est l'opération inverse de la fusion. On part d'un fragment que l'on veut fissionner en deux fragments identiques. Pour reprendre l'exemple donné pour la fusion, si on perd l'information $i = j$ par un élargissement ou par une union, on retrouvera deux expressions désignant le même fragment avant la perte d'information mais plus le même après. On peut alors fissionner le fragment source pour retrouver ces deux fragments auxquels on donnera les mêmes propriétés.

La fission d'un fragment source s en un fragment cible t modifie la fragmentation F en la fragmentation F' vérifiant :

$$F' : \sigma \mapsto \left\{ r : \begin{cases} t \mapsto r_2(s) \\ v \mapsto r_1(v) \text{ si } v \neq t \end{cases} \;\middle|\; r_1, r_2 \in F(\sigma), \forall v \neq s \Rightarrow r_1(v) = r_2(v) \right\}$$

Nous pouvons vérifier qu'il est de nouveau possible de fusionner les deux fragments produits.

Proposition 2 (Cohérence de la fission et de la fusion)**.**
La fission de la variable s en t produit une fragmentation dans laquelle la condition de fusion entre s et t est valide.

Démonstration. Soit F une fragmentation et F' le résultat de la fission de s en t dans F. Soit aussi σ un état concret, et $r \in F'(\sigma)$. Comme r vient d'une fragmentation construite par la fission de s et t, on sait qu'il existe $r_1, r_2 \in F(\sigma)$ servant à la construction de r :

$$r : \begin{cases} t \mapsto r_2(s) \\ v \mapsto r_1(v) \text{ si } v \neq t \end{cases}$$

Construisons r' en inversant le rôle de r_1 et r_2

$$r' : \begin{cases} t \mapsto r_1(s) \\ v \mapsto r_2(v) \text{ si } v \neq t \end{cases}$$

On a par définission de la fission

$$\forall v \neq s \Rightarrow r_1(v) = r_2(v)$$

d'où on déduit par définition de r et r'

$$\forall v \neq s,t \Rightarrow r(v) = r'(v)$$

En outre on a

$$r'(s) = r_2(s) = r(t)$$

ce qui permet de vérifier la condition de fusion. □

4.2.5 Conclusion

Les domaines de fragmentation définissent des fragmentations abstraites et les cinq opérations pour les manipuler. Ces opérations réalisées tout au long de l'interprétation du programme ont des répercutions sur les valeurs abstraites qu'ils doivent décrire. Pour cela ils doivent transcrire les transformations de la fragmentation par des séries de transformations élémentaires qui doivent sous-approximer la véritable transformation.

Le domaine abstrait quant à lui doit implémenter les conséquences des sept transformations élémentaires sur la valeur abstraite.

4.3 Sur-approximation et sous-approximation de fragmentations

Si les domaines de fragmentation sont libres d'approximer sans contraintes les fragmentations symboliques concrètes, la précision de l'analyse dépend avant tout d'un choix d'approximation cohérent. Nous parlerons ici de deux types d'approximation caractéristiques : la sous-approximation et la sur-approximation.

Si on accepte le parti pris au début de ce chapitre, et que l'on considère que toute instruction induit sur les cellules qu'elle manipule une propriété similaire à chaque exécution alors la sous-approximation a un intérêt direct. En effet, si on réussit à sous-approximer les cellules touchées par cette instruction, alors on obtiendra un fragment dont toutes les cellules ont cette propriété induite. Si on ne réussit pas à sous-approximer, on aura des cellules parasites dans le fragment qui affaibliront la propriété que l'on souhaitait synthétiser.

Ceci est particulièrement important dans les cas où on n'a pas le loisir d'affaiblir la propriété cherchée. [GMT08] considère des propriétés qui sont choisies parmi un ensemble de prédicats et qui ne peuvent donc pas être affaiblies. La même publication propose donc des méthodes pour calculer des sous-approximations des fragmentations.

Il est légitime néanmoins de se demander s'il est réellement possible de tirer profit des sous-approximations. Pour reprendre l'exemple de l'instruction, pouvons nous réellement être intéressés par un sous-ensemble strict des cellules lues ou écrites par une affectation ? Existe t-il un exemple dans lequel nous serions satisfaits par des propriétés ne concernant qu'une partie tronquée des structures de données ?

Il est tout à fait possible en imaginant un programmeur faisant preuve de trop de zèle d'inventer un exemple où certaines cellules pourtant manipulées par le programme ne servent pas à la construction du résultat. Cependant, nous n'avons pas, dans notre petite base de tests, de programmes où une abstraction exacte d'un fragment est impossible tandis qu'une sous approximation de ce fragment suffirait à la preuve du programme. En outre, la sous-approximation est un problème généralement difficile sur les abstractions courantes des fragmentations. Les algorithmes pour les calculer peuvent être imprécis et la non-unicité des solutions est dérangeante.

Il y a une autre approche au problème de l'approximation. Si on n'a pas d'abstraction pour représenter exactement l'ensemble des cellules touchées par une instruction, on peut toujours lui rajouter des cellules jusqu'à obtenir un ensemble que l'on peut abstraire. Si les cellules ajoutées n'ont pas les mêmes propriétés que celles du fragment original, nous allons évidement perdre en précision. Mais par chance ces cellules peuvent tout de même avoir une partie de leurs propriétés

```
i ← 1
Tant que i ≤ n faire
    A[i] ← 0
    A[i + 1] ← 0
    i ← i + 2
```

```
i ← 1
Tant que i ≤ A[i] faire
    Si A[i] < 0 alors
        A[i] ← −A[i]
    sinon
        A[i] ← A[i] + 1
    i ← i + 1
```

FIGURE 4.10 : Ces deux programmes sont des cas heureux où la sur-approximation du critère de fragmentation donnera des résultats satisfaisants.

en commun, et on sera capable de conserver ces propriétés communes sur la sur-approximation du fragment. Cette idée est à l'origine des diagrammes de tranches, domaine de fragmentation auquel le chapitre 5 est consacré.

L'avantage de la sur-approximation, c'est qu'à tout moment on décrit absolument toutes les cellules manipulées par le programme et on ne risque donc pas de tronquer les structures de données. L'inconvénient c'est qu'il est possible que les propriétés qu'on obtient puissent être trop faibles pour la preuve du programme.

Nous pouvons donner deux usages heureux de la sur-approximation pour l'analyse des programmes de la figure 4.10 Le premier est un simple déroulement de boucle sur un algorithme d'initialisation de tableau. Le second est un programme qui effectue sur chaque cellule d'un tableau un traitement distinct en fonction du signe de la valeur de cette cellule. Imaginons que nous utilisions notre critère de fragmentation sémantique et que les seuls fragments que nous puissions décrire soient des tranches, c'est à dire des ensembles de cellules consécutives délimités par un indice minimum et un indice maximum.

Pour le premier programme, notre critère de fragmentation demandera que l'on considère deux fragments : ceux modifiés par la première affectation et ceux modifiés par la seconde. Si on disposait d'un domaine de fragmentation capable de distinguer les cellules selon leur parité, on pourrait parler des cellules paires d'indice compris entre 1 et i et celles impaires dans le même intervalle. Mais si on se restreint aux abstractions des fragments en tranches, on ne sera pas capable de décrire exactement ces deux ensembles.

Il y a une sur-approximation évidente de ces deux fragments : leur union, toutes les cellules d'indice compris entre 1 et i. Cette sur-approximation nous permet de dire que le tableau est effectivement constant et égal à 0.

Modifions le programme pour qu'au lieu d'initialiser à 0 la seconde affectation initialise à 1. Le programme construirait alors un tableau dans lequel les cellules d'indice impair valent 1 et celles d'indice pair valent 0. En sur-approximant les deux fragments par leur union, on pourrait dire que toutes les cellules du tableau ont une valeur comprise entre 0 et 1. C'est bien sûr moins précis que si on avait pu discriminer dans la fragmentation les indices pairs et les indices impairs. Mais cette information quoique affaiblie peut suffire pour prouver des propriétés du programme, comme l'absence de débordements arithmétiques, qui ne serait pas directement liée à l'alternance des valeurs.

En revanche, si on essayait avec les mêmes abstractions de calculer une sous-approximation,

le résultat ne serait pas très probant. Une sous approximation d'un des deux fragments ne contiendrait au plus qu'une cellule. Par exemple, pour le premier fragment, on pourrait prendre uniquement de la première cellule lorsque i est strictement supérieur à 1, et considérer ainsi la tranche

$$\{\, A[\ell] \mid 1 \leq \ell < \min\{i, 2\} \,\}$$

Or même si on peut établir à la fin du programme que la première cellule a pour valeur 0, cela risque de ne pas être suffisant.

Dans le second exemple, la réussite du test à chaque itération de la boucle dépend des valeurs de A que nous considérons comme une entrée du programme. Il n'est pas possible de décrire les fragments des cellules validant le test et des cellules ne le validant pas à l'aide de tranches. La seule manière de distinguer ces deux fragments est de parler de « l'ensemble des cellules de valeur négative » et « l'ensemble des cellules de valeur positive ou nulle ». Ceci va au delà de l'expressivité que nous autorise la fragmentation en tranche. Mais cette fois encore, on peut sur-approximer ces deux fragments par l'union des deux. L'union des cellules validant le test et de celles ne le validant pas est l'ensemble des cellules d'indice compris entre 1 et i. Par chance il y a bien une propriété qui convienne à cette sur-approximation : toutes ces cellules, après avoir été mises à jour, deviennent strictement positives. Pour finir, on peut remarquer que comme pour le premier exemple, la sous-approximation n'apporterait rien d'intéressant.

Cette courte discussion sur le choix d'une approximation ne permet pas de conclure sur la supériorité d'une méthode d'approximation. Elle permet tout au plus de comprendre les choix qui nous ont amenés à développer des abstractions des fragmentations basées sur leur sur-approximation comme celles développées dans le chapitre suivant.

5 Diagrammes de tranches

La forme de fragment de tableau la plus courante est la *tranche*. Une tranche est une suite de cellules consécutives d'un même tableau. Elle est omniprésente dans les exemples que nous avons vus jusqu'ici. La forme la plus simple d'un parcours de tableau est de le traverser du premier élément au dernier. Que ce soit dans ce sens ou dans le sens inverse, à tout instant de l'exécution, l'ensemble des éléments énumérés par un tel parcours constitue une tranche.

Il est donc important d'avoir un domaine de fragmentation qui soit en mesure de décrire des tranches. Bien sûr, certains fragments de tableaux ne sont pas de cette forme. Mais on pourra toujours combiner un domaine spécialisé dans la description des tranches avec un domaine décrivant d'autres formes, afin d'obtenir des fragments moins classiques.

Les diagrammes de tranches sont une représentation abstraite d'un ensemble de tranches permettant la résolution d'un certain nombre de problèmes algorithmiques que cette abstraction pose. Les diagrammes permettent uniquement la définition de fragments. Ils ne permettent pas la description d'éventuelles relations entre ces fragments qui pourraient s'exprimer par des contraintes relationnelles sur les indices des cellules.

5.1 Tranches et notations

Une tranche est une suite de cellules consécutives d'un même tableau. On peut la décrire de manière équivalente comme l'ensemble des cellules d'un tableau dont l'indice est compris dans un intervalle fixé. Elle est donc la donnée de trois paramètres : le tableau, (ou le segment de mémoire, suivant le modèle concret de mémoire adopté) ainsi que les bornes inférieures et supérieures d'indice. Ainsi, on peut parler de la tranche des cellules du tableau A d'indice compris entre a et b que nous noterons $A[a..b]$:

$$A[a..b] = \{\, A[\ell] \mid a \leq \ell \leq b \,\}$$

On peut aussi parler directement de l'intervalle d'indices :

$$[a..b] = \{\, \ell \mid a \leq \ell \leq b \,\}$$

Ces intervalles entiers d'indice peuvent être ouverts à droite et/ou à gauche :

$$[a..b] = [a..b+1[=]a-1..b] =]a-1..b+1[$$

Quelle que soit la forme de l'intervalle d'indice, on pourra l'appliquer derrière un symbole de tableau pour construire la tranche correspondante :

$$A[a..b] = A[a..b+1[= A]a-1..b] = A]a-1..b+1[$$

Lorsque cet intervalle est réduit à une seule valeur i, on utilisera la notation $[i]$ permettant la correspondance avec la notation classique des accès aux tableaux.

$$[i] = [i..i]$$

Sauf exception, nous utiliserons généralement la convention « fermé à gauche, ouvert à droite » pour dénoter les intervalles d'indice et les tranches de tableau lorsqu'il ne s'agit pas de singletons.

5.2 Principes

Les tranches qui nous intéressent sont *symboliques*. Leurs bornes peuvent être des variables du programme dont l'évaluation dépend de l'état considéré. En conséquence, lors de l'interprétation d'un programme, on ne sera pas toujours en mesure de savoir si une tranche doit être modifiée ou non par une instruction. Imaginons que nous ayons la tranche $A[1..i[$ et qu'on ait à traiter une affectation à $A[j]$. Il peut y avoir des états du programme pour lequel j est compris entre 1 et i et des états pour lequel il ne l'est pas. Dans les premiers, l'affectation modifiera la tranche, et dans les seconds elle ne la modifiera pas.

Une autre conséquence est qu'on ne peut pas en général comparer de manière totale les bornes des tranches. Par exemple, la vacuité de la tranche $A[i..j[$ peut dépendre des états concrets considérés. Dans certains états i peut être strictement inférieur à j, tandis que dans d'autres non. On ne pourra pas non plus toujours décider du chevauchement de deux tranches, ainsi que d'un certain nombre de propriétés qui nous seront nécessaires pour la manipulation des tranches.

Notre objectif est de construire un domaine de fragmentation qui puisse décrire des collections de tranches. Il va falloir définir les opérations dont les domaines de fragmentations doivent être pourvues. Si nous essayons maintenant de le faire, nous allons nous heurter à la barrière de la complexité. En effet certaines de ces opérations ont un coût exponentiel. Pour rester dans des coûts raisonnables, il va falloir concéder et chercher des opérations simples mais moins précises sur ces collections des tranches. Pour saisir les difficultés que ces opérations entraînent, nous allons considérer deux problèmes qui s'opposent à la construction du domaine de fragmentation : l'approximation des tranches et la réduction des valeurs abstraites.

Premier problème : l'approximation des tranches. Supposons que l'on cherche à approximer un critère de fragmentation comme celui introduit dans le chapitre 4. Supposons également que ce critère nous invite à considérer la tranche $A[1..i[$. C'est à dire que nous sommes dans un cas où il n'est pas nécessaire d'approximer la fragmentation car nous avons l'expressivité pour la décrire. Si à un moment postérieur, la variable i est modifiée par une affectation non-inversible, l'expression $A[1..i[$ de la tranche ne conviendra plus. Il faudra alors décrire cette tranche par une autre expression équivalente. Si une autre variable j a été définie et que sa valeur est égale à celle de i alors on aura une expression de remplacement évidente : on pourra substituer à l'expression originale $A[1..i[$ la nouvelle expression $A[1..j[$. Si en revanche, il n'y a pas de telle variable j ni aucune expression arithmétique que l'on sache égale à i, il faut résoudre un problème. Cela signifie en effet qu'on ne connaît pas d'expression de tranche capable de décrire exactement celle-là.

Alors, au moins trois possibilités s'offrent à nous.

1. Nous pouvons simplement décréter que nous ne sommes de toute façon plus en mesure de désigner la tranche en question. On oublie cette tranche et toute l'information qu'on a pu récolter à son propos.
2. Puisque cette tranche a été désignée par un critère, on admet que ses cellules sont d'un certain intérêt. On essaie de conserver un maximum d'information déjà obtenue sur cette

tranche. Cette information vaut pour toute tranche qui soit incluse dans la tranche originale. On peut donc substituer à celle-ci n'importe quelle tranche strictement incluse. Autrement dit, on cherche une ou des sous-approximations des tranches dont on n'a plus d'expression exacte après l'affectation. Par exemple, si la variable j à défaut d'être égale à i lui est (strictement) inférieure alors la tranche $A[1..j[$ en sera une (stricte) sous-approximation.

3. Inversement, on peut chercher à calculer une sur-approximation de la tranche originale. Pour reprendre l'exemple précédent, cela signifie que si j est (strictement) supérieure à i alors $A[1..j[$ sera une (stricte) sur-approximation de $A[1..i[$. Cette fois en revanche, les propriétés vraies de la tranche originale ne seront pas nécessairement vraies de la tranche de substitution. Mais si par chance les cellules rajoutées dans l'approximation possèdent des propriétés communes avec les cellules de la tranche originale alors on pourra conserver ces propriétés.

La question peut être longuement débattue. Ici, notre choix s'est arrêté sur la recherche de sur-approximation. Détaillons un peu plus comment le choix de la sur-approximation nous permet de répondre à la question initiale Notre problème devient la recherche d'une tranche qui à la fois contienne la tranche originale (sur-approximation) et puisse être désignée par une expression de tranche valide (Ne contienne plus la variable d'indice affectée). On peut raisonnablement rajouter une contrainte supplémentaire : il faut que les cellules que l'on rajoute à la tranche initiale appartiennent déjà à une autre tranche pour laquelle on a aussi de l'information. En effet, les propriétés de la nouvelle tranche seront la disjonction des propriétés des cellules qui la composent, il faut alors que toutes aient une propriété intéressante. Si ne serait-ce qu'une cellule de la nouvelle tranche n'a aucune propriété, on ne pourra rien dire du tout de cette nouvelle tranche et cela n'a pas plus d'intérêt que de simplement oublier la tranche.

On peut, et c'est essentiel, tourner le problème d'une autre manière. La tranche de substitution sera composée de la tranche originale et d'un ajout d'autres cellules de diverses tranches. Elle aura pour propriété la disjonction des propriétés de la tranche originale et des tranches la complétant. Mais quitte à prendre quelques cellules d'autres tranches, on peut les prendre toutes : cela n'affaiblira pas d'avantage les propriétés de cette nouvelle tranche. Autrement dit, on peut prendre l'union de ces tranches comme sur-approximation de la tranche originale.

Par exemple, reprenons notre tranche $A[1..i[$ dont la borne i est invalidée et supposons que nous ayons aussi capturé de l'information à propos de la tranche $A[j..n]$ avec $1 \leq j \leq i \leq n$. Nous pouvons compléter la tranche $A[1..i[$ avec des cellules de $A[j..n]$ afin d'obtenir une tranche, plus grande, dont l'expression ne contiendra pas i. Sur cette nouvelle tranche, les propriétés seront celles vraies des deux tranches à la fois. Mais quitte à compléter ainsi, autant ne pas se limiter à quelques cellules et ajouter toutes les cellules de $A[j..n]$. Cela n'affaiblira pas plus la propriété résultante. On propose ainsi la tranche $A[1..n]$ comme sur-approximation de la tranche $A[1..i[$ et on calcule les propriétés de cette nouvelle tranche comme disjonction des propriétés vraies de $A[1..i[$ et $A[j..n]$.

Cette manière de calculer les sur-approximations nous permet de reformuler le problème précédent. On cherchera désormais des tranches dont l'union avec la tranche originale nous donne une sur-approximation intéressante. Qu'elle soit intéressante signifie ici deux choses. D'une part cette sur-approximation doit elle-même être une tranche afin qu'on puisse toujours la représenter. D'autre part, elle doit être la plus petite possible afin de ne pas trop réduire la qualité de l'approximation. Chose malheureuse, on peut ainsi avoir plusieurs sur-approximations minimales qui soient chacune l'union de plusieurs tranches. Si on devait donner une solution naïve, la

résolution du problème reviendrait donc à énumérer tous les sous-ensembles de tranches et à calculer si possible pour chacun d'eux l'union de leurs éléments. Cette énumération exponentielle de toutes les sur-approximations est coûteuse et ne paraît pas être une solution prometteuse.

Second problème : la réduction. Laissons un moment de coté le problème précédent. Passons maintenant à un autre problème, a priori différent, mais avec lequel nous verrons des similarités. Il s'agit du problème de la réduction. Nous avons vu en section 3.5 que si un ensemble de fragments recouvre un autre fragment, alors on peut dire du fragment recouvert qu'il doit avoir les propriétés qui sont vraies de tous les fragments qui le recouvrent. Dans notre cas, cela signifie que si les cellules d'un fragment $A[1..20[$ ont une valeur inférieure à 3 et que les cellules d'un fragment $A[10..30[$ ont une valeur inférieure à 7, alors on peut dire d'un fragment $A[5..25[$ que ses cellules une valeur elle aussi inférieure à 7.

Quelle conséquence pour notre collection de tranches ? Potentiellement, n'importe quel sous-ensemble de tranches peut recouvrir n'importe quelle tranche. Il faudrait alors énumérer le nombre exponentiel des sous-ensembles de tranches, calculer pour chacun d'eux s'ils recouvrent une ou plusieurs tranches de la collection, et si c'est le cas, calculer les propriétés que ce recouvrement de tranches transfèrent à la tranche recouverte. Cette fois encore, on se heurte à un problème de complexité de l'opération.

Le fondement des diagrammes. Ces deux problèmes ont un point commun. Dans les deux cas, on cherche à énumérer des unions de tranches. C'est cette énumération qui est au cœur de la construction des diagrammes. Le problème général de l'énumération étant intrinsèquement coûteux à résoudre, nous n'allons tenter d'en résoudre qu'une version simplifiée. Nous allons contraindre la collection de tranches. Nous allons lui fixer des propriétés qu'elle doit vérifier. Sous ces contraintes, il sera possible de résoudre les problèmes précédents et d'autres de façon beaucoup plus simple et efficace. En contrepartie, l'analyse perdra en précision.

Le problème principal est d'être capable de décider quand une union de tranches est elle-même une tranche. Commençons par étudier le cas de l'union de deux tranches. Le problème peut s'énoncer de la manière suivante. Étant donnée une tranche, avec quelle autre tranche a-t-elle une union qui soit elle-même une tranche ?

Lorsque l'union de deux tranches est effectivement elle même une tranche, on peut alors établir la relation suivante :

$$A[i..j[\cup A[k..l[= A[\min(i,k)..\max(j,l)[$$

Pour que cette relation soit vraie, il faut que les deux tranches soient « collées » l'une à l'autre, qu'il n'y ait pas de « vide », c'est à dire qu'il n'existe pas d'indice entre ces deux tranches. En général, nous utiliserons des expressions arithmétiques qui ne contiennent pas les opérateurs min et max. La tranche union devra alors avoir soit a soit c en borne inférieure et soit b soit d en borne supérieure. Autrement dit, il faut qu'il soit possible d'ordonner les bornes, c'est à dire que la valeur abstraite implique pour tout état concret une relation entre a et c ainsi qu'entre b et d.

A partir de là, il est possible d'identifier quatre cas.

- Cas 1, $i \leq k$ et $j \geq l$: la première tranche inclut la seconde et leur union est elle-même la première tranche.
- Cas 2, $i \geq k$ et $j \leq l$: cas symétrique au premier, la seconde tranche inclut la première et leur union est elle-même la seconde tranche.
- Cas 3, $i \leq k$ et $j \leq l$: la première tranche est « à gauche » de la seconde. Leur union sera exacte si les deux tranches sont collées ou se chevauchent, c'est à dire si $k \leq j + 1$.

- Cas 4, $i \geq k$ et $j \geq l$: cas symétrique au troisième. L'union des deux tranches est exacte si $i \leq l + 1$.

Ces quatre cas sont facilement caractérisables et on peut espérer, avec une bonne structure de données, être capable de déterminer lequel se présente. Mais nous pouvons simplifier encore le problème sans perdre de généralité. Supposons que nous soyons dans le troisième cas et que nous ayons donc la relation

$$i \leq k \leq j + 1 \leq l + 1$$

Nous pouvons calculer l'union et respectivement l'intersection de ces deux tranches :

$$A[i..j[\cup A[k..l[= A[i..l[$$
$$A[i..j[\cap A[k..l[= A[k..j[$$

On pourrait modifier la fragmentation de sorte qu'au lieu de ces deux tranches, elle contienne leur intersection et leurs complémentaires par rapport à cette intersection. C'est à dire remplacer les tranches $A[i..j[$ et $A[k..l[$ par les tranches $A[i..k[$, $A[k..j[$ et $A[j..l[$. Ce serait une fragmentation strictement plus fine, et on ne perdrait alors pas de précision dans l'analyse. On peut répéter le processus pour toute paire de tranches dont l'intersection est elle-même une tranche. Cela simplifie davantage notre problème. D'abord, les deux premiers cas n'arrivent plus. Lorsqu'une tranche est incluse dans l'autre on garde la tranche incluse et son ou ses deux complémentaires. Les cas trois et quatre sont simplifiés en ce que deux tranches ne peuvent plus se chevaucher, elles ne peuvent qu'être collées. On peut trivialement calculer les unions :

$$A[i..k[\cup A[k..j[= A[i..j[$$
$$A[k..j[\cup A[j..l[= A[k..l[$$

Dans cette configuration notre problème devient lui-même trivial. Si on part d'une tranche $A[i..j[$ trouver une tranche avec laquelle son union sera une tranche se limite à la recherche de tranche de la forme $A[j..k[$ ou de la forme $A[l..i[$. C'est à dire une tranche dont la borne gauche est successeur de la borne droite de la tranche originale ou bien une tranche dont la borne droite est prédécesseur de la borne gauche de la tranche originale.

Il reste toutefois un détail pour que la dernière remarque soit valide. L'égalité

$$A[i..j[\cup A[j..k[= A[i..k[$$

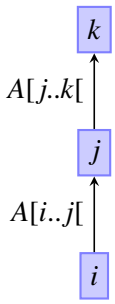
n'est vraie qu'à la condition $i \leq j \wedge j \leq k$, c'est à dire lorsque les bornes inférieures sont inférieures à la borne supérieure. L'inégalité est large et si on a $i = j$ ou $j = k$, l'une ou l'autre des tranches est vide mais l'égalité reste vraie. Afin de pouvoir utiliser l'égalité et calculer trivialement les unions de tranches, nous imposons cette nouvelle condition aux tranches : la borne inférieure d'une tranche devra être inférieure ou égale à la borne supérieure.

Les diagrammes de tranches. Grâce aux simplifications que nous venons de lister, il est maintenant facile d'énumérer un certain nombre d'unions de deux tranches ou plus. Il est possible de chaîner autant d'unions qu'on le désire tant qu'on trouve des tranches à mettre bout à bout :

$$A[i_1..i_2[\cup A[i_2..i_3[\cup \cdots \cup A[i_{n-2}..i_{n-1}[\cup A[i_{n-1}..i_n[$$

On peut chercher une telle séquence de tranches en la construisant itérativement. On cherche successivement des tranches de la collection qu'on puisse mettre au bout de la séquence. Ceci ressemble très fortement à la recherche de chemins dans un graphe orienté. Les arcs de ce graphe sont les tranches et les sommets sont leurs bornes. Les arcs sont orientés de sorte à aller de la borne inférieure vers la borne supérieure. Chaque nouvel arc ajouté à un chemin correspond à une tranche ajoutée à l'union.

Si nous avons les deux tranches $A[i..j[$ et $A[j..k[$, nous pouvons les représenter par un graphe à trois sommets i, j et k et deux arcs entre i et j et entre j et k étiquetés par $[i..j[$ et $[j..k[$ respectivement.



Si on souhaite construire la tranche $A[i..k[$, on peut chercher les chemins du sommet étiqueté i au sommet étiqueté k. On trouve le chemin passant par les arrêtes $[i..j[$ et $[j..k[$, l'union de ces deux tranches dans le tableau A donnant bien $A[i..k[$.

Ce graphe nous donne une modélisation agréable des deux problèmes initialement posés. Si une affectation non-inversible invalide une borne, les tranches délimitées par cette borne doivent alors être retirés du graphe. On va sur-approximer chacune de ces tranches en prenant les unions d'une tranche d'un arc entrant avec une tranche d'un arc sortant. On obtient ainsi une sur-approximation minimale (se limitant à l'union de deux tranches) qui est une tranche et dont l'expression ne contient pas la borne invalidée. Si dans l'exemple précédent, la borne j est invalidée, les deux tranches $A[i..j[$ et $A[j..k[$ le sont également et on peut les remplacer par leur union, $A[i..k[$. L'algorithme complet est détaillé en section 5.5.4.

Le problème de la réduction quant à lui demandera un peu plus de travail et on lui consacrera la section 5.5.3.

La relation d'ordre sous-jacente. La valeur abstraite induit une relation d'ordre partiel $\preceq$ entre les bornes : on aura $i \preceq j$ dans une valeur abstraite $\widetilde{X}$ si pour tout état $\sigma \in \gamma(\widetilde{X})$ de la concrétisation on a $\mathcal{E}[i](\sigma) \leq \mathcal{E}[j](\sigma)$. La contrainte que nous avons fixée sur les diagrammes qui nous autorise à ne considérer que des tranches dont la borne inférieure est inférieure à la borne supérieure s'écrit effectivement avec cette relation d'ordre.

Un cas particulier de diagramme est celui des diagrammes de Hasse de cette relation d'ordre. [1] Dans ce diagramme, des sommets i et j vérifiant $i \preceq j$ sont reliés par un chemin et il n'y a d'arc (i, j) que s'il n'y a pas d'autre chemin reliant ces deux sommets. Considérons deux tranches $A[i..j[$ et $A[k..l[$ se chevauchant et dont l'intersection est $A[k..j[$. Autrement dit, on a $i \preceq k \preceq j \preceq l$. Alors, dans le diagramme de Hasse, il y aura un chemin entre i et k, entre k et j, entre j et l mais il n'y aura pas d'arc (k, l) et (k, j) puisqu'on peut les décomposer avec

1. Le diagramme de Hasse d'une relation d'ordre est un graphe transitivement réduit dans lequel les sommets sont les éléments ordonnés et un chemin relie toute paire de sommet dont le premier est inférieur au second.

les chemins cités. Par conséquent, ces deux tranches qui se chevauchent ne sont pas dans le diagramme, mais on peut les obtenir par union de tranches.

De manière générale, le diagramme de Hasse vérifie deux propriétés :

- toute tranche dont les bornes sont des sommets du diagramme peut être obtenue par union de tranches présentes dans le diagramme, et
- aucune intersection de tranches présentes dans le diagramme n'est elle même une tranche.

Ces propriétés nous assurent une certaine simplicité des opérations liées aux diagrammes de tranches. Nous chercherons autant que possible à avoir des diagrammes de cette forme. Les diagrammes de Hasse seront donc la forme canonique des diagrammes de tranches. Pour obtenir cette forme canonique, il faudra parfois rajouter des tranches, quitte à ne pas associer de variable de synthèse à ces tranches. Certaines opérations sur les diagrammes de tranches nous feront perdre cette forme canonique. Nous aurons temporairement des diagrammes dont le graphe n'est pas transitivement réduit.

Rappelons enfin que le graphe d'un diagramme de Hasse est unique à isomorphisme près. Pour un ensemble de sommets fixé et une relation d'ordre partiel entre ces sommets, il n'existe qu'un seul graphe transitivement réduit dans lequel deux sommets sont reliés par un chemin si le premier est inférieur au second. Nous aurons besoin par la suite de construire ce graphe unique, notamment lors de l'unification de deux diagrammes de tranches. (Cf. section 5.5.6)

5.3 Domaine de bornes

Les sommets des diagrammes représentent les bornes des tranches. Ce sont des expressions symboliques que nous restreindrons aux expressions de la forme $i + c$ ou simplement c, où i est une variable du programme et c une constante. S'il est possible d'élargir l'ensemble des expressions acceptées, la famille de bornes que l'on s'autorise à utiliser dans les diagrammes doit répondre à certaines contraintes que nous détaillerons.

Sur cet ensemble de bornes, nous décrirons un certain nombre d'opérateurs qui nous seront utiles dans la manipulation des diagrammes. La donnée de l'ensemble des bornes admises et de ces opérateurs constitue un *domaine de bornes*. Ce domaine de bornes est un paramètre des diagrammes qui peut être substitué.

Nous notons $\mathcal{B}$ le domaine de bornes choisi. Nos expressions $i + c$ peuvent être désignées par les ensembles de couples d'une variable et d'une constante entière. En outre, on désignera par $(0, c)$ les bornes constantes :

$$\mathcal{B} = \Big(\mathcal{V} \cup \{0\}\Big) \times \mathbb{Z}$$

où $\mathcal{V}$ est l'ensemble des variables entières du programme.

Dans un état concret donné, ces bornes peuvent être évaluées. Nous noterons $\dot{\gamma}(b)$ la concrétisation de la borne b c'est à dire la fonction qui à un état σ associe la valeur de la borne dans cet état.

$$\dot{\gamma}(v, c) : \sigma \mapsto \begin{cases} \sigma(v) + c & \text{si } v \in \mathcal{I} \\ c & \text{si } v = 0 \end{cases}$$

A partir de cette fonction $\dot{\gamma}$ on peut définir la concrétisation d'une tranche dans un diagramme. Si un arc relie le sommet b_1 au sommet b_2, alors il représente l'intervalle d'indices $[b_1..b_2[$ qui

s'évalue pour un état σ en l'ensemble d'indices

$$\{\, \ell \mid \dot{\gamma}(b_1) \leq \ell < \dot{\gamma}(b_2) \,\}$$

Nous présentons maintenant les opérateurs que les domaines de bornes doivent implémenter. La correction de ces opérateurs se vérifie grâce à la fonction $\dot{\gamma}$ que nous venons d'introduire. Leur implémentation dépend du domaine abstrait. Il faut donc que le choix du domaine abstrait soit cohérent avec le choix du domaine de bornes. Pour des bornes de la forme $i + c$, le domaine des zones sera adapté.

Comparaison des bornes Les diagrammes reposent sur la relation binaire $\preceq$ introduite plus tôt et permettant la comparaison de deux bornes. Cette relation doit être transitive et réflexive. En outre, elle doit abstraire correctement la relation d'ordre induite par la valeur abstraite que les diagrammes fragmentent. Autrement dit, pour une valeur abstraite $\widetilde{X}$, $\preceq$ doit vérifier

$$\forall \sigma \in \gamma(\widetilde{X}),\ \forall b_1, b_2 \in \mathcal{B}^2, \quad b_1 \preceq b_2 \ \Rightarrow\ \dot{\gamma}\,[b_1]\,(\sigma) \leq \dot{\gamma}\,[b_2]\,(\sigma)$$

L'implication ne va que dans un sens : l'opérateur n'est pas être nécessairement complet et certaines bornes comparables peuvent ne pas être comparées.

A partir de cette relation, on peut construire l'ordre strict et l'équivalence notés respectivement $\prec$ et $\equiv$.

$$b_1 \equiv b_2 \Leftrightarrow b_1 \preceq b_2 \wedge b_2 \preceq b_1$$
$$b_1 \prec b_2 \Leftrightarrow b_1 \preceq b_2 \wedge \neg(b_2 \preceq b_1)$$

Lorsque deux bornes de $\mathcal{B}$ sont équivalentes, on peut utiliser l'une ou l'autre indifféremment dans un diagramme sans en changer le sens.

Dans le domaine des zones, (Cf. section 2.3.3 la comparaison de deux bornes $i + c$ et $j + d$ est directe :

$$i + c \leq j + d \ \Leftrightarrow\ i - j \leq d - c$$

Création de bornes La singularisation de cellules dans les diagrammes requiert qu'on ajoute deux bornes aux diagrammes : la borne inférieure et supérieure de la tranche singleton à ajouter. Nous utiliserons deux opérateurs *inf* et *sup* qui à l'expression d'indice de la cellule à singulariser associent respectivement ces deux bornes. Pour une valeur abstraite $\widetilde{X}$ ces opérateurs sont corrects si

$$\forall \sigma \in \gamma(\widetilde{X}),\ \dot{\gamma}\,[\mathit{inf}(\mathit{exp})]\,(\sigma) = \dot{\gamma}\,[\mathit{sup}(\mathit{exp})]\,(\sigma) - 1 = \mathcal{E}\,[\mathit{exp}]\,(\sigma)$$

Ces bornes n'existent pas toujours. Si l'expression d'indice *exp* n'est sémantiquement équivalente à aucune expression de la forme $i + c$ ou c, nous ne pourrons pas trouver ces deux bornes dans notre ensemble $\mathcal{B}$. On autorise donc les fonctions *inf* et *sup* à être partielles.

Fission des bornes Lorsque deux bornes sont équivalentes, on notera indifféremment l'une ou l'autre des bornes dans un diagramme. Mais plus tard dans l'analyse, deux expressions qui étaient équivalentes peuvent cesser de l'être. Ceci arrive dans deux situations : on a perdu l'information de leur équivalence par élargissement ou par union. Dès que deux expressions de bornes ne sont plus équivalentes et à défaut de pouvoir choisir laquelle garder, on peut conserver les deux expressions. Dans le graphe, le sommet représentant l'expression peut être dédoublé et on associera avec chacune de ces copies sa nouvelle expression.

Soient $\preceq_1$ et $\preceq_2$ deux relations d'ordre partiel sur $\mathcal{B}$. Nous utiliserons un opérateur $\mathit{fission}_{\preceq_1,\preceq_2}$ qui associe à une expression de borne b l'ensemble des expressions de bornes équivalentes à b pour $\preceq_1$ mais pas pour $\preceq_2$. Autrement dit, si on note $\equiv_1$ et $\equiv_2$ les deux relations d'équivalence pour $\preceq_1$ et $\preceq_2$ respectivement, on doit avoir

$$b' \in \mathit{fission}_{\preceq_1,\preceq_2}(b) \;\Rightarrow\; b \equiv_1 b' \wedge \neg(b \equiv_2 b')$$

Une fois encore, l'opérateur n'est pas nécessairement complet et certaines bornes qui vérifient les conditions peuvent ne pas être énumérées.

Cet opérateur est celui qui nous empêche d'avoir un ensemble de bornes $\mathcal{B}$ trop général. Par exemple, si on autorise n'importe quelle expression linéaire et dès qu'il y a une relation linéaire entre les variables du programme, le nombre de bornes équivalentes devient infini. On n'a alors pas de moyen a priori de choisir lesquelles conserver. Notre ensemble de bornes réduit a l'avantage de désigner des bornes couramment utiles.

Dans le domaine des zones, trouver ces les bornes équivalentes à $i + c$ revient à lister les variables j telles que $j = i + d$ dans la première valeur abstraite mais pas dans la seconde. Les expressions pouvant être substituées à $i + c$ seront donc $j - (d + c)$.

Opérateurs sémantiques Les affectations à des variables d'indice changent inévitablement les évaluations des expressions de bornes et donc la concrétisation des diagrammes. Dans notre cadre, seules les instructions d'instrumentation doivent modifier la fragmentation. Nous chercherons à modifier les bornes de sorte à inverser l'effet des affectations et conserver une fragmentation constante.

Considérons par exemple la borne i. Si l'instruction $i \leftarrow i + 1$ est interprétée, on pourra remplacer la borne i par la borne $i-1$ dont la valeur après l'instruction sera la même que la valeur de la borne originale avant l'instruction. Un peu différemment, si $i = j$ et que i est affectée de manière non-inversible, ($i \leftarrow 0$ par exemple) on pourra remplacer la borne i par la borne j.

Si on note $[\mathit{ins}](\sigma)$ l'application de la sémantique de l'instruction ins sur l'état σ, on doit avoir un opérateur $[\dot{\mathit{ins}}](b)$ associant à la borne $b \in \mathcal{B}$ une nouvelle borne b' dont l'évaluation dans l'état résultat est la même que celle de b dans l'état initial.

$$\forall \sigma \in \gamma(\widetilde{X}),\ \dot{\gamma}\,[b]\,(\sigma) = \dot{\gamma}\left[[\dot{\mathit{ins}}](b)\right]([\mathit{ins}](\sigma))$$

Lors de certaines affectations non-inversibles, il arrivera qu'aucune borne de remplacement ne puisse être trouvée. Dans ce cas, on n'aura d'autre choix que de supprimer la borne et par suite de supprimer le sommet auquel elle était attachée. Comme pour les opérateurs de création de bornes, les opérateurs sémantiques seront donc partiels.

5.4 Définition

Un diagramme de tranche est un graphe dont les arcs représentent des tranches d'un tableau fixé et les sommets représentent leurs bornes. Il est paramétré par un domaine de bornes définis-

sant l'ensemble $\mathcal{B}$ des bornes utilisables, une relation $\preceq$ sur cet ensemble, la fonction $\dot{\gamma}$ ainsi que les autres opérateurs décrits dans la section précédente.

Définition 13 (Diagramme de tranches)**.**
Un diagramme de tranches D sur $\mathcal{B}, \preceq$ est la donnée d'un couple (B, T) formé d'un ensemble de bornes

$$B \subseteq \mathcal{B}$$

et d'une fonction partielle

$$T : B^2 \dashrightarrow \mathcal{S} \cup \{\top, \bot\}$$

définissant les tranches du diagramme et associant à chaque couple de bornes (b_1, b_2)
- ou bien une variable $s \in \mathcal{S}$ si la tranche (b_1, b_2) doit être associée à s
- ou bien $\top$ si la tranche (b_1, b_2) ne doit être associée à aucune variable
- ou bien $\bot$ si la tranche (b_1, b_2) est vide

et vérifiant

$$\begin{aligned} \forall b_1, b_2 \in B^2,\ (b_1, b_2) \in \mathrm{dom}(T) \ &\Rightarrow\ b_1 \preceq b_2 \\ \wedge\ T(b_1, b_2) = \bot \ &\Rightarrow\ b_1 \equiv b_2 \end{aligned}$$

Un diagramme, s'il est associé à un tableau donné A, représente une fragmentation abstraite dont l'interprétation est donnée par la fonction de concrétisation $\widehat{\gamma}$:

$$\widehat{\gamma}(B, T) : \sigma \mapsto \left\{ r \;\middle|\; \begin{array}{l} \forall s \in \mathcal{S}, \\ \quad r(s) \text{ indéfini } \vee \\ \quad \exists (b_1, b_2) \in B^2, \exists \ell, \\ \qquad r(s) = A[\ell] \wedge \\ \qquad T(b_1, b_2) = s \wedge \\ \qquad \dot{\gamma}(b_1) \leq \ell < \dot{\gamma}(b_2) \end{array} \right\}$$

5.5 Domaine de fragmentation des diagrammes de tranches

Dans cette section, nous allons nous employer à définir les différentes opérations constituant un domaine de fragmentation. A chaque tableau du programme, on associe son propre diagramme. On peut ainsi avoir une fragmentation distincte pour chaque tableau. Pour qu'un diagramme décrive effectivement une fragmentation, on va associer des variables de synthèse aux arcs.

La construction du domaine de fragmentation consiste en l'élaboration d'une succession d'algorithmes. Nous commencerons par introduire des algorithmes de manipulation des diagrammes qui serviront de matériau pour l'élaboration des algorithmes du domaine. En général, nous ne chercherons pas l'optimalité : la plupart des algorithmes présentés ici peuvent être améliorés. Néanmoins, ils ont tous une complexité polynomiale.

Dans cette section, nous ne nous préoccuperons pas encore des critères de fragmentation qui pourront alors être choisis librement. Nous traiterons de l'application aux diagrammes de notre

critère de fragmentation dans la section 5.6 où nous verrons que la somme de travail supplémentaire à fournir sera faible. Nous chercherons donc à obtenir des fragmentations aussi fines que possible. L'application du critère de fragmentation pourra alors librement choisir où il faut rendre la fragmentation plus grossière et où il faut garder la précision obtenue.

Pour des raisons de simplicité, deux arcs différents devront forcément être associés à deux variables de synthèse distinctes. On peut vouloir supprimer cette contrainte, pour réduire le nombre de variables de synthèse et augmenter l'efficacité de l'analyse. Cela sera toujours possible en prenant certaines précautions. Si deux arcs sont associés à la même variable et qu'on a besoin de procéder à une opération différente sur chacun de ces deux arcs, il faudra au préalable décomposer ou fissionner la variable. Ainsi, on pourra, au moins temporairement, réaffecter deux variables de synthèse distinctes aux deux arcs.

5.5.1 Type abstrait de données

La fragmentation peut être décrite comme une liste associative. A chaque symbole de tableau on associe un diagramme. Nos algorithmes ne travaillent jamais sur la liste entière de diagrammes et sont appliqués individuellement sur chaque tableau. Ils prendront en entrée des couples (B, T) représentant des diagrammes.

Nos algorithmes sur les diagrammes utiliseront essentiellement des itérations sur les sommets ainsi que sur les arcs adjacents à ces sommets. Il faut donc que nous disposions de ces itérations sur B et T. En outre, il faudra pouvoir décider de la présence d'une borne dans B et d'un arc dans T. Pour un diagramme $D = (B, T)$, on utilisera les notations raccourcies $b \in D$ et $(b_1, b_2) \in D$ pour ces tests.

$$(b_1, b_2) \in D \;\Leftrightarrow\; (b_1, b_2) \in Dom(T)$$
$$b \in D \;\Leftrightarrow\; b \in B$$

En outre, on doit pouvoir ajouter et supprimer des sommets ou arcs à ces deux ensembles. On notera

$$T(b_1, b_2) \;\twoheadleftarrow$$

le retrait de l'arc (b_1, b_2).

Enfin, les algorithmes peuvent prendre en entrée une relation $\preceq$ définie sur $\mathcal{B}$. Il doit être possible de tester la relation sur deux éléments de $\mathcal{B}$, ainsi que d'énumérer les éléments de B en relation avec une borne $b \in \mathcal{B}$ donnée.

5.5.2 Transformations élémentaires

Nous décomposerons toute transformation des diagrammes en une suite de transformations élémentaires. Celles-ci serviront de briques algorithmiques pour construire les opérations que nous devons implémenter dans le domaine de fragmentation.

À chaque transformation élémentaire d'un diagramme, la fragmentation obtenue par concrétisation du diagramme change également. Il faut informer le domaine abstrait des transformations qui s'opèrent de sorte qu'il puisse en répercuter correctement les conséquences. À chaque transformation élémentaire des diagrammes, on associe une ou plusieurs transformations élémentaires de la fragmentation (Cf. section 4.2.4). Ce sont ces dernières qui sont interprétées par le domaine abstrait.

Nous devons prouver que les transformations élémentaires des diagrammes doivent être correctes. Cette preuve consiste à vérifier que ces transformations que l'on indique au domaine abstrait sont bien celles que l'on applique sur les diagrammes. Supposons que l'on parte d'un diagramme D_0 et qu'on lui applique une transformation élémentaire $\widehat{t}$ dont le résultat est un diagramme D_1. Notons F_0 la fragmentation concrète correspondant au diagramme initial, et F_1 celle correspondant au diagramme final, c'est à dire $F_0 = \widehat{\gamma}(D_0)$, et $F_1 = \widehat{\gamma}(D_1)$. L'algorithme opérant la transformation $\widehat{t}$ doit générer une suite de transformations élémentaires t sur F_0 produisant une fragmentation F'. Pour que l'algorithme soit correct, il faut que F' soit plus grossière que F_1, c'est à dire $F' \supseteq F_1$. Ainsi, le domaine abstrait assurera que la valeur abstraite décrit des états de synthèses valides pour les choix de représentants appartenant à F' et donc a fortiori appartenant à F_1.

$$\begin{array}{ccc} D_0 & \xrightarrow{\quad\widehat{t}\quad} & D_1 \\ \Big\downarrow{\scriptstyle \widehat{\gamma}} & & \Big\downarrow{\scriptstyle \widehat{\gamma}} \\ F_0 & \xrightarrow[t]{} & F' \supseteq F_1 \end{array}$$

Les transformations élémentaires des diagrammes forment une couche au sens logiciel. Elles seront les seules à transformer directement les diagrammes. Ce sont donc les seules dont nous avons besoin de prouver la correction. Les transformations plus complexes bâties à partir de celles-là seront à leur tour correctes.

Nous définissons quatre transformations élémentaires des diagrammes, la composition, la décomposition, la fission et la fusion, basées sur les transformations élémentaires des fragmentations du même nom.

Composition et décomposition

Dans les diagrammes, la composition aura toujours lieu entre deux tranches adjacentes et disjointes. En termes de graphe, cela signifie que les tranches à composer sont représentées par deux arcs adjacents. On peut donc désigner une composition par deux arcs dont le sommet destination de l'un est le sommet origine de l'autre, ou bien par la donnée de trois sommets : le sommet origine, le sommet destination tous deux définissant la cible de la composition, et un sommet intermédiaire permettant de caractériser les deux tranches à composer. La figure 5.1 illustre la composition et la décomposition.

La composition se contente de composer les variables de synthèse tout en modifiant le diagramme pour refléter l'effet de la composition, c'est à dire en ajoutant un nouvel arc correspondant à la tranche composée. Si l'arc existe déjà, on fusionne le résultat de la composition avec la tranche existante. L'algorithme permettant l'ajout ou la fusion de l'arc est donné figure 5.2 et se contente d'énumérer les cas possibles. L'algorithme de composition est donné quant à lui figure 5.3.

La décomposition a les mêmes entrées que la composition. A partir des trois sommets i, j, k, elle décompose l'arc (i, k) en les arcs (i, j) et (j, k). Lorsque les arcs cibles de la décomposition existent déjà, on fusionnera les tranches résultat avec les tranches existantes. L'algorithme est donné figure 5.4.

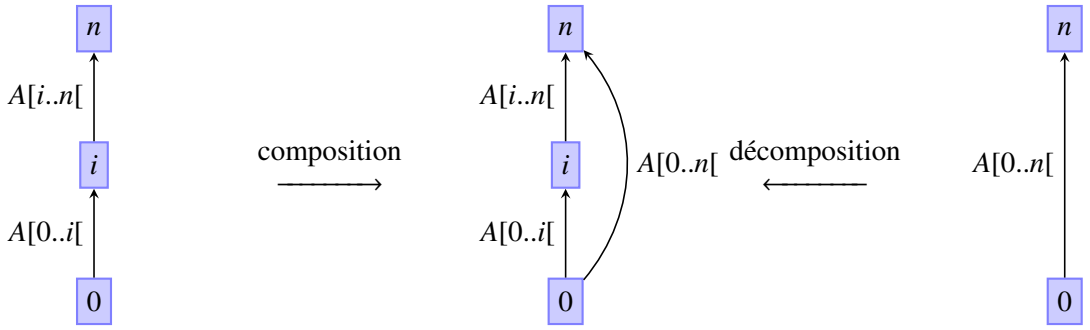


Figure 5.1 : Depuis le diagramme de gauche on compose $A[1..i[$ et $A[i..n[$ en $A[1..n[$ afin d'obtenir le diagramme central. Tandis que depuis le diagramme de droite on décompose $A[1..n[$ en $A[1..i[$ et $A[i..n[$ pour obtenir la même chose.

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq)$, $b_1, b_2 \in D$ tels que $b_1 \preceq b_2$ et $s \in \mathcal{S} \cup \{\top, \bot\}$
Si $T(b_1, b_2)$ *est indéfini* **alors**
  $T(b_1, b_2) \leftarrow s$
sinon
  Selon *la valeur de* s
    Cas $\bot$
      Si $T(b_1, b_2) \in \mathcal{S}$ **alors**
        Supprimer la variable $T(b_1, b_2)$
      $T(b_1, b_2) \leftarrow \bot$
    Cas $\top$
      Ne rien faire
    Cas $s \in \mathcal{S}$
      Selon *la valeur de* $T(i, j)$
        Cas $\bot$
          Supprimer la variable s
        Cas $\top$
          $T(b_1, b_2) \leftarrow s$
        Cas $t \in \mathcal{S}$
          Fusionner s dans t
          Supprimer la variable s

Figure 5.2 : Assignation d'une nouvelle valeur à un arc dans un diagramme.

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq)$ et $i, j, k \in D$ tels que $(i, j) \in D$ et $(j, k) \in D$
Selon *la valeur de* $T(i, j), T(j, k)$

- **Cas** $(\top, x)$ *ou* $(x, \top)$
  - Assigner $\top$ à l'arc (i, k)
- **Cas** $(\bot, \bot)$
  - Assigner $\bot$ à l'arc (i, k)
- **Cas** $(\bot, s)$ *ou* $(s, \bot)$ *avec* $s \in \mathcal{S}$
  - Choisir une nouvelle variable de synthèse t
  - Fissionner s en t
  - Assigner t à l'arc (i, k)
- **Cas** $(s_1, s_2) \in \mathcal{S}^2$
  - Choisir une nouvelle variable de synthèse t
  - Composer s_1 et s_2 dans t
  - Assigner t à l'arc (i, k)

FIGURE 5.3 : Algorithme de composition des arcs (i, j) et (j, k) en (i, k).

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq)$ et $i, j, k \in D$ tels que $(i, k) \in D$ et $i \preceq j \preceq k$.
Selon *la valeur de* $T(i, k)$

- **Cas** $\top$
  - Assigner $\top$ à l'arc (i, j)
  - Assigner $\top$ à l'arc (j, k)
- **Cas** $\bot$
  - Assigner $\bot$ à l'arc (i, j)
  - Assigner $\bot$ à l'arc (j, k)
- **Cas** $s \in \mathcal{S}$
  - Choisir deux nouvelles variables de synthèse t_1 et t_2
  - Décomposer s en t_1 et t_2
  - Assigner t_1 à l'arc (i, j)
  - Assigner t_2 à l'arc (j, k)

FIGURE 5.4 : Algorithme de décomposition de l'arc (i, k) en les arcs (i, j) et (j, k).

Proposition 3.
La composition et la décomposition sont correctes.

Démonstration. Il faut prouver que l'application des différentes compositions/décompositions de variables de synthèse transforment la fragmentation concrète en une fragmentation plus grossière que celle associée au diagramme résultat.

Nous donnons la preuve dans les cas où les arcs à composer et à décomposer ne sont associés ni à $\top$ ni à $\bot$. Dans ces cas, la composition et la décomposition ne modifient pas la fragmentation autrement que par des éventuelles fissions et fusion, et la preuve consiste à passer tous les cas en revue.

Nous utiliserons la définition de la concrétisation d'un diagramme donnée en section 5.4. La définition de la composition et de la décomposition dans une fragmentation sont donnés en section 4.2.4. Notons D_0 le diagramme initial et D_1 le diagramme obtenu par application de l'algorithme et $F_0 = \widehat{\gamma}(D_0)$, $F_1 = \widehat{\gamma}(D_1)$. Notons F' la fragmentation obtenue par application des transformations élémentaires. Nous devons prouver $F_1 \subseteq F'$.

Nous commençons la preuve pour traiter les cas où aucune fusion n'est à réaliser.

Prouvons la correction de la composition. Soit un choix de représentants $r \in F_1$, nous devons prouver $r \in F'$. Par définition le composition, il faut trouver r' et r'' dans F_0 coïncidant sur les variables autres que s_1 et s_2, c'est à dire

$$\forall v \neq s_1, s_2,\ r'(v) = r''(v) \tag{5.1}$$

et tels que r est égal à r' sur toutes les variables sauf t et $r(t)$ est égal ou bien à $r''(s_1)$ ou bien à $r''(s_2)$:

$$\forall v \neq t,\ r(v) = r'(v)\ \wedge \Big(r(t) = r''(s_1) \vee r(t) = r''(s_2)\Big) \tag{5.2}$$

On prendra r' comme la restriction de r à son domaine privé de t de sorte qu'elle vérifie trivialement 5.2 et que l'appartenance de r' à F_0 se vérifie aisément. Pour r'', il faut distinguer les deux cas induits par la définition de la concrétisation.

- Ou bien $r(t)$ est indéfini. On prend pour r'' la restriction de r à son domaine privé de s_1 (choix arbitraire, s_2 convenant aussi) et de t.
- Ou bien $r(t)$ est défini et il existe un ℓ tel que $r(t) = A[\ell]$ et vérifiant $\dot{\gamma}(i) \leq \ell < \dot{\gamma}(k)$. Suivant que $\ell \geq \dot{\gamma}(j)$ ou $\ell < \dot{\gamma}(j)$, t vient de la première ou la seconde tranche. Dans le premier cas on prend

  $$r'' = \begin{cases} v \mapsto r(v) \text{ si } v \neq s_1, t \\ s_1 \mapsto r(t) \\ \text{indéfini en } t \end{cases}$$

  et dans le second

  $$r'' = \begin{cases} v \mapsto r(v) \text{ si } v \neq s_2, t \\ s_2 \mapsto r(t) \\ \text{indéfini en } t \end{cases}$$

Dans les deux cas, on peut vérifier 5.1 et 5.2 et par conséquent que la composition est correcte lorsqu'il n'y a pas de fusion.

Prouvons la décomposition, en employant la même structure de raisonnement. Soit un choix de représentants $r \in F_1$, nous devons prouver $r \in F'$. Par définition de la décomposition, il faut trouver r', r_1 et r_2 coïncidant sur les variables autres que s, c'est à dire

$$\forall v \neq s \; r'(v) = r_1(v) = r_2(v) \tag{5.3}$$

et tels que r est égal à r' sur toutes les variables sauf t_1 et t_2 et $r(t_1) = r_1(s)$ et $r(t_2) = r_2(s)$:

$$\forall v \neq t_1, t_2, \; r(v) = r'(v) \; \wedge \; \Big(r(t_1) = r_1(s) \wedge r(t_2) = r_2(s)\Big) \tag{5.4}$$

On choisit r' comme la restriction de r à son domaine privé des variables t_1, t_2. On choisit r_1 (resp. r_2) suivant les cas.

- Ou bien $r(t_1)$ (resp. $r(t_2)$) est indéfini. On prend pour r_1 (resp. r_2) la restriction de r à son domaine privé de s, de t_1 et de t_2.
- Ou bien $r(t_1) = A[\ell]$ (resp. $r(t_2) = A[\ell]$) et vérifiant $\dot\gamma(i) \leq \ell < \dot\gamma(j)$ (resp. $\dot\gamma(j) \leq \ell < \dot\gamma(k)$) et donc a fortiori vérifiant $\dot\gamma(i) \leq \ell < \dot\gamma(k)$. On peut alors prendre

$$r_1 = \begin{cases} v \mapsto r(v) \text{ si } v \neq s, t_1, t_2 \\ s \mapsto r(t_1) \\ \text{indéfini en } t_1, t_2 \end{cases} \qquad \left(\text{resp. } r_2 = \begin{cases} v \mapsto r(v) \text{ si } v \neq s, t_1, t_2 \\ s \mapsto r(t_2) \\ \text{indéfini en } t_1, t_2 \end{cases} \right)$$

Dans les deux cas, on peut vérifier 5.3 et 5.4 et par conséquent que la décomposition est correcte lorsqu'il n'y a pas de fusion.

Dans le cas où des fusions doivent avoir lieu, il nous faut vérifier la validité de la condition de fusion. Celle-ci est possible dans une fragmentation F si pour tout choix de représentant $r \in F$ il existe un autre choix de représentant r' tel que

$$\begin{cases} \forall v \neq s, t & r'(v) = r(v) \\ & r'(s) = r(t) \end{cases}$$

Or nous venons de prouver qu'après la composition (resp. la décomposition) la variable t (resp t_1, t_2) pouvait être associée à toutes les cellules de la tranche $A[i..k[$. (resp. $A[i..j[$, $A[j..k[$) Par conséquent, pour tout choix de représentant r, $r(t)$ (resp $r(t_1)$, (t_2)) appartient à cette tranche, et on peut donc trouver un r' identique partout à ceci près que $r'(T(i,k)) = r(s)$. (resp. $r'(T(i,j)) = r(t_1)$, $r'(T(j,k)) = r(t_1)$) □

Fission et fusion

La fission et la fusion surviennent respectivement quand deux tranches deviennent distinctes ou identiques. Par exemple, si on a $i = j$, la tranche $A[1..i]$ peut aussi bien s'écrire $A[1..j]$. Tant que l'égalité entre les indices est conservée, on peut choisir l'une ou l'autre des expressions indifféremment. Mais on peut perdre cette égalité, dans deux situations : élargissement de la valeur abstraite ou bien union avec une valeur abstraite n'ayant pas cette égalité. Avant de perdre l'égalité, il est possible de dupliquer la tranche et lui donner les deux expressions qui deviendront bien distinctes $A[1..i]$ et $A[1..j]$. Dans ces deux cas, nous n'avons pas a priori de raison de choisir l'une ou l'autre des expressions. C'est pourquoi nous voudrons en général garder les deux. À l'inverse, si on retrouve l'information $i = j$ grâce à une garde on peut fusionner à nouveau les deux tranches et les bornes.

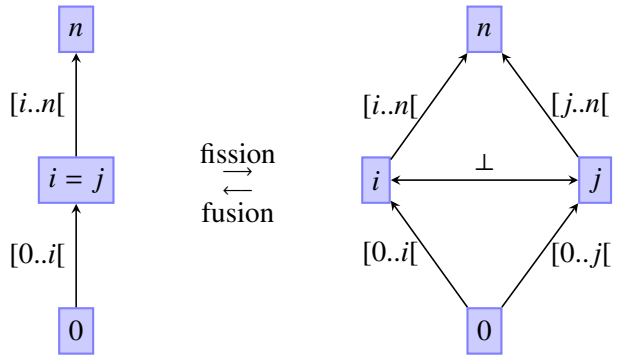


Figure 5.5 : On obtient le diagramme de droite à partir de celui de gauche en fissionnant le sommet i en les sommets i et j. Inversement, on obtient le diagramme de gauche à partir de celui de droite en fusionnant les sommets i et j en le sommet i.

Sur les diagrammes, les cas précédents se traduisent par une duplication d'arc pour la duplication de tranche, et par une duplication de sommet pour la duplication de borne. La fission d'une borne nécessite l'ajout dans le diagramme des sommets associés aux bornes produites de la fission. Puis pour chaque arc allant vers ou partant du sommet initial, on crée un des arcs allant vers ou partant de chacun des sommets dupliqués. Enfin, on ajoute également des arcs entre toute paire de sommets fissionnés en les associant à $\bot$. Ces arcs représentent effectivement tous des fragments vides. Les ajouter brise l'acyclicité du graphe. La fusion à l'inverse supprimera tout arc entre les sommets fusionnés et fusionnera les arcs entrants et sortants. La figure 5.5 illustre la fission et la fusion, la figure 5.6 donne l'algorithme de la fission et la figure 5.7 celui de la fusion.

En général, la fission génèrera plus d'arcs que nécessaire. Mais seul l'algorithme utilisant la fission et la fusion sait lesquels sont intéressants : à ce niveau d'abstraction, nous n'avons pas de moyen de savoir lesquels conserver. Certains arcs ne serviront qu'à réaliser des compositions, d'autres seront conservés tels quel, et d'autres devront être supprimés. Une implémentation efficace voudra éviter de générer trop de fissions de tranches qui peuvent être coûteuses dans les domaines abstraits et utiliser une évaluation paresseuse de la fission.

5.5.3 Clôture et réduction

Rappelons que si on tient à ce que les diagrammes de tranches soient transitivement réduits c'est parce que si un arc du diagramme peut s'obtenir par transitivité c'est que la tranche associée à cet arc peut s'obtenir par composition. Or un fragment qui peut être obtenu par composition n'apporte aucune précision supplémentaire à la fragmentation : ce que l'on sait sur la tranche composée, on le sait déjà sur les tranches composantes.

Cependant il est utile de temporairement oublier cette contrainte et de considérer des graphes qui ne sont pas transitivement réduits. Après une composition ou une décomposition, le diagramme n'est jamais réduit, comme le montrait la figure 5.1.

Quoi qu'il en soit, si après avoir terminé de travailler avec un diagramme on se retrouve dans une situation où le graphe n'est pas réduit, on cherchera à retirer les arcs en trop. Pour le faire sans perdre d'information, il faudra auparavant être sûr que toute l'information acquise sur la variable associée à l'arc que l'on retire ne sera pas perdue, c'est à dire que toute information sur la tranche composée est bien déjà contenue dans les tranches composantes.

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq)$, $b \in B$ et $B' \subseteq \mathcal{B}$ un ensemble de bornes équivalentes pour $\preceq$ vérifiant $B \cap B' = \{b\}$

$B \leftarrow B \cup B'$

Pour chaque $b' \in B'$ **faire**

- **Pour chaque** $(b, b_s) \in D$ **faire**
  - **Si** $T(b, b_s)$ *est* $\top$ *ou* $\bot$ **alors**
    - $T(b', b_s) \leftarrow T(b, b_s)$
  - **sinon**
    - Choisir une nouvelle variable de synthèse t
    - Fission de $T(b, b_s)$ en t
    - $T(b', b_s) \leftarrow t$
- **Pour chaque** $(b_p, b) \in D$ **faire**
  - **Si** $T(b_p, b)$ *est* $\top$ *ou* $\bot$ **alors**
    - $T(b_p, b') \leftarrow T(b_p, b)$
  - **sinon**
    - Choisir une nouvelle variable de synthèse t
    - Fission de $T(b_p, b)$ en t
    - $T(b_p, b') \leftarrow t$

Pour chaque $b_1, b_2 \in B'^2$ **faire**

- $T(b_1, b_2) \leftarrow \bot$

FIGURE 5.6 : Algorithme de fission d'un sommet b en l'ensemble des sommets B'.

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq)$, $B' \subseteq B$ un ensemble de bornes équivalentes pour $\preceq$ et $b \in B'$

Pour chaque $b' \in B'/\{b\}$ **faire**

- **Pour chaque** $(b', b_s) \in D$ **faire**
  - **Si** $T(b', b_s)$ *n'est ni* $\top$ *ni* $\bot$ **alors**
    - **Si** $T(b, b_s)$ *est défini et n'est ni* $\top$ *ni* $\bot$ **alors**
      - Fusion de $T(b', b_s)$ dans $T(b, b_s)$
      - Supression de $T(b', b_s)$
    - **sinon**
      - $T(b, b_s) \leftarrow T(b', b_s)$
  - $T(b', b_s) \twoheadleftarrow$
- **Pour chaque** $(b_p, b') \in D$ **faire**
  - **Si** $T(b_p, b')$ *n'est ni* $\top$ *ni* $\bot$ **alors**
    - **Si** $T(b_p, b)$ *est défini et n'est ni* $\top$ *ni* $\bot$ **alors**
      - Fusion de $T(b_p, b')$ dans $T(b_p, b)$
      - Supression de $T(b_p, b')$
    - **sinon**
      - $T(b_p, b) \leftarrow T(b_p, b')$
  - $T(b_p, b') \twoheadleftarrow$

Pour chaque $b_1, b_2 \in B'^2$ **faire**

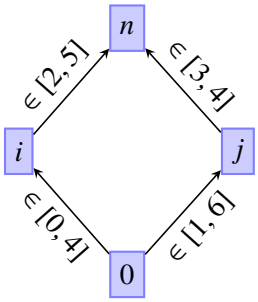
- **Si** $T(b_1, b_2)$ *n'est ni* $\top$ *ni* $\bot$ **alors**
  - Supression de $T(b_1, b_2)$
- $T(b_1, b_2) \twoheadleftarrow$

$B \leftarrow B/(B'/\{b\})$

Figure 5.7 : Algorithme de fusion d'un ensemble de sommets B' en un sommet b.

Dans cette section, nous définirons deux opérations. La première est une généralisation de la composition et permet de construire un arc dans le graphe qui peut être obtenu par transitivité. Durant cette construction, on tâchera de donner à la nouvelle tranche le maximum d'information connue à son sujet. La second est l'opération inverse. Elle consiste à déconstruire un arc en le retirant du graphe tout en récupérant toute l'information accumulée à son propos.

Avant de détailler ces deux opérations et afin de mieux les comprendre il faut observer le lien qu'elles ont avec le problème de la réduction des valeurs abstraites. Les domaines abstraits ont déjà leurs propres réductions. Mais un domaine abstrait n'a aucune connaissance des fragmentations et ne peut être capable de dériver des propriétés impliquées par les singularités d'un diagramme. Considérons par exemple le diagramme suivant.



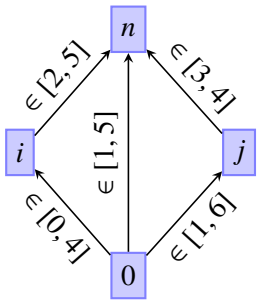
Nous utiliserons ce diagramme en conjonction avec le domaine abstrait des intervalles. A coté de chacun des arcs figure un intervalle de valeurs possibles pour les cellules de chacune des tranches. Ce diagramme représente une caractéristique souvent rencontrée : il y a différentes manières de construire une même tranche. Ici, la tranche $[0..n[$ peut être produite par la composition de $[0..i[$ et $[i..n[$ aussi bien que par la composition de $[0..j[$ et $[j..n[$. Si on pose la question « que sait t-on de $[0..n[$ » on pourra répondre en utilisant aussi bien l'une que l'autre des possibilités.

Construisons la tranche $[0..n[$ dans ce diagramme. On compose d'abord les tranches $[0..i[$ et $[i..n[$ du premier chemin en $[0..n[$. L'algorithme de composition va informer le domaine abstrait que l'on veut une nouvelle variable dont les propriétés sont celles vérifiées par ces deux tranches à la fois. On fait la même chose pour les tranches $[0..j[$ et $[j..n[$ du second chemin. Mais cette fois ci, puisque il y a déjà une tranche $[0..n[$, l'algorithme de composition va devoir fusionner le résultat des deux compositions. La fusion va permettre de dire au domaine abstrait que la tranche $[0..n[$ possède à la fois les propriétés de la première et de la seconde composition.

Pour le domaine abstrait, l'intervalle pour la tranche $[0..n[$ se calculera avec des unions pour les compositions et une intersection pour la fusion :

$$([0,4] \sqcup [2,5]) \sqcap ([1,6] \sqcup [3,4]) = [0,5] \sqcap [1,6] = [1,5]$$

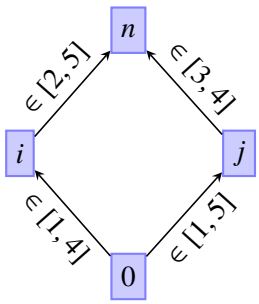
Dans notre exemple, l'intervalle obtenu pour la tranche $[0..n[$ est optimal. En outre, l'intervalle obtenu peut servir à renforcer celui des quatre tranches originales. En effet, chacune de ces quatre tranches est incluse dans $[0..n[$ et hérite donc des propriétés calculées sur $[0..n[$.



Pour signifier au domaine abstrait qu'il doit renforcer les propriété des quatre tranches avec la tranche nouvellement construite, on utilise la seconde opération, la déconstruction. On va déconstruire la tranche $[0..n[$, c'est à dire la retirer du diagramme tout en transférant ses propriétés aux tranches qui la composent.

Pour déconstruire on suit l'exact procédé inverse de celui utilisé par la construction. Il y avait deux chemins pour construire $[0..n[$, il y a les deux même chemins pour la déconstruire. On commence donc par fissionner la tranche $[0..n[$ en deux pour obtenir deux tranches identiques. Puis on décompose chacune de ces deux tranches le long des deux chemins. Enfin, on peut fusionner les produit de la décomposition avec les tranches originales. Les deux premières opérations permettent de demander au domaine abstrait quatre variables correspondant aux tranches originales mais avec les propriétés de la tranche $[0..n[$. La fusion quant à elle permet effectivement de transmettre cette propriété aux tranches originales.

Les intervalles des tranches $[0..i[$ et $[j..n[$ deviennent respectivement $[1, 4]$ et $[1, 5]$ tandis que tranches $[i..n[$ et $[0..j[$ restent inchangées.



En utilisant la construction et la déconstruction on peut partiellement calculer une réduction de la valeur abstraite. Nous donnerons ici une version peu efficace des deux algorithmes, puis nous proposerons des pistes d'optimisations possibles.

Clôture

La construction est une opération qui consiste à ajouter un arc entre deux sommets lorsqu'il existe au moins un chemin entre ces deux sommets. Pour le moment, réaliser cette construction ne sera pas différent de construire tous les arcs obtenus par transitivité, c'est à dire de calculer la clôture transitive du diagramme.

Pour récupérer un maximum d'information il sera nécessaire que tout chemin permettant d'obtenir un arc à construire soit pris en compte. Le nombre de chemins entre deux sommets étant généralement exponentiel, il faudra réutiliser une astuce similaire à celle utilisée dans les

Entrées : $D = (B, T)$ un diagramme sur $\mathcal{B}, \preceq$
Pour *n de* 1 *à* $|B|$ **faire**
  Pour $(i, j) \in D$ *et* $(j, k) \in D$ **faire**
    Composer les arcs (i, j) et (j, k) en (i, k)

FIGURE 5.8 : Algorithme de clôture.

algorithmes de recherche de chemin dans un graphe qui permet de ne pas considérer plusieurs fois le même sous chemin.

L'algorithme de base est similaire à celui du calcul du nombre de chemins dans un graphe par exponentiation de la matrice d'adjacence. Soit A la matrice d'adjacence du graphe, $A_{i,j}$ le coefficient sur la i-ème ligne et la j-ème colonne contient 1 s'il y a un arc (i, j) dans le graphe, 0 sinon. Alors A^n est la matrice telle que le $A^n_{i,j}$ est le nombre de chemins distincts et de longueur exactement n allant de i à j dans le graphe. Le calcul de A^{n+1} à partir de A^n,

$$A^{n+1}_{i,j} = (A^{n+1} \times A)_{i,j} = \Sigma_k A^n_{i,k} \times A_{k,j}$$

s'interprète en termes de graphes : pour chaque sommet k, les chemins de longueur $n + 1$ allant de i à j en ayant k comme avant dernier sommet sont les chemins de longueur n allant de i à k augmentés de l'arc (k, j) s'il existe. En outre, le nombre de chemins de longueur $n + 1$ est égal à la somme pour tous les sommets k du nombre de ces chemins.

Nous allons utiliser une méthode similaire, à ceci près qu'au lieu de compter les chemins, nous allons les construire. A chaque itération de l'algorithme, nous allons construire un nouveau diagramme dont les tranches sont les composées d'une tranche du diagramme original, et d'une tranche de l'itération précédente. Autrement dit, à l'itération n, on obtiendra toutes les tranches produites par composition de n tranches. Sur un diagramme sans cycle, le chemin le plus long ne peut excéder en longueur le nombre de sommets. La figure 5.8 donne cet algorithme.

L'algorithme est inefficace car il va calculer plusieurs fois les mêmes compositions. A l'itération n il va non seulement calculer les compositions de n tranches, mais également celles de moins de n tranches, et donc refaire le travail des itérations précédentes. Sans optimisation, la complexité de cet algorithme est $O(|B|^4)$, la boucle interne itérant au pire pour chaque triplet de sommet du graphe.

Réduction

La réduction est l'exact inverse de la clôture. Chaque arc (i, k) peut potentiellement être décomposé en deux arcs (i, j) et (j, k), si j vérifie $i \preceq j \preceq k$.

Construction et déconstruction

Les algorithmes présentés construisent et déconstruisent tous les arcs de la clôture transitive d'un graphe. Comme annoncé en introduction de cette section, on aura besoin par la suite de deux cas particulier de ces algorithmes : construire et détruire un unique arc. Ces problèmes ne sont pas plus simples : pour construire ou déconstruire un arc, il faudra toujours considérer tous les arcs intermédiaires.

Entrées : $D = (B, T, \preceq)$ un diagramme
Pour n *de* 1 *à* $|B|$ **faire**
  Pour $i, j, k \in B$ *tels que* $i \prec j \prec k$ **faire**
    Décomposer les arcs (i, k) en (i, j) et (j, k)

Figure 5.9 : Algorithme de réduction.

Pour passer de la clôture et de la réduction à la construction et déconstruction, il faut modifier les algorithmes en deux points.

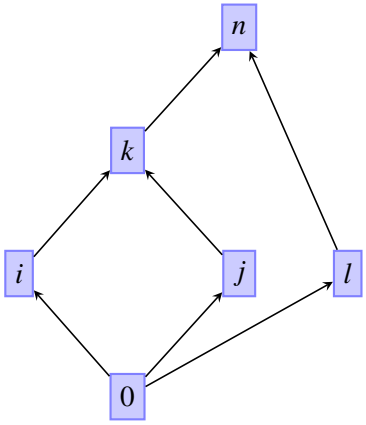
1. D'abord, on peut ignorer tout sommet et arc du graphe qui ne soit pas « entre » ces deux points. Autrement dit, dans les boucles interne des deux algorithmes, on se restreindra aux sommets i, j et k qui sont effectivement encadrés par les deux extrémités i_0,k_0 de l'arc à construire. Si on veut construire ou déconstruire l'arc (i_0, k_0) on prendra des triplets de sommets vérifiant $i_0 \preceq i \prec j \prec k \preceq k_0$.

2. Il faudra également supprimer toute construction intermédiaire utile à l'algorithme. La construction ne doit construire que l'arc demandé, tandis qu'on exigera de la déconstruction qu'elle réduise effectivement le diagramme entre les deux sommets de l'arc à déconstruire.

Optimisations

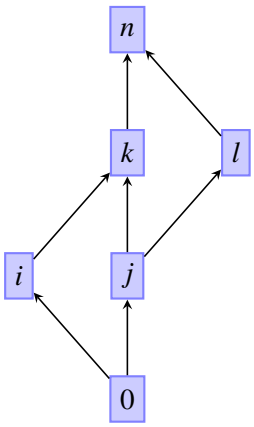
Il y a plusieurs améliorations possibles des algorithmes précédents. Par exemple, composer une fois pour toutes les arcs autour des sommets de degré entrant et sortant égal à 1. Ou bien, mémoriser les compositions déjà réalisées pour ne pas réaliser deux fois les même. Ou encore, ignorer les parties du graphe qui ne sont pas entre deux sommets où divergent et se rejoignent plusieurs chemins.

En effet, les précédents algorithmes effectuent plusieurs fois les mêmes compositions ou les mêmes décompositions. C'est parfois nécessaire : entre deux compositions identiques, on peut avoir acquis plus d'information sur les tranches composantes et il est alors nécessaire de reproduire une nouvelle fois la composition. Sur certains diagrammes en revanche, il est possible de trouver un ordre d'itération des compositions ou des décompositions, de sorte qu'il ne soit pas nécessaire de refaire deux fois la même opération. Considérons par exemple le diagramme suivant.



Dans ce diagramme, on peut construire la tranche $[0..k[$ par composition de $[0..i[$ et $[i..k[$ et par composition de $[0..j[$ et $[j...k[$. Ensuite, on peut construire $[0..n[$ par composition de $[0..k]$ et $[k..n[$ et par composition de $[0..l[$ et $[l..n[$. Toute autre composition est inutile. Cela est dû au fait qu'il y a une hiérarchie. On peut réduire le losange $0, i, j, k$ en l'arc (i, k). En effet, une fois l'arc i, k construit, il sera par la suite inutile de calculer toute composition avec les quatre tranches qui ont servi à sa composition. On obtient alors un autre losange $0, k, l, n$ qu'on peut réduire à son tour.

Tous les diagrammes ne sont pas hiérarchiques de cette manière. Le diagramme suivant par exemple, ne peut être vu comme une imbrication de losanges.



Néanmoins, supposons qu'un diagramme de cette forme est inclus dans un autre, connecté par un arc entrant sur la borne inférieure et un arc sortant sur la borne supérieure. Il est alors possible comme avec les losanges de commencer par clore cette partie du diagramme avant de clore le reste.

Ceci nous donne une idée générale d'optimisation à réaliser. On identifie la hiérarchie dans un diagramme, et on clôt les uns après les autres les différents niveaux de la hiérarchie. Pour construire cette hiérarchie, il faut identifier les différents intervalles. Ces intervalles seront définis par les paires de sommets constituant leur borne inférieure et supérieure et sont telles qu'aucun chemin ne sort ou n'entre dans la composante sans passer par ces bornes. De cette manière, on réduit significativement le nombre de transformations élémentaires à réaliser.

5.5.4 Ajout et retrait de sommets

Suivant le critère de fragmentation choisi, il faudra ajouter et retirer, à la demande du critère, des nouvelles bornes pour les tranches et les sommets correspondants. Il sera également nécessaire de rajouter des sommets pour unifier des diagrammes. Enfin, une affectation non-inversible à une variable d'indice va probablement invalider une borne, nécessitant son retrait du diagramme.

L'ajout et le retrait de sommet doivent préserver les contraintes imposées par la définition des diagrammes, et en particulier ne pas créer d'arc incompatible avec la relation d'ordre entre les bornes. En outre, il serait appréciable que le diagramme reste réduit s'il était réduit auparavant.

Ajout d'un sommet.

Pour ajouter un sommet, il faut d'abord trouver sa place dans le graphe. On compare la nouvelle borne à toutes les autres déjà présentes dans le diagramme. On va chercher à ce que le nouveau sommet puisse être accessible de tout autre sommet correspondant à un minorant, et qu'on puisse accéder depuis ce nouveau sommet à tout sommet correspondant à un majorant. Pour garder un diagramme aussi réduit que possible, on n'ajoutera pas un arc entre le nouveau sommet et tous les sommets comparables, mais seulement avec les sommets associés aux minorants maximaux et aux majorants minimaux. Si (B, T) est un diagramme et qu'on souhaite ajouter la borne b, alors, on calculera les deux ensembles de sommets

$$sup(b) = \{\, b_s \in B \;\middle|\; b \preceq b_s \wedge \nexists b' \in B/\{b_s\}, b \preceq b' \preceq b_s \,\}$$
$$inf(b) = \{\, b_i \in B \;\middle|\; b_i \preceq b \wedge \nexists b' \in B/\{b_i\}, b_i \preceq b' \preceq b \,\}$$

Le nouveau sommet devra être connecté par un arc entrant depuis chaque sommet de $inf(b)$ et par un arc sortant vers chaque sommet de $sup(b)$. Se pose maintenant la question de l'information qu'il faudra attacher à ces arcs. Les bornes de $inf(b)$ sont inférieures aux bornes de $sup(b)$. On peut donc chercher des chemins entre les sommets correspondant aux premières et ceux correspondant aux seconds. Ces chemins ne sont pas nécessairement réduits à un arc, car il peut y avoir entre $inf(b)$ et $sup(b)$ des sommets qui ne sont pas comparables à b.

Considérons d'abord le cas où $sup(b)$ et $inf(b)$ sont tous les deux non vides. Pour toute paire de sommets respectivement dans ces deux ensembles, ou bien il y a déjà un arc correspondant dans le diagramme, ou bien on peut le construire en utilisant l'algorithme de construction présenté en section 5.5.3. Notre nouveau sommet va alors venir décomposer cet arc en deux et on peut appliquer l'algorithme de décomposition pour créer les arcs composants. On répète ces deux opérations pour tout couple de sommet de $inf(b) \times sup(b)$ quitte à fusionner les résultats de la décomposition avec les arcs déjà construits.

Dans le cas où soit $sup(b) = \emptyset$ soit $inf(b) = \emptyset$, cela signifie qu'on n'a pas d'information à associer aux arcs venant ou sortant de b. On peut tout de même les créer en leur associant $\top$. L'algorithme général est décrit figure 5.10.

Retrait d'un sommet.

Le retrait d'un sommet présente moins de subtilités. Pour conserver l'information des arcs entrants et sortants du sommet à retirer, on se contente de les composer. Tous les arcs composants

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq)$ et $b \notin B$
$B \leftarrow B \cup \{b\}$
Si $inf(b) = \emptyset$ **alors**
  Pour $b_s \in sup(b)$ **faire**
    $T(b, b_s) \leftarrow \top$

Si $sup(b) = \emptyset$ **alors**
  Pour $b_i \in inf(b)$ **faire**
    $T(b_i, b) \leftarrow \top$

Pour $(b_i, b_s) \in inf(b) \times sup(b)$ **faire**
  Construire si nécessaire l'arc (b_i, b_s)
  Décomposer l'arc (b_i, b_s) en (b_i, b) et (b, b_s)
  Supprimer l'arc (b_i, b_s)

Figure 5.10 : Algorithme d'ajout de la borne b dans un diagramme.

peuvent ensuite être supprimés sans risques. Il sera parfois nécessaire de déconstruire l'arc composé si celui-ci peut être obtenu autrement par transitivité. L'algorithme, tout à fait symétrique à celui de l'ajout de sommet est donné figure 5.11.

Exemple 12.
Supposons qu'on ait un diagramme D composé des sommets $B = \{0, 1, i, i+1, n\}$ et la relation $\preceq$ définie par

$$0 \preceq i \preceq n - 1$$

ainsi que les tranches $[0]$, $[1..n[$, $[0..i[$ et $[i..n[$. Le diagramme réduit correspondant à ces données est représenté à gauche de la figure 5.12

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq)$ et $b \in B$
Pour $(b_i, b) \in D$ *et* $(b, b_s) \in D$ **faire**
  Composer les arcs (b_i, b) et (b, b_s) en (b_i, b_s)
  Déconstruire si nécessaire (b_i, b_s) dans le diagramme

Pour $(b_i, b) \in D$ **faire**
  Supprimer l'arc (b_i, b) et les éventuels symboles associés

Pour $(b, b_s) \in D$ **faire**
  Supprimer l'arc (b, b_s) et les éventuels symboles associés

$B \leftarrow B/\{b\}$

Figure 5.11 : Algorithme de supression d'une borne b dans un diagramme.

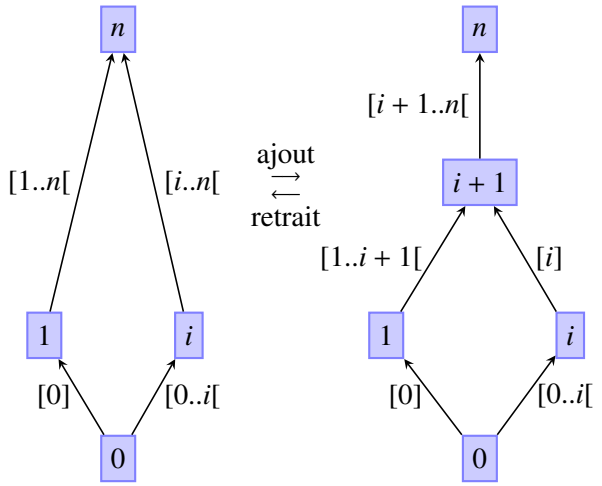


Figure 5.12 : En partant du diagramme de gauche, on ajoute le sommet $i + 1$ au diagramme et on décompose la tranche $[0..n[$ en $[0..i + 1[$ et $[i + 1..n[$ et la tranche $[i..n[$ en $[i..i + 1[$ et $[i + 1..n[$ pour obtenir le diagramme de droite. Inversement, on passe du diagramme de droite au diagramme de gauche en retirant le sommet $i + 1$ et en composant les arcs adjacents.

Ajoutons le sommet $i + 1$ au diagramme. Dans les sommets existants, un seul est supérieur à $i + 1$, et on a $sup(i + 1) = \{n\}$. Les trois autres sommets en revanche sont tous inférieurs à $i + 1$ mais seuls deux d'entre eux sont maximaux : 0 n'est pas maximal et on a $inf(i + 1) = \{1, i\}$. Les arcs (i, n) et $(1, n)$ sont déjà dans le diagramme et il est donc inutile de les construire. On décompose l'arc (i, n) en $(i, i + 1)$ et $(i + 1, n)$ et l'arc $(1, n)$ en $(1, i + 1)$ et $(i + 1, n)$. On obtient deux fois comme produit de la décomposition l'arc $(i + 1, n)$, on doit donc fusionner les deux tranches obtenues par la décomposition. Enfin, on peut supprimer les deux arcs décomposés.

L'arc $(i, i+1)$ hérite donc des propriétés de (i, n) tandis que l'arc $(1, i+1)$ hérite des propriétés de $(1, n)$. L'arc $(i + 1, n)$ en revanche hérite des deux et ses propriétés sont la conjonction des propriétés des deux arcs.

Si on veut revenir au diagramme initial en retirant le sommet $i + 1$, il suffit de joindre les arcs entrants aux arcs sortant du sommet. Ainsi, on composera les arcs $(1, i + 1)$ et $(i + 1, n)$ en l'arc $(1, n)$ et on composera les arcs $(i, i + 1)$ et $(i + 1, n)$ en l'arc (i, n). L'arc $(i + 1, n)$ est utilisé dans deux compositions différentes. Une fois qu'on a opéré les deux compositions, on peut sans crainte supprimer le sommet ainsi que les arcs incidents.

5.5.5 Transformations de diagrammes

La relation d'ordre entre les bornes est amenée à changer durant l'analyse. Elle peut être renforcée par des passages de gardes ou des intersections et elle peut être affaiblie par des unions ou des élargissements. Il faudra transformer les diagrammes en conséquence.

La forme canonique d'un diagramme est fixée par cette relation d'ordre. Il est toujours possible de recalculer complètement cette forme canonique du diagramme. On part de la clôture transitive, le graphe composé des sommets B et dans lequel on a un arc (b_1, b_2) si et seulement si $b_1 \preceq b_2$. Puis on applique un algorithme de réduction transitive [AGU72] sur ce graphe.

Lorsque la relation d'ordre change, il nous suffira alors d'appliquer un algorithme de transfor-

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq_1)$ et $\preceq_2$ une relation d'ordre telle que $\preceq_1 \supseteq \preceq_2$
Sorties : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq_2)$
Pour $i \in B$ **faire**
  $B' \leftarrow fission_{\preceq_1, \preceq_2}(b)$
  Fissionner b en B' dans D

Construire G le graphe canonique pour l'ensemble de sommets B et la relation d'ordre $\preceq_2$
Pour $(i, j) \in G$ **faire**
  Construire dans D l'arc (i, j)

Pour $(i, j) \in D$, $(i, j) \notin G$ **faire**
  Retirer l'arc (i, j) de D

Figure 5.13 : Algorithme de généralisation d'un diagramme.

mation permettant de passer de la forme actuelle du diagramme à sa forme canonique.

Nous décrirons deux algorithmes à utiliser suivant que la relation d'ordre s'affaiblit ou se renforce. L'ensemble des sommets ne variera pas durant la transformation, seul l'ensemble des tranches est affecté. Ces algorithmes requièrent ceux de la construction et déconstruction d'arc dans un diagramme. (cf. section 5.5.3)

Transformation vers un diagramme plus grossier

La transformation d'un diagramme vers un diagramme plus grossier implique que la relation d'ordre $\preceq_1$ du diagramme initial soit plus forte que la relation $\preceq_2$ du diagramme final.

$$\preceq_1 \supseteq \preceq_2$$

En conséquence chaque fois qu'un arc (i, j) est présent dans le diagramme final on a $i \preceq_2 j$ et on a donc également $i \preceq_1 j$. Cela ne signifie pas nécessairement que l'arc existe dans le diagramme initial, mais seulement qu'il y a un chemin entre i et j. On peut donc appliquer l'algorithme de construction pour ajouter cet arc par composition et fusion des arcs composants. En répétant cette construction pour tout arc du graphe canonique, on obtient un graphe dont l'ensemble des arcs inclut celui qu'on veut obtenir. Il ne restera alors plus qu'à retirer les arcs inutiles.

Avant d'ajouter et de retirer des arcs et puisque la relation d'ordre est affaiblie, certains sommets peuvent être fissionnés. Pour chaque sommet b, les bornes b' qui étaient équivalentes à b pour $\preceq_1$ ne sont plus nécessairement équivalentes pour $\preceq_2$. On fissionne b en les sommets

$$B' = fission_{\preceq_1, \preceq_2}(b)$$

grâce à l'algorithme de fission.

La figure 5.13 donne l'algorithme de généralisation d'un diagramme.

Transformation vers un diagramme plus fin

Le raffinement d'un diagramme est tout à fait symétrique au cas précédent. On commence par fusionner les sommets associés à des bornes équivalentes pour la nouvelle relation $\preceq_2$.

Entrées : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq_1)$ et $\preceq_2$ une relation d'ordre telle que $\preceq_1 \subseteq \preceq_2$
Sorties : $D = (B, T)$ un diagramme sur $(\mathcal{B}, \preceq_2)$
Pour $i \in B$ **faire**
  $B' \leftarrow \{ b' \mid b' \preceq_2 b \wedge b \preceq_2 b' \}$
  Fusionner B' en b dans D

Construire G le graphe canonique pour l'ensemble de sommets B et la relation d'ordre $\preceq_2$
Pour $(i, j) \in G$ **faire**
  Si $(i, j) \notin D$ **alors**
    Ajouter l'arc (i, j) dans D associé à une nouvelle variable

Pour $(i, j) \in D$, $(i, j) \notin G$ **faire**
  Déconstruire l'arc (i, j)
  Retirer l'arc (i, j)

Figure 5.14 : Algorithme de généralisation d'un diagramme.

Cette relation $\preceq_2$ est incluse dans la relation d'ordre initiale $\preceq_1$. Par conséquent, certains arcs du diagramme initial peuvent maintenant être décomposés. On utilisera pour cela l'algorithme de déconstruction. Pour que la déconstruction fonctionne, il faudra au préalable compléter le diagramme avec les nouveaux arcs, quitte à leur associer $\top$. La figure 5.14 donne cet algorithme.

5.5.6 Unification

L'unification de diagrammes a pour but de transformer deux diagrammes de sorte qu'ils soient finalement identiques. Ainsi, les fragmentations décrites doivent être en tout point pareilles et en particulier désigner les mêmes variables de synthèse. Une fois cette unification réalisée, il est possible pour les domaines abstraits d'utiliser leurs opérateurs binaires : l'union, l'élargissement et la comparaison. On ne cherchera pas à calculer une unification pouvant être utilisée avant le calcul d'une intersection.

N'importe quel diagramme unifié peut faire l'affaire pour peu que la transformation vers ce diagramme unifié soit correcte. On cherchera tout de même à conserver un maximum d'information durant la transformation et donc ne pas choisir un diagramme d'unification complètement arbitraire.

Unification de la relation d'ordre

Pour que les transformations des diagrammes soient correctes, nous n'avons d'autres choix que d'unifier vers un diagramme plus général. En effet, on peut toujours affaiblir la relation d'ordre entre les bornes, en prendre une sur-approximation, mais on ne peut pas la renforcer : les opérations sur les diagrammes sont correctes pourvu que l'abstraction de la relation d'ordre soit elle aussi correcte.

On prendra donc comme relation d'ordre pour le diagramme unifié, la plus forte relation d'ordre qui soit plus faible que les deux relations d'ordre $\preceq_1$ et $\preceq_2$ des diagrammes à unifier.

C'est à dire, on prendra la relation d'ordre :

$$\preceq = \preceq_1 \cap \preceq_2$$

et pour tout $i, j \in \mathcal{B}$

$$i \preceq j \;\Leftrightarrow\; i \preceq_1 j \wedge i \preceq_2 j$$

Pour les opérateurs de comparaison et d'élargissement, affaiblir la relation d'ordre assurera seulement la correction des transformations. Pour l'union de deux valeurs abstraites en revanche, affaiblir la relation d'ordre est essentiel puisque le diagramme résultant de l'unification sera celui de la valeur abstraite union et doit donc être compatible avec la relation d'ordre de l'union.

Ce dernier point soulève un problème. On a besoin d'unifier les diagrammes pour pouvoir calculer l'union et il faut donc calculer l'unification avant de calculer l'union. On ne peut donc pas interroger la valeur abstraite union pour construire $\preceq$. Il est important d'être capable de le calculer à partir des deux relations d'ordres.

L'affaiblissement de la relation d'ordre affaiblit du même coup la relation d'équivalence entre les bornes, ce qui a pour effet de diviser les classes d'équivalence des bornes utilisées dans le diagramme.

$$\equiv = \equiv_1 \cap \equiv_2$$

Sommets du diagramme d'unification

Une fois fixées les relations d'ordre et d'équivalence, la donnée d'un ensemble de sommets suffit à définir un unique diagramme. Il s'agira donc de notre diagramme unifié. Il ne reste en réalité qu'à choisir l'ensemble de sommets du diagramme.

Il y a en particulier deux unifications remarquables : le diagramme dont les sommets sont ceux présents dans les deux diagrammes à la fois, et le diagramme dont les sommets sont ceux présents dans l'un ou l'autre des diagrammes. Le premier peut être utilisé pour conserver des diagrammes simples : chaque fois qu'on doit unifier des diagrammes venant de deux branches différentes du programme, on ne garde que les bornes qui ont été introduites dans les deux branches à la fois. Cela permet en particulier de ne pas avoir de diagramme dont la complexité s'agrandit en présence de boucles. La seconde unification est en revanche plus précise. C'est un raffinement du premier qui permet de conserver un maximum d'information.

Ici, nous comptons sur un critère de fragmentation pour éviter la croissance infinie des diagrammes. Aussi, nous choisirons un diagramme d'unification composé de l'ensemble des sommets du premier et du second diagramme, quitte à simplifier ce résultat si le critère de fragmentation le juge nécessaire. On peut faire varier aisément le résultat de l'unification en changeant librement l'ensemble de sommets à intégrer dans le diagramme unifié.

Il reste un point de détail sur cet ensemble de sommet. Comme nous l'avons noté dans la section précédente, la relation d'équivalence du diagramme unifié va être affaiblie et les classes d'équivalence de bornes sont divisées. En conséquence, si plusieurs bornes étaient équivalentes avant l'union et leur classe d'équivalence attachée à un sommet, ce n'est plus nécessairement le cas après unification. On peut donc choisir une ou plusieurs expressions pour remplacer la borne originale. Si $i \equiv_1 j$, que i est attachée à un sommet dans le premier diagramme mais que $i \not\equiv j$ on peut choisir d'avoir i ou j dans le diagramme unifié, ou même d'avoir les deux bornes à la fois.

Entrées : $D_1 = (B_1, T_1)$ un diagramme sur $(\mathcal{B}, \preceq_1)$ et $D_2 = (B_2, T_2)$ un diagramme sur $(\mathcal{B}, \preceq_2)$
$\preceq = \preceq_1 \cap \preceq_2$
$B = \bigcup_{b_1 \in B_1} \mathit{fission}_{\preceq_1, \preceq_2}(b_1) \cup \bigcup_{b_2 \in B_2} \mathit{fission}_{\preceq_2, \preceq_1}(b_2) \,/\, \equiv$
Pour *tout* $b \in B$ **faire**
  Ajouter si nécessaire b dans D_1
  Ajouter si nécessaire b dans D_2

Généraliser le diagramme D_1 pour $\preceq$
Généraliser le diagramme D_2 pour $\preceq$
Pour *tout* (b_1, b_2) *dans* D_1 *et* D_2 **faire**
  Si $T_1(b_1, b_2)$ *est* $\top$ *ou* $\bot$ **alors**
    Choisir une nouvelle variable de synthèse s
    $T_1(b_1, b_2) \leftarrow \top$
  Si $T_2(b_1, b_2)$ *est* $\top$ *ou* $\bot$ **alors**
    Choisir une nouvelle variable de synthèse s
    $T_2(b_1, b_2) \leftarrow \top$
  Unifier les variables de synthèse $T_1(b_1, b_2)$, $T_2(b_1, b_2)$

Figure 5.15 : Algorithme d'unification de deux diagrammes.

Nous remettons ce choix au domaine de borne, en appliquant ses fonctions $\mathit{fission}_{\preceq_1, \preceq_2}$ et $\mathit{fission}_{\preceq_2, \preceq_1}$ respectivement aux sommets des deux diagrammes. Finalement, on calcule l'ensemble des sommets B du diagramme à partir des ensembles de sommets B_1 et B_2 des deux diagrammes à unifier :

$$B = \bigcup_{b_1 \in B_1} \mathit{fission}_{\preceq_1, \preceq_2}(b_1) \cup \bigcup_{b_2 \in B_2} \mathit{fission}_{\preceq_2, \preceq_1}(b_2) \,/\, \equiv$$

Il faut quotienter l'ensemble B par la relation d'équivalence $\equiv = \equiv_1 \cap \equiv_2$, car si les ensembles de sommets de chacun des cotés de l'union sont tous différents pour $\equiv$, il peut y avoir une borne à gauche de l'union qui soit équivalente mais non égale à une borne à droite de l'union.

L'algorithme d'unification

L'algorithme d'unification commence par calculer la relation d'ordre unifiée et l'ensemble des sommets. L'algorithme de transformation ne faisant que changer l'ensemble d'arcs, il faut impérativement ajouter les sommets manquants à l'un et l'autre des diagrammes. On peut ensuite appliquer l'algorithme de transformation défini en section 5.5.5. Enfin, il faut unifier les variables associées aux sommets. L'algorithme est donné figure 5.15

5.5.7 Conclusion

Si on oublie l'interprétation des instructions d'instrumentation, les algorithmes que nous avons définis suffisent à implémenter le domaine de fragmentation des diagrammes de tranches. Nous rappelons les cinq opérations qu'un domaine de fragmentation doit implémenter.

- Les effets des instructions et des gardes sont évalués sur chacun des sommets, modifiant éventuellement la borne associée à ce sommet. Lorsque une borne de remplacement ne peut être trouvée pour un sommet après une affectation non-inversible, on supprime le sommet en appliquant l'algorithme de retrait de sommet.
- Les effets des instructions d'instrumentation seront abordés dans de la section suivante. Ils n'impliquent que des ajouts ou des retraits de sommets.
- L'unification des fragmentations est réalisée grâce à l'algorithme d'unification des diagrammes.
- La généralisation et la spécialisation des diagrammes ont leurs algorithmes propres.
- La réduction (partielle) de la valeur abstraite s'obtient par la clôture et la réduction successives des diagrammes.

5.6 Application du critère de fragmentation

Afin d'assurer que la croissance du nombre de variables de la fragmentation décrite par nos diagrammes soit finie, il y a au moins trois solutions.

1. On peut inventer une sorte d'élargissement sur les diagrammes. C'est à dire un opérateur qui à deux diagrammes obtenus par des itérations successives pour le même point de contrôle donne un diagramme « simplifié » de sorte que le nombre de variables de la fragmentation ne croisse pas ou pas au delà d'une certaine borne. Par exemple, on peut retirer tout sommet ajouté par la seconde itération dont l'expression ne serait pas équivalente à celle d'un sommet présent à la première itération. Sous l'hypothèse à vérifier que la fission d'un sommet ne peut se produire qu'un nombre fini de fois, on aurait alors un nombre de sommets borné dans les diagrammes et par suite un nombre borné de tranches.
2. On peut utiliser un algorithme d'unification qui n'ajoute pas de nouveaux sommets. On peut restreindre l'usage d'une telle unification aux seuls points d'élargissement, ce qui revient à peu de chose près à la solution précédente.
3. Enfin, on peut se baser sur un critère de fragmentation capable d'assurer en tant que tel que le nombre de variables ne croît pas.

Nous allons appliquer aux diagrammes le critère de fragmentation défini en section 4.1.4. Cela va nécessiter de modifier légèrement les diagrammes afin d'y ajouter quelques informations supplémentaires.

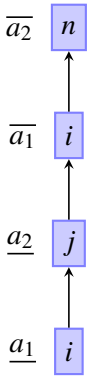
Dans le cas idéal, le critère de fragmentation choisit des fragments qui ont la forme de tranches. Ce ne sera évidemment pas toujours le cas et le critère pourra parfois désigner des fragments composés de cellules non consécutives d'un tableau. Nous chercherons à sur-approximer ces fragments par des tranches.

En outre, pour plusieurs raisons, l'ensemble des variables de synthèse défini par un diagramme peut ne pas coïncider avec celui défini par le critère de fragmentation. D'une part, les diagrammes imposent que deux tranches ne se chevauchent pas, et il faudra alors décomposer le fragment en l'intersection des tranches et leurs complémentaires. D'autre part, à cause de la sur-approximation, il est possible qu'on ne parvienne plus à distinguer deux fragments pourtant bien distincts dans le critère de fragmentation.

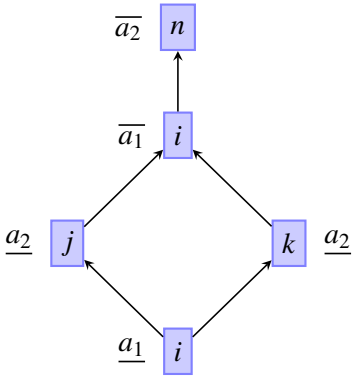
Ainsi, il est possible qu'une variable du critère de fragmentation corresponde à plusieurs tranches du diagramme. Ou inversement, la même tranche du diagramme peut correspondre

à plusieurs variables. Si nous devons retrouver une tranche désignée par le critère, nous pouvons néanmoins l'identifier par sa borne supérieure et sa borne inférieure dans le diagramme. Supposons que le critère nous incite à définir une variable a_1 associée à la tranche $A[1..i[$ et une variable a_2 associée à la tranche $A[j..n[$ et qu'on ait $1 \preceq j \preceq i \preceq n$. Introduire ces deux tranches dans le diagramme nous force à considérer leur intersection $A[j..i[$ et ses complémentaires $A[1..j[$ et $A[i..n[$. Le diagramme serait le suivant.

Pour mémoriser que ce sont les tranches $A[1..i[$ et $A[j..n[$ qui nous intéressent, nous pouvons attacher à leurs bornes inférieures et supérieures une décoration. Nous noterons $\underline{a_1}$ ou $\overline{a_1}$ à coté d'un sommet pour signifier qu'il est la borne inférieure ou supérieure respectivement du fragment désignée dans le critère de fragmentation par la variable a_1. En décorant notre diagramme avec ces nouvelles notations, nous obtenons :



Imaginons maintenant, que j ait été égal à k jusqu'à présent, mais que nous perdions cette information et que j et k soient maintenant incomparables. Nous pouvons aussi bien conserver j que conserver k comme borne inférieure de a_2. Il n'y a a priori pas moyen de choisir lequel des deux sera le bon. En pratique, ce cas est courant lors des premières itérations d'une boucle et il n'est possible de véritablement choisir la borne convenable qu'aux itérations suivantes. En attendant, il vaut mieux conserver les deux bornes.



Nous avons donc pour chaque variable du critère de fragmentation non pas une borne inférieure et une borne supérieure mais un ensemble de bornes inférieures et un ensemble de bornes supérieures. On ne conserve que les bornes extrémales. Si par exemple, on a $b_1 \preceq b_2$ alors on ne les prendra pas toutes les deux comme bornes supérieures ou comme bornes inférieures. S'il s'agit de bornes supérieures, on ne conservera que la plus grande b_2 et s'il s'agit de bornes inférieures, on ne conservera que la plus petite. En conséquence, on conserve à tout moment une sur-approximation du fragment que l'on cherche à décrire.

Le fait d'avoir des bornes de la forme $i+c$ est essentiel ici : cela nous assure que les ensembles de bornes ne pourront pas être infinis. En effet, il ne peut y avoir plus de bornes que de variables puisque $i+c$ et $i+d$ sont forcément comparables. On a donc au plus autant de bornes inférieures et autant de bornes supérieures que de variables dans le programme.

On va chercher à retirer du diagramme toute borne qui ne délimite pas un fragment, c'est à dire tout sommet qui n'a pas de décoration. Ces bornes augmentent « inutilement » la complexité de la fragmentation en distinguant des tranches qui n'ont pas été désignées par le critère de fragmentation.

Grâce à cela, si on supprime du diagramme tout sommet non décoré, on peut assurer que le nombre de sommets du diagramme est borné. Rappelons que le critère définit un nombre borné de fragments (c'est son rôle) et plus précisément un fragment par accès. Le nombre de sommets du diagramme ne peut donc excéder deux fois le nombre de variables du programme multiplié par le nombre d'accès. Le nombre de tranches quant à lui, ne peut excéder le carré du nombre de sommets. Cette borne sur le nombre de tranches peut être élevée même si en pratique le nombre de tranches reste généralement raisonnable.

Notre application du critère de fragmentation va donc consister à

1. décorer les diagrammes pour associer les bornes aux fragments du critère qu'elles délimitent,
2. supprimer tout sommet du diagramme qui n'est la borne d'aucun fragment du critère et
3. maintenir la décoration en interprétant les instructions d'instrumentation et en recalculant les annotations aux points de jointure.

Nous traitons maintenant le cas des points de jointure avant d'énumérer les effets des quatre instructions d'instrumentation.

Points de jointure. Lorsque deux branches du programme se rejoignent en un point, la fragmentation symbolique F en ce point est l'union des fragmentations F_1 et F_2 venant de chacune

de ces branches.

$$\forall \sigma \; F(\sigma) = F_1(\sigma) \cup F_2(\sigma)$$

Nous ne sommes pas capables en général de calculer exactement cette union. Nous allons donc la sur-approximer. Soit A un tableau et D_1 et D_2 les diagrammes pour deux branches du programme se rejoignant. Soit également a une variable de synthèse définie par le critère de fragmentation. Notons S_1 et I_1 les ensembles de bornes supérieures et inférieure pour a dans le premier diagramme et S_2 et I_2 dans le second. Au point de jointure, les deux diagrammes vont être unifiés par l'algorithme d'unification. Il nous reste à savoir quels sont les sommets S et I du diagramme unifié que nous désignons comme bornes inférieures et supérieures pour la variable a.

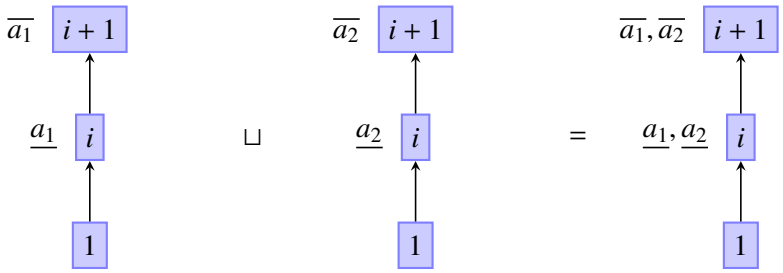
La manière la plus directe de choisir S et I est de prendre les unions $S_1 \cup S_2$ et $I_1 \cup I_2$. Il faut néanmoins prendre en compte un détail supplémentaire. Lors de l'unification, la relation d'ordre initiale $\preceq_1$ et $\preceq_2$ des deux diagrammes va être affaiblie. A ce moment, l'algorithme de généralisation des diagrammes va fissionner les sommets des diagrammes. Par défaut, nous pouvons attribuer à chacun de ces sommets les même décorations qu'au sommet original. Il est néanmoins possible de faire mieux lorsque ces sommets fissionnés sont comparables dans la relation d'ordre unifiée $\preceq = \preceq_1 \cap \preceq_2$. En effet, dans un ensemble de bornes fissionnées, nous pouvons conserver comme bornes supérieures uniquement les sommets minimaux et comme bornes inférieures uniquement les sommets maximaux. Ainsi, nous nous assurons que les fragments seront minimaux pour la relation d'ordre $\preceq$. Formellement, nos ensembles S_1 et I_1 deviennent les ensembles S'_1 et I'_1 après la fission :

$$\begin{aligned} S'_1 &= \bigcup_{b \in S_1} \min_{\preceq} \left(\mathit{fission}_{\preceq_1, \preceq_2}(b) \right) \\ I'_1 &= \bigcup_{b \in I_1} \max_{\preceq} \left((\mathit{fission}_{\preceq_1, \preceq_2}(b) \right) \end{aligned}$$

On calcule S'_2 et I'_2 de la même manière. On peut enfin calculer nos ensembles S et I comme union des ensembles calculés. Il ne reste plus qu'à les réduire pour ne garder que les bornes supérieures maximales et les bornes inférieures minimales.

$$\begin{aligned} S &= \max_{\preceq}(S'_1 \cup S'_2) &&= \max_{\preceq} \left(\bigcup_{b \in S_1} \min_{\preceq} \left(\mathit{fission}_{\preceq_1, \preceq_2}(b) \right) \cup \bigcup_{b \in S_2} \min_{\preceq} \left(\mathit{fission}_{\preceq_2, \preceq_1}(b) \right) \right) \\ I &= \min_{\preceq}(I'_1 \cup I'_2) &&= \min_{\preceq} \left(\bigcup_{b \in I_1} \max_{\preceq} \left((\mathit{fission}_{\preceq_1, \preceq_2}(b) \right) \cup \bigcup_{b \in I_2} \max_{\preceq} \left((\mathit{fission}_{\preceq_2, \preceq_1}(b) \right) \right) \end{aligned}$$

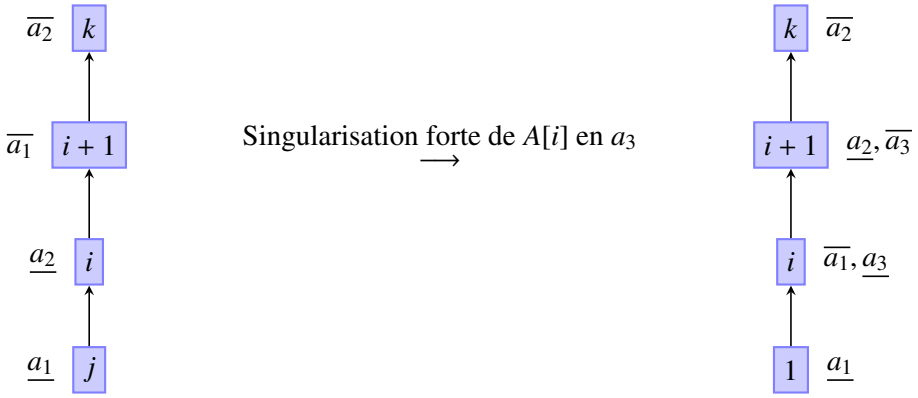
Nous pouvons vérifier que le comportement que nous donnons ainsi aux diagrammes est conforme à celui que l'on prônait dans la section 4.3. Dans le second exemple qui y était présenté, on a une boucle qui à chaque itération peut selon une condition modifier d'une manière ou d'une autre une cellule $A[i]$. Le critère de fragmentation désigne a_1 et a_2 les variables correspondant aux accès selon que la condition est vraie ou non. Si la condition est vraie, a_1 est le singleton $A[i]$ et a_2 le fragment vide. Le diagramme correspondant associe $\underline{a_1}$ à i et $\overline{a_1}$ à $i+1$. Si la condition est fausse on a l'inverse. Pour joindre les deux fragmentations, on associera simplement $\{\underline{a_1}, \underline{a_2}\}$ à i et $\{\overline{a_1}, \overline{a_2}\}$ à $i+1$. Autrement dit on ne distingue plus a_1 de a_2, et on a en fait un seul fragment qui se trouve être l'union des deux.



Une fois le diagramme redécoré, certains sommets peuvent n'avoir aucune décoration. On peut alors supprimer ces sommets, car ils ne représentent la borne d'aucun fragment auquel s'intéresse le critère.

Singularisation faible. La singularisation faible nous demande d'ajouter un fragment singleton a correspondant à une cellule $A[exp]$ accédée dans une instruction. S'ils ne sont pas déjà dans le diagramme, on ajoute alors les sommets $inf(exp)$ et $sup(exp)$ (Cf. section 5.3) dans le diagramme et on associe $\underline{a}$ avec le premier et $\overline{a}$ avec le second.

Singularisation forte. La différence avec la singularisation faible est qu'il faut retirer la cellule singularisée de tout autre fragment. C'est facile lorsque la cellule est à l'extrémité d'une tranche. Il suffit alors de prendre les bornes inférieures déjà associées à $inf(exp)$ et de les affecter à $sup(exp)$, en réduisant alors effectivement la taille de ces fragments de 1. De même on prend les bornes supérieures déjà associées à $sup(exp)$ pour les affecter à $inf(exp)$. A la fin, $inf(exp)$ et $sup(exp)$ ne doivent être la borne inférieure et supérieure respectivement du fragment singularisé.



Le problème se pose quand un fragment contient la cellule à singulariser, mais que celle-ci n'en est pas une extrémité. On ne peut alors pas retirer la cellule sans créer un trou dans le fragment. On se contentera alors de sur-approximer le fragment privé de la cellule par le fragment original, autrement dit, en ne faisant rien.

Réinitialisation. La réinitialisation rend un fragment vide. Pour nous, cela revient à retirer du diagramme toute indication de borne supérieure ou inférieure correspondant à ce fragment. Si un sommet se retrouve ainsi privé d'annotation, nous pourrons le supprimer.

La composition. Nous prendrons logiquement pour composition de deux fragments l'union des bornes supérieures et l'union des bornes inférieures. Comme pour les points de jointure, il

faudra réduire ces deux ensembles en n'en prenant que les éléments maximaux et minimaux respectivement.

Si les deux fragments composés sont disjoints, l'union est une sur-approximation du fragment composé. Ceci correspond aux comportements décrits pour le premier exemple de la section 4.3. Il s'agissait d'une boucle effectuant des traitements potentiellement différents sur les cellules paires et impaires. Le critère de fragmentation demande à ce que ces ensembles de cellules soient distingués, mais c'est impossible dans les diagrammes. A défaut, les diagrammes regrouperont dans les même tranches les cellules indépendamment de la parité de leur indice.

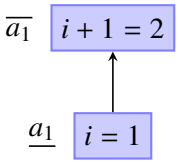
A l'issue de la composition, les bornes du fragment à gauche de l'opérateur de composition ayant changé, certaines bornes du diagramme ne sont plus associées à aucun fragment et on pourra les retirer.

Conclusion et exemple L'application du critère de fragmentation aux diagrammes demande un travail supplémentaire. Le fait d'interpréter les instructions d'instrumentation nous permet d'accepter les diverses variations du critère. Si on change ou déplace ces instructions, nous sommes toujours en mesure de les interpréter.

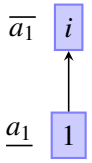
En l'occurrence, nous verrons dans l'exemple suivant que la nécessité de modifier le critère peut venir aussi de la manière dont on abstrait les fragmentations. Prenons une simple boucle itérant sur un tableau qu'on instrumente avec la version développée du critère de fragmentation.

Pour i *de* 1 *à* n **faire**
  $a_2 \cup\!: a_1$
  $a_1 :: \{A[i]\}$
  $A[i] \leftarrow 0$

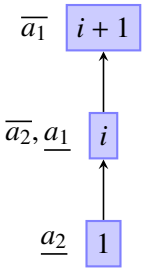
Le critère de fragmentation introduit deux variables : a_1 correspondant au singleton couramment accédé et a_2 accumulant les cellules déjà accédées. Nous partons d'un diagramme vide. Lors de la première itération nous avons $i = 1$. Avant l'accès à la cellule, le critère de fragmentation nous invite à singulariser $A[i]$, ce qui nous amène à ajouter i et $i + 1$ dans le diagramme et à les annoter pour qu'on se souvienne qu'ils représentent les bornes du fragment correspondant à a_1. On a $i \equiv 1$ et $i + 1 \equiv 2$ on peut donc choisir n'importe laquelle de ces expressions dans la représentation du diagramme.



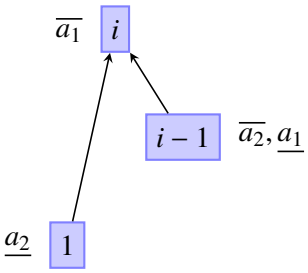
En sortant de la boucle, les sommets i et $i + 1$ sont renommés en $i - 1$ et i alors que la variable i est incrémentée. Une fois l'itération terminée, nous devons unifier ce diagramme avec le diagramme vide qu'on a en entrée de boucle. La relation d'ordre est affaiblie par l'union. On perd les équivalences entre les bornes, et il faut donc fissionner les sommets. La relation d'ordre unifiée vérifie $i - 1 \preceq 1$ et $i \preceq 2$. On choisit la borne maximale 1 pour $\underline{a}$ et la borne minimale i pour $\overline{a}$.



On peut interpréter une seconde fois la boucle. L'élargissement aura été appliqué sur la valeur abstraite, mais ne changera pas notre diagramme. Avant de singulariser une nouvelle fois $A[i]$ le critère de fragmentation nous demande de composer a_2 encore vide avec a_1 : les bornes de a_1 deviennent celles de a_2. Puis on traite la singularisation comme lors de la première itération.

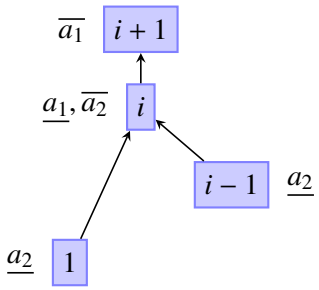


Comme pour la première itération, on renommera $i + 1$ en i et i en $i - 1$ avant de sortir de la boucle. Au point de jonction cette fois, il n'y a plus de sommet à fissionner. Mais comme la relation d'ordre n'induit plus $1 \leq i - 1$, ces deux sommets ne seront donc plus reliés dans le diagramme. L'algorithme d'unification joindra alors $A[1..i - 1[$ et $A[i - 1]$ pour former $A[1..i[$.

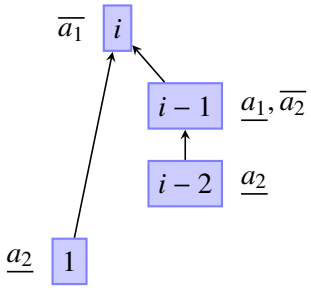


La décoration du diagramme rappelle que a_1 doit représenter le singleton $A[i - 1]$ et qu'il s'agit conformément au critère de la dernière cellule accédée. Plus précisément, il s'agit d'une sur-approximation du fragment voulu pour a_1. Lors de la première itération, notre définition de a_1 était plus précise. La tranche $A[1..i[$ correspondait exactement à ce que le critère exigeait pour a_1. Si le programme a déjà exécuté une fois la boucle, on a $i = 2$ et $A[1..i[= A[i - 1]$. Si le programme n'a encore jamais exécuté la boucle, on a $i = 1$ et $A[1..i[$ est vide. Cela est conforme au critère puisque avant d'être entrée dans la boucle, aucune singularisation n'a été rencontrée pour a_1. Mais maintenant qu'on a perdu l'inégalité $1 \leq i \leq 2$, les diagrammes ne nous donnent pas l'expressivité nécessaire pour désigner un fragment qui serait vide si c'est la première fois qu'on entre dans la boucle et qui serait singleton $A[i - 1]$ sinon.

On interprète une nouvelle fois la composition. On calcule l'union des bornes inférieures de a_1 et de a_2 soit $\{1, i - 1\}$ dont on n'a pas de minimum et l'union des bornes supérieures $\{i - 1, i\}$, dont le maximum est i. Puis on singularise $A[i]$.



Une nouvelle unification produira les mêmes effets qu'à l'itération précédente et nous forcera à composer $A[1..i-1[$ avec $A[i-1]$. On aura au point de jonction le diagramme

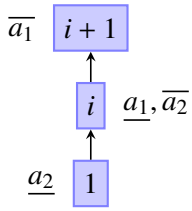


La composition de a_1 et a_2 nous fera perdre le sommet $i-1$ qui ne délimite plus rien. Le sommet $i-2$ sera néanmoins conservé. Si on continuait les itérations, $i-2$ serait transformé en $i-3$, $i-4$ etc.

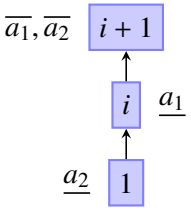
Dans la valeur abstraite, la tranche $A[i-2..i-1[$ n'aura aucune propriété et il ne s'agit donc que d'un fragment parasite qui ne change en rien le déroulement de l'analyse. Il est néanmoins décevant de n'être pas parvenu à ne conserver que des tranches utiles sur un exemple aussi simple. La présence de cette tranche parasite vient de l'approximation que nous faisons à chaque unification sur le fragment a_1. Chaque composition vers a_2 souffre de cette approximation. Si on déplace la composition pour la faire avant de sortir de la boucle

Pour *i de* 1 *à n* **faire**
  $a_1 :: \{A[i]\}$
  $A[i] \leftarrow 0$
  $a_2 \cup: a_1$

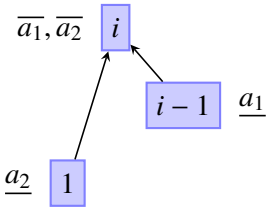
on obtiendra dans la boucle après l'affectation le diagramme suivant :



Cette fois-ci, la composition de a_1 et a_2 sera exacte et donnera $A[1..i+1[$. Elle ne modifie pas le diagramme puisque i reste la borne inférieure de a_1.



a_1 reste inclus dans a_2 jusqu'au point de jointure où la perte de $1 \preceq i$ nous force comme dans l'autre configuration à composer les tranches $A[1..i-1[$ et $A[i]$:



Mais cette fois, la singularisation de $A[i]$ déplacera $\underline{a_2}$ de $i-1$ vers i. $i-1$ ne délimite plus aucune partie et peut être retiré. On retombe sur le diagramme initial. Nous avons donc trouvé une solution permettant de supprimer la tranche inutile. C'est cette variation du critère de fragmentation que notre implémentation utilise sur les diagrammes.

5.7 Conclusion

Les diagrammes sont une représentation abstraite des fragmentations conçue pour pouvoir résoudre facilement les problèmes de l'approximation de fragmentation et celui de la réduction partielle. Ils sont fondés sur le parti pris qu'il est plus facile de sur-approximer des fragments que de les sous-approximer. Ceci est particulièrement visible quand on considère l'algorithme d'unification. On ne cherche pas à calculer des tranches qui seraient la sous-approximation d'une tranche venant de chaque diagramme à unifier. On va au contraire chercher à conserver toutes les tranches qui sont des sur-approximations des tranches des diagrammes à unifier.

Le problème à l'origine de la création des diagrammes, est celui des alias des variables d'indice. Lorsque deux variables d'indice peuvent avoir la même valeur, les analyses classiques partitionnent successivement en trois l'ensemble d'état pour tout couple de variables pouvant être aliasées selon que la première est inférieure, égale ou supérieure à la seconde. Ceci entraîne une complexité de l'analyse pouvant être exponentielle dans les pires cas. Les diagrammes permettent de considérer ces indices potentiellement aliasés sans énumérer les cas. La réduction d'une valeur abstraite fragmentée par des diagrammes reste exponentielle mais nous pouvons nous contenter de la réduction partielle proposée dans ce chapitre dont la complexité reste polynomiale.

Perspectives. Les diagrammes offrent plusieurs perspectives que l'on peut obtenir en généralisant, non sans problème, les concepts manipulés. La plus évidente des perspectives est la possibilité de modifier l'ensemble de bornes $\mathcal{B}$. On peut chercher par exemple à généraliser les expressions de bornes aux expressions affines. Le problème à surmonter pour cela est celui de la fission des bornes. Pour peu qu'on ait une relation linéaire entre des variables d'indice, alors il y a une infinité d'expressions affines équivalentes. Lors de la fission, le choix des bornes à garder dans cette infinité risque d'être difficile. On peut imaginer des moyens de remettre ces

choix à plus tard. Sur notre base de tests utilisés pour l'expérimentation, il suffit d'attendre une itération supplémentaire pour être capable de choisir la borne à garder parmi celles de la fission. Le problème semble inhérent à la méthode d'itération.

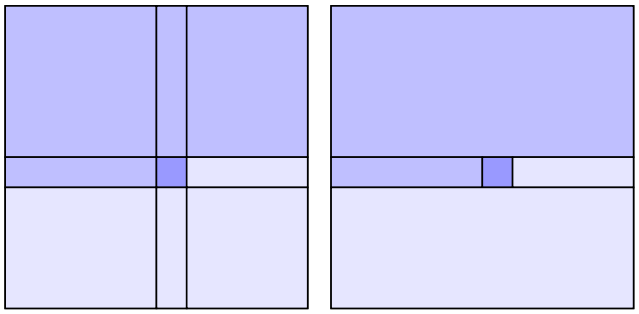
Plus intéressant serait de changer la définition des tranches en même temps que la définition de l'ensemble de bornes. Les bornes pourraient être des expressions de pointeurs et les tranches les éléments accessibles depuis le pointeur origine d'un arc avant d'accéder au pointeur destination de l'arc. Ceci a toutefois peu de chances de fonctionner sur d'autres structures que des listes chaînées pour des raisons que nous évoquerons plus loin dans cette section. Dans le cas des listes, l'idée pourrait être intéressante si on envisage de traiter des programmes manipulant des listes et des pointeurs pouvant pointer sur le même élément de la liste : les diagrammes permettraient toujours d'éviter l'énumération combinatoire des cas d'alias.

En restant sur les tableaux, on peut imaginer une version des diagrammes adaptée aux représentations des arbres binaires complets en tableaux et notamment aux tas. Il s'agit d'une utilisation des tableaux où chaque cellule représente un nœud de l'arbre et si son indice dans le tableau est i alors son fils gauche est la cellule d'indice $2 \times i$ et son fils droit la cellule d'indice $2 \times i + 1$. On pourrait alors imaginer qu'une tranche (i, j) représente le chemin dans l'arbre entre le nœud i et le nœud j. On adapte la relation $\preceq$ de sorte qu'elle ordonne deux éléments si l'un est ascendant de l'autre dans l'arbre.

S'il est intéressant d'expérimenter différentes versions de $\mathcal{B}$ et $\preceq$, le changement de la relation d'ordre entre les sommets du graphe a des conséquences importantes. Notre relation $\preceq$ étant une sur-approximation de $\leq$, on en utilise quelques unes des propriétés. Par exemple, lorsque deux arcs du diagramme sont sur un même chemin, c'est qu'ils ne peuvent pas se chevaucher : les indices des cellules de l'un sont strictement inférieurs aux indices des cellules de l'autre. Par conséquent, on peut facilement déterminer quelles sont les tranches qui peuvent potentiellement contenir des cellules communes. Ceci nous permet de savoir quelles sont les affectations faibles à réaliser. Avec une autre relation $\preceq$ énumérer les tranches qui peuvent être modifées par une affectation peut être moins aisé. Dans le cas où on prend comme bornes des pointeurs et comme relation d'ordre l'accessibilité, le problème peut être pire. Sur des structures de données avec cycle, les compositions ne sont plus forcément disjointes. Ceci rend difficile la recherche de propriétés d'agrégation (Cf. chapitre 8). Ce sont autant de problèmes qu'il faudrait résoudre.

Pour la représentation des structures de données chaînées autres que les listes, le problème qui empêche l'utilisation directe des diagrammes est similaire à celui rencontré avec les tableaux multidimensionnels. Pour un tableau à deux dimensions, on peut imaginer prendre un ensemble des couples de bornes de $\mathcal{B} \times \mathcal{B}$. Une tranche $[i_1..j_1, i_2..j_2[$ serait alors l'ensemble des cellules $A[\ell_1, \ell_2]$ telles que $i_1 \leq \ell_1 \leq j_1$ et $i_2 \leq \ell_2 \leq j_2$. Ce serait un mauvais choix, néanmoins, puisque si on a un diagramme dont les sommets sont $(0, 0)$, $(n, 0)$ et (n, m), la composition de la tranche $A[0..n, 0]$ avec la tranche $A[n, 0..n[$ ne donne certainement pas la tranche $A[0..n, 0..n[$. La composition de deux arcs adjacents dans le graphe ne donne pas l'arc obtenu transitivement. C'est le problème que nous évoquions avec les structures de données générales. Si on compose l'ensemble des éléments accessibles par un pointeur p et pouvant accéder à un pointeur q et l'ensemble des éléments accessibles par q et pouvant accéder à r on obtient un ensemble plus petit que l'ensemble des éléments accessibles depuis p et accédant à r.

La méthode la plus simple pour traiter les tableaux à n dimensions serait probablement d'utiliser un n-uplet de diagrammes, chacun décomposant une dimension. La fragmentation qui nous intéresse serait alors la fragmentation produit. La figure 5.16(a) illustre ce que serait le découpage d'un tableau à deux dimension pour un programme simple itérant en ligne puis en colonne



(a) Fragmentation produit (b) Fragmentation hierarchique

Figure 5.16 : A gauche on a fragmenté indépendamment la première et la seconde dimension du tableau. A droite, on a fragmenté d'abord par ligne, puis chacune des lignes a été fragmentée séparément.

sur chacune des cellules. Ce découpage produirait quatre fragments pour les cellules déjà itérées, un fragment pour la cellule en cours d'itération, et quatre fragments pour les cellules pas encore itérées qui peuvent n'être associés à aucune variable pour des soucis d'optimisation.

Une autre manière de faire est d'utiliser un diagramme pour décomposer le tableau en fragments de lignes. Puis à chaque tranche, associer un diagramme propre permettant de subdiviser la tranche selon des groupes de colonnes. On obtiendrait alors la fragmentation illustrée figure 5.16(b) dans laquelle seulement deux fragments seraient définis pour les cellules déjà itérées et deux également pour les cellules pas encore itérées.

Cette manière de découper ne suppose pas nécessairement qu'il y ait un ordre entre les dimensions. Cela suppose néanmoins que nous sachions choisir à l'exécution dans quel ordre fragmenter les dimensions. La « direction » d'une fragmentation n'est d'ailleurs pas nécessairement orthogonale aux dimensions d'un tableau. On peut imaginer vouloir des fragments de la forme $\{A[\ell_1, \ell_2] \mid 1 \leq \ell_1 + \ell_2 \leq i + j\}$.

Une piste intéressante serait alors de concevoir les diagrammes comme une hiérarchie de fragmentations. Chaque niveau est un diagramme découpant le tableau selon une direction. Chacune des tranches de ce diagramme peut ensuite être refragmentée à son tour dans une autre direction, possiblement déjà utilisée pour fragmenter à un niveau supérieur. L'intérêt de cette formalisation est que la composition est toujours possible à un même niveau de la hiérarchie. En revanche, on ne peut pas composer une tranche d'un niveau avec une tranche d'un niveau inférieur. Lors de la composition de deux tranches, les subdivisions peuvent être conservées à condition qu'elles soient dans la même direction et qu'elles aient été unifiées.

Cette idée est dans le prolongement de celles qui ont conduit à l'élaboration des diagrammes. Elle donnerait une solution très imprécise, se concentrant uniquement sur ce qu'il est facile de calculer. Le problème des tableaux à plusieurs dimensions est intéressant pour l'analyse des algorithmes basés sur la programmation dynamique. [Bel52]

6 Recherche de relations de translation entre les tranches de diagrammes

Les diagrammes découpent les tableaux en désignant fragment par fragment l'ensemble des cellules que peuvent contenir ces fragments. Mais ils ne permettent pas de lier entre eux les choix de représentants des différents fragments. On peut désigner une relation entre toutes les cellules d'un fragment et toutes les cellules d'un autre. Mais on ne peut pas par exemple restreindre cette relations aux seuls couples de cellules dont les indices sont égaux.

On peut établir le parallèle avec les domaines abstraits non-relationnels. Ceux-ci permettent d'abstraire indépendamment les ensembles de valeurs de chaque variable mais pas de désigner une relation entre ces variables. Et comme pour les domaines abstraits, les possibilités pour le développement des domaines de fragmentations sont vastes.

Nous nous intéressons dans ce chapitre à une classe particulière de fragmentations, celle décrivant des relations de translation. Une relation de translation est une relation de la forme

$$\left\{ \left(A_1[\ell], A_2[\ell + c_2], \ldots, A_n[\ell + c_n]\right) \;\middle|\; \ell \in \mathcal{L} \right\}$$

où $\mathcal{L}$ est un ensemble d'indices et $c_2, \ldots, c_n$ sont des termes ne contenant pas ℓ. Ici, nous serons encore plus restrictifs en imposant que ce soient des constantes, ne dépendant donc pas de σ. Il s'agit d'une relation qui lie les fragments $A_1[\mathcal{L}]$, $A_2[\mathcal{L} + c_2]$, …, $A_n[\mathcal{L} + c_n]$ de telle sorte que chaque cellule de chaque fragment n'est toujours en relation qu'avec une seule cellule des autres fragments.

Par exemple, un algorithme qui copie le contenu d'un tableau A dans un tableau B pourra être analysé en utilisant pour fragmentation la relation de translation

$$\left\{ \left(A[\ell], B[\ell]\right) \;\middle|\; \ell \in [1..n] \right\}$$

Un autre exemple est celui du programme qui initialise un tableau avec des valeurs successives allant de c à $c + n$, la cellule $A[\ell]$ ayant pour valeur $c + \ell - 1$. On peut complètement définir l'état du tableau en disant d'une part que $A[1]$ a pour valeur c et d'autre part que toute cellule du tableau a pour valeur celle de la cellule précédente augmentée de 1, c'est à dire

$$\forall \ell, 2 \leq \ell \leq n \Rightarrow A[\ell] = A[\ell - 1] + 1$$

Pour découvrir cette propriété, on utilisera comme fragmentation la relation de translation

$$\left\{ \left(A[\ell], A[\ell - 1]\right) \;\middle|\; \ell \in [2..n[\right\}$$

Enfin, mentionnons comme dernier exemple, celui des algorithmes de tri dont le résultat peut s'énoncer : « toute cellule du tableau a une valeur inférieur à celle de la cellule suivante. » Pour exprimer cette propriété, on pourra fragmenter le tableau avec la même relation de translation que dans l'exemple précédent.

Pour *i de* 1 *à n* **faire**
  $A[i] \leftarrow B[i]$

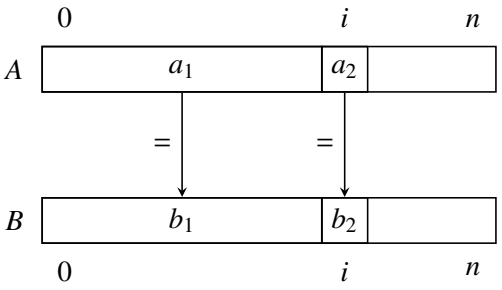


FIGURE 6.1 : À gauche, un programme réalisant la copie d'un tableau A dans un tableau B. À droite, une représentation de la fragmentation du tableau adaptée à l'analyse de la boucle.

Notre critère de fragmentation introduit section 4.1.1 est capable de désigner de telles relations et le fera chaque fois que deux cellules présentes dans la même instruction ont un écart d'indice constant. On peut donc conserver le critère de fragmentation.

L'abstraction de cette fragmentation quant à elle ne peut être réalisée par les diagrammes seuls. Nous pouvons néanmoins conserver les diagrammes pour la description de l'ensemble d'indices $\mathcal{L}$. Il nous faudra encore les combiner avec un autre domaine de fragmentation qui sera spécialisé dans l'expression des relations entre indices.

Ce chapitre introduit un tel domaine de fragmentation, dont le rôle est de désigner des relations affines entre les indices des cellules. Nous commencerons par décrire les problèmes que posent l'introduction de ces relations avant de décrire ce domaine puis nous traiterons de sa combinaison avec les diagrammes.

6.1 Problématique

Commençons par observer l'exemple de la copie de tableau de la figure 6.1.

L'analyse de ce programme avec des diagrammes de tranches et notre critère de fragmentation, donnera dans la boucle un partitionnement du tableau en quatre fragments $A[1..i[$, $A[i]$, $B[1..i[$, $B[i]$ et nous noterons les variables associées a_1, a_2, b_1, et b_2 respectivement. Ainsi nous aurions la fragmentation suivante.

$$F_d(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[\ell_2] \\ b_1 \mapsto B[\ell_3] \\ b_2 \mapsto B[\ell_4] \end{cases} \;\middle|\; \begin{cases} 1 \leq \ell_1 < \sigma(i) \\ \ell_2 = \sigma(i) \\ 1 \leq \ell_3 < \sigma(i) \\ \ell_4 = \sigma(i) \end{cases} \right\}$$

Afin de pouvoir analyser le programme, il est nécessaire de raffiner la fragmentation en imposant la relation $\ell_1 = \ell_3$. Pour ajouter cette relation, on combine notre première fragmentation F_d obtenue sur les diagrammes avec une autre fragmentation F_r capable d'exprimer cette relation,

mais n'exprimant rien d'autre.

$$F_r(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A_1[\ell_1] \\ a_2 \mapsto A_2[\ell_2] \\ b_1 \mapsto A_3[\ell_3] \\ b_2 \mapsto A_4[\ell_4] \end{cases} \;\middle|\; \forall A_i \in \mathcal{A},\ \ell_1 = \ell_3 \right\}$$

où $\mathcal{A}$ est l'ensemble des symboles de tableau. Ainsi F_r décrit toutes les fragmentations dans lesquelles l'indice de la cellule associée à a_1 et celui de la cellule associée à a_3 est le même. La fragmentation qui nous intéresse est l'intersection des deux fragmentations

$$F_d(\sigma) \cap F_r(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ a_2 \mapsto A[\ell_2] \\ b_1 \mapsto B[\ell_3] \\ b_2 \mapsto B[\ell_4] \end{cases} \;\middle|\; \begin{cases} 1 \leq \ell_1 < \sigma(i) \\ \ell_2 = \sigma(i) \\ 1 \leq \ell_3 < \sigma(i) \\ \ell_4 = \sigma(i) \end{cases}, \ell_1 = \ell_3 \right\}$$

Avec cette fragmentation, l'invariant du programme pourra être exprimé par les égalités $a_1 = b_1 \wedge a_2 = b_2$.

Notre premier objectif sera donc de définir le domaine de fragmentation permettant la sélection de relations entre les indices des cellules d'un même choix de représentants. La définition de ce domaine et sa combinaison avec le diagramme de tranches entraîneront toutefois un certain nombre de problèmes que nous tenterons d'identifier dans le reste de cette section.

Supposons d'abord que nous souhaitions que ces relations entre indices puissent être n'importe quelle inégalité affine. Ces relations s'ajouteront aux contraintes déjà imposées par les diagrammes. Les diagrammes définissent des tranches (j_1, j_2) par des formules φ de la forme

$$\varphi(\ell, \sigma) \;\equiv\; \dot{\gamma}\,[j_1]\,(\sigma) \leq \ell < \dot{\gamma}\,[j_2]\,(\sigma)$$

et produisent ainsi des fragmentations de la forme

$$F(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A_1[\ell_1] \\ \vdots \\ a_n \mapsto A_n[\ell_n] \end{cases} \;\middle|\; \begin{cases} \varphi_1(\ell_1, \sigma) \\ \vdots \\ \varphi_n(\ell_n, \sigma) \end{cases} \right\}$$

où $A_1, \ldots, A_n \in \mathcal{A}$ sont des symboles de tableau. On complète cette formule pour y intégrer les contraintes sur les relations entre indices. Notons $\varphi_r(\ell_1, \ldots, \ell_n)$ le système d'inégalités affines liant les indices des cellules d'un même choix de représentants. La fragmentation devient :

$$F(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A_1[\ell_1] \\ \vdots \\ a_n \mapsto A_n[\ell_n] \end{cases} \;\middle|\; \begin{cases} \varphi_1(\ell_1, \sigma) & \\ \vdots & \wedge\, \varphi_r(\ell_1, \ldots, \ell_n) \\ \varphi_n(\ell_n, \sigma) & \end{cases} \right\}$$

L'ensemble de toutes ces contraintes, liant $\ell_1, \ldots, \ell_n$ et σ constituent un système d'inéquations affines. L'ensemble des valeurs que peuvent prendre les variables peut être vu comme un polyèdre convexe. Cette abstraction de l'ensemble d'indices est utilisée dans certaines méthodes

d'analyse [GMT08]. En raisonnant ainsi, on généralise notre analyse. En effet, pour nous, chaque tranche n'est définie que par deux contraintes : sa borne inférieure et sa borne supérieure. Dans un système affine général, on peut avoir plus de contraintes.

Toutefois, réaliser cette généralisation nous fait perdre les propriétés sur lesquelles reposaient la construction des diagrammes de tranche. Pour approximer une tranche, il suffisait de trouver une tranche adjacente et d'en calculer l'union : l'union de deux tranches adjacentes est elle-même une tranche. Cela ne fonctionne pas si les fragments peuvent être délimités par des polyèdres convexes quelconques : l'union de deux polyèdres convexes ne sera en général pas convexe.

On peut toujours résoudre le problème autrement puisque notre méthode qui consiste à approximer un fragment par son union avec d'autres fragments n'est pas l'unique possibilité. Le problème de l'union des fragments reviendra néanmoins quand il sera question de calculer la réduction. (Cf. section 3.5) Rappelons que lorsque un ensemble de fragments recouvre un fragment cible, il est possible de renforcer les propriétés du second avec la disjonction des propriétés des premiers. Nous avions trouvé une solution dans les diagrammes simplement en considérant toutes les suites de tranches adjacentes. Cet algorithme n'est pas directement applicable dans le cas où les fragments sont représentés par des polyèdres quelconques.

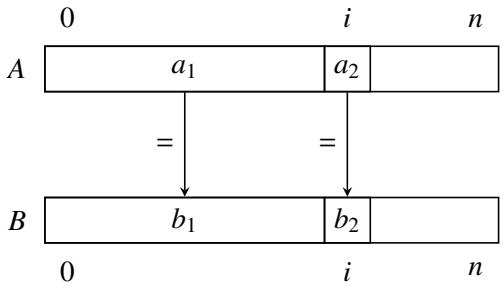
Il est tout à fait possible d'ignorer ces problèmes. Lorsqu'on ne sait pas approximer un fragment, on peut toujours le retirer de la fragmentation et on peut ne calculer aucune réduction. Bien que la précision de l'analyse s'en trouve affaiblie, certains exemples peuvent tout de même être analysés dans ces conditions. Il est néanmoins facile de construire des exemples courts où cela ne suffira pas. Nous avons dans notre base de tests des exemples dans lesquels la réduction est nécessaire à la preuve du programme. (Cf. section 10.2)

Il est donc tout de même intéressant de garder ce qui est acquis sur les diagrammes de tranches, quitte à les combiner avec un domaine désignant des relations entre indices. Pour ce qui concerne ces relations entre indices notons que, dans le cas qui nous intéresse – les relations de translation – les problèmes évoqués plus tôt sont très simples. On n'a plus des systèmes d'inéquations affines mais des systèmes d'équations affines de la forme

$$\ell_i = \ell + c_i$$

décrivant des variétés affines. Or l'union de variétés affines est rarement une variété affine elle même à moins que les deux variétés dont on calcule l'union soient incluses l'une dans l'autre. Les problèmes d'approximation et de réduction deviennent triviaux : les seules approximations pertinentes sont le retrait où l'ajout d'une équation affine et une réduction n'est possible que lorsqu'une variété affine en contient une autre.

Nous choisissons donc de développer un domaine de fragmentation qui soit la combinaison des diagrammes de tranches et d'un domaine spécialisé dans l'expression des relations affines, ce qui inclut les relations de translation. Nous décrirons plus loin ce second domaine. La combinaison des deux domaines se heurtera à un obstacle, la rigidité des transformations élémentaires (Cf. section 4.2.4). Pour le comprendre, reprenons l'exemple de la copie de tableau.



Après chaque copie de cellule, on doit avoir $a_1 = b_1$ et $a_2 = b_2$. Après la copie, on doit composer les fragments a_1 avec a_2 et b_1 avec b_2. C'est cette composition qui pose problème. Les deux compositions doivent être réalisées l'une après l'autre. Si on compose a_1 et a_2 en premier, la variable a résultante aura pour propriété la disjonction de celles de a_1 et celles de a_2. Or ces deux variables n'ont aucune propriété en commun : l'une est égale à b_1 tandis que l'autre est égale à b_2. Si on compose b_1 et b_2 en premier, le problème est le même. Il ne faudrait pas faire les deux compositions l'une après l'autre, il faudrait les faire en même temps.

Dans l'état actuel de notre théorie, nous n'avons pas encore de telle transformation élémentaire. Mais il existe déjà une solution plus simple utilisée dans [HP08]. Il s'agit de décomposer la valeur abstraite en autant de sous-valeurs abstraites qu'il y a de relations. Dans notre exemple, il y avait la relation entre a_1 et b_1 ainsi que la relation entre a_2 et b_2. On peut donc décomposer la valeur abstraite en deux, la première partie contenant les variables a_1 et b_1 et la seconde partie contenant les variables a_2 et b_2. Les fragmentations respectives pour ces deux valeurs abstraites seraient

$$F_1(\sigma) = \left\{ r : \begin{cases} a_1 \mapsto A[\ell_1] \\ b_1 \mapsto B[\ell_3] \end{cases} \;\middle|\; \begin{cases} 1 \leq \quad \ell_1 < \sigma(i) \\ 1 \leq \quad \ell_3 < \sigma(i) \end{cases} \wedge\, l_1 = l_3 \right\}$$

$$F_2(\sigma) = \left\{ r : \begin{cases} a_2 \mapsto A[\ell_2] \\ b_2 \mapsto B[\ell_4] \end{cases} \;\middle|\; \begin{cases} \ell_2 = \sigma(i) \\ \ell_4 = \sigma(i) \end{cases} \right\}$$

Dans la première valeur abstraite on aurait $a_1 = b_1$ et dans la seconde $a_2 = b_2$. Pour réaliser les compositions, en parallèle, il suffit de calculer l'union des deux valeurs abstraites après avoir renommé les variables. On va renommer a_1 et a_2 en a, b_1 et b_2 en b dans chacune des valeurs abstraites. Après renommage les deux valeurs vérifient $a = b$. Puis on calcule l'union des deux valeurs abstraites, qui donnera donc bien la disjonction des propriétés de a_1 et a_2 pour a et celle des propriétés de b_1 et b_2 pour b. L'équation $a = b$ est conservée dans l'union.

La fragmentation pour ce résultat peut être calculée de la manière suivante. Le fragment pour la variable a est logiquement l'union des fragments pour a_1 et a_2 de même que le fragment pour b sera l'union des fragments pour b_1 et b_2. Les relations entre indices doivent également être composées. Or on a la même relation entre indices dans les deux valeurs abstraite : on a aussi bien $\ell_1 = \ell_3$ pour la première que $\ell_2 = \ell_4$ impliquée par la seconde. On peut donc sans approximation conserver la même relation entre indices dans le résultat de la composition.

$$F(\sigma) = \left\{ r : \begin{cases} a \mapsto A[\ell_1] \\ b \mapsto B[\ell_2] \end{cases} \;\middle|\; \begin{cases} 1 \leq \quad \ell_1 \leq \sigma(i) \\ 1 \leq \quad \ell_2 \leq \sigma(i) \end{cases} \wedge\, l_1 = l_3 \right\}$$

On a bien $F = F_1 \cup F_2$ et il était donc correct de prendre l'union des deux valeurs abstraites.

L'idée de décomposer la valeur abstraite peut être intégrée à notre méthode d'analyse. Nous devons pour cela définir quand et comment effectuer la décomposition, et quand recomposer les différentes valeurs abstraites.

Nous commençons par décrire formellement le domaine des relations affines avant d'aborder sa combinaison avec les diagrammes de tranches et par la même occasion le cas général correspondant à notre exemple de la copie de tableau.

6.2 Domaine de fragmentation des relations entre indices affines

Le domaine de fragmentation des relations entre indices affines permet sélectionner des relations particulières entre les cellules désignées par un même choix de représentants. Il permet de ne mettre en relation que des cellules dont les indices vérifient des égalités affines. On ne cherche à sélectionner aucun fragment particulier, seulement des relations. On ne cherche pas non plus à préciser sur quel tableau ces relations s'appliquent : ce serait possible sans difficulté mais on doit de toute façon combiner ces relations à un domaine désignant des tranches qui réalisera déjà la sélection des tableaux. Les relations affines sont plus générales que les relations de translation, mais cette généralité n'ajoute pas de complexité et il suffira de conserver un système de la forme souhaitée.

Dans ce domaine, une fragmentation abstraite est un système d'équations affines. La concrétisation d'un tel système $\varphi(\ell_1, \ldots, \ell_n)$ pour l'ensemble de variables $\{s_1, \ldots, s_n\}$ sera :

$$\widehat{\gamma}(\varphi) : \sigma \mapsto = \left\{ r : \begin{cases} s_1 \mapsto A_1[\ell_1] \\ \vdots \\ s_n \mapsto A_n[\ell_n] \end{cases} \;\middle|\; \forall A_i \in \mathcal{A}, \varphi(\ell_1, \ldots, \ell_n) \right\}$$

La concrétisation n'est absolument pas symbolique : les choix de représentants désignés ne dépendent pas de σ. Ceci a pour conséquence de simplifier notablement la plupart des opérations du domaine de fragmentation décrites dans la section 4.2.3. D'une part l'interprétation des instructions et des gardes n'a absolument aucune conséquence sur la fragmentation. D'autre part, il n'y aura ni simplification, ni spécialisation de la fragmentation. Le développement du domaine se réduit donc à deux tâches : interpréter le critère de fragmentation et définir l'opération d'unification.

L'interprétation du critère nécessite l'interprétation des quatre instructions d'instrumentation définies section 4.1.2. Nous introduisons dans notre fragmentation les mêmes variables que dans le critère. Rappelons que ce n'était pas le cas pour les diagrammes qui pouvaient associer les mêmes tranches à deux variables distinctes, ou au contraire décomposer une même variable en plusieurs tranches.

En présence d'une singularisation faible ou forte, la variable singularisée est introduite sans relation dans le système d'équations. Si elle y était déjà, on doit retirer ses anciennes relations. Lorsque l'on rencontre une réinitialisation, on pourra retirer complètement la variable du système affine. La composition est la seule des quatre opérations qui nécessite un peu de travail.

Supposons que nous ayons à traiter la composition suivante :

$$s_1, s_2, \ldots, s_n \cup\!\!: s'_1, s'_2, \ldots, s'_n$$

Rappelons que cette instruction d'instrumentation permet d'ajouter les cellules représentées par $s'_1, \ldots, s'_n$ aux fragments représentés par $s_1, \ldots, s_n$. Dans la fragmentation résultat, la relation qu'on obtient entre $s_1, s_2, \ldots, s_n$ est la disjonction de leur relation originale et de la relation entre $s'_1, s'_2, \ldots s'_n$.

D'une part, on commence par interroger le domaine abstrait pour savoir s'il connaît une relation affine $\varphi_1(\ell'_1, \ldots, \ell'_n)$ entre les indices $\ell'_1, \ldots, \ell'_n$ de ces cellules représentées respectivement par $s'_1, \ldots, s'_n$. D'autre part, on regarde si on a déjà trouvé une relation $\varphi_2(\ell_1, \ldots, \ell_n)$ entre les indices $\ell_1, \ldots, \ell_n$ de ces cellules représentées respectivement par $s_1, \ldots, s_n$. Cela consiste à extraire du système affine les équations liant les indices de ces variables. Si les deux relations sont les mêmes, c'est à dire si $\varphi_1 = \varphi_2$ alors on peut conserver cette relation à l'issu de la composition. Par ailleurs, si l'un des fragments représentés par $s_1, \ldots, s_n$ est vide, alors n'importe quelle relation sera valide et on pourra également conserver φ_1. Pour notre critère, lors de la première itération d'une boucle, les fragments $s_1, s_2, \ldots, s_n$ seront toujours vides. Ainsi, on acceptera toujours la relation entre indices initiale entre les cellules représentées par $s_1, \ldots, s_n$.

Si on est dans aucun de ces deux cas, nous ne pouvons pas désigner exactement la relation qui lie ces variables. On devra donc approximer cette relation. Nous pourrions prendre une sur-approximation de cette relation en calculant l'union affine [1] des deux relations φ_1 et φ_2.

Terminons la définition de notre domaine de fragmentation par l'opération d'unification de deux fragmentations. Le problème est le même que celui de la composition. Si nous avons dans les deux systèmes à unifier les mêmes relations, alors on peut les conserver. Si une relation est présente dans l'un mais pas dans l'autre, c'est qu'on ne peut représenter exactement les relations désignées par le critère. La solution sera la même. On conserve les relations d'un système qui impliquent les fragments vides dans l'autre fragmentation. Puis on sur-approxime les relations restantes en calculant l'union affine des systèmes.

Pour terminer cette section, vérifions que nous obtenons bien ce qui était avancé sur l'exemple de la copie de tableau. Nous pouvons partir de la version instrumentée de ce programme pour le critère de fragmentation simplifié. (Cf. section 4.1.1)

Pour *i de* 1 *à n* **faire**
  $a' : \{A[i]\}, b' :: \{B[i]\}$
  $B[i] \leftarrow A[i]$
  $a, b \cup: a', b'$
  $a' : \emptyset, b' : \emptyset$

La première fois que l'on entre dans la boucle, les deux singularisations nous forcent à considérer les variables a' et b' dont nous ne chercherons pas à tirer une relation pour le moment. On reste donc avec un système d'équations vide dont la concrétisation est la suivante :

$$F_1(\sigma) = \left\{ r : \begin{cases} a' \mapsto A_1[\ell_1] \\ b' \mapsto A_2[\ell_2] \end{cases} \;\middle|\; A_i \in \mathcal{A} \right\}$$

Puis on doit interpréter la composition $a, b \cup: a', b'$. Les cellules $A[i]$ et $B[i]$ représentées par a' et b' sont évidemment de même indice, et on s'intéressera donc à la relation $\ell_1 = \ell_2$. Comme a

1. Un algorithme d'union affine peut être trouvé dans [Kar76]

et b ne sont pas encore définis, ils représentent des fragments vides et on conservera la relation que nous venons de trouver pour a et b :

$$F_2(\sigma) = \left\{ r : \begin{cases} a' \mapsto A_1[\ell_1] \\ b' \mapsto A_2[\ell_2] \\ a \mapsto A_3[\ell_3] \\ b \mapsto A_4[\ell_4] \end{cases} \;\middle|\; A_i \in \mathcal{A},\ \ell_3 = \ell_4 \right\}$$

Les réinitialisations nous invitent ensuite à retirer a' et b' de la fragmentation :

$$F_3(\sigma) = \left\{ r : \begin{cases} a \mapsto A_3[\ell_3] \\ b \mapsto A_4[\ell_4] \end{cases} \;\middle|\; A_i \in \mathcal{A},\ \ell_3 = \ell_4 \right\}$$

Ce qui termine la première itération sur la boucle. On doit unifier cette fragmentation à la fragmentation initiale dans laquelle tous les fragments sont vides, ce qui nous permet de conserver la fragmentation F_3.

Lors de la seconde itération, les singularisations nous ramènent à la fragmentation F_2. Pour traiter la composition $a, b \cup\!\!: a', b'$ on vérifie que la relation entre indices entre a' et b' est bien la même qu'entre a et b. Comme lors de la première itération, les deux cellules singularisées représentées par a' et b' ont le même indice $\ell_1 = \ell_2$. Or il y a entre a et b la même relation entre indices $\ell_3 = \ell_4$. On peut donc à l'issue de cette composition conserver la relation d'égalité entre les indices des cellules représentées par a et celles représentées par b et on retrouve la fragmentation F_3.

Les itérations suivantes ne changeront pas les fragmentations qui resteront F_2 avant la composition et F_3 après.

6.3 Combinaison des diagrammes de tranches et des relations affines

Nous cherchons maintenant à combiner les diagrammes de tranches et notre nouveau domaine de fragmentation. Cette combinaison est elle-même un domaine de fragmentation dont les opérations sont basées sur les opérations respectives de ces deux domaines.

Nous réalisons cette combinaison en deux temps. Nous traitons d'abord du cas simple où on ne cherche pas à décomposer la valeur abstraite. Dans un second temps nous généraliserons ce cas en décrivant quand et comment décomposer la valeur abstraite selon les relations trouvées.

Pour construire les fragmentations du domaine combinaison, on utilisera un *produit de fragmentation*. Le produit de deux fragmentations F_1 et F_2 est une fragmentation dans laquelle les variables de synthèse sont les couples de variables de synthèse de l'une et l'autre des fragmentations respectivement. L'intention est que chacun de ces couples de variables représente l'intersection des deux fragments représentés par ces variables.

Prenons l'exemple du programme suivant :

$A[0] \leftarrow 0$
Pour *i de* 1 *à n* **faire**
  $a'_1 : \{A[i]\}, a'_2 :: \{A[i+1]\}$
  $A[i+1] \leftarrow A[i] + 1$
  $a_1, b_1 \cup: a'_1, b'_1$
  $a'_1 : \emptyset, b'_1 : \emptyset$

Les diagrammes de fragmentation et les relations affines produiront les deux fragmentations F_1 et F_2 respectivement :

$$F_1(\sigma) = \left\{ r : \begin{cases} s_1 \mapsto A[\ell_1] \\ s_2 \mapsto A[\ell_2] \end{cases} \middle| \begin{cases} 1 \leq & \ell_1 \leq \sigma(i) \\ 2 \leq & \ell_2 \leq \sigma(i)+1 \end{cases} \right\}$$

$$F_2(\sigma) = \left\{ r : \begin{cases} s_3 \mapsto A_3[\ell_3] \\ s_4 \mapsto A_4[\ell_4] \end{cases} \middle| A_i \in \mathcal{A} \wedge \ell_4 = \ell_3 + 1 \right\}$$

On peut calculer leur produit

$$(F_1 \times F_2)(\sigma) = \left\{ r : \begin{cases} (s_1,s_3) \mapsto A[\ell_{1,3}] \\ (s_1,s_4) \mapsto A[\ell_{1,4}] \\ (s_2,s_3) \mapsto A[\ell_{2,3}] \\ (s_2,s_4) \mapsto A[\ell_{2,4}] \end{cases} \middle| \begin{cases} 1 \leq & \ell_{1,3} \leq \sigma(i) \\ 1 \leq & \ell_{1,4} \leq \sigma(i) \\ 2 \leq & \ell_{2,3} \leq \sigma(i)+1 \\ 2 \leq & \ell_{2,4} \leq \sigma(i)+1 \end{cases} \wedge \begin{cases} \ell_{1,4} = \ell_{1,3} + 1 \\ \ell_{1,4} = \ell_{2,3} + 1 \\ \ell_{2,4} = \ell_{1,3} + 1 \\ \ell_{2,4} = \ell_{2,3} + 1 \end{cases} \right\}$$

Ce produit est en général trop grossier pour ce que l'on veut en faire. Plusieurs variables de ce produit ne nous intéressent pas. Nous raffinons ce produit en sélectionnant les couples de variables à conserver. Un couple de variables nous intéresse s'il a été désigné par le critère de fragmentation, c'est à dire si ces deux variables de synthèse sont issues de la même variable du critère de fragmentation. Dans notre exemple s_1 et s_3 sont issues de la variable a_1 tandis que s_2 et s_4 sont issues de la variable a_2. On peut donc se limiter aux seules variables (s_1, s_3) et (s_2, s_4) dans le produit :

$$F(\sigma) = \left\{ r : \begin{cases} (s_1,s_3) \mapsto A[\ell_{1,3}] \\ (s_2,s_4) \mapsto A[\ell_{2,4}] \end{cases} \middle| \begin{cases} 1 \leq & \ell_{1,3} \leq \sigma(i) \\ 2 \leq & \ell_{2,4} \leq \sigma(i)+1 \end{cases} \wedge \ell_{2,4} = \ell_{1,3} + 1 \right\}$$

La fragmentation F est satisfaisante pour l'analyse de cet exemple puisqu'elle met en relation chaque cellule du tableau avec sa cellule suivante et permettra au domaine abstrait d'exprimer que la seconde vaut 1 de plus que la première. En pratique, pour sélectionner les couples, il faut pouvoir interroger les domaines de fragmentation pour savoir d'où viennent les fragments qu'ils désignent.

A partir de ce produit, on pourrait essayer d'écrire complètement toutes les opérations d'un domaine de fragmentation. On peut commencer par appliquer les opérations sur chacune des composantes du produit. Les deux domaines combinés génèreront leurs transformations élémentaires que nous devons traduire sur notre fragmentation produit.

Néanmoins, comme nous l'avons montré plus tôt (section 6.1), les transformations élémentaires ne sont pas assez puissantes en général pour traduire précisément les transformations sur la fragmentation produit. Pour contourner ce problème, nous allons suivre la piste proposée plus tôt et décomposer les valeurs abstraites.

Nous allons créer autant de sous-valeurs abstraites $\widetilde{X}_1, \ldots, \widetilde{X}_n$ que de relations entre indices découvertes. Nous utiliserons également une valeur abstraite supplémentaire $\widetilde{X}'$ pour conserver les propriétés valables quelles que soient les relations entre indices. La valeur abstraite $\widetilde{X}$ globale sera le produit de toutes celles-là.

$$\gamma(\widetilde{X}) = \gamma'(\widetilde{X}') \cap \gamma_1(\widetilde{X}_1) \cap \ldots \gamma_2(\widetilde{X}_n)$$

Toutes ces valeurs abstraites sont fragmentées par le domaine des diagrammes. On maintiendra donc une fragmentation F' issue du domaine des diagrammes. En outre, les valeurs $\widetilde{X}_i$ sont également fragmentées par des relations entre indices φ_i. Chaque $\widetilde{X}_i$ a donc une fragmentation propre qui est la combinaison, au sens défini au début de cette section, de F' et de la fragmentation associée à la relation φ_i. C'est parce que les fragmentations pour chacune des valeurs $\widetilde{X}_i$ sont distinctes que nous avons introduit autant de fonctions γ_i que de valeurs abstraites.

Nous noterons $(v_1, \ldots, v_n, \varphi_i)$ les fragmentations associées aux $\widetilde{X}_i$. Il s'agit de la fragmentation obtenue à partir des diagrammes en prenant les tranches associées à $v_1, \ldots, v_n$ et en liant les indices par la relation $\varphi_i(\ell_1, \ldots, \ell_n)$. Formellement, si F' est la fragmentation décrite par les diagrammes pour la valeur abstraite globale, on aura :

$$(v_1, \ldots, v_n, \varphi_i) = \left\{ r : \begin{cases} v_1 \mapsto A_1[\ell_1] \\ v_2 \mapsto A_2[\ell_2] \\ \vdots \\ v_n \mapsto A_n[\ell_n] \end{cases} \;\middle|\; \exists r' \in F' \begin{cases} r'(v_1) = A_1[\ell_1] & \\ \wedge \quad r'(v_2) = A_2[\ell_2] & \\ \quad \vdots & \wedge \ \varphi_i(\ell_1, \ldots, \ell_n) \\ \wedge \quad r'(v_n) = A_n[\ell_n] & \end{cases} \right\}$$

Parfois, la relation entre indices φ_i sera incompatible avec le choix des tranches, ce qui impliquera $(v_1, \ldots, v_n, \varphi_i) = \emptyset$. Dans cette situation, la valeur abstraite $\widetilde{X}_i$ associée à cette fragmentation doit être $\bot$. Dans le cas qui nous intéresse, les relations de translations, il est facile de tester si les tranches translatées ont une intersection non vide.

On commence l'analyse des programmes avec une seule sous-valeur abstraite, $\widetilde{X}' = \top$. On conduit l'analyse comme s'il s'agissait d'une analyse basée sur les diagrammes de tranches. Ceux-ci vont générer des transformations élémentaires de la fragmentation F' que nous répercutons sur $\widetilde{X}'$. Au moment opportun, on ajoutera des sous-valeurs abstraites $\widetilde{X}_i$. Pour poursuivre l'analyse, il faudra aussi évaluer l'influence des transformations élémentaires sur les $\widetilde{X}_i$.

Le moment opportun pour ajouter une nouvelle sous-valeur abstraite est celui que nous avons identifié en section 6.2 : lors de l'interprétation d'une composition. Si on doit traiter la composition

$$v_1, \ldots, v_n \cup\!\!: v'_1, \ldots, v'_n$$

on cherche une éventuelle relation entre indices affine entre $v'_1, \ldots, v'_n$. Dans le cas où il y a effectivement une relation φ_i, on crée une nouvelle valeur abstraite $\widetilde{X}_i$. On extrait de $\widetilde{X}'$ les propriétés qui lient les variables $v'_1, \ldots, v'_n$ de cette nouvelle valeur abstraite. Autrement dit, $\widetilde{X}_i$ est la projection de $\widetilde{X}'$ sur les variables $v'_1, \ldots, v'_n$. $\widetilde{X}_i$ sera associée à la fragmentation $(v'_1, \ldots, v'_n, \varphi_i)$.

Dès lors que nous ajoutons des sous-valeurs abstraites $\widetilde{X_i}$ nous introduisons la difficulté d'interpréter les transformations élémentaires des fragmentations auxquelles elles sont associées. Qu'arrive t-il quand les diagrammes demandent de composer ou fusionner une variable d'une des $\widetilde{X_i}$ avec d'autres variables qui n'y sont pas présentes ?

Traitons d'abord le cas plus facile de la fission et de la décomposition. Supposons que nous avons une valeur abstraite $\widetilde{X_i}$ sur la fragmentation $(s, v_1, \ldots, v_n, \varphi)$. Si les diagrammes fissionnent ou décomposent la variable s en une variable cible t, on peut construire une nouvelle valeur abstraite $\widetilde{X_j}$ à partir de $\widetilde{X_i}$ en substituant s par t. On associe à cette nouvelle sous-valeur abstraite la fragmentation $(t, v_1, \ldots, v_n, \varphi)$. La nouvelle valeur $\widetilde{X_j}$ traduira bien les propriétés de t qui sont identiques à celles de la variable originale.

Pour la composition, on ne pourra pas se contenter de créer une nouvelle sous-valeur abstraite. Supposons encore que nous avons une valeur abstraite $\widetilde{X_i}$ sur la fragmentation $(s_1, v_1, \ldots, v_n, \varphi)$. Si la variable s_1 est composée avec la variable s_2 en la variable t, on commence par chercher une valeur abstraite $\widetilde{X_j}$ sur la fragmentation $(s_2, v_1, \ldots, v_n, \varphi)$. Si on renomme dans ces deux valeurs abstraites s_1 et s_2 en t et qu'on calcule l'union des deux valeurs abstraites, alors on obtient effectivement une valeur abstraite décrivant correctement les propriétés de la composée t. Cette valeur abstraite union peut être ajoutée à notre ensemble de sous-valeurs abstraites avec la fragmentation

$$(t, v_1, \ldots, v_n, \varphi) = (s_1, v_1, \ldots, v_n, \varphi) \cup (s_2, v_1, \ldots, v_n, \varphi)$$

Il y a un cas particulier mais courant où la composition est plus simple. Si $(s_2, v_1, \ldots, v_n, \varphi) = \emptyset$ alors la fragmentation originale avec s_1 et celle avec t sont identiques. On peut donc directement utiliser $\widetilde{X_i}$ comme valeur abstraite pour la fragmentation avec t.

La fusion est similaire à la composition. Il faudra trouver une autre sous-valeur correspondante, mais cette fois-ci, au lieu de calculer leur union, on calculera leur intersection.

Enfin, lorsque le domaine des diagrammes décide de retirer une tranche, nous n'aurons d'autre choix que de supprimer la tranche en question de toutes les sous-valeurs abstraites. Après cette suppression, certaines sous-valeurs abstraites deviennent inutiles ou redondantes et on pourra simplifier l'ensemble des sous-valeurs abstraites.

Abordons maintenant le problème de l'unification des fragmentations. Supposons que nous ayons d'un coté les valeurs abstraites $\widetilde{X'}, \widetilde{X_1}, \ldots, \widetilde{X_n}$ dont le produit est $\widetilde{X}$ et de l'autre les valeurs abstraites $\widetilde{Y'}, \widetilde{Y_1}, \ldots, \widetilde{Y_m}$ dont le produit est $\widetilde{Y}$. Nous appliquons normalement l'algorithme d'unification des diagrammes, de sorte que $\widetilde{X'}$ et $\widetilde{Y'}$ utilisent la même fragmentation. Pour chacun des $\widetilde{X_i}$ en revanche, il faudra trouver une valeur $\widetilde{Y_j}$ qui partage la même fragmentation. Comme nous avons déjà unifié les diagrammes, les symboles de variables sont les mêmes, et on doit donc trouver deux sous-valeurs abstraites sur les mêmes variables et sur la même relation entre indices. Si $\widetilde{X_i}$ est basée sur la fragmentation $(v_1, \ldots, v_n, \varphi)$ il faudra trouver $\widetilde{Y_j}$ sur la même fragmentation. Il est toutefois possible que $(v_1, \ldots, v_n, \varphi)$ soit vide dans $\widetilde{Y}$, auquel cas on peut également prendre $\widetilde{Y_j} = \bot$. On répète la recherche pour tous les $\widetilde{X_i}$ et tous les $\widetilde{Y_j}$. Puis, s'il y en a, on supprime les $\widetilde{X_i}$ et $\widetilde{Y_j}$ qui ne pourraient pas trouver de correspondant. A la fin on a construit une liste de couples de sous-valeurs abstraites basées sur les mêmes fragmentations et sur lesquelles on peut donc appliquer les opérateurs binaires $\sqcup$, $\sqcap$, $\sqsubseteq$ ou $\triangledown$.

Nous pouvons déjà vérifier que notre méthode fonctionne sur l'exemple de la copie de tableau de la section 6.1.

Pour *i de* 1 *à n* **faire**
  $a' : \{A[i]\}, b' :: \{B[i]\}$
  $B[i] \leftarrow A[i]$
  $a, b \cup: a', b'$
  $a' : \emptyset, b' : \emptyset$

Nous commençons l'analyse avec une unique valeur abstraite $\widetilde{X'} = \top$. On entre dans la boucle et les deux singularisations nous forcent à ajouter deux variables a' et b' dans la fragmentation. Le domaine des diagrammes construit deux tranches singletons pour chacune de ces variables. La fragmentation à ce point est

$$F'(\sigma) = \left\{ r : \begin{cases} a' \mapsto A[\ell_1] \\ b' \mapsto B[\ell_2] \end{cases} \middle| \ \ell_1 = \ell_2 = 1 \right\}$$

Le domaine des diagrammes demande en même temps à ce que ces deux variables soient introduites dans la valeur abstraite $\widetilde{X'}$. On peut ensuite appliquer la sémantique abstraite de l'affectation. La valeur abstraite $\widetilde{X'}$ devient

$$\widetilde{X'} \equiv a' = b'$$

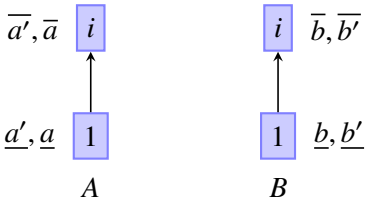
Lors de la composition qui suit, et puisque a et b sont encore indéfinies dans la fragmentation, on cherche une relation entre indices entre la cellule représentée par a' et celle représentée par b'. En consultant F', on vérifie que a' et b' représentent des cellules de même indice $\ell_1 = \ell_2$. Pour cette relation, on crée une nouvelle valeur abstraite $\widetilde{X}_1$ obtenue par projection de $\widetilde{X'}$ sur a', b', puis on renomme a' et b' en a et b puisque ce sont bien les cibles de la composition.

$$\widetilde{X}_1 \equiv a = b$$

On associe à cette nouvelle sous-valeur abstraite la fragmentation

$$F_1 = (a, b, \ell_1 = \ell_2)$$

Dans les diagrammes, la composition n'a aucune conséquence. Les variables a et b étaient pour le moment indéfinies et donc pour les diagrammes considérés comme représentant des fragments vides. Par conséquent, leur composition n'a eu pour conséquence qu'une simple annotation des diagrammes pour noter que la tranche $A[i]$ correspond aussi bien à la variable a qu'à la variable a' et que la tranche $B[i]$ correspond aussi bien à b qu'à b'.



Les deux réinitialisations qui suivent retirent les décorations associées à a' et b' dans les diagrammes, toujours sans conséquence sur la fragmentation hormis un éventuel renommage des variables pour conserver la cohérence avec le critère de fragmentation. La fragmentation F' à cette fin d'itération est

$$F'(\sigma) = \left\{ r : \begin{cases} a \mapsto A[\ell_1] \\ b \mapsto B[\ell_2] \end{cases} \middle| \ \ell_1 = \ell_2 = 1 \right\}$$

En fin de boucle pour le moment, on a donc deux valeurs abstraites identiques $\widetilde{X'} \equiv \widetilde{X_1}$ associées à deux fragmentations elles aussi identiques puisque la relation $\ell_1 = \ell_2$ est déjà impliquée dans F'.

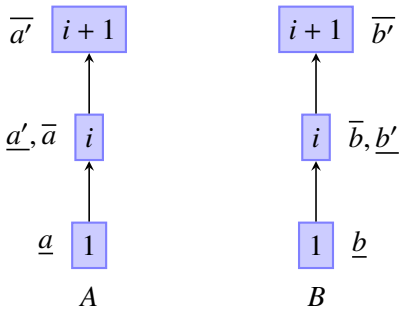
De retour en tête de boucle, il faut unifier les fragmentations avant de pouvoir calculer l'union de la valeur abstraite de fin d'itération et celle d'entrée de boucle. On commence par unifier les diagrammes. On a en entrée de boucle des diagrammes vides et $i = 1$ tandis qu'on avait en fin d'itération les tranches $A[1]$ et $B[1]$ et $i = 2$. L'unification des diagrammes donnera les tranches $A[1..i[$ et $B[1..i[$. Ces deux tranches sont vides en entrée de boucle quand $i = 1$ et par conséquent les fragmentations F' et F_1 sont également vides en entrée de boucle. On peut donc conserver $\widetilde{X}$ et $\widetilde{X_1}$ inchangées.

On repart donc avec la fragmentation :

$$F'(\sigma) = \left\{ r : \begin{cases} a \mapsto A[\ell_1] \\ b \mapsto B[\ell_2] \end{cases} \;\middle|\; \begin{cases} & 1 \le \ell_1 < \sigma(i) \\ \wedge & 1 \le \ell_2 < \sigma(i) \end{cases} \right\}$$

Mais avant d'entrer de nouveau dans la boucle, on doit élargir la valeur abstraite. On élargit logiquement chacune des sous-valeurs abstraites, sans effet sur les variables a et b : elles étaient indéfinies à l'itération précédente et on conserve leurs propriétés à l'élargissement. En revanche, cela a des effets sur la variable d'indice i qui était soit 1 soit 2. Après élargissement, on considère qu'elle peut prendre toute valeur supérieure ou égale à 1. En conséquence, la fragmentation $F_1 = (a, b, \ell_1 = \ell_2)$ est désormais plus fine que F'. Cela dit, on conserve encore l'égalité $a = b$ dans $\widetilde{X'}$ puisque elle était valide avant l'élargissement.

Traitons la seconde itération de la boucle. Après les deux singularisations, les diagrammes définissent quatre tranches $A[1..i[$, $A[i]$, $B[1..i[$ et $B[i]$ associées respectivement à a, a', b et b'.



La fragmentation F' devient

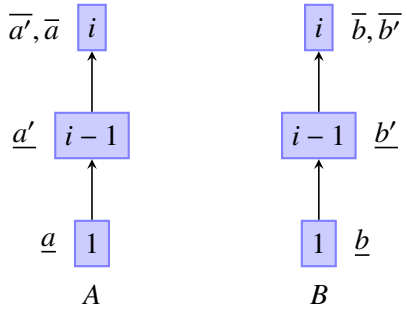
$$F'(\sigma) = \left\{ r : \begin{cases} a \mapsto A[\ell_1] \\ b \mapsto B[\ell_2] \\ a' \mapsto A[\ell_3] \\ b' \mapsto B[\ell_4] \end{cases} \;\middle|\; \begin{cases} & 1 \le \ell_1 < \sigma(i) \\ \wedge & 1 \le \ell_2 < \sigma(i) \\ \wedge & \ell_3 = \ell_4 = \sigma(i) \end{cases} \right\}$$

L'affectation qui suit est une affectation forte sur a' qui ne touche aucune tranche. Elle laisse donc inchangée $\widetilde{X_1}$ et modifie $\widetilde{X'}$ pour y intégrer l'égalité entre a' et b' :

$$\widetilde{X'} \equiv a = b \wedge a' = b'$$

Après l'affectation, la composition $a, b \cup: a', b'$, comme lors de la première itération, on introduit une nouvelle valeur abstraite $\widetilde{X_2}$ sur la fragmentation $(a, b, \ell_1 = \ell_2)$. L'interprétation

de cette composition dans les diagrammes n'a encore une fois pour effet que de changer la décoration.



En revanche, les réinitialisations qui suivent vont retirer les décorations pour a' et b' et donc les sommets $i-1$ des diagrammes. Cela a pour conséquence de composer les tranches $A[1..i[$ avec $A[i]$ et $B[1..i[$ avec $B[i]$ avant de supprimer chacune de ces tranches composantes. Nous noterons a'' et b'' les tranches $A[1..i]$ et $B[1..i]$ le temps de la composition, avant d'utiliser à nouveau a et b, afin de bien distinguer ces variables avant et après la composition.

Sur $\widetilde{X'}$ cette opération implique la perte totale de toute information : la variable a n'a aucune propriété commune avec a', tout comme b n'a aucune propriété commune avec b'. On ne trouve alors aucune propriété sur a'' et b'' et on perd le reste des propriétés lorsque l'on supprime a, a', b et b'. A ce point $\widetilde{X'} = \top$.

Nous devons également répercuter ces deux compositions sur $\widetilde{X_1}$ et $\widetilde{X_2}$. Ces deux valeurs abstraites sont définies sur les fragmentations $(a, b, \ell_1 = \ell_2)$ et $(a', b', \ell_1 = \ell_2)$ respectivement. Commençons par la composition de a et a' en a''. Si on substitue a par a' et inversement dans les deux fragmentations on obtient des fragmentations vides

$$(a', b, \ell_1 = \ell_2) = (a, b', \ell_1 = \ell_2) = \emptyset$$

En effet a' représente $A[1..i[$ et b représente $B[i]$, et il n'est donc pas possible de trouver un couple de cellules de l'un et l'autre des fragments dont l'indice est égal. Le cas de a et b' est symétrique. Par conséquent, on peut prendre $\widetilde{X'_1} \equiv a'' = b$ obtenue par renommage de a en a'' dans $\widetilde{X_1}$ pour $(a'', b, \ell_1 = \ell_2)$. De même, on peut prendre $\widetilde{X'_2} \equiv a'' = b'$ obtenue par renommage de a' en a'' dans $\widetilde{X_2}$ pour $(a'', b', \ell_1 = \ell_2)$. Une fois a et a' composées, les deux variables sont retirées de la fragmentation, nous forçant à abandonner $\widetilde{X_1}$ et $\widetilde{X_2}$ et à ne conserver que $\widetilde{X'_1}$ et $\widetilde{X'_2}$

Passons maintenant à la composition de b et b' en b''. Le cas est différent du précédent. Si on substitue b par b' dans la première fragmentation $(a'', b, \ell_1 = \ell_2)$, on ne tombe plus sur une fragmentation vide. En revanche, on retrouve la seconde fragmentation. On a bien

$$(a'', b'', \ell_1 = \ell_2) \;=\; (a'', b, \ell_1 = \ell_2) \cup (a'', b', \ell_1 = \ell_2)$$

et on peut donc ajouter une nouvelle valeur abstraite $\widetilde{X_3}$ calculée comme l'union de $\widetilde{X'_1}$ et de $\widetilde{X'_2}$ après renommage de b et b' en b''. On a $\widetilde{X_3} \equiv a'' = b''$ qu'on associe à la fragmentation $(a'', b'', \ell_1 = \ell_2)$. Après quoi, le retrait des variables b' et b'' nous force à abandonner $\widetilde{X'_1}$ et $\widetilde{X'_2}$ pour ne conserver $\widetilde{X_3}$.

On termine donc la boucle avec

$$\widetilde{X'} \equiv \top \qquad \text{et} \qquad \widetilde{X_3} \equiv a = b$$

avec a représentant la tranche $A[1..i[$, b représentant $B[1..i[$ et $\widetilde{X_3}$ associée à la fragmentation $(a, b, \ell_1 = \ell_2)$. Ceci suffit à exprimer l'invariant de boucle recherché. Une nouvelle itération

donnera lieu aux mêmes calculs et on produira le même résultat, prouvant effectivement que celui-ci est un invariant.

De nombreuses améliorations peuvent être apportées à notre méthode. D'abord, un certain nombre de réductions sont possibles entre les sous-valeurs abstraites. Si entre deux sous-valeurs abstraites, la fragmentation de l'une est plus fine que celle de l'autre, alors on peut renforcer la première avec la seconde. En particulier, il est toujours possible de renforcer les X_i avec X'.

En outre, il est également possible d'améliorer les unifications. Au lieu de chercher à associer des sous-valeurs abstraites basées sur la même fragmentation, on peut chercher à associer des valeurs abstraites dont la fragmentation est commune sur un sous ensemble de variables. C'est à dire que si on peut trouver deux fragmentations dont les projections sur un sous ensemble de variables coïncident, alors on peut projeter les sous-valeurs abstraites associées sur cet ensemble de variables afin de les rendre compatibles. La technique peut également être utilisée pour améliorer la précision lors des compositions ou fusions de tranches.

6.4 Conclusion

La solution que nous proposons est une application des techniques déjà développées dans d'autres travaux pour la recherche de propriétés relationnelles. Il s'agissait donc d'adapter ces techniques pour les diagrammes de tranches et pour notre critère de fragmentation.

L'intérêt de se limiter aux relations de translation est qu'il est facile de savoir quand ces relations sont vides : il suffit de vérifier si les translations des tranches de la relations ont une intersection vide ou non. Pour d'autres types de relations, le test du vide peut être plus subtil. Lorsque les coefficients sont différents de 1 ou -1, le test du vide peut tirer profit de propriétés de congruences. Par exemple, considérons la relation $\ell_1 = 2\ell_2$ où où ℓ_1 et ℓ_2 sont les indices utilisés pour les cellules des tranches $A[1..i[$ et $A[j]$ respectivement. Cette relation est vide si et seulement si j est impair.

On pourrait également étendre le domaine de fragmentation des relations affines aux fragmentations symboliques. Ceci permettrait de considérer des équations de la forme $\ell_1 = n - \ell_2$ où n est une variable du programme. Cette relation serait utile à l'analyse d'un programme qui inverse l'ordre des éléments d'un tableau.

7 Adaptation des domaines abstraits à la synthèse de propriétés

Les chapitres 3 et 4 fixent un cadre pour l'analyse des programmes manipulant des structures de données. Ce cadre est contraignant pour les domaines abstraits qui doivent être adaptés en conséquence. Ce chapitre traite de cette adaptation.

Elle peut être décomposée en trois tâches. Tout d'abord, le processus de synthèse requiert la modification des fonctions d'abstraction et de concrétisation. Dans le cas classique, on abstrait des ensembles d'états concrets. Dans notre système on doit abstraire des ensembles d'états de synthèse. La différence essentielle entre un état concret et un état de synthèse, c'est que les états de synthèse d'un même ensemble n'ont pas tous le même domaine de définition. En particulier un état de synthèse peut être indéfini en certaines variables, lorsque ces variables représentent des fragments vides.

Comment abstraire un ensemble qui contient un tel état de synthèse ? Supposons qu'on ait démontré que toute cellule d'un fragment $A[1..i]$ d'un tableau A ait une valeur inférieure à celle d'une variable x. Peu nous importe que le fragment $A[1..i]$ soit vide ou non pour affirmer cette propriété. Que i soit supérieur à 1 ou non, on peut toujours dire que toutes les cellules du fragment ont une valeur supérieure à x. Seulement, lorsque le fragment est vide, les choix de représentants sont indéfinis en ce fragment. La manière naturelle d'abstraire la propriété serait donc d'ignorer les états de synthèse dans laquelle la variable symbolisant le fragment est indéfinie. Ainsi, de manière générale, le domaine abstrait prétendra vraie une propriété sur un ensemble de variables pour peu que cette propriété soit vraie sur les seuls états de synthèse où toutes ces variables sont définies, c'est à dire sur les seuls états où on peut effectivement la vérifier.

En pratique, ceci nous force à réécrire les fonctions d'abstraction et de concrétisation. La fonction d'abstraction reste certainement identique dans le cas où les états de synthèse sont toujours définis partout. Il s'agit donc d'étendre ces fonctions aux cas des fragments vides.

La transformation de la correspondance de Galois ainsi produite va forcément avoir des effets sur les domaines. La fonction de réduction, composée de l'abstraction et de la concrétisation est forcément impactée. Reprenons l'exemple précédent. Si en plus d'avoir prouvé $A[1..i] < x$ on a également prouvé $A[1..i] > y$, il ne faut à aucun prix en déduire que y est inférieur à x. La propriété $x < y$ serait effectivement vraie et dérivable lorsque le fragment $A[1..i]$ n'est pas vide. Puisque le fragment est non vide, on peut prendre n'importe quelle cellule du fragment et étant donné que sa valeur est à la fois inférieure à x et supérieure à y on peut en déduire la propriété. Si le fragment est vide en revanche on n'a pas de telle cellule et la déduction de la propriété nous est impossible par ce moyen.

La deuxième tâche pour l'adaptation des domaines abstraits est l'implémentation des transformations élémentaires (Cf. section 4.2.4). Celles-ci permettent au domaine abstrait de refléter les modifications de la fragmentation sur les valeurs abstraites. Au nombre de six, leur implantation est assez directe et est en fait un exercice classique dans le champ de l'analyse par interprétation

abstraite de programme avec structures de données. La librairie de domaines abstraits Apron [JM09] implémente déjà certaines de ces opérations, et les autres sont tout à fait similaires.

Enfin, la dernière tâche consiste à adapter la sémantique abstraite. Ceci se limite à implémenter une nouvelle opération, l'affectation faible (Cf. section 3.6). Rappelons que cette opération nous permet de traiter les cas où une affectation touche une cellule qui peut selon les états appartenir ou non à un fragment fixé. Dans certains états, ce fragment est donc modifié, dans d'autres non et on ne peut conserver que les propriétés vraies quelle que soit la situation.

Nous commencerons par décrire le cas des domaines abstraits non relationnels que l'on peut facilement et systématiquement adapter. Puis nous traiterons du cas général en proposant des pistes sur le moyen d'adapter les domaines relationnels. Cette adaptation n'est pas systématique et doit être réalisée pour chaque domaine. Enfin, une section sera consacrée à l'exemple de l'adaptation du domaine abstrait des zones.

7.1 Adaptation des domaines non-relationnels

Tout le travail d'adaptation se simplifie considérablement dans le cas des domaines non-relationnels. On peut remarquer que le problème de réduction cité en introduction n'existe pas dans le cas non-relationnel. Ce qui rend le cas des domaines non-relationnels simples c'est le fait de pouvoir faire la distinction sur la vacuité de chaque fragment individuellement. Il n'est jamais nécessaire de considérer ce que cette vacuité implique sur les éventuelles relations que ces fragments ont entre eux.

Cette section suffira à décrire ce qu'il est nécessaire de développer sur les domaines non-relationnels. Pour illustrer les faits évoqués, nous prendrons comme exemple le domaine des intervalles [CC76] dont nous rappelons la définition en section 2.3.2

Une valeur abstraite $\widetilde{X}$ dans un domaine non-relationnel est la donnée pour chaque variable v d'une abstraction $\widetilde{X}(v)$ indépendante. Chacune de ces abstractions représente l'ensemble des valeurs atteignables par la variable associée. On combine les abstractions de chaque variable pour former la valeur abstraite. On étend les fonctions d'abstraction α et de concrétisation γ à cette combinaison. La fonction d'abstraction projette la valeur concrète sur chacune des variables avant d'abstraire ces ensembles indépendamment tandis que la fonction de concrétisation recombine les ensembles de valeurs atteignables :

$$\alpha(X) = \widetilde{X} : v \mapsto \alpha(\{\, \sigma(v) \mid \sigma \in X \,\})$$
$$\gamma(\widetilde{X}) = \left\{ \sigma \;\middle|\; \forall v,\ \sigma(v) \in \gamma(\widetilde{X}(v)) \right\}$$

Revenons maintenant à la synthèse des propriétés des structures de données. Ce que nous devons faire c'est étendre la définition de α afin qu'elle s'applique aux états de synthèse. Il faut interpréter les cas où les états sont indéfinis pour une variable donnée. Comme présenté dans l'introduction de ce chapitre, on se contentera d'ignorer ces cas. Si on doit associer une abstraction à un fragment, ce sera l'abstraction des valeurs prises par n'importe quelle cellule du fragment lorsque ce fragment est non vide.

Formellement, cela ne change pas grand chose à la fonction α. Elle abstraira toujours de la même manière l'ensemble des valeurs $\sigma(v)$ pour tous les σ définis en v. Ainsi, notre nouvelle fonction $\widetilde{\alpha}$ sera :

$$\widetilde{\alpha}(\overline{X}) = \widetilde{X} : \overline{v} \mapsto \widetilde{\alpha}(\left\{ \overline{\sigma}(\overline{v}) \;\middle|\; \overline{\sigma} \in \overline{X} \right\})$$

Ce qui a changé par rapport à α c'est que cet ensemble de valeurs peut maintenant être vide même si X ne l'est pas : il suffit que tous les états de X soient indéfinis sur v. Cela arrive lorsqu'une variable représente un fragment vide dans tous les états.

Pour les intervalles, cela nous demande simplement d'associer un intervalle vide chaque fois que l'ensemble de valeurs à abstraire est vide. Il y a dans notre domaine plusieurs manières de désigner un intervalle vide. Il suffit que ce soit un couple (l, u) avec $l > u$. Il est classique lorsque l'on introduit le domaine des intervalles de regrouper tous les intervalles vides sous le même élément $\bot$ du treillis abstrait.

Néanmoins, pour que nous puissions dans la section 7.3 établir un parallèle entre le domaine des intervalles et celui des zones nous prendrons une autre formalisation. Nous choisissons de conserver distinctes dans le treillis abstraits toutes les versions de l'intervalle vide. Lorsque la fonction $\widetilde{\alpha}$ doit abstraire un ensemble vide de valeur, nous prenons néanmoins soin d'utiliser l'élément minimum pour $\sqsubseteq$, l'intervalle $(+\infty, -\infty)$ qui n'appartient pas au domaine tel que nous l'avons défini. Nous devons généraliser la définition du domaine pour qu'il autorise $+\infty$ comme borne inférieure et $-\infty$ comme borne inférieure. Le treillis abstrait est désormais

$$(\mathbb{Z} \cup \{-\infty, +\infty\})^2$$

L'intérêt de cette définition c'est qu'il n'est pas nécessaire de modifier les opérateurs $\sqcup$, $\sqcap$ et $\sqsubseteq$. On peut vérifier que $(+\infty, -\infty)$ est bien le plus petit élément pour $\sqsubseteq$ et a fortiori l'élément neutre de $\sqcup$ ainsi que l'élément absorbant de $\sqcap$. Toute autre écriture de l'ensemble vide dans ce domaine n'est pas neutre pour $\sqcup$.

Pour l'analyse, l'existence d'une abstraction représentant un ensemble de valeurs vide est cohérente. Si on doit calculer l'union de deux valeurs abstraites, l'une dans laquelle un fragment est toujours vide, l'autre dans laquelle le fragment n'est pas toujours vide, il faut que les propriétés de ce fragment dans l'union soient les propriétés du fragment dans la seconde valeur abstraite.

La fonction γ quant à elle nécessite quelques modifications si on souhaite conserver une correspondance de Galois. Rappelons que $\widetilde{\gamma}(\widetilde{X})$ doit, dans une correspondance de Galois, donner le plus grand ensemble d'états de synthèse dont l'abstraction est $\widetilde{X}$. Il va falloir inclure tous les états de synthèse qu'il est possible d'obtenir en restreignant le domaine de définition, puisque ceux-ci ne changeront pas l'abstraction. Notre fonction $\widetilde{\gamma}$ doit être la suivante.

$$\widetilde{\gamma}(\widetilde{X}) = \left\{ \overline{\sigma} \;\middle|\; \forall \overline{v},\ \overline{\sigma}(\overline{v}) \in \widetilde{\gamma}(\widetilde{X}(\overline{v})) \ \vee\ \overline{\sigma}(\overline{v}) \text{ indéfini} \right\}$$

Lorsque $\widetilde{\gamma}(\widetilde{X}(\overline{v}))$ est un ensemble vide, par exemple parce que $\widetilde{X}(v)$ est l'intervalle vide, l'ensemble d'états de synthèse produit n'est pas forcément vide. Seulement, il ne contient que des états $\overline{\sigma}$ indéfinis en v.

Ce dernier point a des implications pour le calcul de la réduction. Auparavant, la réduction sur le domaine des intervalles consistait essentiellement à dire « si un intervalle est vide, alors l'ensemble d'états atteignables est vide. » Désormais, ce n'est plus nécessairement le cas. Rappelons nous cependant que le domaine de fragmentation a pour rôle de nous indiquer le type des variables et en particulier de nous indiquer les variables qui représentent des fragments qui ne peuvent pas être vides. Dans ce dernier cas, il ne peut pas être associé à un intervalle vide, à moins qu'effectivement, l'ensemble d'états atteignables soit lui même vide.

Cela se vérifie lorsque on applique la fonction $\overline{\gamma}$ définie en section 3.4. Cette fonction va chercher à construire des états concrets, mais ne trouvera aucun état de synthèse dans lequel chercher des valeurs pour les cellules de ce fragment non-vide. Par conséquent, elle ne construira aucun état concret et on pourra conclure à la vacuité de la valeur concrète.

L'opérateur d'élargissement ne nécessitera généralement aucune modification, mais on prendra soin de vérifier que son comportement est cohérent vis à vis des ensembles vides valeurs. Nous considèrerons qu'il n'est pas nécessaire d'élargir l'abstraction lorsque celle-ci représente un ensemble vide. En effet, lorsque ceci arrive, c'est que le fragment vient d'être créé. On ne connaît pas encore l'évolution de son ensemble de valeurs et il serait vain de tenter de chercher un élargissement à ce stade.

Dans les intervalles, on étend naturellement la définition de $\triangledown$:

$$(a,b) \triangledown (c,d) = \left(\begin{cases} c \text{ si } a = +\infty \\ -\infty \text{ si } c < a \\ a \text{ sinon} \end{cases} , \begin{cases} d \text{ si } b = -\infty \\ +\infty \text{ si } d > b \\ b \text{ sinon} \end{cases} \right)$$

Les transformations élémentaires des fragmentations ont une expression simple sur les domaines abstraits non relationnels. Elles peuvent être exprimées en termes d'union et d'intersection des abstractions individuelles.

- Le fragment composé hérite des propriétés vraies des deux composants. Il suffira donc de calculer la borne supérieure des deux abstractions de ces fragments. Si on est parvenu à déterminer deux intervalles de valeurs pour deux fragments $A[1..i[$ et $A[i..n]$ alors il suffira de calculer l'union de ces intervalles pour obtenir un intervalle valide pour leur composition $A[1..n]$.
- Les fragments produits de décomposition ou de fissions héritent des propriétés du fragment original et on se contentera de dupliquer l'abstraction pour l'associer aux différents produits.
- Enfin, la fusion combine deux fragments identiques en un seul. Les propriétés du fragment fusionné sont obtenues par intersection des abstractions des deux fragments originaux. Si on a découvert deux intervalles de valeur pour les fragments $A[1..i[$ et $A[1..j[$ et qu'une garde nous apprend $i = j$ alors on ne gardera que les états où les deux fragments sont identiques et on pourra conclure que l'intervalle de valeur de ces fragments est l'intersection des intervalles qu'on avait obtenus.

L'affectation faible peut également se traiter avec l'opérateur d'union sur l'abstraction d'une variable, à la manière dont on l'avait exprimée en section 3.6. Un fragment cible d'une affectation faible est selon les états ou bien modifié ou bien laissé intact. On calculera l'union de l'abstraction initiale et de l'abstraction modifiée par la sémantique abstraite de l'affectation.

7.2 Adaptation des domaines relationnels

Pour les domaines relationnels, les simplifications du cas non-relationnel ne sont plus possibles. Il n'y aura pas de méthode automatique pour transformer les fonctions d'abstraction et de concrétisation. Il faudra les redéfinir et par conséquent réécrire la réduction et prouver que les nouveaux opérateurs abstraits sont corrects pour ces fonctions. Néanmoins, il existe un moyen de décrire les transformations élémentaires et l'affectation faible à l'aide des opérateurs ensemblistes abstraits et en s'inspirant du cas non-relationnel.

Notations et conventions. La définition des transformations élémentaires pour les domaines relationnels reposent sur deux opérations courantes sur les domaines abstraits. La première est le renommage d'une dimension. On notera $\widetilde{X}[v \to v']$ le renommage de la variable v en la variable v' dans la valeur abstraite $\widetilde{X}$. L'ancienne variable v reste non-contrainte. Cette opération est définie formellement dans le treillis concret :

$$\widetilde{\gamma}(\widetilde{X}[v \to v']) = \left\{ \sigma : \begin{cases} w & \mapsto \sigma'(w) \text{ si } w \neq v, v' \\ v' & \mapsto \sigma'(v) \\ v & \mapsto x \in \mathbb{V} \end{cases} \middle| \; \sigma' \in \widetilde{\gamma}(\widetilde{X}) \right\}$$

où $\mathbb{V}$ est l'ensemble des valeurs possibles. La seconde opération est la projection qui permet d'oublier toute information associée à une variable. On notera $\widetilde{X}[\not{v}]$ l'oubli de la variable v dans la valeur abstraite $\widetilde{X}$.

$$\widetilde{\gamma}(\widetilde{X}[\not{v}]) = \left\{ \sigma : \begin{cases} w & \mapsto \sigma'(w) \text{ si } w \neq v \\ v & \mapsto x \in \mathbb{V} \end{cases} \middle| \; \sigma' \in \widetilde{\gamma}(\widetilde{X}) \right\}$$

La composition. Lorsque le domaine de fragmentation affirme qu'un fragment est la composition de deux autres, il faut répercuter le fait que les propriété du fragment composé sont la disjonction des propriété des deux arguments. Pour les propriétés non relationnelles, on se servait de l'opérateur d'union pour calculer cette disjonction. Pour reproduire cela ici, il faut faire attention à ce que la disjonction ne s'applique qu'aux variables concernées par la composition. Ceci peut s'exprimer à l'aide de substitutions de la manière suivante. Si on compose les fragments associés aux variables s_1 et s_2 en un fragment associé à la variable t, la valeur abstraite $\widetilde{X}$ après composition devient

$$\widetilde{X} \sqcap \left(\widetilde{X}[s_1 \to t] \sqcup \widetilde{X}[s_2 \to t]\right)$$

On peut vérifier que la composition ne nous apprend rien sur les relations entre s_1, s_2 et t.

La décomposition et la fission. Les composants d'un fragment décomposé, tout comme les produits de la fission doivent hériter des propriétés du fragment source décomposé ou fissionné. Pour les domaines non relationnels, il s'agissait de dupliquer l'abstraction associée aux variables. Pour chaque variable t représentant le résultat d'une fission ou d'une décomposition et pour la variable s représentant le fragment source, la valeur abstraite devient

$$\widetilde{X} \sqcap \widetilde{X}[s \to t]$$

Encore une fois, cette contrainte ne nous apprend aucune relation entre s et t.

La décomposition et la fission diffèrent dans un cas particulier. Si le fragment source est sous-scalaire, c'est à dire s'il contient au plus une cellule, on peut dériver plus de propriétés de la transformation élémentaire.

- Une décomposition implique que la somme des tailles des fragments composants soit égale à la taille du fragment décomposé. Comme la taille du fragment décomposé est au plus un, on peut en déduire des conséquences. On sait qu'il est impossible que les deux produits de la décomposition soient simultanément non-vides. Lorsque l'un d'entre eux est non vide, il est obligatoirement un singleton égal au fragment original. Il n'est pas possible de trouver un couple d'éléments pris respectivement dans chacun des deux composants. Par conséquent le domaine abstrait est libre de dériver toute propriété liant ces deux variables. Nous en verrons un exemple dans la section 7.3.
- La fission duplique un fragment en un fragment identique. S'il s'agit d'un singleton, on aura deux fois le même singleton et on pourra dire sans risque que la valeur du premier singleton est égale à celle du second. Il est possible pour le domaine abstrait de conclure à l'égalité des deux variables représentant ces fragments.

Ces deux cas justifient d'une part qu'on sépare la décomposition de la fission dans les transformations élémentaires, d'autre part qu'on ait besoin de différencier les fragments sous-scalaires des autres.

La fusion. La fusion nous demande de renforcer les propriétés d'un fragment avec les propriétés acquises sur un autre fragment dont on découvre qu'il est le même. La fusion de la variable s en la variable t a la même expression que la fission :

$$\widetilde{X} \sqcap \widetilde{X}[s \to t]$$

La différence avec la fission se situe au niveau de l'implémentation de ces opérations : quand on opère une fission c'est pour introduire une nouvelle variable, tandis que la fusion a pour but d'en faire disparaitre une. Le calcul d'une intersection ne sera nécessaire que dans le second cas.

L'affectation faible. L'affectation faible $t \leftsquigarrow exp$ d'une variable t par une expression exp s'exprime très bien de manière générale et à l'aide de l'affectation forte. Ceci a déjà été abordé dans la section 3.6. Il s'agit d'une disjonction de cas : où bien t est affectée, ou bien elle ne l'est pas. On pourra donc calculer le résultat de l'affectation en appliquant la sémantique suivante :

$$[t \leftsquigarrow exp](\widetilde{X}) = \widetilde{X} \sqcup [t \leftarrow exp](\widetilde{X})$$

Il est difficile d'aller plus loin en généralités pour les domaines relationnels. Il y a toujours une part du travail d'adaptation qui doit être fait spécifiquement pour chaque domaine abstrait. Bien que l'effet des diverses transformations élémentaires s'expriment comme des intersections et des unions elles sont généralement plus simples et demandent moins de calcul que ces opérateurs ensemblistes. Nous voudrons donc souvent en implémenter des versions optimisées.

La section suivante donne un exemple d'adaptation d'un domaine abstrait relationnel, celui des zones.

7.3 Application au domaine des zones

Dans cette section, nous appliquerons notre technique d'adptation au domaine abstrait des zones [Dil89] (Cf. section 2.3.3), abstraites par des matrices de bornes de différences. Nous

rappelons que dans ce domaine, les valeurs abstraites sont des matrices dans

$$(\mathbb{Z} \cup \{+\infty\})^{(n+1)^2}$$

que les fonctions d'abstraction et de concrétisation sont

$$\alpha(X) = \left(\min\left\{ c \in \mathbb{Z} \cup \{+\infty\} \;\middle|\; \forall \sigma \in X,\ \sigma(v_i) - \sigma(v_j) \leq c \right\}\right)_{i,j}$$

$$\gamma(\widetilde{X}) = \left\{ \sigma \;\middle|\; \forall v_i, v_j \in \mathcal{V}\ \sigma(v_i) - \sigma(v_j) \leq c_{i,j} \right\}$$

et que les opérateurs binaires sont

$$(c_{i,j})_{i,j} \sqsubseteq (d_{i,j})_{i,j} \Leftrightarrow \forall i, j\ c_{i,j} \leq d_{i,j}$$

$$(c_{i,j})_{i,j} \sqcup (d_{i,j})_{i,j} = \left(max\{c_{i,j}, d_{i,j}\}\right)_{i,j}$$

$$(c_{i,j})_{i,j} \sqcap (d_{i,j})_{i,j} = \left(min\{c_{i,j}, d_{i,j}\}\right)_{i,j}$$

$$(c_{i,j})_{i,j} \triangledown (d_{i,j})_{i,j} = \left(\begin{cases} +\infty \text{ si } d_{i,j} > c_{i,j} \\ c_{i,j} \text{ sinon} \end{cases}\right)_{i,j}$$

Adaptation de la correspondance de Galois Comme pour les intervalles, commençons par augmenter le treillis abstrait et la fonction d'abstraction. Encore une fois, la seule question qui se pose est l'interprétation des états indéfinis pour certaines variables. Encore une fois on choisira d'interpréter comme vraies les formules où ces variables interviennent. Plus précisément on considèrera toujours vraie dans l'état σ la contrainte

$$v_i - v_j < c_{i,j}$$

si v_i ou v_j sont indéfinies dans σ. Par conséquent, si dans tout état ou bien v_i ou bien v_j est indéfinie, c'est à dire si elles ne sont jamais définies simultanément, alors n'importe quelle constante convient. Puisqu'il faut prendre la plus petite, comme pour les intervalles, on étendra le domaine abstrait afin de pouvoir choisir $c_{i,j} = -\infty$. Ainsi, la contrainte lorsque v_i et v_j ne sont jamais définies simultanément devient

$$v_i - v_j < -\infty$$

Cette contrainte n'est jamais satisfiable et le fait qu'elle n'ait pas de modèle traduit donc bien qu'on ne peut trouver simultanément de cellule dans chacun des fragments associés. Avec cette extension, les valeurs abstraites sont maintenant des éléments de

$$(\mathbb{Z} \cup \{+\infty, -\infty\})^{(n+1)^2}$$

Formellement, $\widetilde{\alpha}$ et $\widetilde{\gamma}$ deviennent

$$\widetilde{\alpha}(\overline{X}) = \left(\min\left\{ c \in \mathbb{Z} \cup \{+\infty, -\infty\} \;\middle|\; \forall \overline{\sigma} \in \overline{X},\ \begin{cases} & \overline{\sigma}(v_j) \text{ indéfini} \\ \vee & \overline{\sigma}(v_i) \text{ indéfini} \\ \vee & \overline{\sigma}(v_i) - \sigma(v_j) \leq c_{i,j} \end{cases} \right\}\right)_{i,j}$$

$$\widetilde{\gamma}(\widetilde{X}) = \left\{ \overline{\sigma} \;\middle|\; \forall v_i, v_j \in \mathcal{V}\ \begin{cases} & \overline{\sigma}(v_i) \text{ indéfini} \\ \vee & \overline{\sigma}(v_j) \text{ indéfini} \\ \vee & \overline{\sigma}(v_i) - \sigma(v_j) \leq c_{i,j} \end{cases} \right\}$$

Lorsque v_i et v_j ne sont jamais définies simultanément l'ensemble des bornes possibles est tout l'ensemble $\mathbb{Z} \cup \{+\infty, -\infty\}$ dont $-\infty$ est le minimum. On étend les définitions de $\sqsubseteq$, $\sqcup$ et $\sqcap$ en conservant les définitions intactes et en interprétant $-\infty$ naturellement comme l'élément minimum pour $\leq$.

Réduction. La concrétisation $\widetilde{\gamma}$ génère maintenant des états de synthèse dont les domaines de définition sont quelconques. Si un état $\overline{\sigma}$ est dans la concrétisation, alors la restriction de $\overline{\sigma}$ à n'importe quel domaine est également dans la concrétisation.

Ceci a des conséquences sur la réduction. Il n'est plus possible d'utiliser l'implication

$$v_i - v_j \leq c_{i,j} \wedge v_j - v_k \leq c_{j,k} \Rightarrow v_i - v_k \leq c_{i,j} + c_{j,k}$$

En effet, avec la définition de $\widetilde{\alpha}$, il est possible que la relation à droite de l'implication soit fausse. Prenons un état de synthèse où la relation impliquée est fausse et où la variable v_j est indéfinie. Puisque v_j est indéfinie, les contraintes à gauche de l'implication n'empêchent pas cet état d'être dans l'abstraction et pourtant il ne vérifie pas la relation impliquée.

Comme pour le domaine des intervalles, on peut néanmoins profiter des informations de types données par le domaine de fragmentation. Supposons que l'on sache que la variable v_j est super-scalaire, c'est à dire que dans tous les choix de représentants, la variable v_j est toujours associée à une cellule. Alors seuls les états de synthèse où v_j est définie peuvent convenir à de tels choix de représentants. Après réduction, on ne conserve donc que des états où v_j est définie. Dans ces états, ou bien l'une des variables v_i ou v_j est définie et alors la relation à droite de l'implication reste valide. Ou bien, ces deux variables sont définies, elles doivent alors par définition de la concrétisation vérifier les contraintes à gauche de l'implication et par conséquent impliquer la relation à droite.

Nous modifions donc l'algorithme de réduction sur les zones pour qu'il n'utilise l'implication qu'à condition que la variable v_j utilisée en intermédiaire soit de type super-scalaire. Notons néanmoins que si c'est une condition suffisante, elle n'est pas nécessaire et il serait donc possible d'améliorer la réduction.

Pour terminer sur la réduction, mentionnons un cas particulier d'utilisation de cette implication, lorsque $v_i = v_k$.

$$v_i - v_j \leq c_{i,j} \wedge v_j - v_i \leq c_{j,i} \;\Rightarrow\; 0 \leq c_{i,j} + c_{j,i}$$

Lorsque $c_{i,j} + c_{j,i} < 0$, la contrainte à droite de l'implication est insatisfiable. Dans le cas où l'une des deux variables c_i ou c_j est super-scalaire, elle implique donc que l'autre représente un fragment vide. Si les deux variables sont super-scalaires, on revient dans le cas classique où cette contrainte implique la vacuité de l'ensemble d'états atteignables. N'importe comment, cette situation implique que v_i et v_j ne peuvent pas être définies simultanément. Par conséquent on peut poser $c_{i,j} = c_{j,i} = -\infty$ qui est une conséquence de la réduction.

Dans le cas encore plus particulier où on prend $v_i = v_j = v_k$, on peut simplement dire que si $c_{i,i} < 0$ alors v_i représente un fragment vide. La réduction affectera à tous les bornes liés à la variable v_i la valeur $-\infty$.

Élargissement. L'élargissement suit également l'exemple des intervalles. Lorsqu'une borne $c_{i,j}$ passe de $-\infty$ à une autre valeur, c'est qu'on considère pour la première fois une relation entre

les cellules représentées par v_i et celles représentées par v_j. On conservera donc la propriété découverte sans élargir.

$$(c_{i,j})_{i,j} \triangledown (d_{i,j})_{i,j} = \left(\begin{cases} d_{i,j} \text{ si } c_{i,j} = -\infty \\ +\infty \text{ si } d_{i,j} > c_{i,j} \\ c_{i,j} \text{ sinon} \end{cases} \right)_{i,j}$$

Transformations élémentaires. Intéressons-nous maintenant à la définition des transformations élémentaires. Cela se limite à appliquer les formules générales des transformations obtenues section 7.2 et à dérouler la définition pour notre domaine des opérateurs d'union et d'intersection.

Rappelons d'abord la transformation permettant d'ajouter une nouvelle variable à une matrice de bornes de différences. Si on ajoute une nouvelle dimension pour la variable t à une valeur $(c_{i,j})_{i,j}$ on obtient la valeur $(c'_{i,j})_{i,j}$ vérifiant

$$\begin{cases} c'_{t,t} & = 0 \\ c'_{t,j} & = +\infty \text{ si } j \neq t \\ c'_{i,t} & = +\infty \text{ si } i \neq t \\ c'_{i,j} & = c_{i,j} \text{ dans tous les autres cas} \end{cases}$$

Il faut néanmoins prêter attention au cas suivant. La variable ajoutée peut représenter un fragment vide. Ceci arrive typiquement lorsque l'on unifie deux fragmentations, le fragment non vide dans l'une des valeurs abstraite mais vide dans l'autre est introduit dans la seconde. Il faut alors donner à cette variable l'information maximum, ce qui se traduit par l'apparition de $-\infty$ aux bornes liées à cette variable, y compris sur la diagonale.

$$\begin{cases} c'_{t,j} & = -\infty \\ c'_{i,t} & = -\infty \\ c'_{i,j} & = c_{i,j} \text{ dans tous les autres cas} \end{cases}$$

L'oubli d'une variable reste comme dans le cas classique une suppression de ligne et de colonne dans la matrice, éventuellement précédé d'une réduction pour éviter la perte d'information.

La composition. La composition transforme une valeur $\widetilde{X}$ en

$$\widetilde{X} \sqcap \left(\widetilde{X}[s_1 \to t] \sqcup \widetilde{X}[s_2 \to t] \right)$$

Supposons que l'on parte d'une valeur abstraite $(c_{i,j})_{i,j}$ dans laquelle la variable v_t n'existe pas et qu'on cherche à produire une valeur résultat $(c'_{i,j})_{i,j}$ après composition de v_{s_1} et v_{s_2} en v_t. Cette valeur résultat doit vérifier la contrainte précédente, ce qui nous donne

$$\begin{cases} c'_{t,t} & = 0 \\ c'_{s_1,t} & = c'_{t,s_2} = +\infty \\ c'_{t,j} & = max\{c_{s1,j}, c_{s_2,j}\} \text{ si } j \neq s_1, s_2, t \\ c'_{i,t} & = max\{c_{i,s1}, c_{i,s_2}\} \text{ si } i \neq s_1, s_2, , t \\ c'_{i,j} & = c_{i,j} \text{ dans tous les autres cas} \end{cases}$$

Les bornes $c'_{s,t}$ et $c'_{t,s}$ sont fixées à $+\infty$, ce qui traduit bien le fait que la composition n'induit aucune relation entre les composantes et la composée. Toutefois, une réduction peut changer ces valeurs. Par exemple, si les deux fragments composants sont constants, le fragment composé le sera aussi et on pourra dire que les composants sont égaux au composé.

La fission et la décomposition. La fission et la décomposition se contentent dans notre cas de dupliquer les informations acquises sur une variable. Si la variable v_s est décomposée ou fissionnée en la variable v_t, la valeur abstraite $(c'_{i,j})_{i,j}$ résultat vérifiera

$$\begin{cases} c'_{t,t} & = 0 \\ c'_{s,t} & = c'_{t,s} = +\infty \\ c'_{t,j} & = c_{s,j} \text{ si } j \neq s,t \\ c'_{i,t} & = c_{i,s} \text{ si } i \neq s,t \\ c'_{i,j} & = c_{i,j} \text{ dans tous les autres cas} \end{cases}$$

Ceci est l'expression générale de la décomposition et de la fission. On peut renforcer la valeur abstraite dans le cas de la fission ou de la décomposition de fragments sous-scalaires. Si la variable fissionnée est sous-scalaire on aura $v_s = v_t$ et on peut donc fixer $c'_{s,t} = c'_{t,s} = 0$. Pour la décomposition, si la variable décomposée est sous-scalaire, les deux variables composantes ne peuvent être définies simultanément et on pourra avoir $c'_{s,t} = c'_{t,s} = -\infty$.

La fusion Pour la fusion, la variable destination v_t hérite des propriétés de la variable v_s. L'expression générale de la fusion

$$\widetilde{X} \sqcap \widetilde{X}[v_s \to v_t]$$

se traduit sur les zones par un calcul de minimum. La valeur résultat $(c'_{i,j})_{i,j}$ vérifiera

$$\begin{cases} c'_{t,j} & = min\{c_{t,j}, c_{s,j}\} \text{ si } j \neq s,t \\ c'_{i,t} & = min\{c_{i,t}, c_{i,s}\} \text{ si } i \neq s,t \\ c'_{i,j} & = c_{i,j} \text{ dans tous les autres cas} \end{cases}$$

Après la fusion, la variable v_t peut être retirée de la matrice.

L'affectation faible L'affectation $v_t \leftsquigarrow v_s$ affaiblit les propriétés de v_t avec les propriétés de v_s. La valeur résultat $(c'_{i,j})_{i,j}$ vérifiera

$$\begin{cases} c'_{t,j} & = max\{c_{t,j}, c_{s,j}\} \text{ si } j \neq t \\ c'_{i,t} & = max\{c_{i,t}, c_{i,s}\} \text{ si } i \neq t \\ c'_{i,j} & = c_{i,j} \text{ dans tous les autres cas} \end{cases}$$

La forme produite est similaire à celle produite par une composition à ceci près que des relations entre v_s et v_t sont toujours possibles. Observons le calcul des bornes $c_{s,t}$ et $c_{t,s}$. Dans le cas où l'affectation n'a pas lieu, la borne originale convient. Dans le cas où l'affectation a lieu, on a bien $s = t$ et la borne $c_{s,s} = 0$ convient. La valeur retenue est donc bien le maximum des deux, et il n'est pas nécessaire au contraire des transformations élémentaires de distinguer les cas.

Nous conclurons sur l'adaptation du domaine des zones et ce chapitre par deux remarques. D'abord, nous l'avons déjà énoncé, la réduction n'est pas complète. Il est possible d'extraire plus d'information de réduction grâce aux connaissances que l'on peut acquérir sur la fragmentation. Cela demande de généraliser le cadre que nous avons défini dans la section 4.2. Une seconde remarque peut être tirée de la forme des transformations élémentaires. Le calcul de chaque borne nécessite la séparation des cas selon que les variables impliquées sont les variables source ou destination. Il est possible de simplifier ces formes et d'obtenir un calcul plus régulier si l'on change le sens des bornes $c_{i,i}$. Posons qu'ils représentent la différence maximum entre tout couple d'éléments choisis dans le fragment désigné par la variable v_i. Cette borne est donc positive ou nulle. Elle est nulle si et seulement si le fragment est constant et en particulier s'il est sous-scalaire. Cette convention laisse donc les matrices inchangées dans le cas classique ou chaque variable désigne une variable locale. Mais il est maintenant possible de trouver des propriétés entre les composants et le composé d'une composition ou d'une décomposition. En outre, le calcul des compositions et des décompositions nécessite moins de distinctions de cas. Cette solution a été implémentée et testée, puis a été abandonnée : elle introduisait une complexité supplémentaire dans la formalisation et nous n'en avons obtenu aucun résultat intéressant.

8 Propriétés d'agrégation

Jusque ici, nous ne nous sommes attaqué qu'au problème de la recherche de propriétés de chaque cellule de chaque fragment. Ces propriétés s'expriment comme une formule quantifiée universellement sur les cellules. Lorsque l'on considère le problème de l'analyse de programmes manipulant des structures de données, ce ne sont pas les seules propriétés qui nous intéressent.

Penchons nous sur une autre classe de propriétés. Plutôt que de regarder indépendamment les cellules d'un même fragment, combinons les, *agrégeons* les pour former un nouvel objet. Cet objet peut être par exemple

- la taille d'un fragment,
- la somme des valeurs des cellules,
- le maximum des valeurs des cellules,
- la multiplicité d'une valeur dans un fragment,
- l'ensemble des valeurs des cellules ou bien encore
- le multi-ensemble des valeurs des cellules.

De cet objet, nous pouvons chercher les propriétés, qu'on appellera alors des *propriétés d'agrégation*.

Les quatre premiers exemples proposent d'agréger les cellules en une valeur numérique, et ne valent d'ailleurs que si les contenus des cellules sont numériques. Pour exprimer des propriétés sur ces objets, il suffit d'insérer dans la valeur abstraite une nouvelle variable représentant cet objet. Si on ne souhaite pas seulement exprimer ces propriétés mais également les trouver, on aura des contraintes supplémentaires sur le domaine abstrait. Logiquement, on ne pourra profiter pleinement des propriétés de sommes que si le domaine numérique en question sait exprimer des sommes. De même, l'introduction du maximum sera plus pertinente dans un domaine abstrait sachant les manipuler, comme par exemple celui des polyèdres max-plus. [AGG08]

Pour les deux derniers exemples, on ne pourra évidemment pas introduire dans un domaine abstrait numérique une variable représentant l'ensemble ou le multi-ensemble des valeurs d'un fragment. Il faudra recourir à des domaines spécialisés dans les propriétés liant les ensembles et les multi-ensembles respectivement et qui pourront en faire bon usage.

Parmi les travaux s'intéressant à des propriétés d'agrégation, on trouve [PH10] sur les multi-ensembles et [BDE+10] sur les tailles, les sommes et les multi-ensembles.

La découverte de propriétés d'agrégation peut être réalisée dans notre cadre théorique en se reposant sur un critère de fragmentation. Celui-ci définira les fragments desquels on cherchera des propriétés d'agrégation. Il reste cependant à expliciter comment les propriétés seront introduites, et comment les transformations des fragmentations influent sur ces propriétés.

Notre objectif dans ce chapitre est de généraliser les résultats obtenus dans [PH10]. Il s'agissait d'être capable d'exprimer des propriétés sur les multi-ensembles des valeurs des tableaux. Nous généralisons ces résultats aux propriétés sur les multi-ensembles des valeurs des fragments de tableaux. Avant cela, nous commencerons par des généralités sur les propriétés d'agrégation.

$sum \leftarrow 0$
$i \leftarrow 1$

Tant que $i \leq n$ **faire**
  $sum \leftarrow sum + A[i]$
  $i \leftarrow i + 1$

FIGURE 8.1 : Calcul de la somme des valeurs d'un tableau.

8.1 Généralités

8.1.1 Caractérisation d'une propriété d'agrégation

Pour une propriété d'agrégation fixée, notons $\mathbb{A}$ l'ensemble des valeurs d'agrégation possibles. Pour les trois premiers exemples donnés en introduction et pour des cellules contenant des entiers, ce sera $\mathbb{A} = \mathbb{Z}$. Pour les deux derniers exmples $\mathbb{A}$ sera respectivement les ensembles de valeurs et les multi-ensembles de valeurs. À partir d'un fragment, on construit sa valeur d'agrégation en deux temps : d'abord on envoie chaque cellule sur sa valeur d'agrégation individuellement, puis on agrège ces valeurs. On notera f et g les deux fonctions réalisant ces étapes. f sera la fonction qui à chaque cellule associe la valeur d'agrégation du fragment singleton la contenant. g est l'opérateur d'agrégation qui combine les cellules et dont on requiert qu'il soit commutatif et associatif. Ces propriétés de l'opérateur permettent de s'assurer qu'il ne dépend pas de l'ordre des cellules, les fragmentations ne précisant aucune information sur cet ordre. En outre, g doit avoir un élément neutre qui est la valeur d'agrégation du fragment vide.

Propriété recherchée	$\mathbb{A}$	$f : \Sigma \times \mathbb{C} \to \mathbb{A}$	$g : \mathbb{A} \times \mathbb{A} \to \mathbb{A}$
Taille du fragment	$\mathbb{Z}$	$\sigma, c \mapsto 1$	$x, y \mapsto x + y$
Somme des valeurs	$\mathbb{Z}$	$\sigma, c \mapsto \sigma(c)$	$x, y \mapsto x + y$
Maximum des valeurs	$\mathbb{Z}$	$\sigma, c \mapsto \sigma(c)$	$x, y \mapsto \max\{x, y\}$
Multiplicité d'une valeur v	$\mathbb{N}$	$\sigma, c \mapsto 1$ si $\sigma(c) = v$, 0 sinon	$x, y \mapsto x + y$
L'ensemble des valeurs	$2^{\mathbb{V}}$	$\sigma, c \mapsto \{\sigma(c)\}$	$x, y \mapsto x \cup y$
Le multi-ensemble des valeurs	$\mathbb{N}^{\mathbb{V}}$	$\sigma, c \mapsto \{\!\{\sigma(c)\}\!\}$	$x, y \mapsto x \oplus y$

La valeur d'agrégation pour la taille d'un fragment est simplement 1 pour tous les singletons et la fonction d'agrégation est l'addition des tailles. Pour la somme et le maximum la valeur d'agrégation d'un singleton est la valeur de la cellule et l'opérateur d'agrégation sont l'addition et le maximum. Pour la multiplicité la valeur d'agrégation d'un singleton est 1 si la valeur de la cellule est celle recherchée, 0 sinon. On combinera cette valeur comme pour la somme et la taille par addition. Pour les ensembles et multi-ensembles, la valeur d'agrégation d'un singleton est l'ensemble ou le multi-ensemble singleton contenant la valeur de la cellule, et l'opérateur d'agrégation est l'union d'ensemble ou l'union de multi-ensembles $\oplus$ obtenue par addition des multiplicités des deux multi-ensembles. (Cf. section 8.2.1)

8.1.2 Un exemple illustratif

Considérons l'exemple simple et illustratif d'un programme calculant la somme des éléments d'un tableau, celui de la figure 8.1.

Le critère de fragmentation simplifié (Section 4.1.1) donnera comme unique fragment la tranche $A[1..i[$ avant l'affectation et $A[1..i]$ après (mais avant l'incrémentation de i) et considèrera le singleton $A[i]$ le temps d'interpréter l'instruction ajoutant sa valeur à la variable *sum*. Cette fragmentation nous convient parfaitement. Il est néanmoins inutile de parler d'agrégation tant que les fragments ne font pas plus d'une cellule. À la fin de la première itération, l'analyse nous aura donné

$$1 \leq i \leq 2 \wedge A[1..i[= sum$$

où $A[1..i[= sum$ signifie « toutes les cellules de la tranche $A[1..i[$ sont égales à la variable *sum*. » A la première entrée dans la boucle, la propriété est vraie puisque $i = 1$ et la tranche est donc vide. Si on vient de la boucle, $i = 2$ et $A[1..i[= A[i]$. Lors de la seconde itération, l'interprétation de la première affectation nous donnera

$$1 \leq i \leq 2 \wedge sum = A[1..i[+ A[i]$$

Une fois l'affectation passée, le domaine de fragmentation demandera à composer les tranches $A[1..i[$ et $A[i]$. Il semble que ce soit le bon moment pour introduire une propriété d'agrégation car une fois les tranches composées, il sera trop tard. Nous allons introduire une nouvelle variable représentant la somme des cellules de la tranche composée $A[1..i]$. Notons cette variable explicitement $\Sigma A[1..i]$. Étant donné que les tranches $A[1..i[$ et $A[i]$ sont sous-scalaires, on a

$$\begin{aligned} A[1..i[&= \Sigma A[1..i[\\ A[i] &= \Sigma A[i] \end{aligned}$$

En effet, un fragment sous-scalaire contient au plus une cellule : ou bien il n'en contient aucune et on peut donc dire de toute cellule du fragment qu'elle est égale à la somme, ou bien elle en contient une et donc la somme est réduite à la seule valeur de cette cellule. On pourra par conséquent écrire

$$sum = \Sigma A[1..i[+ \Sigma A[i]$$

et lorsque l'on compose les tranches $A[1..i[$ et $A[i]$ en $A[1..i]$, cette égalité devient

$$sum = \Sigma A[1..i]$$

De retour en tête de boucle, notre valeur abstraite est devenue

$$1 \leq i \leq 2 \wedge sum = \Sigma A[1..i[$$

Nous pouvons vérifier qu'une nouvelle itération de l'interprète conservera cette propriété. L'affectation à *sum* donnera

$$sum = \Sigma A[1..i[+ A[i]$$

Comme lors de l'itération précédente, on peut remplacer $A[i]$ par $\Sigma A[i]$, et la composition nous redonnera la propriété invariante.

Cet exemple nous montre que pour arriver à notre objectif, il va falloir préciser deux choses. La première, c'est le moment auquel il faut introduire les variables d'agrégation dans l'analyse. La seconde, c'est comment interpréter les transformations élémentaires des fragmentations.

8.1.3 Introduction des variables d'agrégation

La manière la plus simple d'introduire les variables d'agrégation est de les introduire en même temps que les fragments. Chaque fois que le domaine de fragmentation crée un nouveau fragment, on ajoute la ou les variables d'agrégation correspondant à ce fragment. Nous ne nous y prendrons pas autrement pour les propriétés de multi-ensemble.

Il y a deux raisons pour lesquelles nous voudrions toutefois faire autrement. La première est un besoin d'optimisation. Supposons que l'on s'intéresse à plusieurs propriétés d'agrégation, il va falloir introduire des variables pour chaque fragment et pour chaque propriété. Le nombre de variables peut devenir déraisonnable. Si on considère les propriétés de multiplicité d'une valeur dans un fragment, on peut même avoir une infinité de variables à introduire et la méthode ne marche pas.

La seconde raison, c'est d'éviter d'avoir à redéfinir la sémantique abstraite pour les propriétés d'agrégation. Les critères de fragmentation font généralement en sorte que les cellules affectées soient toujours dans des singletons. Si on peut éviter d'introduire des variables d'agrégations pour les singletons, alors il n'y aura pas besoin de se poser la question des effets des affectations fortes sur les propriétés d'agrégation.

Dans l'exemple précédent, nous n'avons pas introduit de propriété d'agrégation tant que les tranches de la fragmentation étaient sous-scalaires. Tant que les tranches étaient sous-scalaires, on pouvait calculer leur valeur d'agrégation et une nouvelle variable aurait été redondante. On n'a ajouté cette variable que le plus tard possible, au moment où elle cessait d'être redondante. Procéder ainsi permet de réduire le nombre de variables à introduire, mais ne fonctionne que si l'opérateur d'élargissement conserve le type sous-scalaire des fragments.

Une dernière remarque est que l'on peut vérifier la pertinence de l'introduction d'une nouvelle variable en regardant si le domaine abstrait est capable de lui donner des propriétés. Si le programme ne réalisait pas un calcul de la somme, on n'aurait rien su de $\Sigma A[1..i]$ et on aurait pu déterminer qu'il était inutile de l'introduire.

8.1.4 Les transformations élémentaires

Nous sommes libres de choisir les fragments sur lesquels exprimer les propriétés d'agrégation. On voudra certainement réutiliser un critère de fragmentation existant, qu'il s'agisse de celui du chapitre 4 ou d'autres critères de la littérature comme [GDD+04], [HP08] ou [CCL11]. Toutefois, tous ces critères n'ayant pas été développés dans ce but, il peut être utile de réévaluer leur pertinence dans le cas des propriétés d'agrégation.

Ici, on ne s'intéresse dans une fragmentation qu'aux fragments, c'est à dire aux projections de la fragmentation sur les différentes variables de synthèse. On oublie les éventuelles relations entre les fragments que la fragmentation pourrait désigner. Le choix des fragmentations à utiliser et leur évolution restent contrôlées par un domaine de fragmentation. Celui-ci nous informera des transformations élémentaires à répercuter.

Nous décrivons brièvement pour chacune d'entre elles ce qu'elles impliquent sur les propriétés d'agrégation. Ce sera également l'occasion de faire un peu de lumière sur ce qui motivait certains choix dans la formalisation de ces transformations élémentaires.

Ajout et suppression de fragments. À moins qu'on ait décidé de la retarder, l'introduction des variables portant les valeurs d'agrégation se fera en même temps que l'ajout de nouveaux

fragments. La valeur que l'on donne à cette variable pourra être déterminée à l'aide des fonctions f et g définissant la propriété d'agrégation. Selon la situation

- ou bien le fragment introduit est un singleton $\{c\}$ et on donnera à la variable la valeur $f(c)$,
- ou bien le fragment est vide, et la variable sera égale à l'élément neutre de g,
- ou bien le fragment n'est pas sous-scalaire, et on ne connaît pas la valeur de la variable.

Symétriquement, la suppression d'un fragment entraînera le retrait des variables correspondantes.

La compostion. On se restreint au cas où les fragments composants sont disjoints, ce qui est toujours le cas pour les diagrammes de tranches et pour tous les critères de fragmentation produisant des partitions. On doit avoir des variables représentant les valeurs d'agrégation pour chacun des fragments composants. Si ce n'est pas le cas, par exemple parce qu'on a retardé leur introduction, il est toujours temps de les introduire maintenant. On combine alors les valeurs d'agrégation des fragments composants avec la fonction g. Le résultat est la valeur d'agrégation du fragment composé. Si le fragment composé n'a pas de variable associée pour la propriété d'agrégation, on l'ajoute. Dans tous les cas, on établit dans la valeur abstraite l'égalité entre cette variable et la valeur d'agrégation calculée. La correction de cette opération repose sur l'associativité et la commutativité de g.

En pratique, pour réaliser cette opération, il est indispensable que g puisse être évaluée dans le domaine abstrait. En général, on voudra plus que cela et être capable d'exprimer l'égalité impliquée par la composition. Si pour l'exemple illustratif on avait utilisé le domaine des intervalles, on aurait pu évaluer la somme et trouver un intervalle de valeurs pour la somme des éléments du fragment, mais cela ne nous aurait pas été très utile. En utilisant un domaine dans lequel on peut exprimer la somme de deux variables, comme celui des équations linéaires [Kar76], des polyèdres [CH78] ou des octogones [Min06], on peut établir l'égalité

$$\Sigma A[1..i[+\Sigma A[i] = \Sigma A[1..i]$$

C'est précisément l'effet de la composition. À droite de l'égalité, on a la variable associée au fragment composé, et à gauche on a la fonction g appliquée aux variables d'agrégation des fragments composants.

Notons que l'algorithme de réduction des diagrammes a pour effet de rétablir toutes les propriétés d'agrégations triviales : si on a les fragments $A[1..i[$, $A[i..n[$, $A[1..j[$ et $A[j..n[$, la réduction des diagrammes pour les propriétés de somme rétablira l'égalité

$$\Sigma A[1..i[+\Sigma A[i..n[= \Sigma A[1..n[= \Sigma A[1..j[+\Sigma A[j..n[$$

Dans le cas où les fragments composants ne sont pas disjoints, tout ceci ne fonctionne plus en général. On peut noter toutefois l'exception des propriétés d'ensembles.

La décomposition. Une décomposition produira généralement pour les propriétés d'agrégation les mêmes effets qu'une composition. Si on considère toujours des décompositions en deux fragments disjoints, alors appliquer la fonction g aux valeurs d'agrégations des deux fragments composants doit donner la valeur d'agrégation du fragment décomposé. Si on décompose le fragment $A[1..i]$ en $A[1..i[$ et $A[i..n[$ on pourra établir l'égalité

$$\Sigma A[1..i] = \Sigma A[1..i[+\Sigma A[i]$$

Il y a néanmoins une différence marquée entre la composition et la décomposition dans le cas où on ne peut pas décrire précisément l'effet de la décomposition. Pour la composition, on pouvait approximer la valeur d'agrégation du fragment composé à partir de celles des fragments composants. Ici, à partir de la valeur d'agrégation du fragment composé et de la valeur d'agrégation d'un fragment composant, on peut approximer la valeur d'agrégation de l'autre fragment composant.

La fusion et la fission La fusion et la fission garantissent que les cellules des fragments fusionnés ou fissionnés sont exactement les mêmes. Or la valeur d'agrégation est calculée à partir des cellules. On peut donc établir l'égalité des valeurs d'agrégation de ces fragments.

Dans un domaine où on ne peut pas exprimer l'égalité, les effets de la fusion et de la fission diffèreront. La fission dupliquera une approximation de la valeur d'agrégation, tandis que la fusion calculera une borne inférieure sur des approximations de la valeur d'agrégation.

Rappelons que pour la synthèse des structures de données, la fission et la décomposition avaient des effets similaires. Ici, les effets de la fission et de la décomposition sont bien distincts. C'est bien pour cette raison que ces deux transformations ont été distinguées dans le chapitre 4.

8.1.5 La sémantique abstraite

L'effet des instructions sur les propriétés d'agrégation est dépendant de la propriété considérée. Pour certaines propriétés, comme les propriétés de tailles des fragments, les affectations n'ont absolument aucun effet. Pour les autres, il faudra évaluer ces effets.

Comme nous l'avons remarqué à propos de l'introduction des variables d'agrégation, on a intérêt à n'avoir aucune variable d'agrégation de fragments sous-scalaires. Ainsi, les affectations fortes ne s'appliquant qu'à ces fragments, on n'a pas besoin d'envisager leurs effets sur les propriétés d'agrégation.

En revanche, cela ne nous dispense pas d'évaluer les affectations faibles. Dans un domaine numérique sans inégalité, une affectation faible nous fera souvent perdre les propriétés d'agrégation. Sur les autres, on pourra essayer d'encadrer la valeur d'agrégation.

8.2 Propriétés de multi-ensemble

Nous nous intéressons maintenant à la classe des propriétés d'agrégation concernant les multi-ensembles des valeurs d'un fragment.

8.2.1 Notations

Un multi-ensemble est caractérisé par une fonction appelée *multiplicité*. À toute valeur, elle associe le nombre de fois où cette valeur apparaît dans le multi-ensemble. On confond le multi-ensemble avec sa multiplicité et pour un multi-ensemble $\mathcal{M}$, on note $\mathcal{M}(x)$ la multiplicité de la valeur x dans $\mathcal{M}$. Nous considérons ici des *multi-ensembles hybrides* [Syr01], qui généralisent les multi-ensembles classiques en autorisant des multiplicités négatives.

L'addition et la soustraction notées $\oplus$ et $\ominus$ de deux multi-ensembles se calculent respectivement par l'addition et la soustraction valeur par valeur des fonctions de multiplicité :

$$(\mathcal{M} \oplus \mathcal{M}')(x) = \mathcal{M}(x) + \mathcal{M}'(x)$$
$$(\mathcal{M} \ominus \mathcal{M}')(x) = \mathcal{M}(x) - \mathcal{M}'(x)$$

Pour tout tableau A on notera $\widehat{A}$ le multi-ensemble de ses valeurs, c'est à dire

$$\widehat{A}(x) = \sum_i \delta_{x,A[i]} \quad \text{où } \delta_{x,A[i]} = \begin{cases} 1 & \text{si } x = A[i] \\ 0 & \text{sinon} \end{cases}$$

8.2.2 Domaine des égalités linéaires de multi-ensembles

L'objectif initial du domaine des égalités linéaires de multi-ensembles [PH10] était de découvrir automatiquement des propriétés de permutations dans les tableaux. C'est un enjeu important dans la preuve des algorithmes de tris : il faut pouvoir prouver que le tableau initial est une permutation du tableau final. Non seulement l'ensemble des valeurs du tableau initial doit être le même que celui du tableau final, mais également le multi-ensemble de ces valeurs. La multiplicité des valeurs est importante. Si on note $\widehat{A_0}$ le multi-ensemble des valeurs initiales (avant l'exécution de l'algorithme de tri) du tableau A, on doit prouver :

$$\widehat{A} = \widehat{A_0}$$

En général, la preuve d'une telle propriété nécessite de pouvoir exprimer des propriétés un peu plus générales. En effet, beaucoup d'algorithmes de tris utilisant des échanges de cellules, comme celui là :

$x \leftarrow A[i]$
$A[i] \leftarrow A[j]$
$A[j] \leftarrow x$

La propriété de permutation précédemment citée n'est pas valide tout le temps de l'échange. Après l'affectation à $A[i]$, la valeur de $A[j]$ a été dupliquée dans le tableau, tandis que la valeur de $A[i]$ encore stockée dans x a été retirée. En termes de multi-ensembles, on peut dire que le multi-ensemble des valeurs de A est celui des valeurs de A_0 auquel on a ajouté une nouvelle fois la valeur contenue dans $A[j]$ et retiré la valeur contenue dans x. On exprimera ça à l'aide d'une équation linéaire de multi-ensembles :

$$\widehat{A} = \widehat{A_0} \oplus \{\!\{A[j]\}\!\} \ominus \{\!\{x\}\!\}$$

Deux domaines sont proposés dans [PH10] pour la représentations de telles équations. Seul l'un d'entre eux conviendra à notre usage. L'idée est de considérer ces équations comme des équations affines générales et donc de réutiliser le treillis des équations affines. (Cf. section 2.3.1) Dans ces équations, les termes peuvent être

- des multi-ensembles des valeurs (initiales) d'un tableau
- des multi-ensembles singletons contenant la valeur d'une variable du programme
- des multi-ensembles singletons contenant la valeur d'un élément de tableau
- des multi-ensembles singletons contenant une constante

On conserve les caractéristiques du treillis des équations affines. Le treillis étant de profondeur finie, il n'est pas nécessaire d'utiliser un opérateur d'élargissement. La comparaison et le calcul de la borne supérieure reposent sur l'algorithme d'union linéaire [Kar76].

La sémantique abstraite de ce domaine interprète les affectations aux cellules de tableau par deux effets. D'une part, les multi-ensembles singletons sont affectés comme des variables classiques, soit fortement, soit faiblement, selon le cas. D'autre part, les multi-ensembles des valeurs d'un tableau sont affectés de la manière suivante. Pour une affectation à $A[i]$

- si on a dans le système d'équations une égalité $A[i] = exp$, alors l'affectation est inversible et on substitue $\widehat{A}$ par $\widehat{A} \oplus \{\!\{x\}\!\} \ominus \{\!\{A[i]\}\!\}$;
- si on n'a pas de telle égalité, l'affectation n'est pas inversible et on est contraint de retirer $\widehat{A}$ du système.

8.2.3 Application aux fragmentations

On souhaite étendre le domaine des équations linéaires de multi-ensembles. On veut qu'il puisse exprimer des propriétés à propos des multi-ensembles, non plus des valeurs d'un tableau, mais des valeurs de fragments de ce tableau, désignés par une fragmentation. Ceci consiste d'une part à interpréter les transformations élémentaires de la fragmentation et d'autre part à étendre la sémantique abstraite.

Introduction et retrait de fragment Les introductions et retraits de fragments sont traduits par l'introduction et retrait des variables représentant les multi-ensembles de ces fragments. Si le fragment introduit est vide, on ajoutera l'équation affirmant que la variable correspondante est égale au multi-ensemble vide. Dans le cas contraire, il n'y a rien à dire sur la variable, et elle n'apparaîtra dans aucune équation du système. Le retrait d'une variable utilise l'algorithme classique de projection permettant d'éliminer une variable d'un système linéaire en conservant le maximum d'information possible.

Compositions et décompositions Le domaine des équations linéaires de multi-ensemble permet de traiter les compositions et décompositions de manière exacte. Si les fragments f_1 et f_2 sont composés en f_3 ou si f_3 est décomposé en f_1 et f_2, on introduira dans le système l'équation

$$\widehat{f_1} \oplus \widehat{f_2} = \widehat{f_3}$$

où $\widehat{f_1}$, $\widehat{f_2}$ et $\widehat{f_3}$ sont les multi-ensembles des valeurs des fragments f_1, f_2 et f_3. Il n'y a aucune différence entre les effets de la composition et ceux de la décomposition.

Fusion et fission Si un fragment f_1 est fusionné ou fissionné en un fragment f_2, on introduira dans le système l'équation

$$\widehat{f_1} = \widehat{f_2}$$

Ici encore, il n'y a aucune différence entre les effets de la fusion et ceux de la fission.

Sémantique abstraite Les affectations fortes influencent les fragments sous scalaires comme dans le cas [PH10]. L'affectation faible en revanche aura des effets différents selon le cas :

- s'il s'agit d'une affectation à l'une des cellules d'un fragment qui n'est pas sous-scalaire, on peut appliquer la subtistution
- s'il s'agit d'une affectation à une cellule qui suivant les états n'appartient pas toujours au fragment, alors, sauf cas particulier [1] on n'a pas d'autre choix que de retirer la variable faiblement affectée du système.

1. Le cas particulier, c'est quand le fragment affecté a une valeur constante et égale à la valeur affectée, auquel cas l'affectation faible est sans effet. Un exemple d'échange de cellule où ce cas intervient est donné dans [PH10].

Conclusion L'effort nécessaire pour pouvoir découvrir des équations linéaires sur les multi-ensembles des valeurs des fragments d'une fragmentation est très faible. Cette méthode a été implémentée et testée avec succès sur deux exemples simples : l'exemple de la copie de tableau et celui du partitionnement d'un tableau selon le signe. [2]

Pour clore ce chapitre, nous détaillons le déroulement de l'analyse du premier, l'exemple de la copie de tableau.

$i \leftarrow 1$

Tant que $i \leq n$ **faire**
  $B[i] \leftarrow A[i]$
  $i \leftarrow i + 1$

On rentre initialement dans la boucle avec un ensemble vide d'équations de multi-ensembles. Pour traiter l'affectation de $A[i]$ à $B[i]$ le domaine de fragmentation introduit ces deux singletons. On ne sait encore rien d'eux, mais puisqu'ils sont singletons, le domaine abstrait des multi-ensembles peut traiter l'affectation comme une affectation forte classique dont on déduit notre première équation de multi-ensembles

$$\left\{\widehat{A[i]} = \widehat{B[i]}\right.$$

De retour en tête de boucle, l'unification des fragmentations transformera les tranches $A[i-1]$ et $B[i-1]$ en $A[1..i[$ et $B[1..i[$. Pour le système d'équations revenant de l'itération précédente, il ne s'agit que d'un renommage. En revanche, en entrée de boucle, on a un système d'équations vide dans lequel il faut introduire les nouvelles variables. Ces deux tranches sont vides lorsque $i = 0$. On ajoute donc les équations $\widehat{A[1..i[} = \emptyset$ et $\widehat{B[1..i[} = \emptyset$. On doit ensuite calculer l'union linéaire des deux systèmes :

$$\left\{\widehat{A[1..i[} = \widehat{B[1..i[}\right. \quad \sqcup \quad \begin{cases}\widehat{A[1..i[} = \emptyset \\ \widehat{B[i]} = \emptyset\end{cases} \equiv \left\{\widehat{A[1..i[} = \widehat{B[1..i[}\right.$$

Lors de l'itération suivante, on ajoutera une nouvelle fois les variables $\hat{A[i]}$ et $\hat{B[i]}$ et on concluera à leur égalité. Le système à ce point de l'analyse sera

$$\begin{cases}\widehat{A[i]} = \widehat{B[i]} \\ \widehat{A[1..i[} = \widehat{B[1..i[}\end{cases}$$

Après l'affectation, il faut composer les tranches $A[1..i[$ et $A[i]$ et les tranches $B[1..i[$ et $B[i]$. Nous détaillons d'abord la première. Pour traiter la composition de $A[1..i[$ et $A[i]$ en $A[1..i]$, et étant donné que ces tranches sont bien disjointes, on ajoute l'équation $\widehat{A[1..i]} = \widehat{A[1..i[} \oplus \widehat{A[i]}$ au système qui devient

$$\begin{cases}\widehat{A[1..i]} = \widehat{A[1..i[} \oplus \widehat{A[i]} \\ \widehat{A[i]} = \widehat{B[i]} \\ \widehat{A[1..i[} = \widehat{B[1..i[}\end{cases}$$

2. L'exemple du partionnement de tableau selon le signe ne pouvant pas être analysé avec la combinaison de notre critère de fragmentation et des diagrammes de tranches, (Cf section 10.2.3) une autre méthode de partitionnement a été utilisée.

Suivant le critère, les tranches $A[1..i[$ et $A[i]$ seront retirées ou bien directement après la composition ou bien plus tard. Sur cet exemple, cela n'aura pas de conséquences. Le retrait des tranches entraîne l'élimination des variables $\widehat{A[1..i[}$ et $\widehat{A[i]}$ par l'algorithme de projection. Cela réduit le système à une seule équation.

$$\left\{\widehat{A[1..i]} = \widehat{B[1..i[} \oplus \widehat{B[i]}\right.$$

Puis, la composition de $B[1..i[$ et $B[i]$ donnera directement

$$\widehat{A[1..i]} = \widehat{B[1..i[} \oplus \widehat{B[i]} = \widehat{B[1..i]}$$

et l'élimination des deux tranches composantes nous donnera à nouveau

$$\widehat{A[1..i]} = \widehat{B[1..i]}$$

A partir de là, on peut en déduire l'invariant de la boucle $\widehat{A[1..i[} = \widehat{B[1..i[}$.

9 Travaux connexes

Dans le domaine de l'analyse automatique de programmes, le cas des tableaux et des structures de données est un sujet jeune dont l'essentiel des publications s'étale sur les douze dernières années. Nous présenterons d'abord les travaux d'interprétation abstraite concernant la synthèse de propriétés de tableaux et de structures de données puis ceux concernant le choix et l'abstraction des fragmentations. La plupart des travaux présentés dans ces deux sections ont servi de point de départ à cette thèse. Dans une troisième section, nous présenterons plus rapidement diverses méthodes d'analyse statique semi-automatiques dont l'objectif n'est donc pas tout à fait le même, mais partageant des problématiques similaires.

9.1 Synthèse de tableaux et de structures de données

Nous nous intéressons dans cette section aux travaux dont l'objet était de proposer un mécanisme permettant d'utiliser les domaines abstraits classiques dans le cadre d'une analyse du contenu des tableaux ou des structures de données.

9.1.1 Expansion et écrasement de tableau.

Les premières publications sur la synthèse sont liées au développement de l'analyseur statique Astrée [BCC+02]. Celui-ci utilise deux techniques pour traiter les tableaux.

L'expansion [1] d'un tableau, qui consiste à introduire autant de variables qu'il y a de cellules dans le tableau. Elle ne peut être appliquée que si la taille du tableau est constante. Elle a l'avantage de la précision mais coûte cher si le tableau est grand. Pour déterminer un effet précis des boucles sur un tableau expansé, il sera souvent intéressant de dérouler complètement ces boucles. On peut ainsi éviter des affectations faibles, si les accès au tableau utilisent des indices constants après déroulement.

L'écrasement du tableau à l'inverse, associe à l'ensemble d'un tableau une unique variable. On donne à cette variable les propriétés vraies de toutes les cellules du tableau. On gagne en efficacité ce qu'on perd en précision. On introduit beaucoup moins de variables, et il est possible d'utiliser cette technique sur les tableaux dont la taille n'est pas connue statiquement. En contrepartie, toutes les affectations à un tableau écrasé est traité comme une affectation faible. Ainsi, on ne pourra qu'affaiblir les propriétés initiales de ces tableaux.

9.1.2 Partitionnement de tableau.

Les auteurs [GDD+04] proposent un compromis entre ces deux méthodes. Il s'agit de découper les tableaux en un ensemble de tranches formant une partition de celui-ci. On introduit autant de variables de synthèse que de tranches. Chaque cellule du tableau est associée à une et

1. Dans l'article, l'expansion est désignée par « array extension » et l'écrasement par « array smashing ».

une seule variable de synthèse par une surjection qu'on appelle la *fonction de synthèse*. Formellement, les cellules ayant la même image pour la fonction de synthèse sont regroupées dans le même fragment.

Le concept de fonction de synthèse est très proche de la définition que nous utilisons au chapitre 3. Nos choix de représentants correspondent à des inverses de leur fonction de synthèse. Notre définition est néanmoins plus générale en plusieurs points :

- elle permet d'avoir des fragments se chevauchant,
- elle autorise à ignorer une partie des cellules de tableau qu'on ne souhaiterait pas considérer,
- elle autorise des fragments vides, et enfin
- elle permet de choisir des cellules ayant des relations particulières, des relations entre leurs indices.

Les trois premiers points relâchent respectivement les conditions de disjonction, de recouvrement, et de non vacuité contraignant une partition. On peut donc les résumer en disant simplement que notre définition autorise les fragmentations à ne plus être des partitions. Cette contrainte de partitionnement est impliquée par la fonction de synthèse. Puisqu'il s'agit d'une application, chaque cellule a une seule image par la fonction de synthèse et il donc impossible qu'une cellule soit dans plusieurs fragments. Et puisque la fonction est totale, toute cellule appartient au moins à un fragment. Les auteurs nomment « projection » nos choix de représentants. Dans notre cas, ils n'en sont pas toujours : les choix de représentants peuvent associer plusieurs fois la même variable à une cellule, dans le cas où des fragments se chevauchent.

Les transformations de la fragmentations y sont décrites comme l'utilisation de quatre opérateurs, *fold*, *expand*, *add* et *drop*. Le premier permet de combiner deux fragments adjacents en un seul. Elle correspond à notre composition, à ceci près que les fragments composés par l'opération *fold* sont ensuite retirés. La seconde réalise l'opération inverse et correspond à notre décomposition. Encore une fois, on ne conserve que le produit de l'opération *expand*, retirant effectivement le fragment décomposé. Les deux dernières ajoutent et retirent un fragment respectivement.

La description de ces quatre opérations est un peu plus facile que la définition de nos transformations élémentaires. Elles conservent à tout moment une partition du tableau. Tandis que les nôtres conservent à la fois les fragments source et cible de ces opérations et qu'il faut donc porter son attention sur les relations que l'on exprime entre eux. Leurs opérateurs *fold* et *expand* ne servent qu'à faire évoluer la fragmentation durant l'analyse, tandis que les nôtres servent également à donner ou rappeler au domaine abstrait les singularités des fragmentations. Lorsque la fragmentation décrit une partition du tableau, aucune fission ni fusion ne peut avoir lieu puisqu'on ne peut avoir deux fragments identiques.

9.1.3 Synthèse par abstraction de fonction.

Ces résultats sont généralisés dans [JGR05]. Les tableaux et les structures de données en général sont modélisés comme des fonctions des cellules vers leurs valeurs et on cherche des abstractions de ces fonctions. Le partitionnement de tableau devient un partitionnement de l'ensemble de définition de la fonction. La construction d'un domaine abstrait pour ces fonctions repose sur la définition de deux domaines abstraits : l'un décrivant le partitionnement, l'autre décrivant les contenus. Le domaine abstrait décrivant le partitionnement est un treillis généralement plat dont les éléments sont les fragments et leurs concrétisations les ensembles de cellules qu'ils représentent.

Les auteurs proposent d'une part de redécomposer le processus de synthèse en deux parties.

On part d'un état concret, fonction des cellules vers leurs valeurs. On commence par collecter pour chaque variable de synthèse les valeurs des cellules associées à cette variable. On construit ainsi des états qui aux variables de synthèse associent des ensembles de valeurs. C'est la première étape de l'abstraction. La seconde consiste à distribuer ces valeurs. On construit ainsi des états de synthèse en choisissant une valeur dans chaque ensemble. D'autre part, les résultats sont généralisés au cas où la fragmentation n'est pas une partition.

Comparé à notre méthode de synthèse, celle-ci ne permet toujours pas de désigner des cellules liées par des relations d'indice. Les fragmentations considérées ne sont pas symboliques, et ne peuvent contenir de fragment vide.

9.1.4 Synthèse de relations point à point

Dans [HP08], une toute autre manière de synthétiser les propriétés de tableau est utilisée. Le principe du partitionnement de tableau est conservé mais son usage est différent.

Le but est de pouvoir exprimer des propriété point à point de la forme :

$$\forall \ell,\ \varphi(\ell) \Rightarrow \psi(A_1[\ell + c_1], \ldots, A_k[\ell + c_k])$$

Ceci inclut par exemple le fait que deux tableaux soient égaux ou le fait qu'un tableau soit trié :

$$\forall \ell,\ 1 \leq \ell \leq i \Rightarrow A[\ell] \leq A[\ell + 1]$$

L'analyse tentera de trouver une telle implication pour chaque fragment de la partition. La formule φ à gauche de l'implication correspond donc à la définition du fragment dans la partition. À droite de l'implication, on a une propriété ψ qui implique des cellules $A_i[\ell + c_i]$ qui sont donc des translations du fragment original.

La propriété ψ est exprimée par une valeur abstraite. L'utilisateur peut choisir n'importe quel domaine abstrait pertinent pour l'expression de propriétés des contenus des tableaux. On reste bien dans le cadre de la synthèse de propriétés de structures de données. Les valeurs abstraites feront intervenir entre autre des variables de synthèse pour chacun des $A_i[\ell + c_i]$. Elles pourront également introduire des fragments scalaires ou des variables du programme. En revanche, on s'interdit d'exprimer des propriétés à propos de fragments (non-scalaires) qui ne seraient pas des translations l'un de l'autre.

L'intérêt de ne chercher des relations qu'entre des tranches qui sont des translations l'une de l'autre c'est que lorsqu'une tranche est vide, ses translations le sont aussi. Ainsi, dans une valeur abstraite, on ne peut pas simultanément avoir des fragments vides et des fragments non vides. En pratique, $\varphi(\ell)$ est insatisfiable, les tranches translatées sont vides, et on prendra $\bot$ comme valeur abstraite pour la partie droite de l'implication. Si au contraire $\varphi(\ell)$ est satisfiable, toutes les variables de synthèse de la valeur abstraite sont super-scalaires et le comportement classique du domaine abstrait peut s'appliquer. Il n'est donc pas nécessaire de modifier les domaines abstraits comme nous l'avons fait dans cette thèse.

Introduire des variables du programme ou des fragments scalaires dans les valeurs abstraites permet de désigner d'éventuelles relations qu'ils auraient avec les tranches. La valeur abstraite peut également exprimer des propriétés liant les variables du programme entre elles. Mais cela ne nous apprend rien sur ces variables en général : ces propriétés ne sont valables que dans le cas où la partie gauche de l'implication est vérifiée. Par conséquent, si on veut découvrir des propriétés sur les variables du programme, il est de toute façon nécessaire d'avoir une valeur abstraite qui ne fait intervenir que des fragments super-scalaires.

Les auteurs décrivent trois transformations des fragmentations : le raffinement, la simplification et la progression d'indice. La première est utilisée lorsque l'on rentre dans une boucle itérant sur un tableau. On raffine la partition en séparant les cellules suivant qu'elles sont inférieures, égales, ou supérieures à l'indice de la boucle. Dans notre formalisme, celle-ci s'exprime comme une ou plusieurs décompositions. La simplification est l'opération inverse utilisée lorsque l'on sort d'une boucle, les fragments que l'on avait distingués à l'intérieur de la boucle doivent être rassemblés en un seul et on peut donc l'exprimer par des compositions. La progression d'indice intervient lors de la phase d'incrément de la boucle. On cherche à distinguer la cellule d'indice égal à l'indice de boucle : quand cet indice est incrémenté, on doit changer la partition en conséquence. Cela revient à décomposer le fragment dont on extrait la nouvelle cellule et à composer l'ancien singleton avec la tranche adjacente.

Il est donc possible de traduire toutes ces transformations en termes de compositions et décompositions. Celles-ci s'appliquent à toutes les translations du fragment original.

Dans le cadre défini par les auteurs, les compositions ont l'avantage d'être simples. Si on veut composer la tranche dont les indices vérifient φ_i et les contenus vérifient ψ_i avec la tranche dont les indices vérifient φ_j et les contenus vérifient ψ_j on donnera à cette tranche les propriétés $\psi_i \sqcup \psi_j$.

Si cette analyse repose sur un partitionnement du tableau, les translations de tranches introduisent des chevauchements. La translation d'une tranche peut chevaucher la translation d'une tranche adjacente. Par conséquent, les règles de réduction s'appliquent. Lorsqu'une translation de tranche recouvre une autre translation de tranche on peut renforcer les propriétés de la seconde par les propriétés de la première.

Ces travaux incluent également tout un travail sur le choix de la fragmentation. Nous aborderons ce sujet dans la section 9.2.2.

9.1.5 Synthèse de propriétés sur les mots

Les auteurs de [BDE+10] abordent le problème de la synthèse de propriétés sur les listes simplement chaînées. La suite des données associée à chacun des maillons de la liste forme un mot. A partir d'un découpage des listes en fragments, on peut exprimer des propriétés de chacun de ces fragments ainsi que des relations entre ces fragments. Ils traitent aussi bien le cas des propriétés d'agrégation (la somme, la taille et les multi-ensembles) que le cas des propriétés vérifiées cellules par cellules. Dans ce dernier cas les relations entre les fragments de listes peuvent être précisées par des patrons choisis par l'utilisateur. Il est ainsi possible de désigner des relations entre deux cellules d'un mot telles que la première est située avant la seconde. La classe de relations qu'il est ainsi possible de rechercher est plus large que les simples relations de translation.

On retrouvera des opérateurs pour la manipulation des fragments similaires à ceux des précédents travaux :

- l'opérateur `split` qui permet de décomposer un fragment de liste en un singleton tête et le fragment queue,
- l'opérateur `concat` qui permet de composer deux fragments adjacents,
- l'opérateur `sglt` qui permet d'ajouter un nouveau fragment singleton (après l'allocation dynamique d'un nœud de la liste) et
- l'opérateur `proj` qui permet d'éliminer un fragment.

Le concept de mot utilisé dans ces travaux pourrait très bien modéliser le contenu des tableaux. Si on prend la séquence de valeurs d'un fragment de tableau, on pourra utiliser les même moyens pour associer des propriétés à ces fragments. La particularité des mots par rapport aux formalisations précédentes est de conserver une notion d'ordre entre les éléments d'un fragment.

La formalisation proposée en revanche n'autorise pas les fragments à être vides. Il est nécessaire d'énumérer les configurations possibles des listes de sorte que dans chacune d'elle, chaque fragment de liste contienne toujours au moins un élément.

9.1.6 Conclusion

Tous ces travaux à l'exception du dernier, sont antérieurs aux notres et, à l'exception du dernier, ont fortement inspiré l'élaboration de notre formalisation. D'un coté, dans [GDD+04, JGR05] on a une méthode qui permet d'exprimer des relations entre toutes les cellules de tous les fragments. De l'autre, dans [HP08], on a une méthode qui permet l'expression de relations particulières entre des fragments, à condition que quand l'un d'entre eux est vide, l'autre l'est également. Notre formalisation cherche à embrasser l'ensemble de ces possibilités : autoriser l'expression de relations aussi bien générales que particulières sur des fragments qui peuvent se chevaucher et être vide indépendamment les uns des autres. La contrepartie c'est que notre formalisme nécessite que les domaines abstraits soient adaptés en conséquence.

Notre manière de synthétiser les propriétés n'est d'ailleurs qu'une généralisation de la méthode originale de [GDD+04]. Si on impose des contraintes sur notre processus de synthèse, on retrouvera la même correspondance de Galois. D'une part, il faut interdire les fragments vides ou se chevauchant. D'autre part, il faut inclure dans la fragmentation tous les choix de représentant associant à chaque variable de synthèse n'importe laquelle des cellules représentée. Notre manière de présenter la correspondance de Galois diffère également : notre définition est centrée sur les choix de représentants. Ceux-ci n'apparaissent dans l'article précédemment cité que comme inverses de la fonction de synthèse.

Enfin, on peut noter une dernière différence avec le formalisme de [GDD+04] : notre définition fait apparaître le caractère symbolique de la fragmentation. La construction d'états de synthèse à partir d'un état concret repose sur la fragmentation en cet état concret. Cela peut paraître anodin, car on peut proposer une généralisation symbolique directe de leur formalisation. Mais cette généralisation directe n'autorise pas l'introduction de fragments potentiellement vides dont la vacuité dépend de la valeur des variables du programme. Pour introduire un tel fragment, il va falloir partitionner l'ensemble d'états et conduire deux analyses en parallèle. Pour les états où le fragment n'est pas vide, on peut sans problème introduire le fragment. Pour les états où le fragment est vide, on conduira simplement l'analyse en oubliant ce fragment. L'inconvénient, c'est que dans le pire cas, on doit conduire un nombre d'analyses parallèles qui est exponentiel en le nombre de fragments.

La différence de traitement des fragments vides par les méthodes de synthèse résume très bien leur caractéristique.

- Les méthodes présentées dans [GDD+04, JGR05] interdisent les fragments vides et il faudra donc énumérer les cas de vacuité et en payer le coût.
- La méthode introduite dans [HP08] autorise les fragments à être vides, et n'a donc pas ce problème. En contrepartie, il faut une valeur abstraite distincte pour chaque relation entre des fragments.
- Notre méthode autorise des fragments vides, et permet de regrouper toute l'information dans une seule valeur abstraite. En contrepartie, elle nécessite une adaptation du domaine

abstrait.

9.2 Critères de fragmentation et abstractions de fragmentations

Dans cette thèse nous avons tenu à distinguer le critère de fragmentation de son abstraction. Nous avons décrit l'un et l'autre dans des chapitres propres, respectivement les chapitres 4 et 5. Les deux sont tout de même fortement liés : pour qu'un critère de fragmentation soit utile, il faut qu'on soit capable de produire des approximations raisonnables des fragmentations qu'il désigne. C'est probablement la raison pour laquelle les deux ne sont jamais distingués dans la bibliographie. Par conséquent, nous aborderons dans cette section à la fois l'un et l'autre.

Tous les critères de fragmentation cités ici ont un point en commun. Chaque fois qu'une cellule est désignée dans une instruction ou dans un test, un fragment singleton est réservé pour cette cellule avant l'interprétation de cette instruction ou de ce test. La manière d'introduire ce fragment et ce qu'on en fait une fois l'instruction ou le test interprété diffère d'une méthode à une autre. Nous tenterons de mettre en relief ces différences.

9.2.1 Fragmentation par prédicat

L'outil d'analyse statique générique TVLA [LARSW00] applique l'interprétation abstraite à la recherche de propriétés des structures de données. Ces structures de données sont décrites comme des ensembles de nœuds ayant des champs pouvant pointer d'autres nœuds. Ces nœuds sont regroupés en un nombre fini de classes. On décrit les propriétés de chaque classe de nœuds grâce à un ensemble de prédicats que l'on interprète sur une logique tri-valuée. [SRW99] Comme son nom l'indique, cette logique permet d'attribuer aux prédicats trois valeurs :

- ou bien on sait que tous les nœuds d'une classe vérifient le prédicat et on évalue le prédicat à la valeur 1,
- ou bien on sait qu'aucun des nœuds de la classe ne le vérifie et on évalue le prédicat à la valeur 0,
- dans, les autres cas, s'il y à la fois des nœuds vérifiant et ne vérifiant pas le prédicat, on évalue le prédicat à la valeur $1/2$.

Ce dernier cas représente aussi bien une incertitude due à l'approximation réalisée durant l'analyse qu'une possible combinaison de nœuds vérifiant et ne vérifiant pas le prédicat.

On peut avoir des prédicats n-aires qui permettent d'observer des propriétés liant plusieurs classes de cellules. De la même manière, on interprète les prédicats sur les classes en considérant tous les n-uplets de nœuds issus des classes respectives. Les trois valeurs de la logique sont organisées en demi-treillis avec $\top = 1/2$:

$$1 \sqsubseteq 1/2,\ 0 \sqsubseteq 1/2$$
$$0 \sqcup 1 = 1/2, x \sqcup 1/2 = 1/2$$

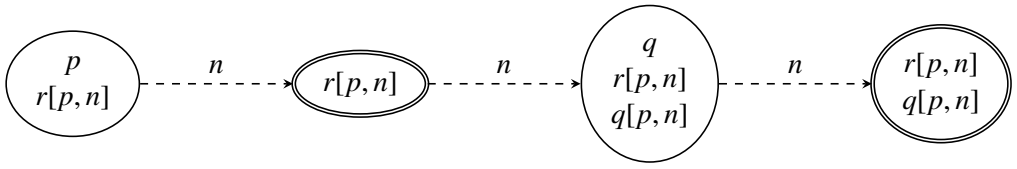
La généricité de l'analyseur TVLA vient de ce que l'on peut lui donner n'importe quel ensemble de prédicats. Leur choix dépendra du type de structures de donnée analysées et des propriétés que l'on cherche à démontrer. On n'utilisera pas les mêmes prédicats selon que l'on considère des listes simplement chaînées, doublement chaînées, ou encore des arbres. Il sera souvent nécessaire d'introduire des prédicats spécifiques au programme que l'on analyse, comme par exemple le prédicat « la liste est triée. » L'ensemble des prédicats peut être généré automatiquement, mais doit rester fini. On ne peut pas ajouter la classe de prédicats « un nœud est à distance n d'un autre dans une liste » pour tout $n \in \mathbb{N}$.

Les auteurs proposent par exemple d'introduire systématiquement certains prédicats :
- pour chaque pointeur x du programme, on introduit un prédicat unaire $x(v)$ qui s'évalue à 1 si la variable x pointe sur v,
- pour chaque pointeur x et chaque champ n, on introduit un prédicat unaire $r[x,n](v)$ qui s'évalue à 1 lorsque v est accessible à partir de x en utilisant uniquement le champ n, et
- pour chaque champ n, on introduit un prédicat binaire $n(u,v)$ qui s'évalue à 1 si le champ n de u pointe sur v.

Ici, ce qui nous intéresse particulièrement à propos de cette théorie, c'est la manière dont les classes de nœuds sont constituées. Les auteurs de TVLA proposent de choisir un sous-ensemble de prédicats unaires et de partitionner l'ensemble des nœuds par ces prédicats. Ainsi, on regroupe les nœuds pour lesquels les prédicats sont évalués à la même valeur. Puisque il y a trois valeurs par prédicats, s'il y a n prédicats, on aura au maximum 3^n classes.

Supposons qu'on veuille partitionner les nœuds d'une liste simplement chaînée avec les deux types de prédicats énoncés plus haut. Un prédicat $x(v)$ ne peut valoir 1 que pour le seul nœud pointé, et permettra donc de distinguer ce nœud dans une classe singleton. Les prédicats $r[x,n]$ quant à eux permettent de séparer les cellules en fonction de leur position dans la liste relativement aux différents pointeurs.

Pour illustrer ce dernier exemple, supposons que nous voulions analyser un programme itérant sur une liste dont le début est pointé par une variable p et l'élément courant pointé par une variable q. On introduit les prédicats $p(v)$, $q(v)$, $r[p,n](v)$, $r[q,n](v)$ et $n(u,v)$ où n désigne le champ pointant sur le nœud suivant dans la liste. La valuation des ces prédicats peut être représentée par un graphe, que nous tracerons avec les conventions de [LARSW00] :



Dans ce graphe, les sommets à bords simples représentent les classes contenant un unique nœud. Les sommets à bords doubles représentent au contraire les classes pouvant contenir plus d'un nœud. On marque à l'intérieur des sommets les prédicats vérifiés par la classe correspondante. Lorsque deux classes vérifient un prédicat binaire, on relie les deux sommets par un arc étiqueté avec le nom du prédicat. Lorsque ce prédicat binaire s'évalue à $1/2$, on trace l'arc en pointillés.

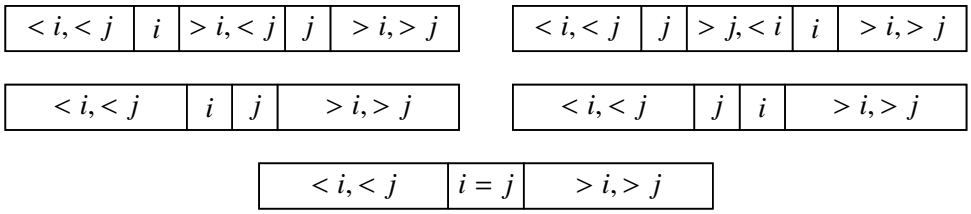
L'analyse ne permet pas à une classe de nœuds d'être vide. Par conséquent, comme nous l'avons déjà mentionné en section 9.1.6, chaque fois qu'une classe peut être vide on devra conduire deux analyses en parallèle : l'une pour le cas où la classe est vide, l'autre pour le cas où elle contient au moins un nœud.

Une partie des auteurs de [GDD+04] ont combiné l'outil TVLA avec leur théorie permettant de synthétiser des propriétés de tableau [GRS05]. Dans leurs travaux, ils remplacent les prédicats automatiquement introduits sur les structures de données par des prédicats similaires mais adaptés au partitionnement des tableaux. Pour chaque tableau et pour chaque variable d'indice utilisée dans un accès à ce tableau ils introduisent une fonction qui à chaque cellule associe une valeur selon que l'indice de la cellule est inférieure, égale, ou supérieure à l'indice. Les cellules

sont partitionnées selon cette fonction, ce qui revient à introduire les prédicats correspondant à chacune des valuations de cette fonction. On trouve le partitionnement familier des tableaux, similaire à celui des listes, dans lequel on sépare la cellule manipulée, les cellules déjà manipulées, et les cellules qui ne le sont pas encore.

Le processus de synthèse de [GDD+04] que nous avons décrit plus tôt n'accepte pas de fragments vides, une caractéristique qu'il partage avec TVLA. On décomposera encore l'analyse suivant la vacuité des différents fragments. Si cette décomposition a un coût potentiellement important, elle peut avoir des effets intéressants dans l'analyse de certains programmes, notamment en présence d'alias en permettant de distinguer les différent cas d'alias et rendre ainsi l'analyse précise.

Pour illustrer la méthode de partitionnement, prenons l'exemple suivant. Supposons que nous ayons un tableau dont les indices vont de 1 à n et que nous ayons deux variables i et j utilisées comme index et vérifiant $1 < i < n$ et $1 < j < n$. Pour chacune de ces deux variables on introduit les prédicats qui discriminent les cellules du tableau en trois sous ensembles suivant que leurs indices sont inférieurs, égaux ou supérieurs à la variable. Nous obtenons donc $3 \times 3 = 9$ fragments distincts. En fonction de leur vacuité on obtient les cinq configurations suivantes :



Les fragments aux extrêmes gauches et droites ne sont jamais vides. On les retrouve donc dans toutes les configurations. Si au lieu d'avoir des inégalités strictes sur i et j on avait les inégalités large $1 \leq i \leq n$ et $1 \leq j \leq n$ le nombre de configurations augmenterait grandement : ces deux fragments pourraient toujours être vides et il faudrait multiplier le nombre de configurations par quatre. Ceci illustre assez bien le coût qu'introduit l'interdiction de considérer des fragments vides.

Ce qui peut toutefois s'avérer intéressant dans certaines analyses de programme, c'est que pour chaque position relative de i et j, on a une analyse complètement indépendante. Cette séparation est souvent nécessaire pour une opération très courante sur les tableaux, l'échange de cellules, dont nous verrons dans la section 10.2.1 qu'elle nécessite au moins la distinction des cas $i = j$ et $i \neq j$. Ici, la séparation des cas est due au fait suivant. Lorsque $i < j$ alors les fragments $A[= i, > j]$, $A[= j, < i]$ et $A[= i, = j]$ sont vides tandis que si $i > j$ ce sont les fragments $A[= i, < j]$, $A[= j, > i]$ et $A[= i, = j]$ et si $i = j$ ce sont les fragments $A[= i, > j]$, $A[= i, < j]$, $A[= j, < i]$ et $A[= j, > i]$. On est donc forcément dans une configuration différente de la partition.

Conclusion. Pour ces deux analyses, on peut dire que le critère de fragmentation est de nature syntaxique. Pour le premier, on cherche dans le programme les pointeurs pour introduire les prédicats correspondants et dans le second on cherche les indices utilisés dans les accès aux tableaux. Ce critère syntaxique peut être développé pour améliorer la précision de l'analyse comme nous le verrons dans la section suivante.

9.2.2 Fragmentation syntaxique

Le critère précédent considérait des programmes dans lesquels les index utilisés dans les accès aux tableaux sont des expressions simples de la forme $A[i]$. Pour des accès plus généraux, ne serait-ce que des accès de la forme $A[i + 1]$, il faut préciser le critère. Cela vaut aussi bien pour les accès aux tableaux que pour les accès aux structures de données chaînées. Si, par exemple, une instruction modifie la valeur d'un nœud pointé par une variable p alors les prédicats de base sont satisfaisants. Mais si au contraire, le nœud modifié est le nœud suivant le nœud pointé, c'est celui là qui doit être distingué et non le nœud pointé.

Modifier les critères pour prendre en compte ces particularités est un point de détail. Cependant la construction d'un critère de fragmentation consiste essentiellement à être capable de prendre en compte toute un ensemble de détails de ce genre. Les critères devenant de plus en plus complexes, il devient difficile de continuer à les généraliser sans briser leurs qualités acquises.

Les auteurs de [HP08] dont nous avons déjà discuté en section 9.1.4 proposent un critère de fragmentation plus précis tenant compte du point dont nous venons de parler, mais également d'autres caractéristiques du programme moins évidentes.

Les auteurs proposent de fixer la syntaxe des programmes à analyser. Cette syntaxe peut paraître contraignante, mais il est tout à fait possible de l'étendre sans mettre en défaut l'analyse. Simplement, elle définit une classe de programmes sur lesquels on a bonne confiance que le critère sera pertinent. C'est d'ailleurs un très bon moyen de décrire les limites d'un critère de fragmentation : on sait où se situe le travail à accomplir lorsque l'on étend la syntaxe.

Nous détaillons brièvement les caractéristiques de ce langage. Les restrictions ne s'appliquent pas aux expressions, qui peuvent être quelconques. D'abord, on sépare les variables du programmes en deux groupes : les variables utilisées pour indexer les tableau, et les variables de contenu, de même type que les cellules de tableau. Ainsi, on sait exactement quelles sont les variables susceptibles d'être utilisées pour partitionner les tableaux. Ensuite, la principale restriction s'applique aux structures de contrôle. On peut employer librement des structures conditionnelles, mais la seule structure répétitive que l'on s'autorise est une boucle simple dont la syntaxe est la suivante :

```
for (i := <Iexp>; <cond>; <progress>)
  <statement>
```

où <Iexp> est une expression pouvant contenir des variables d'indice, <cond> est n'importe quelle condition, <progress> est au choix ++ ou -- et <statement> peut être n'importe quelle instruction du langage. La sémantique est la suivante. On initialisera la variable d'indice i avec <Iexp> avant d'entrer dans la boucle, puis qu'on exécutera <statement> tant que la condition <cond> est vraie, tout en incrémentant ou décrémentant la variable i de 1 entre deux exécutions selon que <progress> est respectivement ++ ou --.

Le langage n'autorise pas de sauts inconditionnels ou de sortie prématurées d'une boucle. Ceci donne une forme bien particulière au graphe de flot de contrôle : les seules composantes fortement connexes sont les boucles et elles n'ont qu'une seule entrée et qu'une seule sortie.

Le critère de fragmentation proposé par les auteurs se base sur cette bonne structuration du programme. Pour chaque boucle, on utilisera un partitionnement différent de l'ensemble des valeurs d'indices, qu'on établira en suivant trois règles :

- Le partitionnement dans une boucle doit être un raffinement du partitionnement au niveau supérieur. Pour une boucle imbriquée, ceci signifie qu'on doit raffiner la partition de la

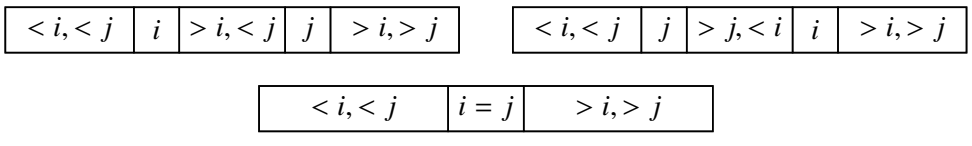
boucle englobante. Une partition est un raffinement d'une autre si les parties de la seconde peuvent être obtenues par union des parties de la première.

- Dans une boucle toujours, la partition doit distinguer les cellules selon que leur indice est inférieur ou supérieur à l'expression d'initialisation $< Iexp >$. Pour les boucles où la progression est ++, on distinguera les cellules d'indice $\ell < Iexp$ et $\ell \geq Iexp$ tandis que si la progression est -- on distinguera les cellules d'indice $\ell \leq Iexp$ et $\ell > Iexp$.
- Chaque fois qu'une expression $A[Iexp]$ apparaît, la partition à ce niveau d'imbrication devra distinguer les cas $\ell < Iexp$, $\ell = Iexp$ et $\ell > Iexp$.

Ce partitionnement de l'ensemble d'indices est utilisé pour produire un partitionnement de chaque tableau. Au cours de l'analyse, on ajoutera également de nouvelles tranches de tableau, construites comme translations non symboliques de ces tranches originales. Ces translations sont ajoutées dans deux cas. D'une part lorsque l'expression d'indice d'une cellule à droite d'une affectation est la même à une addition de constante près que l'expression d'indice d'une cellule à gauche de l'affectation. D'autre part, dans une phase de normalisation quand il est possible d'associer de l'information à une telle translation.

Ce critère permet effectivement de distinguer dans les boucles la cellule en cours de manipulation, les cellules déjà manipulées et les cellules qui ne l'ont pas été. Par rapport aux travaux présentés dans la section précédente, il a plusieurs avantages. D'abord il permet de désigner correctement les fragments singletons en considérant l'expression d'indexation réellement utilisée dans chaque accès. Ensuite, il reste pertinent lorsqu'une boucle ne commence pas aux extrémités d'un tableau. Si une boucle croissante commence à l'indice 2 au lieu de 1, les propriétés que cette boucle construit ne sont valables qu'à partir de la deuxième cellule du tableau. La seconde règle du critère permettra de séparer la première cellule du reste. Pour la même raison, cette seconde règle sera aussi utile dans certains algorithmes utilisant des boucles imbriquées : si la boucle imbriquée ne démarre pas au même indice à chaque itération, on sera en mesure de distinguer les cellules parcourues lors de l'itération courante de la boucle externe et celles parcourues lors de des itérations précédentes.

Enfin, le critère permet, comme dans la section précédente, de distinguer les cas d'alias. Mais cette fois, les tranches vides étant autorisées, le nombre de configurations est bien moindre. Si on doit partitionner un tableau selon deux variables d'indices i et j incomparables, seules 3^2 tranches devront être définies. Suivant les positions relatives de i et j, on n'a qu'à distinguer trois configurations :



Toutes les tranches de ces configurations à l'exception de deux ne sont présentes que dans une seule des configurations. Donner des propriétés à ces tranches revient à ne donner des propriétés à une partie du tableau que dans l'une des trois configurations. On peut en particulier traiter précisément les programmes dont le comportement est différent selon que $i = j$ ou $i \neq j$. En revanche, les tranches $A[< i, < j]$ et $A[> i, > j]$ qui sont présentes dans les trois configurations ne peuvent être traitées différemment selon les cas.

Le critère peut introduire au pire cas un nombre de fragments exponentiel en le nombre d'expressions d'indice. Ce pire cas est rarement atteint et est une réduction importante de la complexité par rapport à la double exponentielle de [LARSW00] qu'on retrouverait en distinguant les configurations selon que les tranches sont vides ou non.

Le critère sémantique de fragmentation utilisé dans cette thèse s'inspire grandement de ce critère syntaxique. Il tente d'en reproduire les effets, notamment dans sa version développée. (Cf. section 4.1.4) Nous avons effectivement cherché à reproduire l'action de la seconde règle, en séparant dans les boucles imbriquées les cellules accédées lors de l'itération courante de la boucle externe et celles accédées lors de ses itérations précédentes.

Nous donnons maintenant quelques différences de notre critère par rapport au critère syntaxique de [HP08]. Notre critère maintient une partition distincte pour chaque tableau et considère de la même manière les cellules lues et les cellules écrites. Cela n'entrainera généralement aucune différence par rapport au critère syntaxique qui choisira astucieusement les bonnes translations à considérer dans chaque tableau. Il s'agit là d'une différence assez subtile entre les deux méthodes, qui produira rarement des différences dans les fragmentations désignées.

Ici, nous nous intéresserons plutôt aux différences inhérentes à la nature syntaxique du critère. En changeant la syntaxe d'un programme sans en changer la sémantique, on peut mettre en difficulté le critère. En particulier, si deux indices sont égaux et qu'on utilise l'un d'entre eux pour accéder à un tableau, il n'est pas toujours évident que c'est celui-là qu'il faut inclure dans la partition. Considérons le programme suivant.

$A[1] \leftarrow 0$
$i \leftarrow 2$
Tant que $i < n$ **faire**
  $A[i] \leftarrow 1$
  $i \leftarrow i + 1$

Le critère syntaxique produira les tranches $A[1]$, $A[2..i[$, $A[i]$, $A]i..n]$ avec lesquelles l'analyse du programme sera précise. On peut modifier le programme sans changer le résultat de son exécution :

$i \leftarrow 1$
$A[i] \leftarrow 0$
$i \leftarrow i + 1$
Tant que $i < n$ **faire**
  $A[i] \leftarrow 1$
  $i \leftarrow i + 1$

Cette fois, le critère syntaxique ne distinguera plus $A[1]$ qui sera confondue dans la tranche $A[1..i[$. Le programme ne peut plus être analysé précisément. Notre critère à l'inverse distinguera les cellules selon l'instruction à laquelle elles ont été accédées et produira les mêmes fragmentations pour les deux programmes ci-dessus.

Un autre exemple de programme qui ne peut directement être analysé par le critère syntaxique est le programme qui cherche non seulement un élément maximum dans un tableau, mais également son indice :

```
max ← A[1]
imax ← 1
Pour i de 2 à n faire
    Si A[i] > max alors
        max ← A[i]
        imax ← i
```

Puisque *imax* n'apparait dans aucun des accès aux tableaux, il ne sera pas présent dans la partition. Tandis que le critère sémantique utilisera l'égalité entre *i* et *imax* comme prétexte pour conserver $A[imax]$. Notons que si on réécrit le test en $A[i] > A[imax]$ alors le critère syntaxique pourra être utilisé.

Nous donnons un dernier exemple pour lequel le critère sémantique peut potentiellement donner un meilleur résultat mais ne le fera pas si la fragmentation est abstraite par des diagrammes de tranches.

```
Pour i de 1 à n faire
    Si i ≤ j alors
        A[i] ← 1
    sinon
        A[i] ← 0
```

Encore une fois, le critère syntaxique ne créera qu'une seule tranche pour toutes les cellules affectées, tandis que le critère sémantique cherchera à séparer les effets des deux affectations. Cependant, les diagrammes interdisent l'utilisation de la tranche $A[j..i]$ car il n'est pas vrai à tout moment de la boucle que $j \leq i$.

En pratique, ce critère donnera très souvent des bonnes partitions de tableau mais son caractère syntaxique lui fait perdre en souplesse et une légère modification de la syntaxe peut le mettre en défaut. On notera toutefois que les critères sémantiques, et pas seulement celui introduit dans cette thèse, peuvent également être mis en défaut par des manipulations de ce genre : des égalités d'indices inopportunes peuvent compliquer significativement le choix des bons indices à utiliser là où le premier choix, celui de l'expression syntaxique, convient. Un tel exemple est donné en section 10.2.3.

9.2.3 Fragmentation par sous-approximation de prédicat

L'approche précédente consistait à deviner une bonne fragmentation des tableaux en observant la syntaxe du programme et à chercher les propriétés des fragments. Une approche opposée est de fixer les propriétés que l'on souhaite trouver et ensuite de chercher quels sont les fragments sur lesquels ces propriétés sont vraies.

Il s'agit de l'approche choisie dans [GMT08] où l'analyse est paramétrée par des modèles de propriétés à chercher. Par exemple, si on veut prouver qu'un tableau est trié, on demande à l'analyse de trouver des propriétés suivant le modèle

$$a[\alpha] \leq b[\beta]$$

L'analyse doit alors trouver pour quels tableaux a et b et pour quels couples d'indices α et β cette propriété est vraie. Quoique la formulation du problème soit différente, il s'agit bien du même problème : celui de la recherche d'une fragmentation adéquate.

Le principe général peut être décomposé en deux. D'une part, on réalise une analyse classique par interprétation abstraite sur le programme en s'autorisant à introduire temporairement des variables représentant une cellule de tableau, ou le champ d'un nœud d'une structure de donnée. D'autre part, tout au long de cette analyse, on extrait des valeurs abstraites les propriétés suivant les modèles de formules choisis. Typiquement, les propriétés qui concernent des cellules de tableaux ou de structures de données seront perdues avant une union. C'est donc un moment opportun pour chercher quelles sont celles qui vérifient les modèles de formule. On regardera dans la valeur abstraite quelles sont les variables qui valident chaque modèle de formule. Ceci implique qu'on ait choisit un modèle de formule cohérent avec le domaine abstrait utilisé.

On collecte ainsi pour chaque modèle de formule n-aire des n-uplets de cellules les validant et constituant une fragmentation. Typiquement, on désignera les ensembles de n-uplets de cellules de tableau par leurs n-uplets d'indices qu'on abstraira dans le domaine des polyèdres convexes [CH78]. D'autres domaines abstraits peuvent être utilisés sous certaines conditions, et les auteurs présentent une abstraction permettant de désigner des morceaux d'un graphe d'accessibilité pour des structures de données chaînées. Notons que ces abstractions permettent de désigner précisément des relations entre les indices des cellules validant les formules modèle. On peut par exemple désigner des couples de cellules $(A[\ell_1], A[\ell_2])$ dont les indices vérifient $\ell_1 \leq \ell_2$, et ainsi instancier le modèle de formule cité plus tôt pour l'exemple des programmes de tris. Si on cible des propriétés sur des listes chaînées, on peut considérer les couples de maillons consécutifs.

Pour que l'analyse soit correcte, il faut que chaque ensemble de cellules dont on prétend qu'elles vérifient l'une des formules modèle soit effectivement une sous-approximation du véritable ensemble. Ceci a des implications sur la manière de manipuler ces ensembles de cellules. Les domaines abstraits nous fournissent des sur-approximation de la borne supérieure, alors qu'ici on en voudra une sous approximation. Par exemple, si deux ensembles de cellules vérifient la même formule modèle, on peut les regrouper en calculant la sous-approximation de leur union. Les auteurs proposent des algorithmes pour calculer ces sous-approximations ne s'appliquant toutefois pas à tous les domaines abstraits et dont la complexité est très contraignante. Dans les treillis classiques, il n'y a pas en général, de meilleure sous-approximation et les algorithmes proposés ne donnent pas non plus des sous-approximations maximales.

Pour comparer avec TVLA [LARSW00], il s'agit toujours d'une fragmentation par prédicat : on regroupe les cellules vérifiant des prédicats choisis par l'utilisateur. Mais cette fois le prédicat est n-aire et est désigné par un modèle de formule. En contrepartie, au lieu d'avoir un fragment par prédicat, on a une fragmentation par prédicat. Si une cellule vérifie deux prédicats, elle sera désignée par les deux fragmentations. Mais on n'aura pas comme dans TVLA de fragment correspondant aux cellules vérifiant les deux prédicats à la fois. Ceci peut être résumé en disant qu'on n'a pas de partitionnement de l'ensemble des cellules.

D'autre part, ici, les prédicats ne servent qu'à choisir les propriétés observées tandis que dans TVLA les prédicats servent également à désigner les fragments. La forme des fragments est d'ailleurs plus générale et ne pourrait être obtenue simplement par une combinaison d'un ensemble fini de prédicats.

Comme TVLA, il s'agit d'une analyse semi-automatique car il faut effectivement choisir les modèles de formule. Par exemple, la formule $a[\alpha] \leq b[\beta]$ est distincte de la formule $a[\alpha] \leq$

$b[\beta] + 1$. Nous sommes limités par le nombre de modèles que nous pouvons utiliser dans une analyse qui doit rester fini. On ne peut donc pas utiliser une famille de formules

$$\{\, a[\alpha] \leq b[\beta] + i \mid i \in \mathbb{Z} \,\}$$

puisqu'il faudrait alors collecter pour chacune d'entre elles des ensembles de cellules différents. Ceci était notre remarque au début du chapitre 4.

En comparaison avec nos travaux, les fragmentations qu'il est possible de décrire sont plus générales. En contrepartie, ces fragmentations sont plus difficiles à manipuler, et il sera plus difficile d'en tirer des propriétés. Les auteurs ne donnent pas de méthode permettant dériver des propriétés généralement obtenues par réduction des valeurs abstraites. Par exemple, si on utilise le modèle de formule proposé plus tôt et que lors d'une analyse on trouve

$$\begin{aligned} &\forall \ell\ 1 \leq \ell \leq n \Rightarrow A[\ell] \leq B[\ell] \\ \wedge &\forall \ell\ 1 \leq \ell \leq n \Rightarrow B[\ell] \leq C[\ell] \end{aligned}$$

on ne dérivera pas pour autant $A[\ell_1] \leq C[\ell_2]$. Tandis qu'avec les méthodes [GRS05, HP08] précédemment citées ainsi qu'avec la méthode présentée dans cette thèse on y parviendra.

Ceci constitue la principale différence avec nos travaux. Pour le choix d'un domaine abstrait, nous sommes capables d'inférer toute une classe de propriétés sur les cellules, et d'en dériver d'autres grâce aux mécanismes de réduction dans les domaines abstraits. En outre, si l'expressivité de nos domaine de fragmentation est beaucoup plus réduite, aussi bien sur la forme des fragments que sur les relations sélectionnées entre ces fragments, nous avons la capacité de déduire des propriétés de la forme de la fragmentation.

Les ensembles de cellules collectées peuvent se chevaucher sans problème. Il n'y a pas de distinction systématique des cas d'alias mais l'analyse pourra déterminer des propriétés propres à certains cas d'alias. Lorsque une affectation modifie une cellule, on retirera la cellule de tous les fragments. Il sera alors parfois nécessaire de calculer une sous-approximation des fragments privés de cette cellule. De ce point de vue, l'approche est similaire à la notre : on autorise les fragments à se chevaucher quitte à prendre les précautions nécessaires pour l'interprétation de la sémantique.

Regrouper des cellules en fonction de leurs propriétés pose un important problème pour l'abstraction de ces ensembles de cellules. Les abstractions classiques sont généralement des ensembles d'indices connexes. Supposons qu'on doive analyser le programme suivant :

Pour *i de* 0 *à* 9 **faire**
  $A[i] \leftarrow 0$

Pour *i de* 20 *à* 29 **faire**
  $A[i] \leftarrow 0$

On utilisera naturellement le modèle de formule $a[\alpha] = 0$. La méthode autorise qu'il y ait plusieurs ensembles de cellules pour le même module de formule. Mais pour assurer la terminaison de la méthode, ils imposent comme contrainte que le nombre d'ensembles de cellules soit borné par un paramètre de l'analyse. Or, si par malchance le nombre d'ensembles de cellules vérifiant

ce modèle de formule atteint déjà la limite, on sera forcé de fusionner ces ensembles de cellules pour ne pas violer la contrainte. Or les deux ensembles d'indices [0..9] et [20..29] sont disjoints et leur union n'est pas connexe. On sera donc forcé de l'approximer, et on pourra difficilement faire mieux que d'utiliser l'un ou l'autre comme sous-approximation de leur union.

Sur ce programme, notre méthode distinguera les deux ensembles correspondant à deux affectations distinctes et n'aura pas ce problème.

Pour finir sur ces travaux, nous pouvons revenir à l'un des exemples donnés dans la section 4.3 où l'on discute du choix de sous-approximation et de sur-approximation. Considérons un programme qui initialise les cellules d'indice pair à 0 et les cellules d'indice impaire à 1. Il serait vain de tenter de sous-approximer les ensembles de cellules d'indice pair et impair avec des abstractions d'ensembles d'indices connexes. Mais on peut choisir le modèle de formule $0 \leq a[\alpha] \leq 1$ et collecter sans problème les cellules le vérifiant. Cette dernière remarque nous permet de souligner le caractère semi-automatique de l'analyse et le fait que l'utilisateur devra sélectionner avec soin les modèles de formules choisis pour l'analyse.

9.2.4 Fragmentation sémantique

Le principe précédent de collecter des cellules lors de l'analyse se retrouve dans [CCL11], mais les cellules n'y sont plus regroupées selon leurs propriétés. Leur analyse construit une partition du tableau, un peu à la manière de [GDD+04, GRS05, HP08]. Les tableaux sont partitionnés selon un ensemble totalement ordonné de bornes. Si i est une de ces bornes, alors on distingue les cellules d'indice inférieur ou égal à i et celles d'indice strictement supérieur. La différence avec les travaux précédents, réside dans le fait que les bornes sont totalement ordonnées, c'est à dire qu'on ne peut partitionner selon deux expressions d'indice qui ne soient pas comparables. Par exemple, si une borne i peut aussi bien être supérieure qu'inférieure à une borne j, alors seule l'une d'entre elles peut servir à partitionner le tableau. En outre, pour être tout à fait précis, ces bornes sont en fait des classes d'équivalences d'expressions. C'est ce qui donne la nature sémantique à ce partitionnement : on peut interchanger une expression de borne avec une autre égale.

Les auteurs se fixent comme objectif de rechercher des propriétés non-relationnelles sur les différents fragments des tableaux. Par conséquent, la possible vacuité des fragments ne pose pas de problème.

Contrairement aux méthodes précédemment cités, il n'y a pas ici d'objectif clairement défini dans le choix de la fragmentation. On sait en revanche comment la partition est calculée à mesure que l'analyse progresse. On part d'une partition composée de deux bornes : la borne inférieure et supérieure des indices valides pour ce tableau, classiquement 0, et la taille du tableau. Cette partition est ensuite modifiée dans trois situations.

- Lorsque une cellule est affectée, on introduit un singleton pour cette cellule dans la partition. On ajoute les bornes i et $i+1$, quitte à supprimer les bornes qui ne seraient pas comparables avec ces deux là.
- Lorsque une variable d'indice est modifiée, on met à jour la partition en conséquence pour la préserver.
- Lorsque l'on doit calculer une union, une intersection, un élargissement ou une comparaison de valeurs abstraites, on unifie les partitions.

La caractéristique principale de l'algorithme d'unification est de ne conserver dans la partition unifiée que des bornes qui soient présentes dans les deux partitions à unifier. À chaque fois qu'on doit calculer l'union de deux valeurs abstraites venant de deux branches distinctes du programme, on ne conservera que les bornes présentes dans les deux branches à la fois. En particulier, une boucle ne peut faire apparaitre de nouvelles bornes et ne conservera que les bornes présentes à l'entrée de la boucle.

Si lors de l'union de deux valeurs abstraites on doit unifier deux partitions et que deux bornes ne sont pas ordonnées de la même manière dans deux partitions à unifier, c'est que les bornes deviennent incomparables après l'union. Dans ce cas, comme nous l'avons indiqué plus tôt, il ne sera pas possible de conserver les deux bornes dans la partition. Le comportement de l'algorithme d'unification dans cette situation est variable et dépend de la forme des deux partitions : ou bien l'algorithme conservera l'un des indices, ou bien il n'en conservera aucun.

On peut voir les diagrammes comme une généralisation de ce partitionnement, en lui ajoutant la capacité de considérer des bornes incomparables. Les partitions telles que présentées dans ces travaux correspondent à un cas particulier de diagramme où l'ensemble des bornes B est totalement ordonné. Pour reprendre le cas précédent, lorsque deux bornes deviennent incomparables lors d'une unification, les diagrammes construiront deux chemins permettant de conserver chacune des deux bornes.

Entre leur abstraction des fragmentations et la notre, la différence se résume à un choix de compromis. Tandis que les diagrammes sont plus précis, ils entraînent un nombre de fragments pouvant être très supérieur. (Au pire quadratique au lieu de linéaire.) Le choix n'est pas évident. Beaucoup de programmes s'analyseront correctement avec ces partitions simples. Mais si il semble qu'on ait rarement besoin de partitionner un tableau selon deux bornes incomparables, le cas arrive plus souvent qu'on ne pourrait le croire : les critères de fragmentation ont tendance à ajouter plus de bornes que nécessaire quitte à les supprimer dans des itérations postérieures. On peut se retrouver temporairement avec des bornes incomparables même si dans la fragmentation finale ce n'est plus le cas.

Comparons maintenant les critères de fragmentation. Celui-là partage avec le notre les défauts dûs à sa nature sémantique par rapport aux critères syntaxiques. Lorsque il y a plusieurs indices égaux, et que l'un d'entre eux est utilisé pour désigner une cellule affectée, les critères sémantiques vont tous les considérer. Cette énumération peut même parfois conduire à l'échec de la recherche de partition. (Le problème est détaillé dans la description de l'une des expérimentations en section 10.2.3) Tandis qu'un critère syntaxique sélectionnera le seul indice pertinent dans la plupart des cas, même s'il échouera dans d'autres.

Le fait de ne conserver que les bornes présentes dans les deux partitions à unifier est important pour la terminaison de l'analyse. C'est ce qui assure que le nombre de tranches reste borné. Mais ceci a des implications négatives sur la précision de l'analyse. Cette caractéristique peut être justifiée pour l'union en disant que, si une tranche n'est pas distinguée dans l'une des deux partitions, alors il y a peu de chances qu'on puisse en déduire quelque chose dans l'union des deux valeurs abstraites. L'argument peut être invalidé par l'exemple suivant. Supposons que l'on ait deux partitions d'un tableau A, d'une part composée des tranches $[0..i[$ et $[i..n[$ et d'autre part de la tranche $[1..n[$. Supposons également qu'on ait les valeurs abstraites pour ces deux partitions

$$A[0..i[\in [0,1], A[i..n[\in [3,4]$$

et

$$A[0..n[\in [1, 3]$$

L'algorithme d'unification ne conservant que les bornes présentes dans les deux partitions, on n'aura dans la partition unifiée que la tranche $A[0..n[$ ayant pour propriété $A[0..n[\in [0, 4]$. Alors que en prenant toutes les bornes, on aurait eu une valeur abstraite plus forte :

$$A[0..i[\in [0, 3], A[i..n[\in [1, 4]$$

C'est le résultat qu'on obtiendrait avec notre critère de fragmentation appliqué aux diagrammes de tranches à la condition que les deux tranches correspondent à deux accès différents.

L'incapacité pour les boucles de faire apparaitre de nouveaux singletons est également handicapante. Si une boucle d'indice i commence à itérer pour $i = 0$, alors on pourra conserver la borne i lors de l'unification puisque 0 est initialement présent dans la partition. En revanche si on commence à itérer pour $i = 1$, quand bien même la première itération de la boucle ajoutera la borne $i = 1$ à la partition, on ne pourra conserver ni l'une ni l'autre puisqu'elles ne sont pas initialement présentes. Dans le cadre de ces travaux, ce n'est pas particulièrement gênant, car si ne pas utiliser la première cellule d'un tableau n'introduit pas nécessairement de bug dans un programme, cela peut toutefois être considéré comme une erreur de programmation.

On aurait un problème similaire pour un programme ressemblant à ceux donnés en début de section 4.3. Il s'agissait de programmes parcourant successivement diverses sections du tableau. Conservons les deux premières boucles et inversons l'ordre d'itération de la première :

Pour i *de* j *à* 0 **faire**
    $A[i] \leftarrow 0$

Pour i *de* $j + 1$ *à* n **faire**
    $A[i] \leftarrow 1$

Si la première boucle n'avait pas été inversée, c'est à dire si elle allait de manière croissante de 0 à j, on aurait eu $i = 0$ en entrant dans la boucle, et donc i aurait été une borne présente dans la partition. En sortant de la boucle, $i = j$ et on peut donc introduire j à la place de i. Mais avec la version inversée, j n'étant égal à aucune borne de la partition initiale, on ne peut pas l'introduire dans la partition.

Un autre exemple, qui n'est lui pas artificiel est celui de la segmentation d'un tableau, comme celle opérée dans un tri par segmentation. Ce programme ne crée que des propriétés relationnelles. Bien que les travaux dont nous parlons s'intéressent avant tout aux propriétés non-relationnelles, le critère de fragmentation peut néanmoins être appliqué.

$i \leftarrow 1$
$j \leftarrow 1$
Tant que $i < n$ **faire**
    Si $A[i] < A[0]$ **alors**
        $A[j] \leftrightarrow A[i]$
        $j \leftarrow j + 1$
    $i \leftarrow i + 1$

$A[j] \leftrightarrow A[0]$

Dans cette version, le pivot $A[0]$ n'est pas mentionné avant d'entrer dans la boucle. La borne 1 n'est donc pas dans la partition, et on ne peut pas décrire précisément l'invariant de la boucle de segmentation.

Terminons par un dernier exemple moins évident sur le problème de conservation de borne lors d'unification. Dans celui-ci, la seconde boucle est tout fait classique, mais l'accès aux cellules n'est réalisé que dans l'une des branches d'une structure conditionnelle.

Pour *i de* 0 *à n* **faire**
  $A[i] \leftarrow 0$

Pour *i de* 0 *à n* **faire**
  Si *cond* **alors**
    $A[i] \leftarrow A[i] + 1$

Supposons que *cond* est une condition qui n'est pas toujours vraie ou toujours fausse. L'affectation ajoutera les bornes i et $i + 1$ dans la partition. Mais $i + 1$ n'est pas dans la partition. Dès que l'on sort de la structure conditionnelle, on doit donc retirer au moins la borne $i + 1$ de la partition. Par conséquent, la valeur abstraite attribue à $A[i..n[$ une valeur comprise entre 0 et 1 à la première itération, et cet intervalle grandit d'une itération sur l'autre. On perd donc la propriété que toutes les valeurs des cellules du tableau sont dans l'intervalle $[0, 1]$.

Tous les programmes que nous venons de citer sont correctement analysés par notre méthode. Celle-ci autorise l'apparition de nouvelles bornes au sein d'une boucle. Notons que ce qui est pertinent pour la recherche de propriétés relationnelles n'est pas forcément pertinent pour la recherche de propriétés non-relationnelles. Dans la segmentation d'un tableau, il n'est pas utile d'avoir $A[0]$ dans la partition pour en tirer des propriétés non-relationnelles : $A[0]$ n'a aucune propriété de ce genre avant la boucle. Mais $A[0]$ a avec les autres tranches la relation $A[1..j[< A[0] \leq A[j..i[$. Puisque $A[1..j[$ et $A[j..i[$ sont vides à l'entrée de la boucle, cette propriété reste valide. La présence de $A[0]$ dans la partition n'est donc utile que si on cherche des propriétés relationnelles. Ceci ne contredit toutefois pas notre première remarque : ce n'est pas parce que les propriétés cherchées sont non-relationnelles qu'il est inintéressant de conserver toutes les bornes lors d'une unification.

Enfin, nous avons cité des exemples de bornes manquantes dans la partition. Mais il est possible, à l'inverse, que le critère apporte des bornes superflues. Si deux bornes égales cessent de l'être après une unification, l'algorithme d'unification conservera toutes ces bornes, même si elles ne délimitent rien.

9.2.5 Conclusion

Notre critère de fragmentation est la tentative de réunir deux méthodes de fragmentation. D'une part, il reprend le principe de collecte de cellule à l'exécution [GMT08, CCL11]. D'autre part, il essaie de reproduire les partitions obtenues dans [HP08] qui donnent très souvent de bons résultats lorsqu'il n'y a pas de subtilités dues à des égalités d'indices. Il en combine certains avantages mais échoue parfois à trouver les partitions adéquates à l'analyse de certains exemples, qui sont pourtant trouvées par au moins l'une de ces méthodes. Quelques uns de ces exemples seront présentés dans la section 10.2 consacré aux résultats d'expérimentation.

9.3 Méthodes d'analyse semi-automatiques

Nous citons dans cette section diverses méthodes semi-automatique pour l'analyse de programmes manipulant des structures de données. L'interprétation abstraite ne nécessite qu'une faible intervention de l'utilisateur. Celui-ci n'aura qu'à choisir les domaines abstraits adéquats à l'analyse de tout un programme. À l'inverse, pour les méthodes d'analyse semi-automatiques, une intervention importante de l'utilisateur est requise et il devra, selon les cas, fournir des invariants ou préciser des paramètres d'analyses pour l'analyse d'une fonction ou d'un morceau de programme.

Si les approches peuvent s'éloigner de l'interprétation abstraite, on retrouvera tout de même des problématiques similaires dans l'analyse de programmes manipulant des structures de données.

9.3.1 Logiques décidables sur les tableaux et les structures de données

Dans le domaine de la vérification, de nombreux travaux portent sur la définition de logiques décidables dans lesquelles s'expriment des propriétés de programmes et sur le développement de procédures de décision dans ces logiques. A partir d'un programme annoté avec des pré-conditions, des post-conditions et des invariants de boucle appartenant à une logique décidable, on utilise ces procédures de décision afin de prouver que les secondes sont bien impliquées par les premières. La plupart de ces méthodes nécessitent que l'utilisateur donne également les invariants de boucles.

Nous décrivons dans cette section quelques travaux récents sur des logiques destinées à la vérification de programmes manipulant des tableaux ou des structures de données. Les logiques que nous évoquerons permettent toutes la quantification universelle sur les cellules.

La logique décidable introduite dans [BMS06] est le fragment APF [2] qui autorise des formules de la forme

$$\forall \mathcal{L},\ \varphi(\mathcal{L}) \Rightarrow \psi(\mathcal{L})$$

où $\mathcal{L}$ est un ensemble de variables entières quantifiées universellement et φ et ψ sont deux formules, elles aussi contraintes :

- les atomes de φ sont ou bien l'égalité $=$ ou bien l'inégalité $\leq$;
- les expressions comparées dans φ sont ou bien une variable $\ell \in \mathcal{L}$ ou bien une expression dans l'arithmétique de Presburger [Pre29] sur les variables du programme ;
- ψ est une formule d'un fragment décidable dans lequel on s'autorise à ajouter des expression $A[\ell]$ pour un tableau A et une variable $\ell \in \ell$ donnés.

Ce fragment contient par exemple la formule

$$\forall \ell_1, \ell_2,\ (i \leq \ell_1 \leq \ell_2 \leq j) \Rightarrow (A[\ell_1] \leq A[\ell_2])$$

exprimant qu'un tableau A est trié entre les indices i et j.

Les auteurs identifient également quelques extensions de cette logique qui deviennent indécidables. Le résultat de décidabilité est brisé

- quand on introduit des accès imbriqués aux tableaux $A[B[\ell]]$ avec $\ell \in \mathcal{L}$;
- quand on introduit dans φ des accès $A[\ell]$ avec $\ell \in \mathcal{L}$;
- quand on s'autorise à utiliser l'arithmétique de Presssburger sur les variables $\ell \in \mathcal{L}$.

2. Acronyme de « Array Property Fragment »

Les auteurs de [HIV08] introduisent la logique LIA [3] incomparable avec AFP. La forme générale des formules est toujours

$$\forall \mathcal{L},\ \varphi(\mathcal{L}) \Rightarrow \psi(\mathcal{L})$$

mais les contraintes sur φ et ψ différent :

- φ peut contenir des contraintes de la forme $\ell = \ell' + c$ avec $\ell \in \mathcal{L}$ et $c \in \mathbb{Z}$, mais contrairement à AFP, ne permet pas d'exprimer d'inégalités entre les variables de $\mathcal{L}$;
- φ peut contenir des contraintes de congruence $\ell \equiv_n c$ avec $c, n \in \mathbb{N}$, $c < n$;
- φ peut contenir des comparaisons entre les variables de $\mathcal{L}$ et des sommes ou différences des variables d'indice du programme et de constantes ;
- ψ est une contrainte de zones pouvant faire intervenir des termes $A[\ell]$.

Notons que puisqu'il est possible d'avoir des contraintes $\ell = \ell' + c$ on peut aussi bien autoriser $A[\ell]$ que $A[\ell' + c]$ dans ψ.

La logique DWL [4] présentée dans [BHJS07] permet de raisonner sur des mots. Les mots pourraient être par exemple des suites de valeurs d'une liste chaînée ou d'un tableau. Les formules de cette logique autorisent la comparaison large et stricte des variables d'indice utilisées dans les accès aux mots ainsi que leur comparaison à zero. Elle autorise également n'importe quel atome ou relation d'une théorie décidable sur les contenus, en étendant cette théorie avec des accès aux tableaux par une variable d'indice. Les auteurs en distinguent un fragment décidable, celui des formules de la forme $\exists^*\forall^*\psi$ où ψ est une formule sans quantificateur.

La logique DWL présente le défaut de ne pas être close par les manipulations de pointeur. La logique CSL [5] [BDES09] n'a pas ce défaut. Cette dernière logique permet de travailler sur des programmes manipulant des structures dynamiquement liées, des tableaux et des combinaisons des deux. Elle permet d'exprimer

- des contraintes sur l'accessibilité entre deux positions du tas en suivant des champs pointeurs,
- des contraintes linéaires sur la taille des listes et les indices des tableaux et
- des contraintes sur les valeurs des données.

La logique UABE [6] introduite dans [ZHWG10] considère le cas où l'ensemble des valeurs possibles pour les cellules de tableau est borné. Ceci est motivé par le fait que dans la pratique, les entiers sont classiquement codés sur un nombre borné de bits. Sous cette condition la logique est décidable et autorise une large classe de formules permettant l'alternance de quantificateurs et les accès imbriqués aux cellules.

Pour appliquer les procédures des décision et vérifier que les post-condtions sont impliquées par les pré-conditions, il est en général nécessaire que l'utilisateur donne également les invariants de boucles. Néanmoins, [BHI+09] propose une méthode pour automatiser la génération d'invariants dans le cas de programme avec des boucles non-imbriquées. Leur méthode permet, dans le cas de la logique LIA [HIV08] d'inférer les post-conditions des boucles réduisant ainsi le besoin d'intervention humaine pour ces programmes.

3. Acronyme de « Logic on Integer Arrays »
4. Acronyme de « Data Word Logic »
5. Acronyme de « Composite Structures Logic »
6. Acronyme de « Unbounded Array with Bounded Element »

Conclusion Le principe est relativement éloigné de celui de l'interprétation abstraite. L'interprétation abstraite permet de synthétiser les invariants et de découvrir des propriétés, sans aide humaine et sur n'importe quelle classe de programme pourvu qu'on ait fourni la sémantique abstraite adéquate. Les méthodes d'analyse par interprétation abstraite sont généralement moins coûteuses que les procédures de décision. Elles n'ont pas besoin de travailler sur un fragment de logique décidable et peuvent donc s'aventurer sur des classes plus vastes de formules. En contrepartie, l'interprétation abstraite ne fournit aucune garantie de résultat : si elle échoue à découvrir une propriété, cela ne signifie pas que la propriété est fausse et on ne peut rien conclure. Notons également que l'interprétation abstraite est souvent plus facile à étendre, par la définition de nouveaux domaines abstraits et leur combinaison.

9.3.2 Abstraction par prédicats

L'abstraction par prédicats [GS97] est une technique d'interprétation abstraite permettant de désigner les ensembles d'états atteignables par les prédicats qu'ils vérifient. On choisit un ensemble de prédicats sur les variables du programme. Les valeurs abstraites sont des ensembles de valuations de ces prédicats, que l'on peut représenter par des formules propositionnelles sur ces prédicats. L'ensemble des valeurs abstraites est donc fini, ce qui a notamment pour avantage de permettre une analyse sans élargissement.

L'ensemble des prédicats peut être choisi par l'utilisateur ou automatiquement à partir du programme analysé. Il peut éventuellement varier selon les points de contrôle considérés. Ceci permet une grande généricité de l'analyse qui pourra donc s'appliquer à une grande classe de propriétés.

En contrepartie, cette généricité implique des difficultés à calculer une approximation correcte des fonctions de transfert. Les méthodes utilisées pour ce calcul reposent sur l'aide d'un démonstrateur de théorèmes. On leur demandera de démontrer si une formule utilisant les prédicats est impliquée par le résultat de l'application d'une fonction de transfert. Plusieurs méthodes ont été développées [DDP99, SS99] en se fixant pour but de minimiser le nombre d'appels au démonstrateur de théorème, mais la méthode reste coûteuse.

L'abstraction par prédicat peut être directement appliquée aux tableaux ou aux structures de données en considérant des prédicats contenant des quantificateurs. Par exemple en choisissant le prédicat $(\forall \ell,\ 1 \leq \ell < i \Rightarrow A[\ell] = 0)$ on peut espérer vérifier un programme d'initialisation de tableau. D'ailleurs, ce prédicat est tellement particulier qu'il ne peut guère servir à autre chose. Mais demander à l'utilisateur de fournir de tels prédicats revient quasiment à lui demander de fournir les invariants de boucle. On n'est plus dans une optique de découverte de propriété. Par ailleurs tout le travail de vérification est en fait réalisé par le démonstrateur de théorème : on ne tire absolument pas profit du cadre défini par l'interprétation abstraite. Dès qu'on introduit des quantificateurs dans les prédicats, ces démonstrateurs doivent d'ailleurs être capables de travailler sur des formules du premier ordre.

Le problème en essence est que les quantificateurs doivent rester à l'intérieur des prédicats. Par conséquent, chaque prédicat utilise une variable quantifiée distincte. Ceci est très limitant dans la recherche d'invariants. Dans la suite de cette section nous énumérons diverses évolutions de l'abstraction par prédicats autorisant certaines classes de formules quantifiées sur les prédicats.

Les auteurs de [FQ02] proposent d'introduire des variables libres dans les prédicats. L'utilisateur fournit un ensemble de prédicats pouvant faire intervenir une ou plusieurs variables libres.

Par exemple, si cet ensemble de prédicats contient les trois prédicats $(1 \leq \ell)$, $(\ell < i)$ et $(A[\ell] = 0)$ sur la variable libre ℓ, alors l'analyse pourra générer la formule

$$(1 \leq \ell \wedge \ell < i) \Rightarrow A[\ell] = 0$$

qui lorsque l'on quantifie universellement ℓ donne l'invariant cité plus tôt. Mais cette fois, d'autres combinaisons de ces prédicats pourront également produire d'autres invariants ce qui rend l'analyse bien plus générique.

Ce type d'ensembles de prédicats est aussi plus facile à générer automatiquement. Les auteurs de [FQ02] proposent des heuristiques pour générer ces ensembles de prédicats et qui rappellent les critères de fragmentations présentés en section 9.2. Nous décrivons brièvement ces heuristiques pour les tableaux. Pour chaque accès $A[exp]$ non constant à une cellule d'un tableau A (l'accès est non constant si la cellule accédée peut changer d'une itération à l'autre) on introduit une nouvelle variable libre ℓ et

- on considère les prédicats $(0 \leq \ell)$ et $(\ell < i)$ pour toute variable entière i dans la portée courante,
- si le type de la cellule accédée est un pointeur, on considère le prédicat $A[\ell] = \texttt{null}$
- si le type de la cellule est entier, on considère les prédicats $A[\ell] \leq c$, $A[\ell] \geq c$, $A[\ell] < \ell$ où c est la valeur de l'expression d'indice exp à l'entrée de la boucle.

La première règle introduisant les prédicats comparant l'indice ℓ à des bornes potentielles ressemble à celle de [GRS05] que nous avons évoquée plus tôt. Néanmoins, beaucoup moins de prédicats sont générés ici, probablement par un souci d'économie : ajouter des prédicats coûte beaucoup plus cher dans le cadre de l'abstraction par prédicat. En conséquence, ces heuristiques ne fonctionneront pas pour de nombreuses variations de la boucle de base : boucle partant de l'indice 1 au lieu de 0, boucle parcourant le tableau en sens inverse, boucle utilisant des expressions d'indice plus compliquées. La génération automatique des ensembles de prédicats à fait l'objet d'autres travaux postérieurs [LB04, JM07].

Si au contraire l'utilisateur choisit lui-même les prédicats, alors la méthode a quand même l'avantage de permettre une très grande expressivité.

Une autre piste pour l'introduction automatique des prédicats est de suivre le principe du raffinement d'abstraction guidé par contre-exemple [CGJ+03]. Dans le cadre de l'abstraction par prédicats, cela consiste à introduire itérativement de nouveaux prédicats à mesure que l'on trouve des contre-exemples aux propriétés que l'on souhaite prouver. Pour les tableaux, le principe a été utilisé dans [JM07, BHMR07, SPW09].

On retrouvera dans ces études une problématique familière. Supposons que la méthode de raffinement nous propose en première approche d'ajouter un prédicat $\varphi(A[c])$ pour décrire des propriétés d'une cellule d'indice c constant. Le faire nous ferait courir le risque de devoir ajouter d'autres prédicats similaires pour des constantes d'indice différentes, et dans le pire cas, une infinité d'autres prédicats. On procèdera donc comme dans [FQ02] et on ajoutera le prédicat $\varphi(A[\ell])$ pour une variable ℓ libre. Ce qui nous ramènera au même problème : celui du choix des prédicats à ajouter sur ℓ. On retrouve alors des idées déjà rencontrées. Par exemple, dans [JM07], on pourra généraliser un prédicat $\varphi(A[1])$ par

$$\forall \ell,\ 1 \leq \ell \leq i \ \Rightarrow\ \varphi(A[\ell])$$

à la condition $i = 1$. Ce comportement est le même que celui de [CCL11] et celui des diagrammes de tranches lors des fissions. Dans [SPW09] on retrouvera l'introduction de prédicats

$\ell < i$ et $i < \ell$ où i est une variable du programme, rappelant les prédicats utilisés pour la fragmentation dans [GRS05].

Enfin, une autre méthode proposée dans [SG09] demande à l'utilisateur, en plus de choisir l'ensemble de prédicats, de choisir un patron de formule indiquant comment doivent être combinés ces prédicats. Ce patron est une formule pouvant contenir des quantificateurs sur un ensemble de variables propositionnelles. L'analyse prend en entrée un programme annoté avec des assertions et découvre automatiquement par quels prédicats ces variables propositionnelles devront être remplacées pour que la formule devienne un invariant, une précondition ou une postcondition.

La classe de formules qu'il est possible de générer grâce à ces patrons est effectivement beaucoup plus large que dans les méthodes précédemment proposées. Mais le choix du patron nécessite une implication importante de l'utilisateur et revient bien souvent à lui demander directement l'invariant.

9.3.3 Abstraction par prédicats paramétriques

L'abstraction par prédicats paramétriques [Cou03, All08] possède un point commun avec les méthodes d'analyse de la section précédente : elle repose sur le choix d'un ensemble de prédicats. La différence, comme son nom l'indique, est que cette fois les prédicats sont paramétriques. Chacun de ces prédicats paramétriques $P(x_1, \ldots, x_n)$ peut être instancié en substituant les variables $x_1, \ldots, x_n$ par des constantes, des variables du programme, etc.

Chaque prédicat paramétrique dénote donc en réalité une classe potentiellement infinie de prédicats. L'analyse est plus générique et nécessite la définition de moins de prédicats.

En revanche, puisqu'on manipule maintenant une infinité de prédicats, les méthodes d'abstraction par prédicats ne s'appliquent plus. On ne peut plus se reposer sur l'usage des démonstrateurs automatiques. On retourne à des pratiques plus classiques en interprétation abstraite où on va devoir définir des algorithmes dédiés pour interpréter l'effet des fonctions de transfert sur les prédicats paramétriques. Cela revient à construire un domaine abstrait pour chaque prédicat paramétrique quitte à combiner ces domaines abstraits par produit réduit. Il faudra également choisir un domaine numérique pour l'instanciation des paramètres $x_1, \ldots, x_n$.

Pour donner un exemple de prédicat paramétrique, citons le prédicat paramétrique *LT* introduit dans [Cou03] :

$$LT(a, i, j, k, l) \;\equiv\; \forall \ell_1, \ell_2,\; i \leq \ell_1 \leq j \wedge k \leq \ell_2 \leq l \Rightarrow a[\ell_1] \leq a[\ell_2]$$

Ce prédicat permet d'exprimer que les valeurs d'une tranche sont inférieures à celles d'une autre.

L'abstraction par prédicats paramétriques présente des problématiques communes avec nos travaux. La recherche des valeurs à donner aux paramètres des prédicats, lorsqu'il s'agit d'indices, est tout à fait similaire à la recherche des bornes à introduire dans les diagrammes. La problématique partagée serait beaucoup plus vaste si on s'autorisait plusieurs instances du même prédicat. Car dès qu'on s'autorise plusieurs prédicats, il faut d'une part trouver un moyen de limiter leur nombre, ce qui est l'objet des critères de fragmentation, et d'autre part savoir comment on les combine, ce qui est l'objet des diagrammes de fragmentation. Quand on ne considère qu'un seul prédicat, ou qu'on ne cherche pas à combiner les prédicats, on n'a pas non plus à se soucier de la possible présence de fragments vides.

10 Implémentation et Expérimentations

Ce chapitre est consacré à la présentation d'un prototype d'analyseur implémentant les méthodes introduites dans cette thèse. Cet analyseur est un clone de Enkidu [Pé10], un autre interprète abstrait développé par Mathias Péron. Ce prototype est entièrement autonome et peut prendre en entrée des programmes écrits dans une version du langage Fast [BFLP03] étendue avec les expressions de tableau ou dans un C très simplifié. Il analyse ces programmes et produit un code source commenté avec les propriétés calculées sur le programme.

Nous commençons par donner une présentation générale du prototype avant de décrire plus en détail les différents composants. Puis nous commenterons les expérimentations réalisées, les résultats obtenus et les difficultés rencontrées à travers la description d'une petite série d'exemples de programmes d'entrée.

10.1 Implémentation

L'analyseur a été cloné en Mai 2008 du prototype Enkidu développé par Mathias Péron. Le but était alors d'y introduire une analyse recherchant des propriétés liant les multi-ensembles des valeurs des tableaux [PH10]. Par la suite trois analyses spécialisées dans la recherche de propriétés de contenus de tableau ont été implémentées dans l'analyseur entre 2008 et 2012. Chacune supplantait la précédente et était construite sur une théorie plus aboutie.

Il est entièrement programmé en OCaml. Il utilise les librairies Fixpoint permettant de résoudre des systèmes d'équations de point fixe et Camllib librairie généraliste, toutes deux développées par Bertrand Jeannet. Il utilise également un analyseur syntaxique pour le langage Fast écrit par Laure Gonnord. [Gon07]

L'analyseur a été développé dans un but purement académique. Une grande attention a été portée à la modularité afin de pouvoir à moindre effort substituer un module à un autre et vérifier plus aisément des variations de la théorie, et intégrer rapidement toute nouvelle idée. Cette modularité a un coût en complexité de l'architecture et en performances. Elle permet toutefois d'éprouver l'une des affirmations de cette thèse : l'indépendance entre les différents composants de l'analyse de structures de données. Le domaine abstrait, le domaine de fragmentation et le domaine de bornes pour les diagrammes ont des interfaces abstraites et chacun ignore l'implémentation des autres.

Si l'analyseur convient pour l'expérimentation, il est de peu d'usage pratique. Les analyseurs syntaxiques sont incomplets et ne comprennent qu'un fragment réduit des langages d'entrée. En outre, l'analyseur est exclusivement intra-procédural et ne peut analyser des programmes étendus sur plusieurs fonctions qui ne sont pas l'objet de cette thèse. Enfin, peu d'attention a été accordée à l'optimisation du temps d'exécution. Les structures de données utilisées ne sont pas toujours adaptées et l'inspection détaillée des temps d'exécution révèlera que c'est dans la manipulation de ces structures de données que l'analyseur passe l'essentiel de son temps.

```
int main(void)
{
  assume(n > 0);

  i = 0;
  while (i < n)
  {
    A[i] = x;
    i++;
  }

  A[n] = x+1;
}
```

```
int main(void)
{
  assume(n > 0);
  /* 0 < n */
  i = 0;
  /* A[0..i[ = A[i-1], A[0..i[ = x, i <= n, 0 <= i, 0 < n */
  while (i < n)
  /* A[0..i[ = A[i-1], A[0..i[ = x, i < n, 0 <= i, 0 < n */
  {
    A[i] = x;
    /* A[0..i[ = A[i], A[0..i[ = x, A[i] = x,
       i < n, 0 <= i, 0 < n */
    i++;
  }
  /* A[0..i-1[ = A[i-1], A[0..i-1[ = x, A[i-1] = x,
     i = n, 0 < i, 0 < n */
  A[n] = x+1;
}
/* A[0..i-1[ = A[n] - 1, A[0..i-1[ = A[i-1], A[0..i-1[ = x,
   A[n] = A[i-1] + 1, A[n] = x + 1, A[i-1] = x,
   i = n, 0 < i, 0 < n */
```

FIGURE 10.1 : À gauche un programme simple en C, à droite le résultat de l'analyseur sur ce programme.

10.1.1 Utilisation de l'analyseur et interprétation du résultat

L'analyseur s'utilise en ligne de commande. Celle-ci permet de désigner le fichier source à analyser ainsi qu'une série d'options permettant de faire varier le déroulement de l'analyse. Ces options sont les suivantes.

- L'option `-analysis` permet de choisir quelle analyse doit être effectuée. Dans sa version actuelle, l'analyseur donne le choix entre une analyse numérique utilisant les DBM, une analyse élargissant la recherche de propriété aux contenus de tableaux et utilisant les DBM également et enfin une analyse se concentrant uniquement sur les propriété liant les multi-ensembles de valeur des tableaux. Par défaut, c'est la seconde analyse qui est retenue. Choisir un type d'analyse rendra disponibles de nouvelles options en ligne de commande, spécifiques à cette analyse.
- L'option `-verbose` permet de demander aux analyses d'afficher une quantité plus ou moins importante d'informations sur leur déroulement. Ces informations sont envoyées sur la sortie d'erreur et peuvent être récupérées pour stockage dans un fichier journal. L'option est essentielle au développement car elle permet d'observer l'exécution de l'analyse pas à pas et ainsi fournit des indices sur les raisons d'éventuels échecs..
- L'option `-expand` permet de conserver un graphe de flot de contrôle avec un maximum de nœuds. L'analyseur donnant les propriétés découvertes en chacun de ces nœuds, ceci permet d'avoir plus de détail sur les valeurs abstraites obtenues à des points de contrôle intermédiaires. Sans cette option, l'analyseur simplifiera le graphe. (Cf. Section 10.1.4)
- Les options `-version` et `-help` affichent respectivement la version de l'analyseur et la liste des options utilisables en ligne de commande sans lancer l'exécution d'une analyse.

L'analyseur accepte des programmes écrits dans des versions modifiées et restreintes des langages FAST ou C. La section 10.1.5 décrit les modifications apportées à ces langages. L'analyseur détecte le langage du fichier d'entrée à travers son extension : l'extension `.fst` instruira l'analyseur de considérer l'entrée comme du code source FAST, et l'extension `.c` comme du code source C.

L'analyseur reproduira en sortie le programme original après y avoir ajouté en commentaire les résultats de l'analyse aux différents points de contrôle. La figure 10.1 montre côte à côte un programme simple donné en entrée et le programme commenté sorti par l'analyseur exécuté avec l'option `-expand`. La sortie reproduite ici est fidèle aux espaces près. Les virgules représentent l'opérateur $\wedge$ dans les conjonctions de propriétés.

Les commentaires sont positionnés aux emplacements du code source correspondants aux nœuds du graphe de flot de contrôle. Ceci n'est pas toujours très naturel, en particulier pour les boucles. Il y a, pour chaque boucle, dans le graphe de flot de contrôle, un nœud tête de boucle. Ce point de contrôle est représenté dans le code source entre le mot clé (ici `while`) et la condition d'entrée dans la boucle. Insérer à cet endroit les résultats obtenus ne serait toutefois pas très lisible ni réellement pertinent dans le cas des boucles `for`.

Dans l'exemple de la figure, il s'agit du nœud auquel on infère la propriété :

$$A[0..i[= A[i-1], A[0..i[= x, i <= n, 0 <= i, 0 < n$$

Ce nœud est accessible aussi bien par l'arc d'entrée dans la boucle que par l'arc revenant du corps de la boucle. C'est à ce nœud qu'on teste la condition d'arrêt de la boucle pour savoir si on doit commencer une itération ou sortir de la boucle. On peut voir dans l'exemple que la propriété de la tête de boucle est affichée avant la boucle elle-même. Il est malheureusement aisé de confondre cette propriété qui est l'invariant de boucle avec la propriété qui serait une précondition avant d'exécuter la boucle.

Lorsque les droits d'accès le permettent, le graphe de flot de contrôle sera généré au format DOT et enregistré dans un fichier créé au même emplacement que le fichier source. Ceci permet de connaître le graphe utilisé pour l'analyse et ainsi mieux comprendre les sorties de l'option `-verbose` si elle a été utilisée.

10.1.2 Organisation du code source

Le code source est composé de 22 000 lignes d'OCAML (interfaces non-comprises) dont 13 000 sont de notre fait. Il est réparti en 65 fichiers et dix répertoires dont nous décrivons les principaux :

- la racine où reposent les interfaces communes, les représentations intermédiaires et le point d'entrée de l'analyseur ;
- un répertoire « langages » regroupant les analyseurs syntaxiques ;
- un répertoire « analyses » où sont décrites les deux analyses statiques disponibles ;
- un répertoire « domaines abstraits » où on trouve les différents domaines servant à paramétrer les analyses ;
- un répertoire « analyses de tableaux » rassemblant les fichiers nécessaires à la construction des domaines abstraits spécialisés capables de traiter des tableaux ;
- un répertoire « librairies » contenant les diverses librairies dont l'analyseur se sert.

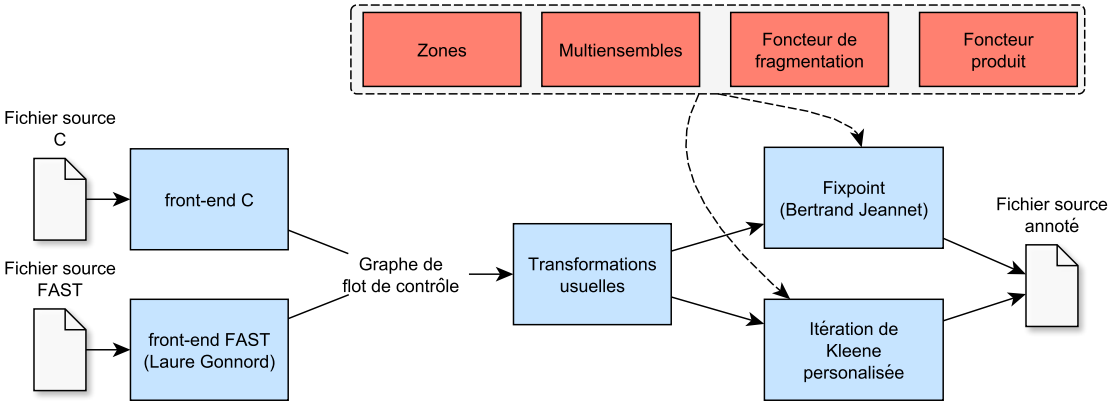


FIGURE 10.2 : Schéma de l'architecture de l'analyseur ENKIDU.

10.1.3 Architecture

L'analyse se déroule en deux temps. D'abord, l'analyse syntaxique convertit le code source en un graphe de flot de contrôle. Dans un second temps, le graphe de contrôle est utilisé pour une analyse par interprétation abstraite. La modularité permet de choisir entre les différents modules disponibles pour l'analyse syntaxique et parmi les différents modules pour l'analyse statique. Les premiers sont décrits dans la section 10.1.5 et les seconds dans la section 10.1.6. Le choix des modules à utiliser est réalisée à l'exécution grâce aux modules de premier ordre de OCAML.

Les optimisations et transformations de graphes de flot de contrôle usuelles étant communes aux différents modules d'analyse syntaxique, elles sont réalisées entre les deux étapes de l'analyse. Ces transformations sont décrites dans la section 10.1.4.

La figure 10.1 résume cette vue globale de l'architecture.

10.1.4 Graphes de flot de contrôle

Notre représentation des graphes de flot de contrôle est celle contrainte par la librairie FIXPOINT. Les nœuds sont les points de contrôle du programme auquel on attache l'information nécessaire pour le retour au fichier source. Les arcs sont les transitions, composées d'une garde et d'une suite de commandes. La figure 10.3 montre un tel graphe de flot de contrôle pour un algorithme de tri par insertion.

Les gardes et commandes qu'il est possible de mettre sur les transitions sont pour le moment limitées aux expressions arithmétiques. Les gardes peuvent être des conjonctions, disjonctions ou négations de comparaisons entre des expressions arithmétiques entières. Les commandes se limitent aux affectations. Les expressions intervenant dans les comparaisons ou les affectations peuvent contenir des sommes, des différences, des produits ou des divisions d'expressions, l'opposée d'une expression ou être des accès à des variables locales ou des cellules de tableau dont l'indice est une expression.

La construction des graphes de flot de contrôle est de la responsabilité des modules de langages. Ceci implique par exemple pour le langage C que les expressions avec effet de bord (comme l'opérateur d'incrémentation) soient décomposées, quitte à introduire des variables temporaires. L'évaluation des opérateurs paresseux se retranscrit directement dans la structure du graphe.

Afin de faciliter l'évaluation de la sémantique abstraite, les expressions peuvent être conver-

```
int main()
{
  int A[];
  int i, j
  int x;

  for (i = 1 ; i < n ; i++)
  {
    x = A[i];

    for (j = i-1 ; j >= 0
         && A[j] > x ; j--)
    {
      A[j+1] = A[j];
    }

    A[j+1] = x;
  }
}
```

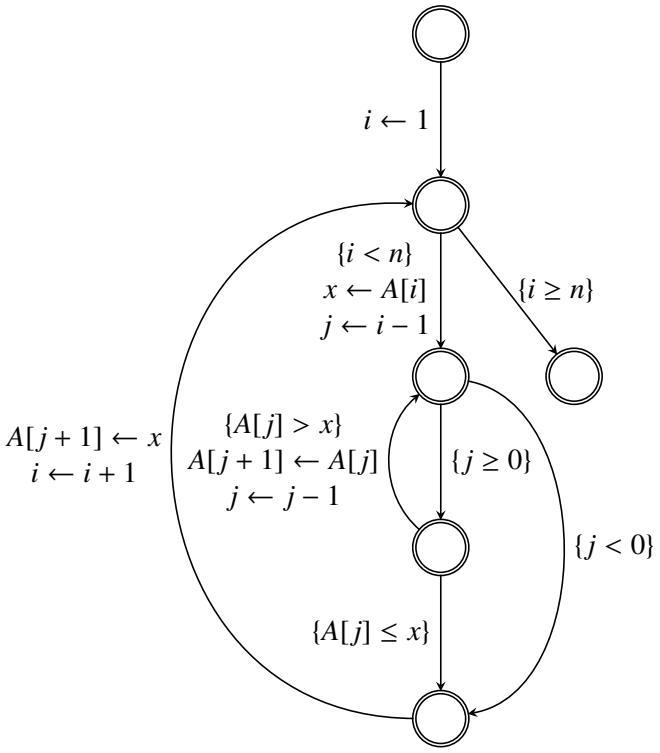


FIGURE 10.3 : A gauche un programme écrit en C, à droite le graphe de flot de contrôle construit à partir de ce programme. Sur les arcs les formules entre accolades sont les gardes des transitions.

ties dans une forme plus régulière. Les sommes de termes, éventuellement multipliés par des constantes, forment des arbres qui peuvent être remplacés par un unique nœud « somme » n-aire dont chaque fils est un couple formé de la constante et du terme de la somme. Ceci est correct pou les langages ou l'addition est effectivement associative, ce qui nécessite parfois de vérifier qu'il n'y a pas de débordements arithmétiques. Grâce à cette transformation, le nombre de cas d'expressions à traiter dans nos domaines abstraits est réduit.

Après l'analyse syntaxique, l'analyseur opère des transformations de simplification ou de décoration du graphe de flot de contrôle. La première consiste à retirer tous les noeuds inaccessibles dans le graphe à partir de l'ensemble des point d'entrées. Ces noeuds sont parfois générés de manière systématique par les analyseurs syntaxiques avant qu'ils ne puissent se rendre compte qu'ils étaient inutiles.

La deuxième transformation permet de retirer les points de contrôle intermédiaires lorsqu'ils ne sont pas utiles. Il s'agit de points de contrôle dans lesquels n'entrent qu'une transition et desquels ne partent qu'une transition. Il est alors possible de composer les deux transitions dans deux cas : lorsque la première transition n'a pas de commande ou lorsque la seconde transition n'a pas de garde. La garde de la transition composée est la conjonction des gardes, et la séquence de commandes est la concaténation des deux séquences.

Enfin, une troisième transformation permet de décorer le graphe en annotant les arcs qui entrent et sortent des boucles. Ceci était nécessaire pour l'implémentation de notre critère de fragmentation.

10.1.5 Langages d'entrée

Deux modules de langages sont implémentés dans l'analyseur. Le premier est un analyseur syntaxique pour le langages FAST. Il s'agit d'un langage décrivant des automates à compteur qui n'ont pas de rapport avec nos travaux. La présence de cet analyseur est purement historique : il est issu d'un autre analyseur, ASPIC développé dans l'équipe par Laure Gonnord [Gon07]. Le langage a été étendu par Mathias Péron [Pé10] pour intégrer les tableaux et leurs accès dans les expressions. Depuis, ce module a fait l'objet d'une activité de maintenance légère pour l'adapter au développement du reste de l'analyseur.

Le second module permet l'analyse de programmes écrits dans un sous-ensemble du langage C. Ce langage, plus largement connu, permet une présentation plus synthétique des programmes que leurs équivalents en FAST. Les structures de contrôle répétitives (`for`, `while`, `do while`) et conditionnelles (`if-else` et `switch`) et les sauts (`goto`) sont supportés. Toutefois, si on souhaite conserver un contrôle direct sur la forme des graphes de flot de contrôle, on préfèrera utiliser le langage FAST.

La classe de programmes traitée demeure très restreinte et de nombreuses limitations sont appliquées. La grammaire n'a qu'une seule limitation : elle prohibe toute définition et utilisation de types qui ne soient pas les types prédéfinis. L'analyseur produit un arbre syntaxique abstrait lui-même ensuite transformé en graphe de flot de contrôle. Dans cette transformation interviennent l'essentiel des limitations. L'ensemble des expressions du C n'est pas totalement compatible avec les expressions autorisées sur les graphes de flot de contrôle de notre implémentation. Par ailleurs, certaines transformations possibles ne sont tout simplement pas implémentées.

L'analyseur ne supporte que les données numériques entières. Aucun domaine n'a été implémenté pour les flottants ni mêmes pour les rationnels. Toutes les variables étant entières, il n'est

```
int main()
{
  for (i = 0 ; i < n ; i++)
  {
    assume(A[i] >= 0);
  }

  ...
}
```

FIGURE 10.4 : Exemple d'un programme donné à l'analyseur dont la première boucle permet de définir comme précondition que toutes les valeurs du tableau A sont positives ou nulles.

plus nécessaire de déclarer les variables. Les valeurs abstraites seront automatiquement étendues lors de la première utilisation des variables.

L'analyseur n'opère aucune vérification sur la validité des accès aux tableaux. Cette analyse serait possible mais n'est pas l'objet de cette thèse. Par conséquent, les tableaux peuvent être déclarés sans taille ou même n'être pas déclarés du tout. Les accès aux tableaux sont représentés par des couples composés d'un symbole de tableau et d'une expression d'indice. L'analyseur prendra comme hypothèse que deux symboles de tableaux distincts représentent toujours deux segments de mémoire disjoints. Autrement dit, deux accès utilisant deux symboles différents seront considérés comme représentant des cellules différentes. Ceci n'est évidement pas correct pour la sémantique originale du langage.

L'analyseur est uniquement intra-procédural. L'ensemble du code exécutable devra donc être contenu dans la fonction `main` du programme C.

Pour modéliser les entrées du programme on utilisera des variables ou des tableaux non initialisés. L'analyseur considèrera que ces variables ou ces tableaux peuvent avoir n'importe quelle valeur au même titre qu'une entrée quelconque.

Si l'on souhaite contraindre ces entrées et ajouter des préconditions au programme, on pourra utiliser une technique propre au langage d'entrée. Avec le langage FAST on peut jouer sur les transitions. En choisissant adéquatement les gardes de ces transitions on peut filtrer les états et ne conserver que ceux répondant aux préconditions souhaitées. Par exemple, si de l'état initial ne part qu'une seule transition avec pour garde $x < 0$ alors on ne conservera effectivement que les états vérifiant cette condition.

Pour le langage C une nouvelle instruction a été ajoutée composée du mot clé `assume` suivi d'une expression entre parenthèses. L'analyseur considèrera que la condition est vérifiée à l'emplacement de l'instruction. Il exclura tout état ne vérifiant pas la condition. La figure 10.4 illustre l'usage de cette instruction pour construire la précondition « toutes les valeurs d'un tableau sont positives ou nulles ». Un autre exemple d'utilisation de l'instruction est donnée plus loin à propos du problème du drapeau hollandais [Dij76]. Elle permet de s'assurer que les valeurs du tableau sont choisies parmi les trois couleurs en ajoutant après l'étiquette `default` de l'instruction `switch` l'instruction `assume(false);`.

10.1.6 Analyses disponibles

L'analyseur implémente une analyse d'atteignabilié réalisée par le solveur de point fixe de la librairie FIXPOINT de Bertrand Jeannet. Cette librairie prend en entrée un graphe de flot de contrôle et calcule en chaque point de contrôle (chaque nœud du graphe) les valeurs abstraites constituant un post-point fixe. Cette librairie générique peut être contrôlée par un grand ensemble de paramètres permettant d'agir sur la manière et la précision avec laquelle le solveur va trouver une solution. En particulier, il est possible de sélectionner des variations de la technique de résolution, dont notamment la résolution par itération guidée [GR07].

Ce solveur, bien qu'adapté à de nombreux de cas, s'est révélé inadapté à certaines situations. Par exemple, les séquences descendantes n'ont lieu qu'une fois un post-point fixe atteint sur l'ensemble du programme. Chaque fois qu'une valeur abstraite est obtenue en un point, on calcule la borne supérieure de cette valeur et de la valeur précédente en ce point, de sorte que la séquence de valeurs abstraites soit toujours croissante. Par exemple, supposons qu'on soit en présence d'une boucle d'indice i itérant de 1 à n. L'élargissement fera classiquement perdre l'information « l'indice i est plus petit que n ». Une fois cette information perdue, il ne sera pas possible de la récupérer avant la phase descendante. On ne pourra pas avoir non plus l'information $i = n$ valide en sortie de boucle. Une des conséquences de cette imprécision est qu'on ne pourra pas conclure à l'égalité des expressions de tranche $A[1..i[$ et $A[1..n[$. Lorsque la valeur de i est écrasée, on ne peut pas substituer l'une par l'autre et on est forcé de retirer la tranche de la partition. Lorsqu'on arrivera à la séquence descendante, on aura perdu trop d'information qu'on ne pourra plus récupérer. Ce cas est relativement courant dans les exemples avec des boucles imbriquées.

Au fil des expérimentations, nous avons voulu faire varier légèrement les méthodes d'itération afin de mesurer les conséquences de ces variations sur les résultats de l'analyse. Il nous a alors semblé plus commode d'avoir une seconde implémentation d'un solveur de point fixe, moins générique, spécialisée pour notre méthode d'analyse et qu'il nous est facile et peu coûteux de modifier. Sur le schéma 10.2, ce solveur est désigné sous le nom « itération de Kleene personnalisée ». Nous décrivons dans la suite quelques caractéristiques de cette implémentation.

L'implémentation a été calquée sur celle de FIXPOINT avec laquelle elle a donc beaucoup de points communs. En particulier, elle se base sur le même algorithme de décomposition en composantes et sous-composantes fortement connexes du graphe de flot de contrôle [Bou93], implémentée dans la librairie CAMLLIB. Elle diffère en deux points.

La première différence tient à ce que nous avons évoqué plus tôt. On ne calcule pas la borne supérieure entre les valeurs abstraites obtenues à l'itération courante et à l'itération précédente. Ceci a deux conséquences. D'abord, l'élargissement n'a plus forcément lieu entre deux valeurs abstraites comparables et l'implémentation de cet opérateur doit en tenir compte. Ensuite, après un élargissement et une itération complète, on peut obtenir une valeur plus petite que celle obtenue par l'élargissement. Lorsque c'est le cas, c'est qu'on a commencé une séquence descendante sur la sous-composante fortement connexe sur laquelle on itère. Sous hypothèse de la monotonie des fonctions de transfert, on peut alors conserver la dernière valeur comme post-point fixe. La principale différence avec FIXPOINT est donc que les séquences descendantes sont « locales » et réalisées composante par composante au lieu d'être réalisées sur tout le programme une fois qu'un post-point fixe a été trouvé.

La seconde modification prend son importance dans le cas où plusieurs boucles sont imbriquées. Le critère de fragmentation est basé sur une bonne structuration du programme. Nous essayons de fonder la stratégie d'itération sur cette structuration dans l'espoir de rendre plus facile l'approximation du critère de fragmentation. La stratégie implémentée recommence à zéro les itérations sur les boucles internes à chaque itération d'une boucle externe. En pratique, chaque fois que l'on commence une itération sur une (sous) composante fortement connexe, on remet à $\bot$ toutes les valeurs abstraites aux points de contrôles de la composante, sauf au point d'élargissement de cette composante. Ainsi, les nouvelles valeurs abstraites sont calculées à partir de la valeur à ce point d'élargissement et avec d'éventuelles contributions des arcs entrants dans la composante. En revanche, les sous-composantes de cette composante devront être entièrement recalculées, avec leurs propres séquences d'itérations et d'élargissements. Les comportements du programme aux différents niveaux de boucle sont ainsi dissociés, ce qui, par rapport à la méthode d'itération initiale, améliore le calcul des fragmentations dans certains exemples.

Ce second solveur est très expérimental, et si il résout certains problèmes rencontrés durant l'expérimentation, il souffre toujours de défauts communs aux méthodes d'itération classiques en interprétation abstraite. Il pourrait être utile de considérer des méthodes d'itérations plus perfectionnéees comme [HH12, HMM12].

10.1.7 Domaines abstraits

L'analyseur implémente deux domaines abstraits et deux *foncteurs* permettant de les combiner pour en produire d'autres. Ces foncteurs sont effectivement implémentés comme des foncteurs OCAML et sont donc tout à fait génériques.

- Le domaine des zones implémente les opérations classiques pour la manipulation des matrices de bornes de potentiel ainsi que les transformations nécessaires pour adapter ce domaine à l'analyse des tableaux telle que décrite dans cette thèse. Lorsque la sémantique doit être approximée, le domaine applique une sémantique d'intervalles peu précise mais simple à implémenter. L'adaptation à l'analyse des tableaux est fidèle à celle présentée au chapitre 7. Un domaine de bornes (Cf. section 5.3) lui est également adjoint pour représenter les bornes de la forme $i + c$ où i est une variable du programme et c une constante. Ce domaine de bornes sait tirer profit des zones pour dériver des propriétés sur l'ordre des bornes.
- Le domaine des équations de multi-ensemble permet d'exprimer des propriétés de permutation de valeur entre différents fragments de différents tableaux [PH10]. Ce domaine existe en deux version : la première permet d'analyser les programmes sans recourir aux fragmentations mais ne peut pas considérer des multi-ensembles des valeurs d'un fragment : il ne peut parler que du multi-ensemble complet des valeurs d'un tableau. La deuxième version est adaptée à l'analyse de tableaux fragmentés et implémente les transformations élémentaires.
- Le foncteur produit permet de combiner deux domaines abstraits conformément à la notion classique de produit en interprétation abstraite. Il ne s'agit évidemment pas d'un produit réduit : s'il faut améliorer la réduction, il faudra le faire pour chaque cas particulier de produit. L'usage actuel de ce foncteur est de permettre l'usage simultané des deux domaines précédents.
- Le foncteur de fragmentation permet de combiner un domaine de fragmentation et un domaine abstrait pour donner au domaine abstrait la capacité d'exprimer des propriétés sur les cellules décrites par la fragmentation. Le domaine abstrait peut être n'importe quel domaine implémentant les transformations élémentaires, donc peut être le domaine des zones

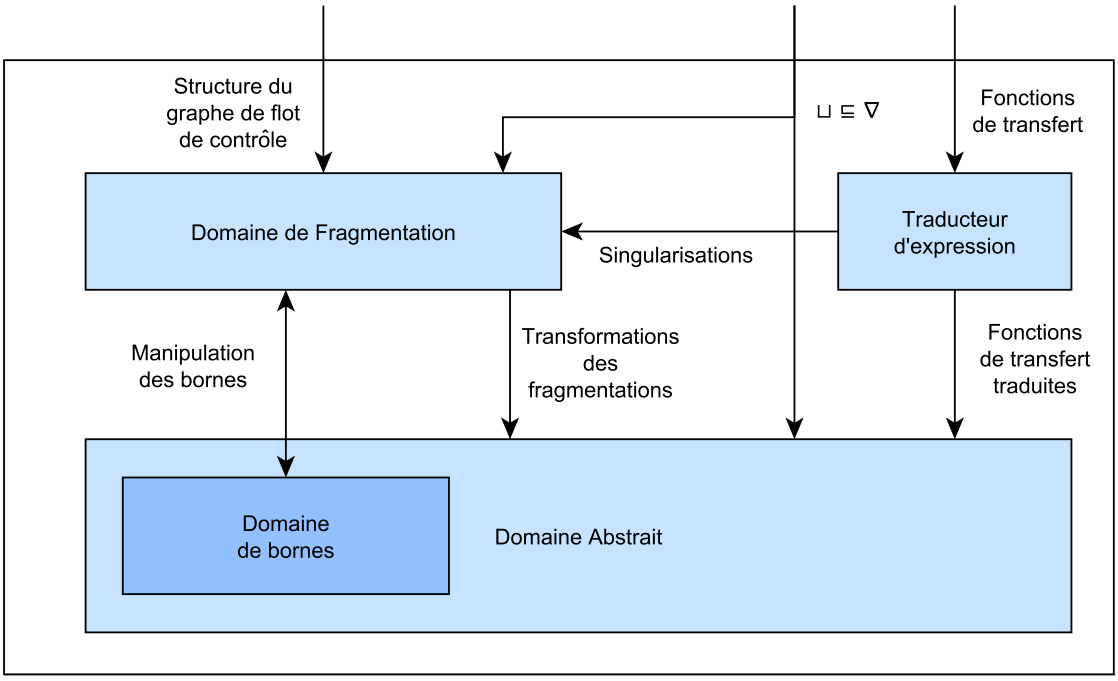


FIGURE 10.5 : Schéma de l'architecture du foncteur de fragmentation.

ou celui des équations de multi-ensembles. En revanche, l'unique domaine de fragmentation implémenté est celui des diagrammes de tranches. Celui-ci requiert que le domaine abstrait ait un domaine de bornes adjoint. Pour utiliser le domaine d'équations de multi-ensemble, il faut utiliser une version spécialisée du foncteur produit pour le combiner avec les zones et désigner le domaine de bornes des zones comme domaine de bornes du produit. La section suivante est consacré à ce foncteur.

Le domaine ou foncteur à utiliser est choisi dans le module d'entrée du programme et est passé en paramètre à l'analyse choisie.

10.1.8 Foncteur de fragmentation

Le foncteur de fragmentation prend en paramètre un domaine abstrait et un domaine de fragmentation et étend le premier pour qu'il puisse dériver des propriétés des cellules désignées par le second. Les appels aux opérateurs binaires $\sqcup$, $\sqsubseteq$ et $\triangledown$ sont retransmis au domaine abstrait après avoir unifié les fragmentations des valeurs sur lesquels ils s'appliquent. Chaque transition du graphe de flot de contrôle est interprétée de la manière suivante :

- Les instructions sont d'abord interprétées par le domaine de fragmentation, qui, dans notre cas les retransmet au domaine de bornes pour évaluer leurs effets sur les sommets des diagrammes.
- Les instructions sont traduites par un module traducteur qui demandera au domaine de fragmentation de singulariser toute nouvelle expression de cellule. Chaque accès à un tableau est remplacé par un accès à une variable de synthèse. Si nécessaire, le module de traduction ajoutera les affectations faibles requises.
- Les instructions traduites sont interprétées par le domaine abstrait original.

- S'il y a dans la transition des décorations indiquant que l'on entre ou que l'on sort d'une (sous) composante fortement connexe, le domaine de fragmentation appliquera les transformations en conséquence.
- Après l'application d'une garde, on demande au domaine de fragmentation de spécialiser la fragmentation pour la nouvelle valeur abstraite.

Toutes les transformations élémentaires générées par le domaine de fragmentation sont retransmises au domaine abstrait. Le foncteur de fragmentation possède un module s'occupant de la liste des variables de synthèse. Il alloue et désalloue les variables de synthèse à la demande du domaine de fragmentation et unifie les noms des variables de synthèse représentant les mêmes fragments dans deux valeurs abstraites distinctes.

L'architecture du foncteur de fragmentation est reproduite figure 10.5.

Pour la découverte des relations d'indice (Chapitre 6) une couche est intercalée entre le foncteur de fragmentation et le domaine abstrait. Cette couche va construire pour chaque valeur abstraite une liste de sous-valeurs abstraites, chacune associée à une relation d'indice et à un ensemble de variables de synthèse. Elle intercepte les transformations élémentaires, puis, suivant le cas, ou bien elle les retransmet aux sous-valeurs abstraites, ou bien elle les interprète en créant de nouvelles sous-valeurs abstraites ou en les combinant.

Quant au domaine de fragmentation des diagrammes, il est construit en trois couches. La couche la plus basse permet une manipulation généraliste des graphes, sans sémantique particulière. La couche intermédiaire fournit les transformations élémentaires des diagrammes (Section 5.5.2) et assure la correction du domaine de fragmentation. La dernière couche enfin fournit l'interface d'un domaine de fragmentation.

10.2 Expérimentations

L'analyseur a été testé sur un ensemble de 32 programmes de test. Il s'agit de programmes de très petites taille (de 3 à 30 lignes de code) tenant dans une seule fonction. On peut les ranger en trois catégories.

- Certains programmes ont été artificiellement créés pour tester des parties précises de l'analyseur et s'assurer que l'expérience coïncide avec la théorie.
- D'autres, ont été également artificiellement créés pour mettre en défaut les autres méthodes d'analyses de la littérature et valider des avantages théoriques de la méthode d'analyse, souvent par petite variation d'un autre programme de la base de test.
- Enfin, certains programmes sont issus d'exemples courants en programmation avec des tableaux, et la plupart sont issus d'autres publications du domaine.

Ces exemples permettent déjà de mettre en évidence un certain nombre de difficultés liées à l'analyse de programmes manipulant des tableaux. Nous en exposerons deux dans cette section. L'ensemble des résultats obtenu est synthétisé dans le tableau de la figure 10.6. Seuls les exemples faisant intervenir des tableaux y sont inclus. Les mêmes paramètres ont été utilisés pour l'analyse de tous les programmes afin de vérifier que les résultats obtenus ne l'ont pas été grâce à un choix de paramètres particulier qui remettrait en cause la généricité de l'analyse. Lorsque des cases sont laissées vides c'est que l'analyseur ne prouve pas l'invariant souhaité, ou qu'un erreur l'empêche de s'exécuter complètement.

Les temps d'exécutions sont bien plus élevés que ce que nous attendions. La saturation des valeurs abstraites avec les propriétés dérivables à partir de la forme de la fragmentation implique un coût important. Nous avons reporté les temps d'analyse avec et sans saturation pour mettre en évidence ce coût. Des travaux sont en cours pour corriger les problèmes de conception de l'analyseur qui sont à l'origine de ces coûts exagérés.

10.2.1 Echanges de cellules

L'échange de deux cellules est une opération courante dans les programmes manipulant des tableaux. Mais déjà elle met en évidence le problème que peuvent poser d'éventuels alias dans les expressions d'accès aux tableaux. La figure 10.7 présente trois programmes de tests sur les échanges. Dans les deux premiers, les cellules $A[1]$ et $A[2]$ sont échangées, et l'analyseur n'a aucun mal à vérifier qu'il s'agit bien de deux cellules distinctes. A la fin, il trouvera que les valeurs ont bien été échangées. Les valeurs affectées aux tableaux servent de témoin pour vérifier les conséquences de l'analyse d'un échange.

On vérifie ainsi que le programme est analysé comme si les cellules de tableau étaient des variables scalaires classiques. On peut remplacer $A[1]$ par x et $A[2]$ par y sans que cela n'ait d'impact sur la précision. Le second exemple, celui de l'échange arithmétique, montre qu'il peut être utile d'utiliser un domaine abstrait qui permette de découvrir plus d'informations que des égalités.

Dans le troisième exemple, les cellules $A[i]$ et $A[j]$ peuvent être aliasées car on n'a pas a priori d'information permettant de réfuter la possible égalité de i et de j. Cet exemple peut être analysé précisément si on accepte de distinguer les cas d'alias sur l'ensemble de l'analyse du programme. Ainsi, les méthodes d'analyse proposées dans [GRS05] et [HP08] sont adéquates pour cet exemple. La notre, en revanche, est imprécise. Nous en détaillons maintenant les raisons.

Observons donc les résultats de notre analyseur sur le troisième exemple. Après l'affectation $x = A[i]$, l'analyseur aura trouvé sans difficulté

$$x = a = A[i] \wedge b = A[j]$$

Dans la seconde affectation, $A[i]$ est fortement affectée, et $A[j]$ est faiblement affectée par elle-même, ce qui ne change pas sa valeur.

$$x = a \wedge A[i] = A[j] = b$$

La dernière affectation en revanche ne sera pas analysée précisément. $A[j]$ est fortement affectée, mais $A[i]$ l'est faiblement et c'est dans cette affectation faible que réside la perte de précision. L'affectation faible à $A[i]$ se calcule en distinguant les cas : si $A(i]$ est affectée, on obtiendra $A[i] = A[j] = x = a$ et si elle ne l'est pas, on obtiendra $A[j] = x = a \wedge A[i] = b$. Le résultat de l'affectation faible est la borne supérieure de ces deux valeurs :

$$\Big(A[i] = A[j] = x = a\Big) \sqcup \Big(A(j] = x = a \wedge A[i] = b\Big) \equiv \Big(A(j] = x = a\Big)$$

On perd l'égalité entre $A[i]$ et b. Pour conserver cette égalité, il aurait fallu que l'on ait l'information

$$i = j \Rightarrow a = b$$

Programme				Avec saturation				Sans saturation			
Nom	Figure	$\|V\|$	$\|E\|$	Temps	K	N	$\|F\|$	Temps	K	N	$\|F\|$
empty.fragment.challenge.c	3.5	3	3	0,00	6	6	2	0,00	6	6	2
exemple1.c	10.1	3	3	0,05	10	6	6	0,01	10	6	3
array.switch.simple.c	10.7(a)	2	1	0,00	4	4	3	0,00	4	4	2
array.switch.arithmetic.c	10.7(b)	2	1	0,00	4	4	3	0,00	4	4	2
array.switch.alias.c	10.7(c)	2	1								
array.init.c	3.2	3	3	0,02	10	6	3	0,00	10	6	3
array.init.partial.c	10.8(a)	5	6	0,22	18	14	10	0,03	18	14	10
array.max.c	10.8(b)	5	6	0,13	18	14	6	0,01	18	14	4
array.range.c	4.1	3	3	0,02	11	6	3	0,01	11	6	3
array.iterate.partial.c		6	8	3,90	24	16	23	0,05	24	16	12
array.iterations.c	4.2(a)	5	7	1,64	22	10	14	0,10	22	10	11
array.iterations.2.c	4.2(c)	6	8	0,17	24	16	6	0,03	24	16	5
array.overapprox.test.c	4.10(b)	5	6	0,11	18	14	10	0,01	18	14	6
array.overapprox.unroll.c	4.10(a)	3	3	0,58	10	6	13	0,02	10	6	6
slice.saturation.c		3	3	0,86	10	6	15	0,04	10	6	7
array.copy.c	6.1	3	3	0,07	10	6	5	0,02	10	6	5
array.copy.2.c		4	5	1,30	16	8	14	0,10	16	8	10
array.range.relation.c	10.8(c)	3	3	0,56	11	6	10	0,04	11	6	8
bubble.c		5	6	2,80	21	16	21	0,05	18	14	11
insertion.c		5	6	0,93	18	13	12	0,03	18	13	8
selection.c		6	8	1,25	23	18	15	0,10	23	18	13
array.index.max.c	4.7	5	6	1,54	21	16	21	0,07	21	16	13
array.segmentation.c	10.8(d)	8	11	128,31	48	40	47				
array.sign.partition.c	10.9	5	6	0,55	18	14	9	0,05	18	14	9
dutch.flag.c	10.8(e)	9	14	549,61	38	33	68	0,79	50	43	32
array.erase.loop.c		5	7	2,18	30	12	25				
bubble.sort.c		6	8								
insertion.sort.c		6	8								
selection.sort.c		6	8								

Légende

- $|V|$: nombre de sommets dans le graphe de flot de contrôle.
- $|E|$: nombre de transitions dans le graphe de flot de contrôle.
- Temps : temps d'exécution en secondes.
- K : nombre de pas d'exécution, correspondant au nombre de fois où une valeur abstraite a été calculée en un sommet.
- N : nombre d'itérations de l'interprète. Lorsqu'une sous-composante est itérée, on additionne les itérations de cette sous-composante avec celles de la composante qui l'inclut.
- $|F|$: maximum du nombre de fragments atteint durant l'analyse.

FIGURE 10.6 : Résultats.

```
A[1] = 3;
A[2] = 7;

x = A[1];
A[1] = A[2];
A[2] = x;
```
(a) array.switch.simple

```
A[1] = a;
A[2] = b;

A[1] = A[1] - A[2];
A[2] = A[1] + A[2];
A[1] = A[2] - A[1];
```
(b) array.switch.arithmetic

```
a = A[i];
b = A[j];

x = A[i];
A[i] = A[j];
A[j] = x;
```
(c) array.switch.alias

FIGURE 10.7 : Trois programmes échangeant des cellules d'un tableau.

Or cette implication ne peut être découverte par notre analyseur. (Elle peut être découverte par l'analyseur de [HP08].) Cet exemple suffit à montrer qu'on ne peut pas toujours se contenter de distinguer les cas d'alias *localement*. Au contraire, il faudrait commencer à distinguer les deux cas $i = j$ et $i \neq j$ au moins à partir de la seconde instruction, $b = A[j]$. Ensuite, on peut conduire deux analyses parallèles pour ces deux cas jusqu'à la dernière instruction de l'échange. A la fin du programme, on peut calculer la borne supérieure des deux valeurs abstraites obtenues, la distinction des cas d'alias n'étant plus nécessaire.

En raison de ce défaut de l'analyseur, nous avons du modifier légèrement nos programmes de test pour que les échanges n'aient lieu qu'entre des cellules que l'on sait distinctes. Aucune modification n'est nécessaire pour les programmes de tris où on échange toujours une cellule et sa voisine (par exemple $A[i]$ et $A[i+1]$). Des modifications ont été nécessaires pour les exemples de la segmentation et du drapeau hollandais.

10.2.2 Quelques exemples de programmes analysés avec succès

Les cinq exemples de la figure 10.8 ont pu être analysés avec succès. Le premier programme initialise conditionnellement un tableau A, en affectant 0 à autant de cellules qu'il n'y a d'éléments non nuls dans B. L'analyseur découvre la propriété $A[0..j[= 0$.

Le second est l'exemple classique de recherche de maximum dans un tableau. L'analyseur séparera la cellule $A[0]$ des autres à cause de l'affectation initiale. La propriété découverte est donc produite par l'analyseur sous la forme $A[0] \leq max \wedge A[1..i[\leq max$.

Le troisième construit un tableau de valeurs successives allant de c à $c + n - 1$. Cette version du programme va forcer l'analyseur à considérer une relation privilégiée entre chaque cellule et sa cellule suivante. On aura bien entendu la propriété $A[0] = c$. L'analyseur ne produisant pas de relations entre les cellules et leurs indices, il ne pourra découvrir $A[\ell] = c + \ell$. En revanche, il découvre la propriété

$$\forall \ell_1, \ell_2, \Big(0 \leq \ell_1 < i - 1 \wedge 1 \leq \ell_2 < i \wedge \ell_2 = \ell_1 + 1\Big) \Rightarrow A[\ell_2] = A[\ell_1] + 1$$

Les deux derniers exemples sont plus conséquents. Le premier est la phase de segmentation d'un tri par segmentation. Il a été modifié pour que les échanges de cellules n'aient lieu qu'entre des cellules différentes. Après la boucle, l'analyseur trouve $A[1..j[< A[0] \wedge A[0] \geq A[j..i[$ mais également $A[1..j[< A[j..i[$ préservé par les compositions successives. Notons que les tranches $A[j]$ et $A[j..i[$ se chevauchent sans que l'une ne soit incluse dans l'autre. Les diagrammes permettent la représentation des deux sans nécessiter la distinction des cas $i > j$ et $i = j$. Néanmoins, pour traiter l'échange, cette distinction est nécessaire.

Le second est une solution au problème du drapeau hollandais [Dij76] et trie un tableau ne contenant que trois valeurs distinctes 0, 1 et 2. L'invariant de la boucle, trouvé par l'analyseur

```
j = 0;
i = 0;
while (i < n)
{
  if (B[i])
  {
    A[j] = 0;
    j++;
  }
  i++;
}
```
(a) array.init.partial

```
max = A[0];
for (i = 1 ; i < n ; i++)
{
  if (A[i] > max)
    max = A[i];
}
```
(b) array.max

```
A[0] = c;
for (i = 1 ; i < n ; i++)
{
  A[i] = A[i-1] + 1;
}
```
(c) array.range.relation

```
assume(n > 0);

i = 1;
j = 1;

while (i < n)
{
  if (A[i] < A[0])
  {
    if (j < i) {
      x = A[i];
      A[i] = A[j];
      A[j] = x;
    }
    j++;
  }
  i++;
}

if (j > 1)
{
  x = A[j-1];
  A[j-1] = A[0];
  A[0] = x;
}
```
(d) array.segmentation

```
assume(n > 0);
low = 0, mid = 0, high = n-1;

while (mid <= high)
{
  switch (A[mid])
  {
    case 0 :
      if (mid > low) {
        x = A[low];
        A[low] = A[mid];
        A[mid] = x;
      }
      low++, mid++;
      break;
    case 1 :
      mid++;
      break;
    case 2 :
      if (high > mid) {
        x = A[mid];
        A[mid] = A[high];
        A[high] = x;
      }
      high--;
      break;
    default:
      assume(0);
  }
}
```
(e) dutch.flag

FIGURE 10.8 : Cinq programmes analysés avec succès. On trouve en fin du programme (a) que chaque cellule de la tranche $A[0..i[$ est initialisée à 0, en fin du programme (b) que la variable `max` est plus grande que toutes les cellules du tableau, en fin du programme (c) que le tableau A est initialisé avec des valeurs comprises entre 0 et $n-1$, en fin du programme (d) l'inégalité $A[0..j-1[< A[j-1] \leq A]j-1..i[$, et en fin du programme (e) que le tableau A est trié.

est

$$A[0..low[= 0 \wedge A[low..mid[= 1 \wedge A]high..n[= 2$$

Encore une fois, pour trouver cette invariant, il a fallu modifier les programmes pour ajouter des tests avant chaque échange afin de ne pas échanger une cellule avec elle-même. L'étiquette `default` de l'instruction `switch` contient l'instruction `assume(0)` qui assure que cette branche n'est jamais prise et ainsi que le tableau ne peut jamais contenir d'autres valeurs.

10.2.3 Partitionnement d'un tableau selon le signe

L'exemple du partitionnement d'un tableau en fonction du signe de ses éléments (figure 10.9) extrait de [KV09] a la particularité de mettre en difficulté les critères de fragmentation sémantiques. En particulier, notre analyseur ne parviendra pas à analyser ce programme précisément. Le tableau A est partitionné en les tableaux B et C : dans le premier, on copie les valeurs positives ou nulles et dans le second celles strictement négatives.

```
assume(n > 0);
i = 0; j = 0 ; k = 0;
while (i < n)
{
  if (A[i] >= 0)
  {
    B[j] = A[i];
    j = j + 1;
  }
  else
  {
    C[k] = A[i];
    k = k + 1;
  }
  i = i +1;
}
```

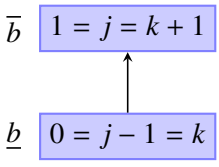
FIGURE 10.9 : Partition d'un tableau selon le signe de ses éléments [KV09].

La difficulté vient de ce qu'il y a trois variables d'indice, i, j et k qui sont initialement égales. Par conséquent, le domaine de fragmentation ne sait pas lesquelles de ces variables choisir comme bornes de chacune des tranches. Le problème ne se pose donc que lors de la première itération. Ensuite, les variables ne sont plus égales et le problème ne se pose plus.

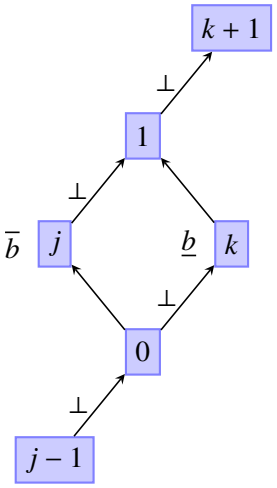
Le moment de l'analyse où le problème apparaît est lors de l'unification des diagrammes après l'instruction `if-then-else`, lors de la première itération. Chacune des branches de la boucle produit des diagrammes différents qu'il faut unifier en sortie de la structure conditionnelle. Dans sa version simplifiée, notre critère de fragmentation va chercher à définir au moins les quatre fragments suivants :

- le fragment a_0 des cellules du tableau A vérifiant le test, c'est à dire les cellules positives ou nulles,
- le fragment a_1 des cellules du tableau A ne vérifiant pas le test, c'est à dire les cellules de valeur strictement négative,
- le fragment b des cellules du tableau B déjà affectées et
- le fragment c des cellules du tableau C déjà affectées.

Observons d'abord les diagrammes que l'on calcule pour le tableau B, ceux pour le tableau C étant symétriques. Après l'exécution de la structure conditionnelle, on aura obtenu le diagramme suivant pour B dans la branche `then`, le diagramme étant vide pour la branche `else`.

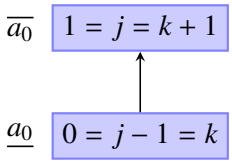


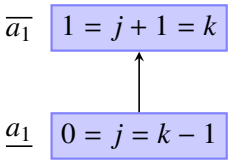
Pour unifier ce diagramme avec le diagramme vide, on commence par généraliser le diagramme pour la relation d'ordre affaiblie. Après la structure conditionnelle, on devra avoir $0 = i \leq j, k \leq 1$, avec j et k incomparables. La généralisation fissionne la borne 0 en 0, $j - 1$ et k et la borne 1 en 1, j et $k + 1$, ce qui donne le diagramme suivant où on a annoté les arcs représentant les tranches vides avec $\perp$:



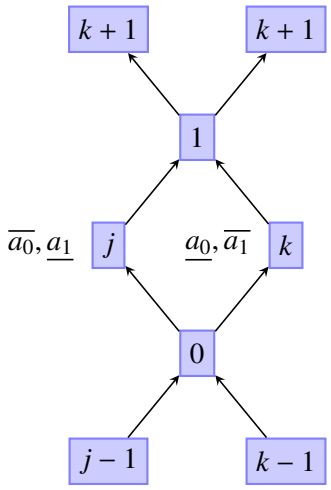
Cependant, il faut sélectionner dans ce diagramme les bornes à conserver et notre méthode ne choisira pas les bonnes. La borne inférieure de b doit être choisie parmi 0, $j - 1$ et k et notre méthode choisira la plus grande, k. La borne supérieure de b doit être choisie parmi 1, j et $k + 1$ et notre méthode choisira la plus petite, j. Autrement dit, la tranche sélectionnée pour b sera $B[j..k[$. Bien que cette tranche convienne, elle n'est pas représentable dans les diagrammes car on n'a pas $j \leq k$. L'implémentation va conserver les sommets j et k et, puisqu'ils ne sont pas ordonnés, aucun arc. L'unification aura donc produit une fragmentation sans fragment. Par ailleurs, bien que $B[j..k[$ soit une abstraction exacte de b lors de la première itération, elle ne conviendra pas aux itérations suivantes.

Quand on observe les diagrammes pour le tableau A, le problème est tout à fait similaire. On obtient pour chacune des branches de la structure conditionnelle les diagrammes suivants.





L'unification de ces deux diagrammes donnera :



Encore une fois, l'interprétation du critère sur les diagrammes ne sélectionnera pas la borne 0 qu'on souhaiterait pourtant conserver.

Nous n'avons pas de solution directe à proposer pour que notre analyseur choisisse de conserver uniquement la tranche $B[0..j[$. Il y a en revanche quelques pistes qui peuvent être suivies :

- La première serait de retarder la supression des sommets, de la même manière qu'on retarde un élargissement. Cela permettrait de remettre à plus tard le choix des sommets à conserver. Toutefois, il serait nécessaire également de modifier la manière dont les annotations sont placées sur les diagrammes afin de pouvoir effectivement réaliser ce choix correctement.
- La seconde serait d'annoter les arcs et non les sommets. Ici, le diagramme unifié possède deux tranches $B[0..j[$ et $B[k..1[$ représentant exactement la tranche singleton $B[0]$ quand le test est vérifié et une tranche vide quand le test ne l'est pas. On peut conserver les deux en attendant de savoir lequel choisir. Cela implique de revoir complètement la manière dont le critère de fragmentation est interprété sur les diagrammes et de retrouver un argument prouvant que le nombre de tranches restera borné.
- Une troisième piste serait de ne conserver qu'une seule variable de synthèse représentant une tranche dont l'expression est incertaine : ici, une tranche représentant ou bien $B[0..j[$ ou bien $B[k..1[$. L'expression reste incertaine jusqu'à ce qu'on ait à notre disposition les moyens de savoir quelle est la tranche réellement désignée par le critère de fragmentation.
- Enfin, une dernière piste serait d'utiliser une autre méthode d'itération que l'itération de Kleene pour l'analyse des programmes.

11 Conclusion

Dans cette thèse, nous avons apporté plusieurs contributions à l'analyse statique de programmes manipulant des tableaux. Ces contributions ne sont pas indépendantes, et ont d'ailleurs été traitées simultanément durant nos travaux de recherche. Dans ce manuscrit, nous avons pourtant choisi de distinguer clairement

1. la sélection d'un critère pour le choix des fragmentations,
2. l'abstraction de la fragmentation choisie et
3. la manière d'associer des propriétés aux fragments.

Ces trois sujets sont étroitement liés. Le critère de fragmentation sera influencé par le type de propriétés que l'on souhaite découvrir. Par ailleurs, on ne développera pas un critère de fragmentation si l'abstraction des fragmentations ne permet pas de l'interpréter précisément. Dans la littérature, la plupart des travaux traitent d'ailleurs des trois sujets à la fois. Les isoler permet néanmoins de voir d'éventuelles combinaisons de ces travaux. Il serait par exemple intéressant d'observer ce que donnerait notre critère de fragmentation appliqué, non pas aux diagrammes de tranches, mais aux partitions utilisées par [CCL11] ou celles distinguant les alias de [HP08].

Notre construction de la correspondance de Galois pour l'interprétation abstraite de programmes manipulant les tableaux généralise celle de [GDD+04, JGR05] notamment en considérant des fragmentations symboliques dans lesquelles certains fragments peuvent être potentiellement vides. Ceci permet l'expression de propriétés relationnelles entre des fragments de tailles différentes sans pour autant nécessiter une distinction combinatoire et systématique des cas de vacuité. En contrepartie, elle requiert plus d'adaptation des domaines abstraits. Toutefois, pour les domaines que nous avons implémentés, la quantité de travail requise a été faible.

Notre première formalisation des fragmentations était plus proche de la formalisation originale de [GDD+04, JGR05]. Dans celle-ci, les fragmentations n'étaient que des ensembles de fragments et non des ensembles de relations entre fragments. Une partie des résultats obtenus sur ces fragmentations ont pu être élargis à celles de ce manuscrit. Cependant, il est certainement possible d'aller plus loin.

Pour la réduction par exemple, nous avions énoncé trois règles. La première concernait les fragments vides : toute propriété est vraie d'un ensemble vide et la valeur abstraite devait pouvoir refléter cela. Ce résultat peut être généralisé aux relations entre fragments. Quand bien même des fragments ne sont pas vides individuellement et dans toute configuration, il est possible que dans tout configuration au moins l'un d'entre eux soit vide. Par exemple les tranches $A[1..i[$ et $A[i..2[$ avec $1 \leq i \leq 2$ ne peuvent pas être simultanément non-vides : lorsque $i = 1$ la première est vide et lorsque $i = 2$, c'est la seconde. Toute propriété relationnelle entre ces deux fragments est alors vraie puisque l'ensemble des couples de cellules prises dans les deux fragments respectif est vide. La même généralisation peut être appliquée aux deux autres règles concernant le chevauchement et le recouvrement de fragments.

En pratique, la généralisation de ces règles est difficile à exploiter. D'abord parce qu'il faut trouver quand appliquer ces règles. Un diagramme de tranches ne montre pas quelles sont les

tranches qui ne peuvent pas être non-vides simultanément. Ensuite, parce qu'il faut pouvoir communiquer ces singularités de la fragmentation au domaine abstrait. Il doit pouvoir connaître les sous-ensembles de fragments tels que dans toute situation l'un au moins des fragments est vide. Même en considérant les sous-ensembles minimaux, leur nombre peut être grand. Cependant, ce n'est pas nécessaire pour les domaines abstraits faiblement relationnels tels que les zones : il suffit de considérer les ensembles de taille deux.

Un problème voisin est celui de l'expressivité des transformations élémentaires de fragmentation, en particulier pour la composition et la décomposition. Il est possible de composer fragment par fragment mais pas relation par relation. Le problème a été exposé au chapitre 6. On voyait notamment dans l'exemple de la copie de tableau qu'il aurait fallu pouvoir composer « en même temps » deux fragments. Il n'y aurait pas de difficulté particulière à formaliser cette composition. Elle aurait l'avantage de permettre une meilleure interprétation du critère de fragmentation dans lequel les compositions se font déjà simultanément. Avec une telle opération, il ne serait plus nécessaire d'utiliser une valeur abstraite pour chaque relation, et on pourrait se contenter d'une seule. En contrepartie, cela peut introduire une redondance d'information dans la valeur abstraite : deux variables représentant le même fragment mais intervenant dans des relations différentes.

Le critère de fragmentation du chapitre 4 produit des fragmentations adéquates sur tous les exemples testés. Il peut être utilisé aussi bien pour des tableaux que pour des structures de données chaînées quelconques, à condition d'avoir les abstractions adaptées et d'interpréter convenablement les instructions d'instrumentation. Notre approximation pour les diagrammes de tranches peut d'ailleurs probablement être améliorée, comme nous l'avions noté dans la section 10.2.3. Au lieu de sélectionner un ensemble de bornes délimitant un fragment, il serait certainement plus adéquat mais moins évident de directement sélectionner les arcs pertinents. D'une part, ceci donnerait la précision qu'il manque à l'analyse de l'exemple de la section 10.2.3. D'autre part parce qu'il serait possible de sélectionner plus finement les tranches qui nous intéressent et par conséquent de réduire le nombre de variables de synthèse à introduire.

L'expérience montre que le calcul des fragmentations est très sensible à la méthode d'itération utilisée. Il est parfois nécessaire de retarder l'élargissement. D'autres fois, l'égalité entre l'indice d'itération d'une boucle et la borne de la boucle doit être établie tôt dans l'analyse pour ne pas perdre toute l'information acquise sur les tableaux. Il est probable que l'on puisse tirer profit des méthodes d'itérations récentes, telles [CGG+05, HMM12].

Le calcul des fragmentations pourrait très bien être réalisé avant la recherche de propriétés sur les contenus des tableaux. Il y a deux raisons pour lesquelles il est intéressant de réaliser les deux simultanément. La première, ce serait de pouvoir traiter des expressions de la forme $A[B[i]]$: il sera utile de pouvoir connaître les propriétés de $B[i]$ pour fragmenter le tableau. Bien sûr, il est toujours possible de réaliser autant d'analyses successives qu'il y a d'imbrications dans les expressions d'accès aux tableaux. Si on peut mettre en avant l'argument de l'efficacité, il n'est pas évident que les analyses successives soient moins efficaces que les analyses simultanées.

Une autre raison, est que l'analyse des contenus des tableaux peut nous apprendre des propriétés sur les indices utilisés dans les accès. Toutefois, bien que la théorie le permette, notre analyseur n'implémente pas encore les algorithmes nécessaires. Il s'agit donc d'une perspective dont nous montrons l'enjeu sur deux exemples.

L'exemple ci-contre montre un parcours de tableau avec sentinelle. On peut prouver la propriété $A[1..i[\neq 0 \wedge A[n] = 0$. Les deux fragments $A[1..i[$ et $A[n]$ ont donc des propriétés incompatibles, et la seconde règle de réduction de la section 3.5 implique que ces deux tranches doivent être disjointes. Par conséquent, on en déduit $i < n$. Ainsi, en découvrant les propriétés de contenu du tableau A on peut en déduire une propriété sur les variables d'indices i et n. À leur tour, ces propriétés sur les variables d'indice pourront impliquer des propriétés sur les contenus de tableaux. Plutôt que d'alterner des analyses sur les indices et sur les contenus, on peut les réaliser simultanément.

```
A[n] ← 0
i ← 1
Tant que A[i] ≠ 0 faire
    i ← i + 1
```

Figure 11.1 : Parcours de tableau avec sentinelle.

L'exemple du tri à bulle est également intéressant. La version ci-contre n'est pas optimisée dans le sens où il suffirait que j aille de 1 à $i - 1$ pour que l'algorithme soit correct. Pour les valeurs de j allant de 1 à $n - 1$, aucun échange n'a lieu. Or, l'analyse du programme pourra déterminer que ces dernières cellules sont plus grandes que le reste du tableau et sont dans un ordre croissant. Par conséquent, l'analyseur peut déduire que si des cellules sont échangées et puisqu'elles ne sont pas dans le bon ordre, leur indice est plus petit que i. Ainsi le fragment des cellules échangées pourra être approximé par $A[1..i[$, introduisant effectivement i parmi les bornes utilisées dans le diagramme. Sans cela, et étant donné que i n'apparaît dans aucune expression de tableau, il aurait été difficile de retrouver automatiquement cette borne.

```
Pour i de n à 1 faire
    Pour j de 1 à n − 1 faire
        Si A[j] > A[j + 1] alors
            A[j] ↔ A[j + 1]
```

Figure 11.2 : Tri à bulle non-optimisé.

Les diagrammes de tranches permettent la représentation efficace d'une collection de tranches en imposant toutefois quelques contraintes sur cette collection. Ils permettent la manipulation de fragments se chevauchant ou pouvant être vides. Il devient alors possible de ne pas systématiquement recourir à une distinction des configurations. Les opérations de manipulation des diagrammes que nous avons proposés ont toutes une complexité polynomiale. Ceci permet en théorie une amélioration de la complexité de l'analyse par rapport aux méthodes qui distinguent les configurations des structures de données. Ces dernières impliquent une complexité qui croit exponentiellement avec les cas d'alias tandis que les diagrammes ont une croissance quadratique.

Certains cas d'alias doivent toutefois être bien distingués au cours de l'analyse. (Cf. section 10.2.1) Il est toujours possible de partitionner l'ensemble d'états selon des cas d'alias bien choisis et d'utiliser une fragmentation distincte pour chacun des cas. Ce qui amène la question « quand distinguer les cas d'alias et quand ne pas les distinguer ? »

Les perspectives de recherche autour des domaines de fragmentation sont nombreuses, notamment en ce qui concerne l'expressivité. Les diagrammes de tranches pourraient être étendus aux tableaux multidimensionnels, aux structures de données chaînées, et les domaines de bornes pourraient être élargis pour accepter des expressions plus générales ou permettre le traitement des accès imbriqués. En dehors des diagrammes de tranches, on peut imaginer des domaines de fragmentations spécialisés pour certaines structures de données ou au contraire généralistes. La combinaison des domaines de fragmentation est également une question qui mériterait d'être

développée. Une idée serait par exemple de combiner les diagrammes de tranches avec un domaine de fragmentation permettant de sélectionner des cellules dont les indices vérifient des congruences.

Bibliographie

[AGG08] Xavier Allamigeon, Stéphane Gaubert, and Éric Goubault. Inferring min and max invariants using max-plus polyhedra. In *Proceedings of the 15th international symposium on Static Analysis*, SAS '08, pages 189–204, Berlin, Heidelberg, 2008. Springer-Verlag. (cité page 161)

[AGU72] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2) :131–137, 1972. (cité page 117)

[All08] Xavier Allamigeon. Non-disjunctive numerical domain for array predicate abstraction. In *Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, ESOP'08/ETAPS'08, pages 163–177, Berlin, Heidelberg, 2008. Springer-Verlag. (cité page 193)

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The essence of computation. chapter Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software, pages 85–108. Springer-Verlag New York, Inc., New York, NY, USA, 2002. (cité pages 24, 25, 57 et 171)

[BDE+10] Ahmed Bouajjani, Cezara Drăgoi, Constantin Enea, Ahmed Rezine, and Mihaela Sighireanu. Invariant synthesis for programs manipulating lists with unbounded data. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 72–88, Berlin, Heidelberg, 2010. Springer-Verlag. (cité pages 161 et 174)

[BDES09] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. A logic-based framework for reasoning about composite data structures. pages 178–195, 2009. (cité page 190)

[Bel52] R. Bellman. On the Theory of Dynamic Programming. In *Proceedings of the National Academy of Sciences*, volume 38, pages 716–719, 1952. (cité pages 2 et 132)

[BFLP03] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast : Fast acceleration of symbolic transition systems. In Warren A. Hunt Jr. and Fabio Somenzi, editors, *CAV*, volume 2725 of *Lecture Notes in Computer Science*. Springer, 2003. (cité page 195)

[BHI+09] Marius Bozga, Peter Habermehl, Radu Iosif, Filip Konecný, and Tomás Vojnar. Automatic verification of integer array programs. In *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2009. (cité page 190)

[BHJS07] Ahmed Bouajjani, Peter Habermehl, Yan Jurski, and Mihaela Sighireanu. Rewriting systems with data. pages 1–22, 2007. (cité page 190)

[BHMR07] Dirk Beyer, Thomas A. Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Path invariants. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 300–309, New York, NY, USA, 2007. ACM. (cité page 192)

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays ? In *Proceedings of the 7th international conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag. (cité page 189)

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993. (cité pages 16 et 202)

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*. Dunod, Paris, 1976. (cité pages 16 et 150)

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977. (cité pages 2 et 9)

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2 :511–547, 1992. (cité pages 2 et 13)

[CC07] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. *Sci. Comput. Program*, 64(1) :115–139, 2007. (cité page 16)

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. *SIGPLAN Not.*, 46(1) :105–118, January 2011. (cité pages 3, 4, 7, 62, 80, 164, 185, 188, 192 et 213)

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986. (cité page 2)

[CGG+05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 462–475. Springer, 2005. (cité pages 23 et 214)

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *JACM*, 50(5) :752–794, September 2003. Preliminary version in CAV'2000, LNCS 1855, Springer-Verlag. (cité page 192)

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978. (cité pages 16, 165 et 183)

[Cou03] Patrick Cousot. Verification by abstract interpretation. In *International Symposium on Verification : Theory and Practice – Honoring Zohar Manna's 64th Birthday*, volume 2772 of *Lecture Notes in Computer Science*, pages 243–268. Springer, June 2003. (cité page 193)

[CR12] Tie Cheng and Xavier Rival. An abstract domain to infer types over zones in spreadsheets. In Antoine Miné and David Schmidt, editors, *SAS*, volume 7460 of *Lecture Notes in Computer Science*, pages 94–110. Springer, 2012. (cité page 3)

[Dav05] Pichardie David. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. In french. (cité page 15)

[DDP99] Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Nicolas Halbwachs and Doron Peled, editors, *CAV*, volume 1633 of *Lecture Notes in Computer Science*, pages 160–171. Springer, 1999. (cité page 191)

[Dij76] Edsger W. Dijkstra. *A discipline of programming / Edsger W. Dijkstra*. Prentice-Hall, Englewood Cliffs, N.J. :, 1976. (cité pages 201 et 208)

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems, in lecture notes in computer science 407. In *Automatic Verification Methods for Finite State Systems, International Workshop*. Springer-Verlag, 1989. (cité pages 6, 16 et 154)

[FQ02] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL 2002*, pages 191–202. ACM, 2002. (cité pages 191 et 192)

[GDD+04] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Shmuel Sagiv. Numeric domains with summarized dimensions. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 512–529. Springer, 2004. (cité pages 3, 4, 7, 28, 50, 80, 83, 164, 171, 175, 177, 178, 185 et 213)

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In George C. Necula and Philip Wadler, editors, *POPL*, pages 235–246. ACM, 2008. (cité pages 3, 5, 6, 7, 62, 87, 136, 182 et 188)

[Gon07] Laure Gonnord. *Acceleration abstraite pour l'amelioration de la precision en analyse des relations lineaires*. PhD thesis, Université Joseph Fourier, Grenoble, France, 2007. (cité pages 195 et 200)

[GR07] Denis Gopan and Thomas W. Reps. Guided static analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 349–365. Springer, 2007. (cité page 202)

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 338–350. ACM, 2005. (cité pages 3, 80, 177, 184, 185, 192, 193 et 206)

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *Conference on Computer Aided Verification CAV'97, Haifa*, volume 1254 of *LNCS*, 1997. (cité page 191)

[HH12] Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In Antoine Miné, editor, *19th International Static Analysis Symposium, SAS'12*, pages 198–213, Deauville, France, September 2012. LNCS 7460, Springer Verlag. (cité page 203)

[HIV08] Peter Habermehl, Radu Iosif, and Tomás Vojnar. What else is decidable about integer arrays ? In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 474–489. Springer, 2008. (cité page 190)

[HMM12] Julien Henry, David Monniaux, and Matthieu Moy. Succinct representations for abstract interpretation. In *Static analysis (SAS)*, 2012. Springer Verlag. (cité pages 203 et 214)

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *ACM Conference on Programming Language Design and Implementation, PLDI 2008*, pages 339–348, Tucson (Az.), June 2008. (cité pages 3, 4, 6, 7, 8, 58, 62, 75, 80, 137, 164, 173, 175, 179, 181, 184, 185, 188, 206, 208 et 213)

[JGR05] Bertrand Jeannet, Denis Gopan, and Thomas W. Reps. A relational abstraction for functions. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 186–202. Springer, 2005. (cité pages 3, 7, 172, 175 et 213)

[JM07] R. Jhala and K. L. McMillan. Array abstractions from proofs. In W. Damm and H. Hermanns, editors, *CAV 2007*, pages 193–206. LNCS 4590, Springer Verlag, 2007. (cité page 192)

[JM09] Bertrand Jeannet and Antoine Miné. Apron : A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *CAV*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009. (cité pages 83 et 150)

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6 :133–151, 1976. (cité pages 16, 17, 139, 165 et 167)

[Kle52] Stephen Cole Kleene. *Introduction to metamathematics*. Bibl. Matematica. North-Holland, Amsterdam, 1952. (cité page 11)

[KV09] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In Marsha Chechik and Martin Wirsing, editors, *FASE*, volume 5503 of *Lecture Notes in Computer Science*, pages 470–485. Springer, 2009. (cité page 210)

[La96] Jacques-Louis Lions and al. Ariane 5 : Flight 501 failure, report by the inquiry board. Technical report, European Space Agency (ESA) and Centre national d'études spatiales (CNES), 1996. (cité page 1)

[LARSW00] Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification : A case study. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, ISSTA '00, pages 26–38, New York, NY, USA, 2000. ACM. (cité pages 176, 177, 180 et 183)

[LB04] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In R. Alur and D. Peled, editors, *CAV 2004*, pages 135–147. LNCS 3114, Springer Verlag, 2004. (cité page 192)

[LF10] Francesco Logozzo and Manuel Fähndrich. Pentagons : A weakly relational abstract domain for the efficient validation of array accesses. *Sci. Comput. Program.*, 75(9) :796–807, 2010. (cité page 16)

[Min01] Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001. (cité page 16)

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1) :31–100, 2006. (cité pages 16 et 165)

[Mon09a] David Monniaux. *Analyse statique : de la théorie à la pratique*. Habilitation to direct research, Université Joseph Fourier, Grenoble, France, July 2009. (cité page 1)

[Mon09b] David Monniaux. A minimalistic look at widening operators. *Higher-Order and Symbolic Computation*, 22(2) :145–154, 2009. (cité page 15)

[Pé10] Mathias Péron. *Contributions à l'analyse statique de programmes manipulant des tableaux*. PhD thesis, Université de Grenoble, septembre 2010. (cité pages 195 et 200)

[PH07] Mathias Péron and Nicolas Halbwachs. An abstract domain extending difference-bound matrices with disequality constraints. In *VMCAI'07 : Eighth International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, pages 268–282. Springer, January 2007. (cité page 4)

[PH10] Valentin Perrelle and Nicolas Halbwachs. An analysis of permutations in arrays. In Gilles Barthe and Manuel V. Hermenegildo, editors, *VMCAI*, volume 5944 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2010. (cité pages 161, 167, 168, 195 et 203)

[Pre29] Mojesz Pressburger. Über die vollständigkeit eines gewissen systems der arithmetik ganzer zahlen, in welchem die addition als einzige operation hervortritt. In *Comptes-Rendus du I CongrÃ¨s de MathÃ©maticiens des Pays Slaves*, pages 92–101, 1929. (cité page 189)

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *International Symposium on Programming*. LNCS 137, Springer Verlag, April 1982. (cité page 2)

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74 :358–366, 1953. (cité page 1)

[SG09] Saurabh Srivastava and Sumit Gulwani. Program verification using templates over predicate abstraction. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 223–234, New York, NY, USA, 2009. ACM. (cité page 193)

[SJR10] Pascal Sotin, Bertrand Jeannet, and Xavier Rival. Concrete memory models for shape analysis. *Electr. Notes Theor. Comput. Sci.*, 267(1) :139–150, 2010. (cité page 34)

[SKH03] Axel Simon, Andy King, and Jacob M. Howe. Two variables per linear inequality as an abstract domain. In *Proceedings of the 12th international conference on Logic based program synthesis and transformation*, LOPSTR'02, pages 71–89, Berlin, Heidelberg, 2003. Springer-Verlag. (cité page 16)

[SPW09] Mohamed Nassim Seghir, Andreas Podelski, and Thomas Wies. Abstraction refinement for quantified array assertions. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 3–18, Berlin, Heidelberg, 2009. Springer-Verlag. (cité page 192)

[SRW99] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM. (cité page 176)

[SS99] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 443–454, Trento, Italy, jul 1999. Springer-Verlag. (cité page 191)

[Syr01] Apostolos Syropoulos. Mathematics of multisets. In *Proceedings of the Workshop on Multiset Processing : Multiset Processing, Mathematical, Computer Science, and Molecular Computing Points of View*, WMP '00, pages 347–358, London, UK, UK, 2001. Springer-Verlag. (cité page 166)

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2) :285–309, 1955. (cité page 11)

[Tar72] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2) :146–160, 1972. (cité page 16)

[ZHWG10] Min Zhou, Fei He, Bow-Yaw Wang, and Ming Gu. On array theory of bounded elements. In Tayssir Touili, Byron Cook, and Paul Jackson, editors, *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 570–584. Springer, 2010. (cité page 190)