Vérification par interprétation abstraite en mémoire faiblement cohérente

Thibault Suzanne

▶ **To cite this version:**

Thibault Suzanne. Vérification par interprétation abstraite en mémoire faiblement cohérente. Cryptographie et sécurité [cs.CR]. Université Paris sciences et lettres, 2019. Français. NNT : 2019PSLEE013 . tel-02503839

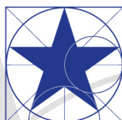
HAL Id: tel-02503839

https://theses.hal.science/tel-02503839

Submitted on 10 Mar 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École normale supérieure

Vérification par interprétation abstraite en mémoire faiblement cohérente

Soutenue par
Thibault Suzanne
Le 26 février 2019

École doctorale n°386
Sciences Mathématiques de Paris centre

Spécialité
Informatique

Composition du jury :

Xavier Rival École normale supérieure, Inria	*Président*
David Monniaux CNRS, Verimag	*Rapporteur*
David Pichardie ENS Rennes, IRISA	*Rapporteur*
Emmanuel Chailloux LIP6, Sorbonne Université	*Examinateur*
Luc Maranget Inria Paris	*Examinateur*
Antoine Miné LIP6, Sorbonne Université	*Directeur de thèse*

ENS | PSL

Demandons à l'Esprit saint, le Père des pauvres, d'illuminer notre cœur et notre intelligence pour nous conduire à la vérité toute entière.

Remerciements

La recherche n'est pas un travail solitaire, et les résultats présentés dans cette thèse doivent beaucoup à ceux que je voudrais remercier ici.

Je pense d'abord à Antoine, qui a dirigé mes travaux. J'ai été impressionné par sa capacité à voir et à présenter les problèmes soulevés sous plusieurs angles complémentaires, ce qui permettait de mieux les comprendre et, souvent, de mettre à profit les solutions envisagées dans des axes auxquels nous n'avions pas pensé. Antoine a en outre su m'encourager dans des moments plus difficiles, qu'ils soient liés à mes recherches ou plus personnels, et je le remercie sincèrement pour ce soutien qu'il m'a apporté.

Je remercie également les chercheurs qui ont accepté d'être rapporteurs de cette thèse, David Pichardie et David Monniaux. Leurs remarques bienveillantes et leurs rapports ont soulevé des questions intéressantes auxquelles j'espère que la recherche saura un jour apporter une réponse. Merci à David Pichardie d'avoir fait partie du jury, ainsi qu'à ses autres membres, Xavier Rival, Emmanuel Chailloux et Luc Maranget. Xavier et Emmanuel ont dirigé les équipes au sein desquelles j'ai eu le plaisir de travailler pendant ces années de thèse, et Luc m'a guidé pendant ma première expérience du monde de la recherche, lors d'un stage à Gallium il y a quelques années. C'était pour moi à la fois un honneur et un plaisir de profiter de leurs points de vue variés lors de la soutenance.

Les équipes Antique à l'ENS et APR au LIP6 m'ont réservé un excellent accueil pendant mon doctorat. Je remercie chaleureusement tous leurs membres avec lesquels j'ai eu le plaisir de passer du temps. Il m'est difficile de nommer tout le monde, mais je pense en particulier aux autres doctorants d'Antoine, Caterina, Abdelraouf, Ghiles, Matthieu, Raphaël et David. Matthieu et Ghiles, sans vous, ce dîner à Édimbourg aurait paru bien long.

Mon intérêt pour la recherche et la science n'est pas venu par hasard : j'ai eu pour professeur au lycée un scientifique passionné, qui m'a appris la rigueur formelle, le goût de l'expérience et l'émerveillement devant l'observation du monde. Je ne sais pas si j'ai réussi à développer au mieux toutes les qualités qu'il a essayé de m'enseigner, mais je remercie sincèrement Jean-Claude Guilloud pour ce qu'il m'a transmis.

Enfin, je remercie ma famille pour tout ce qu'ils m'ont apporté. Je commencerai par mes parents qui, entre autres choses qu'il serait trop long d'énumérer, m'ont les premiers appris à me servir d'un ordinateur. Ils ne se doutaient sans doute pas à l'époque de ce que j'en ferais. Merci également à Ombeline pour son soutien et son amour pendant une période qui n'a pas toujours été facile. Enfin, Agathe ne lira ces remerciements que dans quelques années, mais sa bonne humeur et son sourire ont été une grande joie pendant la fin de mes recherches. Je souhaite qu'elle continue à grandir en sagesse et en intelligence en s'émerveillant du monde qui l'entoure.

À tous, merci.

Table des matières

Introduction **1**
0.1 Vérification de programme et méthodes formelles . . . 1
0.1.1 Vérification automatique, sûreté et complétude . . . 1
0.1.2 Un premier essai : les tests . . . 2
0.1.3 Les méthodes formelles : vers la sûreté mathématique . . . 3
0.2 Modèles mémoire faiblement cohérents . . . 5
0.2.1 Cohérence séquentielle . . . 6
0.2.2 Un comportement contre-intuitif . . . 6
0.3 Plan . . . 7

I Contexte **9**

1 Syntaxe et sémantique séquentielle **11**

2 Interprétation abstraite **15**
2.1 Première approche de l'interprétation abstraite . . . 15
2.1.1 États, éléments et domaine concrets . . . 15
2.1.2 Le domaine des signes . . . 16
2.2 Le domaine des intervalles . . . 17
2.3 Élargissement et rétrécissement . . . 19
2.3.1 Élargir pour accélérer la convergence . . . 19
2.3.2 Élargir par seuils et rétrécir pour regagner de la précision . . . 20
2.4 Domaines relationnels . . . 21
2.4.1 Le domaine des polyèdres . . . 22
2.4.2 Le domaine des octogones . . . 23
2.4.3 Performances des domaines numériques . . . 23
2.5 Booléens et types finis . . . 24
2.6 Structure de treillis . . . 27
2.6.1 Ensembles ordonnés . . . 27
2.6.2 Treillis . . . 28
2.7 Points fixes . . . 29
2.7.1 Définitions et existence . . . 29
2.7.2 Sémantique sous forme de point fixe . . . 30
2.8 Correspondances de Galois . . . 31
2.8.1 Définition . . . 31
2.8.2 Mise en pratique : formalisation du domaine des intervalles . . . 32
2.8.3 Composition d'abstractions . . . 33

2.8.4 Plongements et isomorphismes de Galois 33
2.8.5 Quand α n'existe pas : l'exemple des polyèdres 35
2.9 Application à l'analyse statique 35
2.9.1 Opérateurs abstraits 36
2.9.2 Abstraction des points fixes 38
2.9.3 Élargissement 38
2.9.4 Stratégie d'itération 40
2.10 Conclusion 41

3 Mémoire faiblement cohérente 43
3.1 Cohérence séquentielle 43
3.2 Le modèle TSO : *Total Store Ordering* 43
3.2.1 Un modèle mémoire relâché 43
3.2.2 Tampons d'écriture 44
3.2.3 Barrière 45
3.2.4 Sémantique opérationnelle 46
3.3 Le modèle PSO : *Partial Store Ordering* 47
3.4 Autres modèles 49
3.5 Conclusion 49

II Interprétation abstraite monolithique 51

4 Considérations préalables 53
4.1 Travaux antérieurs 53
4.2 L'importance des transferts distincts 54
4.3 Graphe de contrôle produit 55

5 Sémantique et opérateurs concrets 57
5.1 Domaine concret 57
5.2 Opérateurs concrets 58
5.3 Commutation des transferts 59

6 Partitionnement 65
6.1 Domaines de partitionnement des états 65
6.2 Partitionnement selon la longueur des tampons 66

7 Abstraction des tampons 69
7.1 Condensé des tampons 69
7.1.1 Domaines non relationnels et mises à jour faibles 69
7.1.2 Extension aux domaines relationnels 71
7.1.3 Application aux tampons 72
7.2 Abstraction numérique 76
7.3 Sûreté 77
7.4 Conclusion 78

8 Résultats expérimentaux **79**
8.1 Contexte et objectifs . . . 79
8.1.1 Implémentation de l'analyseur . . . 79
8.1.2 Expériences et objectifs . . . 79
8.1.3 Suppression de barrières . . . 80
8.1.4 Cibles des analyses . . . 80
8.2 Résultats . . . 81
8.3 Interprétation . . . 81
8.3.1 Comparaison avec l'état de l'art . . . 81
8.3.2 Influence du domaine numérique . . . 83
8.3.3 Impact du partitionnement . . . 84
8.4 Conclusion . . . 84

III Interprétation abstraite modulaire **85**

9 Introduction à l'analyse modulaire **87**
9.1 Passage à l'échelle d'une analyse statique . . . 87
9.2 Interférences simples . . . 88
9.3 Interférences relationnelles . . . 89
9.4 État de l'art . . . 91
9.5 Conclusion . . . 92

10 Sémantique concrète modulaire **93**
10.1 Sémantiques transitionnelles . . . 93
10.1.1 Systèmes de transition étiquetés . . . 93
10.1.2 Sémantique monolithique . . . 93
10.2 Sémantique modulaire . . . 96
10.2.1 États locaux . . . 97
10.2.2 Interférences . . . 97
10.2.3 Formulation et méthode de calcul modulaires . . . 98
10.2.4 Clôture par transfert . . . 100

11 Domaines et sémantique abstraits **105**
11.1 Domaine des états locaux . . . 105
11.1.1 Abstraction de mémoire . . . 105
11.1.2 Abstraction de contrôle . . . 108
11.1.3 Domaine numérique final . . . 109
11.2 Interférences abstraites . . . 110
11.3 Opérateurs abstraits . . . 111
11.3.1 Sûreté . . . 112
11.3.2 Conclusion . . . 115

12 Expérimentations **117**
12.1 Implémentation de l'analyse modulaire . . . 117
12.1.1 Nombre d'itérations . . . 118
12.2 Passage à l'échelle . . . 118
12.3 Test de précision . . . 120
12.3.1 TSO et PSO . . . 121
12.4 Conclusion . . . 121

IV Approfondissement et perspectives 123

13 Ordre inter-variables dans TSO 125
13.1 Produit de domaines 125
13.1.1 Analyses séparées 126
13.1.2 Produit direct 127
13.1.3 Produit réduit 127
13.2 Considérations informelles sur l'ordre des variables 130
13.2.1 Ordonnancement des entrées 130
13.2.2 Condensation de l'ordre 131
13.2.3 Partitionnement et treillis des ordres 132
13.2.4 Opérations sur les ordres abstraits 134
13.3 Formalisation 136
13.3.1 Domaine de l'ordre 136
13.3.2 Sémantique de l'ordre 138
13.3.3 Domaine final 140
13.4 Conclusion 141

14 Abstractions non uniformes 143
14.1 Tampons triés 143
14.1.1 Limitation de l'abstraction uniforme 143
14.1.2 Domaine des tampons triés 145
14.1.3 Construction des opérateurs abstraits 145
14.1.4 Conclusion sur le tri 148
14.2 Variables auxiliaires supplémentaires 149
14.2.1 Variables auxiliaires supplémentaires pour le tampon 149
14.2.2 Application 151
14.2.3 Conclusion sur les variables auxiliaires 152
14.3 Conclusion 152

15 Recherches futures 153
15.1 Analyse en parallèle des processus 153
15.2 Mise en production dans un analyseur réel 154
15.3 Vers d'autres modèles 155

Conclusion 157

Annexe 159

A Programmes de test 161

Bibliographie 169

Introduction

Dans notre monde informatisé, les logiciels sont omniprésents. Ils comportent aussi de plus en plus de lignes de code, et combinent des composants et architectures de plus en plus complexes pour des fonctionnalités toujours plus développées. Si écrire un programme demande du temps, dans ces conditions, écrire un programme correct, c'est-à-dire qui respecte l'intention de son programmeur, peut s'avérer particulièrement difficile et fastidieux.

Or certains programmes sont qualifiés de *critiques* : on cherche alors à tout mettre en œuvre pour s'assurer de leur correction. En effet, leur dysfonctionnement a des conséquences catastrophiques : qu'il s'agisse de vies humaines comme l'équipement de radiothérapie Therac-25 [LT93] ou de sommes économiques considérables comme pour la fusée Ariane 5 [LLF+96], le surcoût lié à la vérification est largement inférieur au coût d'un échec. De plus, même les programmes considérés comme non critiques subissent un effort de certification minimal. Ils bénéficieraient donc aussi d'outils de vérification automatiques, simples à mettre en œuvre et performants.

Par ailleurs, l'amélioration des processeurs, depuis quelques années, se traduit plus par la multiplication du nombre de cœurs que par l'amélioration de leur vitesse. Ainsi, parce qu'elles permettent dans certains cas, pour un coût économique équivalent, des performances largement supérieures aux gros systèmes monolithiques, les architectures parallèles s'imposent de plus en plus dans des domaines variés du calcul. Celles-ci définissent un *modèle mémoire* : dans le cas le plus simple, celui-ci spécifie que les comportements possibles du système parallèle correspondent à des entrelacements des exécutions des *processus* qui le composent. Mais certains modèles sont dits *faiblement cohérents* : pour permettre des optimisations critiques à la performance des processeurs, ils autorisent des comportements supplémentaires.

S'assurer qu'un programme concurrent est correct est une tâche difficile, en raison du non-déterminisme des entrelacements et de l'explosion combinatoire du nombre d'exécutions possibles. En rajoutant des comportements, souvent non intuitifs, les modèles mémoire faiblement cohérents compliquent encore plus cette tâche. Il devient alors très difficile d'éviter les erreurs de raisonnement, en particulier de n'oublier aucun comportement problématique. La vérification automatique devient par conséquent nécessaire lorsqu'on exige un degré de confiance élevé.

Dans cette thèse, nous présenterons des méthodes de vérification automatique de programmes concurrents, en présence de mémoire faiblement cohérente, fondées sur l'interprétation abstraite. Nous introduirons le lecteur à notre problématique dans les sections qui suivent, puis donnerons un plan détaillé en section 0.3.

0.1 Vérification de programme et méthodes formelles

0.1.1 Vérification automatique, sûreté et complétude

Dans le cadre le plus général, vérifier un programme consiste à s'assurer que, parmi toutes ses exécutions possibles, aucune ne comporte d'erreur, c'est-à-dire de comportement non conforme

à l'intention initiale du programmeur.

Or vérifier un programme est généralement fastidieux, lent, et demande une rigueur exemplaire pour ne pas introduire d'erreurs dans la preuve elle-même. Il s'agit donc d'une tâche que les ordinateurs semblent particulièrement adaptés à effectuer à notre place. Le domaine de la vérification automatique s'intéresse à de telles analyses effectuées automatiquement, ce qui offre donc le double avantage d'éviter les erreurs dues au raisonnement humain faillible et de permettre d'exécuter ces analyses beaucoup plus rapidement.

Pour des raisons que nous développerons par la suite, de telles analyses peuvent présenter deux défauts : les *faux négatifs* et les *faux positifs*.

Un faux négatif est un programme qui, alors qu'il possède au moins une exécution incorrecte, est détecté comme correct par l'analyse. On prend alors le risque de l'exécuter sans savoir qu'il peut mener à une erreur. Une analyse qui ne peut produire aucun faux négatif est *sûre* : elle n'oublie aucun comportement possible. Si elle certifie un programme comme correct, l'analyse garantit alors qu'aucune erreur ne pourra se manifester lors de son exécution.

À l'inverse, un faux positif est un programme qui ne comporte aucune exécution incorrecte, mais que l'analyse n'arrive pas à vérifier et classe alors comme potentiellement erroné. Par une vérification plus précise, voire manuelle, du programme, on peut s'assurer de sa correction, au prix d'une perte de temps qui peut être significative. Une analyse qui ne peut produire aucun faux positif est une analyse *complète* : elle ne prend en compte aucun comportement en réalité impossible. Si une analyse complète détecte une erreur dans un programme, elle garantit qu'il existe au moins une exécution du programme dans laquelle cette erreur se manifestera.

Ces deux propriétés d'une analyse, la sûreté et la complétude, sont des caractéristiques intéressantes qu'on aimerait idéalement obtenir. Malheureusement, la théorie de la calculabilité [Tur37,Ric53] énonce l'impossibilité de la construction d'une méthode qui serait à la fois sûre et complète. En revanche, obtenir l'une ou l'autre de ces propriétés est possible : par exemple, une analyse qui considère que tous les programmes sont incorrects est trivialement sûre, alors qu'une analyse qui ne détecte jamais d'erreur est trivialement complète.

Les analyses sûres sont spécialement utiles pour la certification des programmes, c'est-à-dire pour garantir l'absence totale d'erreur qui pourrait par exemple compromettre la sûreté d'un système critique. Les analyses complètes sont quant à elles utiles pour le débogage, c'est-à-dire la recherche de bugs : on souhaite alors obtenir des erreurs dont l'existence réelle est avérée, plutôt que perdre du temps à corriger des erreurs fictives.

Notre objectif est la certification des programmes corrects. Pour cette raison, nous chercherons à concevoir des analyses sûres, en essayant de le faire de la façon la plus précise possible afin de réduire au maximum le taux de faux positifs.

0.1.2 Un premier essai : les tests

La première option de vérification automatique qui s'offre à nous, et la plus répandue parce que la plus simple à mettre en œuvre, est le *test*. Un test consiste en une exécution du programme pour un jeu de valeurs d'entrée fixé, instrumentée afin de détecter dynamiquement une éventuelle violation de propriétés de correction établies à l'avance.

Cette méthode présente deux inconvénients majeurs.

D'une part, puisqu'elle nécessite d'exécuter le programme au moins jusqu'à arriver au test de la propriété à valider, elle peut prendre un temps considérable si le programme est lent. Pour la même raison, si le programme ne termine pas, elle ne terminera pas non plus.

D'autre part, c'est une technique qui ne permet que de montrer que la propriété n'est *pas valide*, en révélant une exécution particulière qui ne la vérifie pas. En d'autres termes, la méthode

```
1  /* assume x ∈ [0; 1000] */
2
3  i = 0;
4  while (i < x) {
5      i++;
6  }
7
8  /* assert i ∈ [0; 1000] */
```

Programme 0.1 – Une simple boucle d'incrémentation.

des tests est complète, mais elle n'est pas sûre : elle peut permettre de montrer la présence de bugs, mais jamais leur absence.

Considérons par exemple le programme 0.1. On cherche à vérifier la propriété spécifiée à la ligne 8.

Pour tester ce programme, on l'exécute avec ici une valeur arbitraire choisie initialement pour `x` dans l'intervalle spécifié. Au cours de cette exécution, on peut alors vérifier simplement la validité de la propriété de la ligne 8 lorsque celle-ci est atteinte. Il se trouve par ailleurs que cette propriété est correcte : on dit alors que le programme passe les tests.

Modifions maintenant la propriété recherchée en `/* assert i ∈ [0; 999] */`. Cette propriété n'est pas vérifiée par le programme, mais on peut réaliser de nombreux tests avant de s'en rendre compte : il faut impérativement donner à `x` la valeur `1000` pour exposer un comportement erroné. Si la valeur d'entrée de `x` est choisie aléatoirement par l'outil de test, la probabilité de tirer ce nombre n'est que de 0.001. Les méthodologies de développement logiciel proposent de nombreuses heuristiques permettant de pallier partiellement ce problème, comme la mesure de *couverture* du code, qui vise à maximiser le nombre de chemins d'exécution parcourus par les tests. Si elles permettent d'obtenir des résultats utiles, elles ne sont malgré tout pas suffisantes pour résoudre le problème de fond : en particulier, dans l'exemple du programme 0.1, on peut obtenir une couverture totale avec les seules valeurs d'entrée `0` et `1`, dont aucune ne permet d'observer l'erreur.

S'ils constituent en première approche un moyen efficace et peu coûteux pour avoir une première idée de la correction d'un programme, les tests ne respectent donc pas la contrainte de sûreté que nous nous sommes fixés pour pouvoir certifier leur cible.

0.1.3 Les méthodes formelles : vers la sûreté mathématique

Les *méthodes formelles* sont un ensemble de pratiques et de techniques permettant de raisonner sur les comportements d'un programme en utilisant des outils mathématiques pour prouver rigoureusement les propriétés désirées.

Les systèmes de type : une première approche des méthodes formelles.

L'approche scientifique des *systèmes de types* représente sans doute une des premières applications historiques des méthodes formelles, ainsi que la plus répandue à l'heure actuelle. L'objectif d'un système de types est de rejeter, sûrement et statiquement – c'est-à-dire avant même de les exécuter, les programmes dont l'exécution mènerait à effectuer une opération invalide. Ainsi, la figure 1 présente le typage de trois programmes écrits dans le langage typé OCaml.

Le programme de la figure 1a est correctement rejeté par le typeur : en effet, il contient une erreur consistant à tenter d'additionner un entier et une chaîne de caractères.

```
1 let x =
2   1 + "bonjour"
```

(a) Incorrect et refusé.

```
1 let y =
2   if true
3   then 1
4   else 1 + "bonjour"
```

(b) Correct et refusé.

```
1 let z =
2   1 / 0
```

(c) Incorrect et accepté.

Figure 1 – Trois exemples de comportement du typeur OCaml.

Le programme de la figure 1b est lui aussi rejeté, pour la même raison : le typeur détecte la même opération invalide. Cependant, la branche `else` n'étant jamais empruntée, exécuter ce programme ne conduirait pas à tenter d'effectuer cette opération. Le système de types n'est pas complet : il rejette des programmes qui s'exécuteraient sans erreur. Une analyse plus précise pourrait éventuellement détecter cette impossibilité et attribuer le type `int` des entiers à `y`.

Le programme de la figure 1c met en évidence un comportement à première vue non sûr du système de types : le programme contient une erreur, à savoir une division par zéro. Pourtant, il est accepté par le typeur, qui assignera le type `int` à la variable `z`. Exécuter ce programme conduira effectivement à une erreur : il s'arrêtera brutalement en affichant `Exception: Division_by_zero`. Il convient alors de préciser ce qu'on entend par *sûr* : avec toute analyse statique vient une définition précise des erreurs à détecter. En l'occurrence, les erreurs que le système de types cherche à éviter sont les *erreurs de type*, c'est-à-dire l'emploi dans une opération de valeurs dont la nature est incompatible avec ladite opération (par exemple, une chaîne de caractères dans une addition). L'opération `1 / 0` est une division qui fait intervenir deux entiers : la nature des opérandes est compatible avec la division, elle ne comporte donc pas d'erreur de type. Qu'elle puisse mener à une erreur à l'exécution en raison du comportement de la division fait partie d'une autre classe de problèmes, que le système de types n'est pas conçu pour éviter. Ainsi, ce système de types est sûr : il rejettera obligatoirement un programme comportant une erreur *de type*.

La preuve de programme : une méthode puissante mais exigeante.

Le cœur du système de types d'OCaml présenté dans la section précédente demande très peu d'annotations à l'utilisateur : les types sont *inférés* automatiquement par le compilateur [PR05]. En revanche, il ne permet pas d'exprimer des propriétés très complexes, comme *cette fonction trie correctement la liste qu'elle reçoit en paramètre* ou *cet opérateur est toujours différent de zéro* : même si certaines de ces propriétés peuvent être encodées dans des types, le typeur n'est pas assez puissant pour les prouver seul.

À l'opposé de cet antagonisme entre expressivité et implication de l'utilisateur, on trouve les méthodes de *preuve de programme*. Celles-ci sont mises en œuvre par des outils appelés assistants de preuve, et consistent à décrire dans un langage de programmation le programme souhaité et à fournir une preuve de sa correction dans un langage de preuves. L'assistant de preuve se charge ensuite de vérifier que la preuve est mathématiquement correcte et qu'elle prouve bien la propriété recherchée de correction du programme.

Cette approche puissante permet de prouver des propriétés très expressives sur les programmes cibles, jusqu'à leur correction complète. En revanche, elle demande beaucoup d'annotations et d'interactions avec le programmeur : si des techniques pour automatiser certaines preuves existent, elles ne permettent pas d'éviter un travail conséquent sur la preuve (souvent considéré comme comparable au travail sur le programme lui-même).

Si cette méthode a du succès sur des programmes conçus avec leur preuve dès le départ [KEH+09,Ler09,JLB+15], elle ne s'adapte cependant que difficilement à la preuve de programmes

importants déjà existants.

La vérification de modèle : vers les méthodes automatiques.

La vérification de modèle, ou *model checking* en anglais, propose un compromis entre l'implication de l'utilisateur et l'expressivité de l'analyse. Elle permet en effet de vérifier des propriétés plus ou moins expressives de manière automatique.

La vérification de modèle procède en deux étapes : le programme à vérifier est d'abord *modélisé* par un système de transition d'états, puis un prouveur automatique vérifie que ce modèle satisfait bien la propriété recherchée.

Cette approche est souvent sûre *et* complète par rapport au modèle : pour rester décidable, il doit donc être fini, ou infini mais régulier. Pour réduire l'espace des états à explorer, le prouveur utilise cette propriété structurelle du modèle pour mettre en œuvre des méthodes symboliques efficaces, souvent basées sur des solveurs SAT.

La régularité des modèles les rend nécessairement moins expressifs que le langage de programmation originel : ainsi, si la méthode est sûre et complète sur le modèle considéré, ce dernier ne correspond pas forcément au programme réellement exécuté.

L'interprétation abstraite : approximer pour ne garder que l'essentiel.

Une autre théorie issue de la recherche en méthodes formelles, présentant elle aussi un compromis entre automaticité et expressivité, est l'interprétation abstraite.

Il s'agit d'une théorie générale d'approximation sûre de la sémantique des programmes, proposée par Patrick et Radhia Cousot en 1977 [CC77a]. Elle propose un cadre formel pour unifier la présentation d'analyses statiques déjà existantes, et permet d'en concevoir de nouvelles, qui sont sûres (et calculables, donc incomplètes) et correctes par construction.

L'idée fondatrice de l'interprétation abstraite est la notion d'*abstraction*. Nous avons vu qu'il est impossible de calculer exactement tous les comportements possibles d'un programme. Pour autant, il n'est en général pas nécessaire pour prouver une propriété de disposer d'une information complète sur toutes ces exécutions : une information partielle est souvent suffisante. Par exemple, pour vérifier que l'opérande droit d'une division n'est jamais nul, il est suffisant de savoir qu'il ne peut qu'être supérieur à 3 : connaître exactement toutes ses valeurs possibles (par exemple $\{3, 5, 48\}$) n'est pas nécessaire. L'abstraction consiste alors à oublier les informations qu'on juge inutiles pour ne garder que l'information nécessaire à la preuve du programme analysé. En oubliant suffisamment de détails, on peut alors regagner la calculabilité de l'analyse.

C'est dans le cadre de l'interprétation abstraite, garantissant par construction des analyses statiques sûres, que les résultats de cette thèse ont été établis. Nous présenterons plus en détail cette théorie au chapitre 2.

0.2 Modèles mémoire faiblement cohérents

En raison de leur efficacité pour un coût économique réduit et de leurs performances, l'informatique moderne a vu se développer considérablement les architectures parallèles. Des systèmes distribués aux processus multi-cœurs, ces structures de calcul sont maintenant omniprésentes. Avec ce développement vient le besoin de définir précisément le comportement des programmes qui s'y exécutent, en particulier de la communication entre leurs différents composants individuels répartis sur les nœuds du système.

Nous nous intéresserons en particulier aux programmes parallèles, c'est-à-dire composés de plusieurs processus qui s'exécutent indépendamment et qui communiquent *via* une mémoire par-

```
/* Initial: x = y = 0 */

thread /* T1 */ {
    x = 1;
    r1 = y;
}

thread /* T2 */ {
    y = 1;
    r2 = x;
}

/* Check : not (r1 = 0 && r2 = 0) */
```

PROGRAMME 0.2 – Deux processus communiquant *via* deux variables partagées.

tagée. Le *modèle mémoire* correspond alors à un ensemble de règles régissant cette communication et définissant par là les exécutions autorisées du programme global.

0.2.1 Cohérence séquentielle

Le modèle le plus intuitif est celui de la cohérence séquentielle [Lam79]. Ce modèle considère qu'une exécution d'un programme concurrent est un entrelacement des instructions de chacun de ses processus. Les opérations sur la mémoire apparaissent alors à chaque processus dans l'ordre correspondant à cet entrelacement. Autrement dit, à chaque étape, un processus est choisi de façon non déterministe pour effectuer son prochain pas d'exécution. Si celui-ci lit la mémoire à un emplacement donné, il y verra la dernière modification effectuée par l'ensemble des processus.

Considérons le programme 0.2. En raison du non-déterminisme des entrelacements dans le modèle de cohérence séquentielle, plusieurs états finaux sont possibles : si `T1` et `T2` écrivent respectivement dans `x` et `y` avant de lire ces variables, alors `r1 = r2 = 1` après exécution du programme. Si `T1` s'exécute en premier, écrit dans `x` puis lit la valeur de `y`, on atteint alors un état où `r1 = 0 && r2 = 1`. De même, on peut atteindre l'état `r1 = 1 && r2 = 0`.

En revanche, l'état `r1 = r2 = 0` est impossible : la première instruction exécutée par le programme étant une écriture dans `x` ou `y`, la lecture correspondante ne pourra que lire `1`.

0.2.2 Un comportement contre-intuitif

Cependant, exécuter l'équivalent de ce programme sur un processeur x86 multi-cœurs permet parfois d'observer cet état « interdit ». Ce comportement contre-intuitif, qui ne respecte pas les règles de la cohérence séquentielle, est dû à une optimisation réalisée par le processeur. Le cœur en charge de `T1`, remarquant qu'il doit exécuter deux instructions indépendantes (une écriture suivie d'une lecture à un autre emplacement mémoire), peut choisir de les réordonner afin de gagner du temps (ou, ce qui est équivalent, de commencer à exécuter la lecture alors que l'écriture n'est pas terminée). De même, le cœur en charge de `T2` peut réordonner ses deux instructions.

Chaque cœur voit en effet le processus qu'il doit exécuter comme un programme séquentiel : dans ce cadre, ces optimisations ne changent pas la sémantique de ce programme. Or, ce n'est plus vrai dans le cadre d'un programme concurrent : en raison de ce réordonnancement, les deux lectures de `x` et `y` peuvent être effectuées avant les écritures correspondantes, et donc lire toutes les deux `0`.

Les *modèles mémoire faiblement cohérents* [AG96] (parfois abrégé en « modèle mémoire faible ») cherchent à décrire les exécutions des programmes dans ces environnements où la cohérence séquentielle n'est plus valide. Plus qu'une description exacte d'implémentations matérielles, il s'agit de modèles théoriques dans lesquels l'exécution des programmes donne des résultats équivalents à leur comportement réel sur les processeurs représentés.

Nous chercherons dans cette thèse à définir des analyses statiques qui, en plus des comportements classiques de la cohérence séquentielle, prennent en compte ces comportements relâchés afin de certifier des programmes concurrents s'exécutant dans un modèle mémoire faiblement cohérent.

0.3 Plan

Cette thèse est organisée comme suit.

La partie I (Contexte) présente le contexte de nos travaux.

Le chapitre 1 (Syntaxe et sémantique séquentielle) commence par définir la syntaxe et la sémantique séquentielle des processus : elles nous serviront de base pour les exemples qui suivront et nous les étendrons au fur et à mesure.

Le chapitre 2 (Interprétation abstraite) présente la théorie de l'interprétation abstraite. Nous commençons par donner l'intuition des idées qui la composent, puis en définissons le cadre formel sur lequel nous construirons nos analyses.

Le chapitre 3 (Mémoire faiblement cohérente) définit les modèles mémoire que nous traiterons. Nous y formalisons en particulier la sémantique des programmes concurrents dans TSO et PSO.

En partie II (Interprétation abstraite monolithique), nous construirons une analyse par interprétation abstraite monolithique, par itération sur le graphe produit des processus.

Le chapitre 4 (Considérations préalables) définit formellement ce graphe produit et montre comment il induit une méthode d'itération. Il dresse également l'état de l'art de l'analyse statique en mémoire faiblement cohérente et présente un résultat sur une tentative plusieurs fois observée de simplification erronée du modèle.

Le chapitre 5 (Sémantique et opérateurs concrets) présente la sémantique concrète dans PSO sous la forme d'un système d'équations, plus adaptée à l'interprétation abstraite que les transitions du chapitre 3 (Mémoire faiblement cohérente).

Le chapitre 6 (Partitionnement) définit les domaines abstraits de partitionnement des états et en décrit une application aux états de PSO.

Le chapitre 7 (Abstraction des tampons) décrit le domaine abstrait monolithique à proprement parler. Il commence par présenter les domaines de condensation, puis les applique aux états de PSO, complète le domaine par une abstraction numérique, et formule les opérateurs abstraits ainsi qu'une preuve de sûreté du résultat.

Le chapitre 8 (Résultats expérimentaux) présente les résultats expérimentaux d'un prototype d'implémentation de l'analyse monolithique. Nous nous y comparons à l'état de l'art sur des critères de précision et de performance de l'analyse. Nous y discutons également l'impact expérimental de certains choix de conception.

La partie III (Interprétation abstraite modulaire) présente une analyse modulaire pour résoudre les problèmes de passage à l'échelle de l'interprétation abstraite monolithique.

Nous introduisons le lecteur aux principes de l'interprétation abstraite modulaire au chapitre 9 (Introduction à l'analyse modulaire), dans le cadre formel des interférences qui modélisent l'effet d'un processus sur le reste du programme.

Le chapitre 10 (Sémantique concrète modulaire) définit une sémantique concrète modulaire à l'aide du formalisme des systèmes de transition. Nous montrons comment obtenir une sémantique transitionnelle monolithique depuis les équations opérationnelles, puis la transformons en une définition modulaire qui s'écrit comme un point fixe imbriqué où les processus sont analysés séparément jusqu'à ce que tous leurs effets sur le reste du programme aient été pris en compte.

Nous abstrayons alors les composants de cette sémantique dans le chapitre 11 (Domaines et sémantique abstraits). La particularité par rapport à l'analyse monolithique est l'abstraction supplémentaire des points de contrôle des processus, et la particularité par rapport à l'analyse monolithique en cohérence séquentielle est l'algorithme de clôture par transfert modulaire.

Le chapitre 12 (Expérimentations) présente les résultats expérimentaux de l'analyse modulaire. Nous y discutons notamment le passage à l'échelle, le validant par rapport à l'analyse monolithique tout en montrant ses limites par rapport aux résultats de l'état de l'art en cohérence séquentielle.

La partie IV (Approfondissement et perspectives) présente des pistes d'amélioration de notre analyse. Elle est découpée en trois chapitres indépendants.

Le chapitre 13 (Ordre inter-variables dans TSO) définit un domaine abstrait de l'ordre entre les variables différentes au sein d'un même tampon. Nous montrons comment profiter de ce domaine à l'aide de la construction de produit réduit et formulons les opérateurs abstraits correspondants.

Le chapitre 14 (Abstractions non uniformes) aborde les abstractions non uniformes pour une représentation plus précise des tampons. Nous y montrons comment modéliser l'invariant « un tampon est trié », puis généralisons cette construction à l'aide d'une variable auxiliaire supplémentaire en montrant comment inférer des relations plus générales sans aide de l'utilisateur.

Le chapitre 15 (Recherches futures) enfin présente des pistes de recherche future qu'il est possible de construire sur cette thèse. Nous y discutons notamment une méthode d'analyse modulaire exploitant le parallélisme de la machine faisant tourner l'analyseur, la mise en production dans un analyseur réel et l'extension à d'autres modèles plus complexes.

Première partie

Contexte

Chapitre 1

Syntaxe et sémantique séquentielle

Avant de parler d'analyse de programme et de modèles d'exécution concurrente, nous commencerons par définir dans ce chapitre la syntaxe et la sémantique d'un processus isolé, équivalent à un programme séquentiel. Nous référons le lecteur aux programmes 0.1 (page 3) et 0.2 (page 6) pour des exemples déjà étudiés écrits dans ce langage cible.

Les programmes que nous considérerons seront écrits dans un langage impératif concurrent minimal. Ils sont composés d'un nombre syntaxiquement fixé de processus : il n'est pas possible d'en créer dynamiquement. Le code de chaque processus et l'ensemble des variables du programme sont également statiquement fixés. Les structures de contrôle disponibles sont les boucles `while` et les conditionnelles `if/else` ; il n'existe pas de notion de fonction ou procédure. Variables et expressions peuvent avoir les types *entier* (non borné) ou *booléen*. Il n'existe pas de primitive de synchronisation ; toutefois, nous analyserons plusieurs programmes qui implémentent ces opérations dans le langage.

Ces choix se justifient par un objectif de simplification maximale du langage, afin de concentrer notre étude sur les traits propres à la cohérence faible. De plus, les modèles que nous étudierons plus spécifiquement se définissent au niveau de l'assembleur : nous pouvons donc nous contenter d'un langage minimaliste pour pouvoir les observer complètement.

Syntaxe.

La figure 1.1 définit la syntaxe de nos programmes. Certains modèles lui ajouteront des instructions : nous le préciserons alors en les présentant. Pour ne pas alourdir sa présentation, cette grammaire ne précise pas la priorité des opérateurs. Nous utiliserons la convention habituelle, où $\langle\dagger\rangle$ les liste par priorité décroissante.

Notations. Nous noterons $\mathbb{B} \triangleq \{\texttt{true}, \texttt{false}\}$ l'ensemble des booléens, et $\mathbb{V} \triangleq \mathbb{Z} \cup \mathbb{B}$ l'ensemble des valeurs possibles pour une variable donnée. Par la suite, *Var* dénotera l'ensemble des variables du programme considéré. Pour une fonction $f : A \to B$, $x \in A$ et $y \in B$, on note $f[x \mapsto y]$ la fonction de $A \to B$ qui associe y à x et $f(x')$ à tout élément $x' \in A, x' \neq x$.

$\langle \mathit{prog} \rangle ::= \langle \mathit{thread} \rangle\ *$

$\langle \mathit{thread} \rangle ::= \texttt{thread}\ `\texttt{\{}'\ \langle \mathit{stmt} \rangle\ `\texttt{\}}'$

$\langle \mathit{stmt} \rangle ::=$
- $|\ \langle \mathit{var} \rangle\ `\texttt{=}'\ \langle \mathit{expr} \rangle$
- $|\ \texttt{if}\ \langle \mathit{expr} \rangle\ `\texttt{\{}'\ \langle \mathit{stmt} \rangle\ `\texttt{\}}'\ [\texttt{else}\ `\texttt{\{}'\ \langle \mathit{stmt} \rangle\ `\texttt{\}}']$
- $|\ \texttt{while}\ \langle \mathit{expr} \rangle\ `\texttt{\{}'\ [\ \langle \mathit{stmt} \rangle\ ]\ `\texttt{\}}'$
- $|\ \langle \mathit{stmt} \rangle\ `\texttt{;}'\ \langle \mathit{stmt} \rangle$

$\langle \mathit{expr} \rangle ::=$
- $|\ \langle \mathit{var} \rangle$
- $|\ n \in \mathbb{V}$
- $|\ \langle \mathit{expr} \rangle\ \dagger\ \langle \mathit{expr} \rangle$
- $|\ \star \langle \mathit{expr} \rangle$
- $|\ `\texttt{(}'\ \langle \mathit{expr} \rangle\ `\texttt{)}'$

$\langle \mathit{var} \rangle ::= \texttt{x},\ \texttt{y},\ \texttt{z}\ldots$

$\langle \dagger \rangle ::= `\texttt{*}' \mid `\texttt{/}' \mid `\texttt{+}' \mid `\texttt{-}' \mid `\texttt{=}' \mid `\texttt{!=}' \mid `\texttt{<}' \mid `\texttt{>}' \mid `\texttt{<=}' \mid `\texttt{>=}' \mid `\texttt{\&\&}' \mid `\texttt{||}'$

$\langle \star \rangle ::= \texttt{not} \mid `\texttt{-}'$

Figure 1.1 – Syntaxe des programmes concurrents.

Sémantique séquentielle.

La sémantique isolée de chaque processus est décrite en figure 1.2, sous une forme opérationnelle à petits pas. Celle-ci précise la modification d'une mémoire globale $S : Var \to \mathbb{V}$, associant une valeur à chacune de ses variables, par un pas d'exécution d'un programme donné. Elle est décomposée en une relation $\Rightarrow$ d'évaluation des expressions et une relation $\to$ représentant une étape d'exécution d'un programme depuis un état donné, lui associant l'état obtenu et, le cas échéant, le reste du programme à exécuter. Ces relations ont donc la signification suivante :

- $\langle e, S \rangle \Rightarrow v$: l'expression e s'évalue en v dans l'état S
- $\langle s, S \rangle \to \langle s', S' \rangle$: il existe un pas d'exécution du bloc d'instructions s transformant l'état S en l'état S'. s' est alors le bloc d'instructions à exécuter pour continuer le déroulement du programme.
- $\langle s, S \rangle \to S'$: il existe un pas d'exécution du bloc d'instruction s transformant l'état S en l'état S'. L'exécution du programme sera alors terminée.

Les règles spécifiant les opérations mémoire (lecture et écriture) ont été séparées visuellement afin de faciliter la définition de futurs modèles de cohérence.

La sémantique d'un programme concurrent composé de plusieurs processus étant intrinsèquement liée au modèle mémoire sous-jacent, nous ne la définirons pas ici. Elle sera précisée lors de la présentation de chaque modèle mémoire, en énonçant les règles qu'il spécifie.

Typage. Toujours pour ne pas alourdir la présentation, nous avons omis les règles de typage précisant quelles sont les opérations valides sur les entiers et les booléens. Nous considérerons donc dans la suite de cette thèse que tous les programmes considérés sont correctement typés au sens usuel, ce qui peut par ailleurs être vérifié par une analyse statique sommaire.

Maintenant que la syntaxe de nos programmes et la sémantique de leurs processus sont définies, nous pouvons procéder à leur analyse. Les chapitres suivants préciseront à cette fin la sémantique, soit en en donnant une présentation différente plus adaptée à certains problèmes, soit en définissant des comportements supplémentaires au fur et à mesure que nous introduirons les spécificités de nos modèles.

$$\textsc{Read}\quad \frac{x \in \mathit{Var} \qquad S(x) = v}{\langle x, S\rangle \Rightarrow v} \qquad\qquad \textsc{Write}\quad \frac{\langle e, S\rangle \Rightarrow n}{\langle x \texttt{ = } e, S\rangle \rightarrow S[x \mapsto n]}$$

(a) Sémantique des opérations mémoire.

$$\textsc{Const}\quad \frac{v \in \mathbb{V}}{\langle v, S\rangle \Rightarrow v} \qquad\qquad \textsc{Paren}\quad \frac{\langle e, S\rangle \Rightarrow v}{\langle (e), S\rangle \Rightarrow v}$$

$$\textsc{Unop}\quad \frac{\langle e, S\rangle \Rightarrow v}{\langle \star e, S\rangle \Rightarrow \star v} \qquad\qquad \textsc{Binop}\quad \frac{v, w \in \mathbb{V} \qquad \langle e_1, S\rangle \Rightarrow v \qquad \langle e_2, S\rangle \Rightarrow w}{\langle e_1 \dagger e_2, S\rangle \Rightarrow v \dagger w}$$

(b) Évaluation d'une expression dans un état mémoire S.

$$\textsc{Seq1}\quad \frac{\langle s_1, S\rangle \rightarrow S'}{\langle s_1 ;\ s_2, S\rangle \rightarrow \langle s_2, S'\rangle} \qquad\qquad \textsc{Seq2}\quad \frac{\langle s_1, S\rangle \rightarrow \langle s_1', S'\rangle}{\langle s_1 ;\ s_2, S\rangle \rightarrow \langle s_1' ; s_2, S'\rangle}$$

$$\textsc{IfTrue}\quad \frac{\langle e, S\rangle \Rightarrow \texttt{true}}{\langle \texttt{if } e \texttt{ \{ } s \texttt{ \}}, S\rangle \rightarrow \langle s, S\rangle} \qquad\qquad \textsc{IfFalse}\quad \frac{\langle e, S\rangle \Rightarrow \texttt{false}}{\langle \texttt{if } e \texttt{ \{ } s \texttt{ \}}, S\rangle \rightarrow S}$$

$$\textsc{IfElseTrue}\quad \frac{\langle e, S\rangle \Rightarrow \texttt{true}}{\langle \texttt{if } e \texttt{ \{ } s \texttt{ \} else \{ } s' \texttt{ \}}, S\rangle \rightarrow \langle s, S\rangle} \qquad\qquad \textsc{IfElseFalse}\quad \frac{\langle e, S\rangle \Rightarrow \texttt{false}}{\langle \texttt{if } e \texttt{ \{ } s \texttt{ \} else \{ } s' \texttt{ \}}, S\rangle \rightarrow \langle s', S\rangle}$$

$$\textsc{WhileEnter}\quad \frac{\langle e, S\rangle \Rightarrow \texttt{true}}{\langle \texttt{while } e \texttt{ \{ } s \texttt{ \}}, S\rangle \rightarrow \langle s ; \texttt{while } e \texttt{ \{ } s \texttt{ \}}, S\rangle} \qquad\qquad \textsc{WhileLeave}\quad \frac{\langle e, S\rangle \Rightarrow \texttt{false}}{\langle \texttt{while } e \texttt{ \{ } s \texttt{ \}}, S\rangle \rightarrow S}$$

(c) Étapes d'exécution séquentielle.

FIGURE 1.2 – Sémantique séquentielle d'un processus isolé.

Chapitre 2

Interprétation abstraite

Nous avons présenté très brièvement l'interprétation abstraite en introduction de cette thèse. Nous l'exposerons plus en détail dans ce chapitre, en commençant par une approche intuitive basée sur des exemples avant d'en définir le cadre formel.

2.1 Première approche de l'interprétation abstraite

Le programme 2.1 contient deux propriétés que l'on souhaite vérifier. Ce programme répète simplement une boucle d'incrémentation qui ajoute un nombre aléatoire entre 1 et 10 compris à la variable `x` (`x += [1, 10]`) tant que celle-ci ne dépasse pas 100. Cette boucle sera donc répétée entre 10 et 100 fois, et à la fin de l'exécution du programme, `x` est donc compris entre 101 et 110. Les deux propriétés à vérifier sont plus relâchées (donc *a fortiori* correctes) : elles stipulent respectivement que `x` doit être positif et inférieur à 200. Nous chercherons à les vérifier par interprétation abstraite.

2.1.1 États, éléments et domaine concrets

Durant l'exécution d'un programme, il passe par différents *états concrets*, c'est-à-dire des ensembles de valeurs pour chaque variable du programme ; couplés à une localisation dans son code appelée *point de contrôle*.

On appelle alors *élément concret* un ensemble d'états concrets. La *sémantique d'états concrète* du programme associe, à chaque point de contrôle, l'élément concret qui contient tous les états accessibles. On appelle *domaine concret* l'ensemble auquel appartiennent les éléments concrets, c'est-à-dire l'ensemble des ensembles d'états concrets.

```
1  x = 1;
2
3  while (x <= 100) {
4      x += [1, 10];
5  }
6
7  /* assert x >= 0 */
8  /* assert x < 200 */
```

Programme 2.1 – Deux propriétés à vérifier.

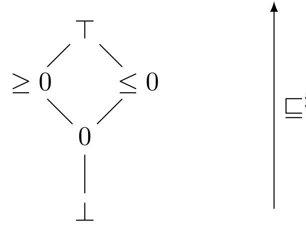


FIGURE 2.1 – Diagramme de Hasse du domaine des signes.

Lorsque les variables sont à valeur numérique, on parle de domaine numérique. Une représentation « visuelle » de ces domaines consiste à considérer, pour un environnement à n variables, un espace géométrique de dimension n où chaque variable représente une dimension. Un état concret est alors associé à un point de cet espace, dont chaque coordonnée est la valeur de la variable correspondant à la dimension en question dans l'état du programme. Un élément concret est donc un ensemble de points. Nous verrons par la suite comment visualiser ainsi différents domaines numériques abstraits.

2.1.2 Le domaine des signes

Revenons au programme 2.1. Pour la première propriété, à la ligne 7, nous abstrairons les valeurs possibles de la variable `x` en ne retenant que leur signe. Nous définirons alors un *domaine abstrait*, représentant les différents signes possibles : `x` peut être positif (au sens large), négatif ou nul. À ces différentes possibilités, nous ajouterons deux autres niveaux possibles d'information :

- $\top$ (prononcer *top*) signifie que tous les signes sont possibles.
- $\bot$ (prononcer *bottom*) signifie qu'aucun signe n'est possible : `x` n'est alors pas défini.

Ces différents éléments abstraits sont *partiellement ordonnés*, selon la précision de l'information qu'ils apportent. Ainsi, $x = 0$ est plus précis que $x \geq 0$, et $x \geq 0$ n'est pas comparable avec $x \leq 0$. On considère que $\top$ représente l'information la moins précise, et $\bot$ la plus précise. Cet ordre partiel est représenté en figure 2.1 à l'aide d'un *diagramme de Hasse*. $\sqsubseteq^\sharp$ est la relation « est une information plus précise que ».

Nous allons maintenant interpréter le programme à vérifier, mais dans une *sémantique abstraite* : la variable `x` prendra ses valeurs dans notre domaine abstrait, et les différents opérateurs du langage seront redéfinis en des *opérateurs abstraits* travaillant sur ce domaine des signes. La figure 2.2 présente l'opérateur abstrait $+^\sharp$ qui réalise l'addition de deux signes. Il s'agit essentiellement d'une transcription des règles usuelles de l'addition : l'addition de deux nombres positifs est un nombre positif, l'addition d'un nombre positif et d'un nombre négatif peut a priori avoir n'importe quel signe (d'où $\top$)… et si x n'est pas défini, l'addition de x et de y ne l'est pas non plus.

Déroulons alors l'interprétation abstraite du programme 2.1. À chaque point du programme, on calculera l'ensemble des valeurs possibles pour `x`.

- Initialement, nous n'avons aucune information sur la valeur de `x` : $x = \top$.
- On affecte ensuite un nombre positif à `x` : $x = (\geq 0)$.
- On rentre dans la boucle.

$+^\sharp$ est commutatif, et :

$$\begin{array}{ccc}
\forall x, x +^\sharp \bot = \bot & 0 +^\sharp 0 = 0 & (\geq 0) +^\sharp (\geq 0) = (\geq 0) \\
\forall x \neq \bot, x +^\sharp \top = \top & 0 +^\sharp (\geq 0) = (\geq 0) & (\leq 0) +^\sharp (\leq 0) = (\leq 0) \\
 & 0 +^\sharp (\leq 0) = (\leq 0) & (\leq 0) +^\sharp (\geq 0) = \top
\end{array}$$

FIGURE 2.2 – Addition abstraite dans le domaine des signes.

- On additionne un nombre positif à `x` : le résultat est positif.
- On stocke le résultat dans `x` : $x = (\geq 0)$.
- On retourne en tête de boucle : on a toujours $x = (\geq 0)$ comme seule valeur possible au point de programme « tête de la boucle ».
- On peut ainsi répéter la boucle : on obtiendra toujours le même résultat. En observant après la deuxième itération que ce résultat ne change pas, on considère donc qu'on a atteint un « point fixe » : le résultat restera le même et il est alors inutile de calculer des itérations supplémentaires. Ainsi, le nombre d'itérations abstraites effectives (ici 2) peut être différent de celui dans le concret (ici de 10 à 100), ce qui permet au passage de regagner la calculabilité lorsqu'un passe de l'infini à un nombre borné.

- On sort ensuite de la boucle : à la ligne 7, $x = (\geq 0)$.

On peut alors vérifier que la propriété énoncée à la ligne 7 est correcte pour le programme considéré. En revanche, cette analyse n'est pas assez précise pour vérifier la propriété de la ligne 8 : il nous faut utiliser un domaine plus précis, comme les intervalles.

2.2 Le domaine des intervalles

La figure 2.3 présente le diagramme de Hasse du domaine des *intervalles*. Les éléments de ce domaine sont les intervalles d'entiers habituels (éventuellement non bornés à gauche ou à droite), auxquels on a ajouté la valeur $\bot$ représentant un intervalle vide. $\sqsubseteq^\sharp$ est la relation d'inclusion usuelle.

Sur ce domaine, on dispose d'une opération de *borne supérieure* $\sqcup^\sharp$. La borne supérieure de deux intervalles est le plus petit intervalle qui contient leur union au sens ensembliste :

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

On dispose également d'une opération d'addition abstraite, calculant l'intervalle de toutes les valeurs possibles lorsqu'on additionne des nombres dans deux intervalles :

$$[a, b] +^\sharp [c, d] = [a + c, b + d]$$

Une fois le domaine muni de ces opérations, nous pouvons procéder à l'interprétation abstraite du programme 2.1. Contrairement à une interprétation classique, l'interprétation abstraite, en présence de boucles, accumulera à chaque étape les valeurs de l'itération précédente à celle de la nouvelle itération. On procède ainsi parce qu'on cherche, par exemple, un invariant en tête de boucle, donc valable à toute itération.

- À l'entrée dans le programme, $x =]-\infty, +\infty[$.

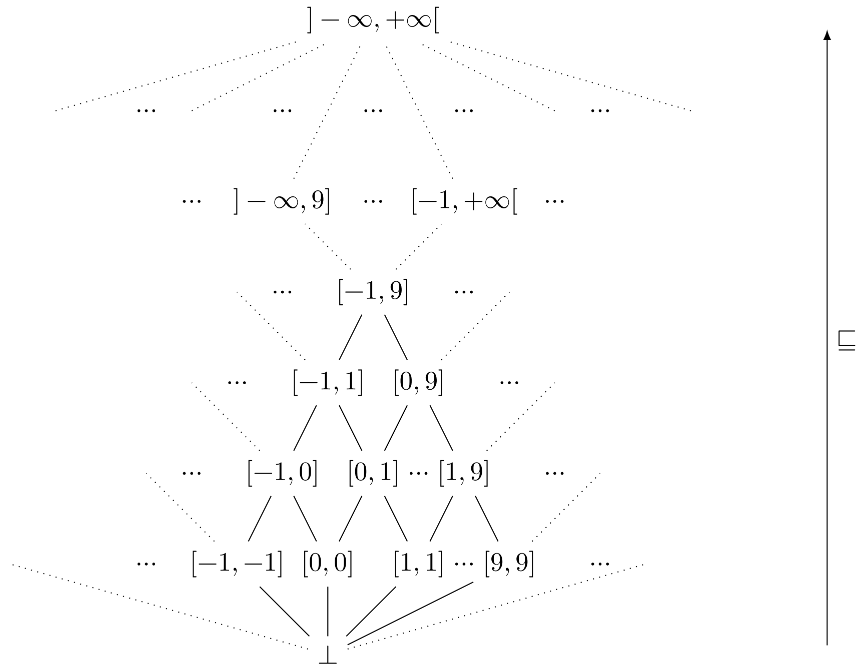


Figure 2.3 – Diagramme de Hasse du domaine des intervalles.

- Après l'exécution de la première instruction, $x = [1, 1]$.
- On entre alors dans la boucle : pour l'instant, les valeurs possibles de x au point « tête de la boucle » sont donc représentées par l'élément abstrait $[1, 1]$.
  - La condition d'entrée dans la boucle est $x <= 100$. Pour l'instant, toutes les valeurs possibles de x respectent cette condition : au début du corps de la boucle, on a donc $x = [1, 1]$.
  - Après exécution du corps de la boucle, $x = [2, 11]$.
  - On retourne alors en tête de boucle. Pour mettre à jour par accumulation la valeur possible de `x` en ce point du programme, on procède alors au calcul de la borne supérieure de la valeur précédemment connue et de la valeur calculée après un tour : $x = [1, 1] \sqcup^\sharp [2, 11] = [1, 11]$.
  - On recommence alors l'interprétation de la boucle avec la valeur de x mise à jour. Après 10 tours, on obtient le résultat $x = [1, 101]$. Ce résultat contient une valeur qui ne permet pas de rentrer dans la boucle : 101. Il est alors possible de commencer à calculer les valeurs en sortie de boucle ; toutefois, nous préférerons les calculer directement une fois que toutes les valeurs possibles en tête de boucle seront connues.
  - Au prochain tour de boucle, la condition d'entrée permet en revanche de filtrer les valeurs possibles : en entrée de boucle, $x = [1, 100]$.
  - Après un tour supplémentaire, on obtient en tête de boucle $x = [2, 110] \sqcup^\sharp [1, 101] = [1, 110]$.

```
1  x = 1;
2
3  while (x <= 1000) {
4      x = x + 1;
5      /* assert x < 2000 */
6  }
7
8  /* assert x > 100 */
9  /* assert x < 2000 */
```

PROGRAMME 2.2 – Une boucle répétée de nombreuses fois.

- L'analyse de la boucle continue encore un tour de plus, et on retrouve à nouveau $[1, 110]$ comme valeur en tête de boucle. Le point fixe est atteint et la valeur de x en tête de boucle stabilisée : on peut maintenant passer à la suite du programme.

• On sort alors de la boucle : la condition de sortie depuis la valeur $[1, 110]$ permet d'obtenir $x = [101, 110]$ en sortie de boucle.

On peut alors vérifier la propriété de la ligne 8.

2.3 Élargissement et rétrécissement

Considérons une variation de l'exemple précédent, le programme 2.2. Le code et les propriétés sont similaires, mais l'incrémentation de `x` ne se fait que par pas de 1 et continue jusqu'à 1000. Pour prouver les propriétés demandées, nous pourrions donc simplement utiliser les intervalles comme précédemment : après 1000 itérations de la boucle `while` et applications de l'opérateur $\sqcup^\sharp$, l'analyse inférerait l'invariant $x = [1, 1001]$ en tête de la boucle (c'est-à-dire avant de tester la condition du `while`). On peut alors sortir de la boucle, et la condition de sortie permet d'inférer $x = [1001, 1001]$ en fin d'exécution du programme. Les trois propriétés souhaitées sont ainsi vérifiables. Cette analyse serait toutefois particulièrement lente : elle nécessiterait en effet d'itérer 1000 fois le calcul des abstractions au niveau de la boucle.

2.3.1 Élargir pour accélérer la convergence

On peut accélérer l'analyse du programme en utilisant un opérateur d'élargissement (en anglais *widening*) ∇ [CC77a]. Cet opérateur a pour opérandes deux éléments abstraits, obtenus lors de deux itérations successives comme la valeur associée à un même point du programme. Il a pour rôle d'accélérer la convergence en extrapolant les bornes variables de ses opérandes pour atteindre immédiatement une valeur stable.

Ainsi, les deux valeurs abstraites obtenues à deux itérations successives en tête de boucle pour x sont $[1, 1]$ et $[1, 2]$. L'opérateur d'élargissement standard des intervalles garde la borne inférieure, qui ne change pas, et extrapole la croissance de la borne supérieure vers l'infini :

$$[1, 1] \ \nabla \ [1, 2] = [1, +\infty[$$

Lors de la prochaine itération, l'application de la condition d'entrée dans la boucle à la valeur élargie $[1, +\infty[$ infère $x = [1, 1000]$ avant l'exécution de l'instruction `x = x + 1`, soit $x = [2, 1001]$

après cette instruction. La propriété de la ligne 5 est alors vérifiée, et l'application de l'opérateur $\sqcup^\sharp$ infère alors en tête de boucle :

$$x = [1, +\infty[\ \sqcup^\sharp \ [2, 1001] = [1, +\infty[$$

Cette valeur est donc stable : l'interprétation abstraite de la boucle est terminée. En sortie de boucle, on obtient alors la valeur abstraite $x = [1001, +\infty[$. Cette analyse est donc également suffisante pour prouver la propriété de la ligne 8, tout en étant nettement plus rapide que l'analyse sans élargissement.

Le choix du point d'élargissement a une importance non négligeable pour la précision de l'analyse : ainsi, si on choisit d'élargir après l'exécution de l'instruction `x = x + 1` dans le corps de la boucle, on obtient comme valeur abstraite $x = [2, +\infty[$ en ce point. On ne peut alors plus vérifier la propriété de la ligne 5. Nous développerons le choix des points d'élargissement et les contraintes sur l'opérateur ∇ en section 2.9.4 ; notons simplement que l'intuition indique que l'application de la condition permet de circonscrire la valeur obtenue après élargissement lorsque celui-ci est réalisé en tête de boucle.

2.3.2 Élargir par seuils et rétrécir pour regagner de la précision

L'analyse avec élargissement nous a permis de vérifier les propriétés des lignes 5 et 8, mais la valeur abstraite $x = [1001, +\infty[$ calculée en fin d'exécution du programme ne permet pas de vérifier celle de la ligne 9. L'extrapolation due à l'élargissement a perdu l'information nécessaire.

Il existe plusieurs techniques pour tenter de regagner une précision suffisante lors d'une analyse sans pour autant dégrader la rapidité de la convergence.

La première méthode est simplement de retarder l'élargissement en commençant par calculer un certain nombre d'itérations avant de commencer à l'appliquer. Cette méthode permet de garder une précision maximale pour les boucles qui convergent rapidement même sans accélération, mais elle n'est pas applicable pour le programme 2.2 : il serait en effet nécessaire de réaliser 1000 étapes de calcul avant de stabiliser la valeur de `x`. En revanche, pour un programme qui ferait intervenir des boucles courtes et d'autres plus lentes à converger, elle permettrait de garder une bonne précision pour les premières tout en accélérant le calcul des secondes. Enfin, même sur les boucles qu'elle conduit à élargir, cette méthode peut mener à la stabilisation de certaines contraintes lors des premières itérations, permettant à l'élargissement de les conserver plutôt que les extrapoler à l'infini, ce qui permet encore une meilleure précision.

Une autre méthode est l'élargissement par seuils : plutôt que de sauter directement à l'infini, l'opérateur ∇ passera par un nombre fini d'étapes lors de l'extrapolation d'une borne. Ainsi, on pourrait choisir comme seuils $\{100, 500, 1500, +\infty\}$ pour l'analyse du programme 2.2. Les valeurs abstraites calculées pour `x` en tête de boucle seront alors les intervalles successifs suivants :

$$\begin{aligned}
x_0 &= [1, 1] \\
x_1 &= [1, 1] \ \nabla \ [1, 2] = [1, 10] \\
x_2 &= [1, 10] \ \nabla \ [1, 11] = [1, 100] \\
x_3 &= [1, 100] \ \nabla \ [1, 101] = [1, 500] \\
x_4 &= [1, 500] \ \nabla \ [1, 501] = [1, 1500] \\
x_5 &= [1, 1500] \ \nabla \ [1, 1001] = [1, 1500]
\end{aligned}$$

La valeur de `x` est donc stabilisée au bout de 5 itérations, et on peut alors vérifier la propriété de la ligne 9 : après la boucle, l'analyse infère $x = [1001, 1500]$.

```
1  x = 1;
2  y = 1;
3
4  while (x <= 1000) {
5      x = x + 1;
6      y = y + 1;
7  }
8
9  /* assert x > 100 */
10 /* assert y > 100 */
```

Programme 2.3 – Un programme faisant intervenir deux variables.

Une dernière méthode est l'utilisation d'un opérateur de rétrécissement Δ. Cet opérateur consiste, à partir du résultat stable obtenu après élargissement, à effectuer une itération supplémentaire de la boucle et à raffiner les bornes initialement infinies qui auraient été ramenées à des bornes finies après cette itération.

Reprenons par exemple notre première analyse avec élargissement. Elle infère après stabilisation $x = [1, +\infty[$ en tête de boucle. En exécutant la boucle une seconde fois, grâce au filtre de la condition d'entrée et après application de $\sqcup^\sharp$ avec la valeur initiale $x = [1, 1]$, on obtient alors $x = [1, 1001]$. L'opérateur de rétrécissement raffine alors la borne supérieure redevenue finie, et on obtient en tête de boucle :

$$x = [1, +\infty[\ \Delta\ [1, 1001] = [1, 1001]$$

En sortie de la boucle, on retrouve alors la valeur abstraite $x = [1001, 1001]$. On remarque qu'on a ici retrouvé la valeur abstraite la plus précise pour x, tout en n'ayant effectué que deux itérations de calcul de la boucle (une pour l'élargissement et une pour le rétrécissement). On peut alors vérifier la propriété de la ligne 9.

De plus, ce nombre d'itérations abstraites (2) est indépendant de la valeur (`1000`) utilisée dans la condition de boucle : on peut par exemple la remplacer par `100000000` et être aussi efficace dans l'abstrait.

2.4 Domaines relationnels

Le programme 2.3 est semblable au programme 2.2. Toutefois, en plus de la variable `x`, une variable `y`, absente de la condition, est incrémentée à chaque tour de boucle.

À la fin de l'exécution de ce programme, la propriété concrète la plus précise qu'on puisse établir est $x = 1001 \wedge y = 1001$. On remarquera que les propriétés des lignes 10 et 9 sont strictement plus faibles, donc bien valides. Nous allons chercher à la vérifier à l'aide d'une analyse par interprétation abstraite.

Considérons d'abord le domaine des intervalles. Lorsque l'état du programme comporte plusieurs variables, ce domaine est simplement étendu en représentant chacune de ces variables par un intervalle propre. Les opérations faisant intervenir plusieurs variables sont alors calculées à l'aide des intervalles les représentant. Sans élargissement, cette analyse commencera par tenter de stabiliser la boucle de façon similaire au programme 2.1. Ainsi, après 1000 tours, elle calculera l'invariant en entrée de boucle $x = [1, 1001] \wedge y = [1, 1001]$. Or la condition d'entrée de boucle ne concerne que `x` : l'analyse entre dans le tour suivant avec l'invariant, au début du corps de la boucle, $x = [1, 1000] \wedge y = [1, 1001]$. Après ce 1001$^{\text{e}}$ tour, l'invariant accumulé en tête de boucle

est donc $x = [1, 1001] \wedge y = [1, 1002]$. Et ainsi de suite : si la valeur de x est stabilisée, celle de y ne cesse d'augmenter, n'étant pas filtrée par la condition de la boucle.

Il est donc ici nécessaire d'utiliser un élargissement pour assurer la terminaison de l'analyse. En l'appliquant dès la première itération, on obtient l'invariant de fin de programme $x = [1001, +\infty[\wedge\ y = [1, +\infty[$. S'il permet de prouver la propriété de la ligne 9, il est insuffisant pour vérifier celle de la ligne 10.

Pour pouvoir prouver cette propriété sur y, il faut une abstraction qui soit capable de représenter (et d'inférer) l'invariant $x = y$ en tête de boucle, c'est-à-dire une *relation* entre les variables x et y. Ce n'est pas le cas du domaine des intervalles, qui est dit *non relationnel* : il représente ces deux variables dans deux intervalles différents sans corrélation entre eux.

Remarque 2.1.
Si l'élargissement mène inévitablement à des pertes de précision, il n'est pas ici responsable de l'impossibilité de prouver la propriété sur y. En effet, même en appliquant des techniques d'amélioration de la précision comme l'élargissement retardé (après 1000 itérations) ou le rétrécissement, l'invariant le plus précis possible qu'on puisse obtenir à l'aide des intervalles est $x = [1001, 1001] \wedge y = [1, +\infty[$.

S'il permet de borner supérieurement x (à l'instar de ce qui a été observé pour l'analyse du programme 2.2), cet invariant n'apporte aucune information supplémentaire sur y : la perte de précision est irrémédiable et intrinsèquement liée au domaine choisi. $\triangleleft$

2.4.1 Le domaine des polyèdres

Un premier exemple de domaine relationnel est celui des *polyèdres* [CH78] : il représente un ensemble d'environnements mémoire sur des variables $x_1, x_2, \dots, x_n$ par une conjonction de relations de la forme :

$$\alpha_1 x_1 + \alpha_2 x_2 + \dots + \alpha_n x_n \geq \beta$$

Les coefficients α et β sont donc les paramètres de contraintes linéaires respectées par les valeurs des variables, dont la conjonction définit un polyèdre convexe clos au sens géométrique (on notera que ces polyèdres ne sont pas nécessairement bornés). Le domaine inférera ces coefficients lors de l'analyse, déterminant ainsi une sur-approximation des valeurs possibles des variables, formulée comme un ensemble de contraintes qu'elles respectent.

Nous ne rentrerons pas ici dans les détails du calcul des opérateurs abstraits dans le domaine des polyèdres, car il fait intervenir des algorithmes géométriques sensiblement plus complexes que le calcul des intervalles présenté dans les sections précédentes [Che68, CH78, LV92, Sch98, BHRZ03, MFK+16, Mar17]. Nous nous contenterons d'indiquer que ce domaine permet, sur le programme 2.3, d'inférer entre autres la relation $x - y = 0$ en entrée de boucle. Une analyse sans élargissement permettra donc d'en déduire, après stabilisation et sortie de la boucle, $y = 1001$ à la fin de l'exécution du programme : en effet, la condition d'entrée dans la boucle permet maintenant également de filtrer la valeur de y puisqu'on la sait identique à celle de x. La propriété de la ligne 10 est alors vérifiée.

Quant à l'élargissement dans ce domaine relationnel, il fonctionne sur le même principe général que pour les intervalles : les contraintes stables sont gardées, les autres sont projetées à l'infini, c'est-à-dire oubliées. En l'occurrence, la contrainte $x = y$ serait maintenue à la fin du programme, mais les contraintes qui bornent x subiraient le même traitement que dans l'analyse non relationnelle. Ainsi, si l'élargissement permet dans tous les cas de borner inférieurement y (à l'aide de la combinaison de la contrainte $x = y$ et de la borne inférieure de x inférée par sortie de la boucle), il serait nécessaire d'utiliser par exemple un rétrécissement pour le borner supérieurement (de même qu'il fallait l'utiliser pour borner supérieurement x à l'aide des intervalles).

2.4.2 Le domaine des octogones

Un autre exemple de domaine permettant d'inférer des relations entre les variables est celui des *octogones* [Min06, Jou16]. Ce domaine représente les ensembles d'états concrets sur les variables $x_1, x_2, \dots, x_n$ par une conjonction de relations de la forme (i et j n'étant pas nécessairement différents) :

$$\pm x_i \pm x_j \leq c$$

La forme de ces inéquations appelle plusieurs remarques.

Tout d'abord, ces relations ne portent que sur deux variables au plus : le domaine des octogones est parfois dit *faiblement relationnel*. Toutefois, des combinaisons linéaires de ces relations peuvent montrer que ce domaine est aussi capable d'encoder certaines relations faisant intervenir plus de deux variables.

Ensuite, pour un état de programme composé de n variables, on a au plus $\binom{n}{2} \times 4 + n \times 2$, soit $2n^2$ relations de ce type, chacune étant représentée simplement par son coefficient c. Les octogones ont donc une représentation bornée, contrairement aux polyèdres qui pouvaient décrire un nombre arbitrairement grand d'inégalités. Ils permettront ainsi des calculs en général plus rapides.

Enfin, on pourra remarquer que la définition des polyèdres contient en particulier les octogones : autrement dit, les octogones correspondent à une conjonction de relations autorisées pour les polyèdres, sur lesquelles on a simplement appliqué des contraintes supplémentaires. On dit que les octogones sont une *abstraction* des polyèdres. En règle générale, l'analyse obtenue sera donc moins précise. Ce n'est toutefois pas systématique : il existe des contre-exemples, notamment à cause de l'élargissement. Par exemple, considérons une suite infinie de polyèdres dont la limite est un octogone : avec le domaine des octogones, l'itération pourrait converger rapidement vers cette limite, alors que les polyèdres finiraient par élargir et sauter à l'infini.

De même, on remarquera que les intervalles sont quant à eux une abstraction des octogones : en effet, on peut représenter l'intervalle $x = [a, b]$ par les relations $x + x \leq 2b \wedge -x - x \leq -2a$.

La représentation dans des espaces géométriques permet de visualiser cette relation entre les domaines numériques. Ainsi, en figure 2.4, nous avons représenté les valeurs abstraites les plus précises possibles pour un ensemble de points concrets en deux dimensions (ou, de façon équivalente, un ensemble d'états de programme à deux variables). On constate que ces abstractions décrivent également des états additionnels qui ne font pas partie des états concrets voulus : il s'agit d'une perte de précision qui peut mener à des fausses alarmes si ces états supplémentaires ne vérifient pas les propriétés recherchées. On constate également que le polyèdre est bien inclus dans l'octogone, qui est lui-même inclus dans le produit cartésien des intervalles ; ce qui signifie que les abstractions moins précises décrivent des états supplémentaires qui ne font pas partie des abstractions plus fines.

Le domaine des octogones permet, tout comme le domaine des polyèdres, de prouver la propriété exprimée à la ligne 10 du programme 2.3. Il représenterait donc sans doute un bon choix ici.

2.4.3 Performances des domaines numériques

La figure 2.4 permet de visualiser la précision relative des polyèdres, octogones et intervalles. Un autre critère entre en jeu pour choisir un domaine numérique adapté : ses performances en temps et mémoire. Ainsi, les complexités des opérations sur un domaine numérique de dimension n (donc qui abstrait un élément concret à n variables $x_1, \dots, x_n$) sont données par le tableau suivant, qui récapitule également la forme des invariants inférés :

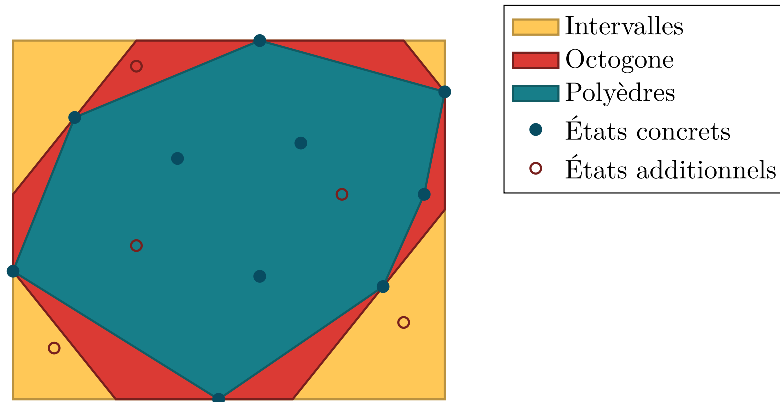


FIGURE 2.4 – Comparaison de différents domaines numériques.

Domaine	Invariants	Complexité mémoire	Complexité temps
Intervalles	$x_i \in [a, b]$	$\mathrm{O}(n)$	$\mathrm{O}(n)$
Octogones	$\pm x_i \pm x_i \leq c$	$\mathrm{O}(n^2)$	$\mathrm{O}(n^3)$
Polyèdres	$\sum_i \alpha_i x_i \geq \beta$	∞, *exponentielle en pratique*	

Remarque 2.2.
Le nombre de contraintes qui composent un polyèdre est non borné, ce qui explique le coût a priori lui-même non borné des opérations. Toutefois, on observe en pratique qu'il est exponentiel [NQ10]. De plus, tant qu'on n'effectue que des opérations linéaires (contraintes linéaires, projections, enveloppes convexes…), on peut borner le nombre de contraintes du polyèdre obtenu [Mon10]. Dans tous les cas, l'élargissement garantit la terminaison et limite la taille de la représentation. ◁

Ainsi, alors que les intervalles passeront à l'échelle sur des dizaines de milliers de variables, les octogones en accepteront une centaine et les polyèdres seront limités à une dizaine. Le choix du domaine numérique doit donc se faire suivant un compromis entre coût et précision.

Remarque 2.3.
Il existe de nombreux autres domaines numériques, permettant d'exprimer autant de formes d'invariants, que nous n'avons pas utilisés dans le cadre de cette thèse et que nous ne développerons donc pas.

Il est également possible (et souvent souhaitable) d'assembler plusieurs domaines abstraits afin de combiner les informations qu'ils contiennent, par exemple à l'aide d'une opération appelée produit réduit [CC77a], que nous développerons au chapitre 13 (l'amélioration qu'il décrit étant la seule utilisation de cette technique dans le cadre de cette thèse). ◁

2.5 Booléens et types finis

Lorsqu'un programme fait intervenir des variables booléennes, qui peuvent prendre les valeurs `true` (vrai) ou `false` (faux), il est possible de les encoder comme des entiers dans un domaine numérique comme ceux des sections précédentes, en les y intégrant comme des variables entières où `true` est représenté par 1 et `false` par 0. Cependant, cette représentation n'est pas sans inconvénient : à moins de transformer le domaine et ses opérateurs (notamment l'élargissement)

pour le rendre conscient des booléens qu'il manipule, elle mène vite à des pertes de précision conséquentes.

Il est donc préférable de représenter directement les booléens dans un domaine qui leur est propre. On peut utiliser à cette fin les *diagrammes de décision binaires* (souvent abrégés BDD, pour *binary decision diagrams*).

Cette approche consiste à représenter un état formé de n variables booléennes $x_1, \ldots, x_n$ par la formule logique caractérisant la valeur de chacune de ces variables. Considérons par exemple l'état obtenu après l'exécution du programme suivant :

```
1 x1 = false;
2 x2 = true;
3 x3 = true;
```

Il correspond à la formule logique suivante :

$$\neg x_1 \wedge x_2 \wedge x_3$$

Pour représenter plusieurs états, on utilisera comme formule caractéristique la disjonction des formules caractéristiques de chaque état.

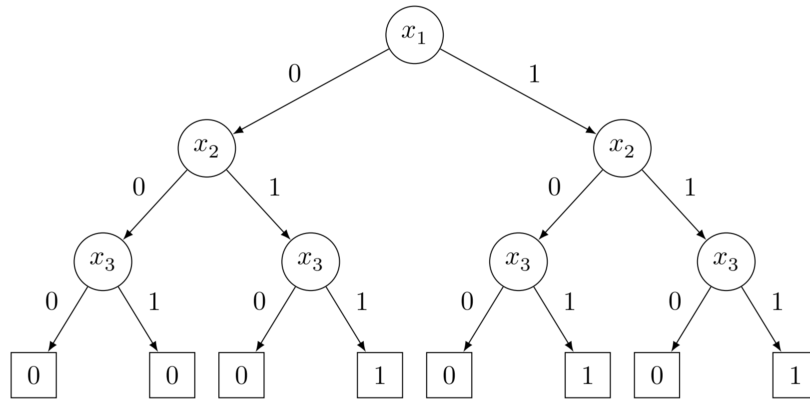
Pour représenter ces formules dans un domaine abstrait, on passe par la structure d'arbre binaire de décision ou arbre de Shannon. Un arbre binaire de décision a ses nœuds étiquetés par une variable de la formule, et la même variable est utilisée pour tous les nœuds d'une même profondeur. Les branches sont étiquetées par 0 ou 1. Les feuilles sont également étiquetées par 0 ou 1, et l'étiquette de chaque feuille correspond à la validité de la formule lorsque les variables prennent les valeurs correspondant aux branches du chemin suivi pour arriver à ladite feuille. La figure 2.5a présente un exemple d'arbre de décision pour une formule logique quelconque.

Tel quel, l'arbre de décision binaire est une structure très peu compacte, qui demande de stocker de nombreux nœuds : $2^{N+1} - 1$ pour un ensemble d'états à N variables booléennes. Les *diagrammes de décision binaire* [Lee59, Bry86] sont une version compressée de ces arbres de décision, qui cherchent à partager de façon maximale des nœuds équivalents (c'est-à-dire à partir desquels on obtient les mêmes valeurs de feuilles pour les mêmes chemins). Les nœuds ayant deux fils identiques sont également supprimés : la valeur de la variable correspondante ne change pas la valeur de vérité de la formule représentée. La figure 2.5b représente un diagramme de décision binaire équivalent à l'arbre de décision binaire de la figure 2.5a. Comme dans le cas de l'arbre, pour connaître la valeur de vérité associée à une valuation des variables, on part de la racine x_1 et, à chaque nœud rencontré, on emprunte l'arête étiquetée par la valeur de la variable correspondante jusqu'à arriver sur la feuille indiquant la valeur de la formule. Contrairement à l'arbre, ces chemins ne passent pas nécessairement par un nœud pour chaque variable : par exemple, pour obtenir 0 à partir de la valuation $\neg x_1 \wedge \neg x_2 \wedge x_3$, on ne passe pas par x_3.

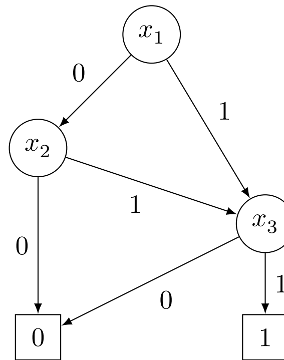
On notera qu'un arbre de décision binaire est lui-même un diagramme de décision binaire de la formule correspondante : la représentation par un diagramme n'est généralement pas unique (en particulier, l'ordre des variables compte), et pour obtenir la plus compacte possible, il existe des algorithmes que nous ne détaillerons pas [Bry86].

Cette représentation d'états booléens est *concrète* : elle permet de représenter exactement un ensemble d'états composés de variables booléennes, sans aucune perte de précision due à l'abstraction. Cela ne compromet pas la calculabilité de l'analyse en raison du nombre fini (2^N) d'états (et donc de diagrammes) possibles impliquant N variables booléennes.

Pour la même raison, on peut utiliser la même représentation pour encoder des variables dont le type est *fini*, c'est-à-dire habité par un nombre fini de valeurs. On peut ainsi représenter une variable d'un type à N valeurs par un nœud à N branches (mais le diagramme n'est alors plus binaire), ou de l'encoder par $\lceil \log_2 N \rceil$ nœuds, chacun représentant un *bit* de la variable.

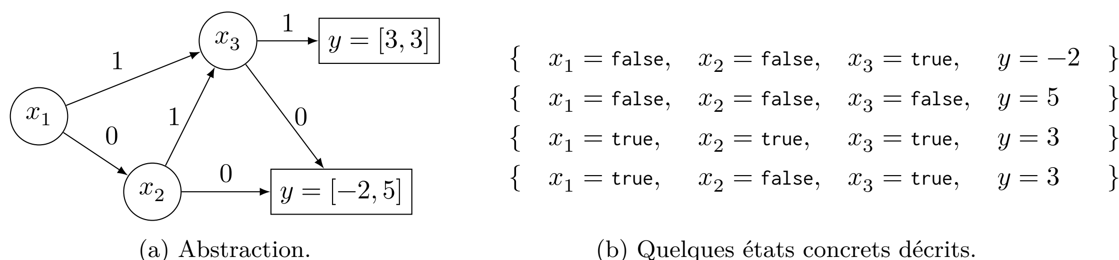


(a) L'arbre de décision binaire.



(b) Un diagramme de décision binaire.

FIGURE 2.5 – Arbre et diagramme de décision binaires pour la formule $(x_1 \vee x_2) \wedge x_3$.



(a) Abstraction.

(b) Quelques états concrets décrits.

FIGURE 2.6 – Exemple d'une abstraction logico-numérique.

Enfin, lorsqu'on souhaite abstraire des états faisant intervenir à la fois des variables booléennes (ou de type fini) et des variables numériques, on peut utiliser cette même représentation pour les variables booléennes, en remplaçant les 0 et 1 étiquetant les feuilles du diagramme par une abstraction de la partie numérique des états dont la partie booléenne correspond au chemin menant à la feuille en question [Jea03, SJ11]. Au lieu d'un unique élément numérique décrivant tous les états concrets possibles, on obtient alors une disjonction de plusieurs éléments numériques, chacun associé à une valuation donnée des variables booléennes (avec partage des éléments en commun). Il s'agit d'un partitionnement du domaine par la valeur de ces variables booléennes, abstraction que nous présenterons plus en détail dans le chapitre 6.2, partie II. La figure 2.6 présente un exemple d'un tel élément abstrait, utilisant le domaine des intervalles aux feuilles, ainsi que quelques éléments concrets qu'il décrit.

2.6 Structure de treillis

Après avoir introduit de manière informelle divers aspects de l'interprétation abstraite dans les sections précédentes, nous allons maintenant en donner une présentation formelle. Pour cela, nous aurons besoin de la structure mathématique de treillis, sur laquelle nous ferons un bref rappel ici.

2.6.1 Ensembles ordonnés

Nous rappellerons d'abord quelques définitions liées aux ensembles ordonnés, ce qui nous permettra entre autres de fixer les notations correspondantes.

Définition 1 (Majorant, minorant, bornes).
Soit un ensemble E muni d'une relation d'ordre $\sqsubseteq$. Soient $a, b, m \in E$.

- m est un majorant de a et b si et seulement si $a \sqsubseteq m \wedge b \sqsubseteq m$.
- m est un minorant de a et b si et seulement si $m \sqsubseteq a \wedge m \sqsubseteq b$.
- m est une borne supérieure de a et b si et seulement si m est un majorant de a et b et, pour tout majorant M de a et b, $m \sqsubseteq M$.
- m est une borne inférieure de a et b si et seulement si m est un minorant de a et b et, pour tout minorant M de a et b, $M \sqsubseteq m$. $\lozenge$

Théorème 2.1 (Unicité des bornes).
Soit un ensemble E muni d'une relation d'ordre $\sqsubseteq$, soient $a, b \in E$. La borne inférieure de a et b, si elle existe, est unique et on la note $a \sqcap b$. De même, la borne supérieure de a et b, si elle existe, est unique et on la note $a \sqcup b$.

Remarque 2.4.
La notion de borne supérieure $\sqcup S$ s'entend également pour un sous-ensemble quelconque $S \subseteq E$: $\forall x \in S, x \sqsubseteq \sqcup S$, et $\forall M \in E, (\forall x \in S, x \sqsubseteq M) \implies \sqcup S \sqsubseteq M$. $\lhd$

Définition 2 (Chaîne).
Soit un ensemble ordonné $(E, \sqsubseteq)$. Un sous-ensemble $C \subseteq E$ est une chaîne si et seulement si C est totalement ordonné par $\sqsubseteq$:

$$\forall x, y \in C, x \sqsubseteq y \vee y \sqsubseteq x$$

Définition 3 (Ordre partiel complet).
Un ordre partiel complet (*complete partial order*, abrégé **CPO**) est un ensemble partiellement (ou totalement) ordonné dont toutes les chaînes ont une borne supérieure. $\Diamond$

Définition 4 (Conditions de chaîne).
Un ensemble ordonné $(E, \sqsubseteq)$ satisfait la condition de chaîne ascendante (**ACC**) si toute suite croissante (c_n) d'éléments de E est stationnaire : $\exists N \in \mathbb{N}, \forall n \geq N, c_n = c_N$.

On définit de même la condition de chaîne descendante (**DCC**). $\Diamond$

2.6.2 Treillis

Définition 5 (Treillis).
Un treillis $(E, \sqsubseteq, \sqcup, \sqcap)$ est un ensemble ordonné $(E, \sqsubseteq)$ pour lequel toute paire d'éléments (a, b) admet une borne inférieure $a \sqcap b$ et une borne supérieure $a \sqcup b$. $\Diamond$

Nous avons déjà donné des exemples de treillis, en représentant visuellement la relation d'ordre sous-jacente à l'aide des diagrammes de Hasse : le treillis des signes en figure 2.1 et le treillis des intervalles en figure 2.3.

Définition 6 (Treillis complet).
Un treillis E est complet si et seulement si tout sous-ensemble $S \subseteq E$ admet une borne supérieure $\sqcup S$.

Proposition 2.1.

1. *De façon équivalente à la définition 6, un treillis E est complet si et seulement si tout sous-ensemble $S \subseteq E$ admet une borne inférieure $\sqcap S$.*

2. *Un treillis complet admet un plus petit élément $\bot$ et un plus grand élément $\top$.*

Exemple 2.1.
Pour tout ensemble S, l'ensemble $\wp(S)$ des parties de S est un treillis pour l'inclusion ensembliste $\subseteq$: la borne inférieure est l'intersection $\cap$ et la borne supérieure l'union $\cup$. Il s'agit de plus d'un treillis complet. Le plus petit élément est l'ensemble vide $\varnothing$ et le plus grand élément est S lui-même.

En prenant pour S l'ensemble des états d'un programme, nous utiliserons fréquemment cette structure par la suite pour justifier de résultats sur leur analyse statique. $\bigcirc$

Définition 7.
Soient deux treillis $(E_1, \sqsubseteq_1, \sqcup_1, \bot_1)$ et $(E_2, \sqsubseteq_2, \sqcup_2, \bot_2)$ et une fonction $f : E_1 \to E_2$.
f est :

- Croissante si elle préserve l'ordre : $\forall x, y \in E_1, x \sqsubseteq_1 y \implies f(x) \sqsubseteq_2 f(y)$.
- Si E_1 et E_2 sont des CPO : continue si pour toute chaîne $C_1 \subseteq E_1, C_2 \triangleq \{f(x) \mid x \in C_1\}$ est une chaîne de E_2 et $f(\sqcup_1 C_1) = \sqcup_2 C_2$.

Soit un ensemble ordonné $(E, \sqsubseteq)$ et une fonction $f : E \to E$. f est :

- Extensive si $\forall x \in E, x \sqsubseteq f(x)$
- Réductrice si $\forall x \in E, f(x) \sqsubseteq x$ ♢

Ces définitions nous seront également utiles par la suite : elles interviendront dans plusieurs définitions et théorèmes en raison de leur aspect fondamental.

2.7 Points fixes

Après avoir rappelé dans cette section la définition des points fixes ainsi que quelques résultats qui les concernent, nous montrerons en quoi ils permettent de formaliser la sémantique des programmes.

2.7.1 Définitions et existence

Définition 8 (Point fixe).
Un point fixe d'une fonction $f : E \to E$, pour un ensemble E quelconque, est un élément $x \in E$ tel que $f(x) = x$.
On note :

- $\text{fp}(f) \triangleq \{x \in E \mid f(x) = x\}$ l'ensemble des points fixes de f.
- $\text{lfp}_x(f) \triangleq \min_{\sqsubseteq} \{y \in \text{fp}(f) \mid x \sqsubseteq y\}$ le plus petit point fixe de f supérieur à x, s'il existe.
- $\text{lfp}(f) \triangleq \text{lfp}_{\bot}(f)$ le plus petit point fixe de f, s'il existe. ♢

Définition 9 (Post-point fixe, Pré-point fixe).
Soient un ensemble ordonné $(E, \sqsubseteq)$, une fonction $f : E \to E$ et un élément $x \in E$.

- x est un pré-point fixe de f si et seulement si $x \sqsubseteq f(x)$
- x est un post-point fixe de f si et seulement si $f(x) \sqsubseteq x$ ♢

Plusieurs théorèmes permettent, sous certaines conditions, de montrer qu'une fonction f admet un point fixe et de donner des caractéristiques de $\text{fp}(f)$. Nous en citerons deux.

Théorème 2.2 (Knaster-Tarski).
Si une fonction $f : E \to E$ est croissante sur un treillis complet E, alors $fp(f)$ est un treillis complet non vide.

Théorème 2.3 (Inspiré de Kleene).
Si $f : E \to E$ est une fonction continue sur un ordre partiel complet E, pour tout $a \in E$, si $a \sqsubseteq f(a)$, alors $lfp_a f$ existe.

Il existe encore de nombreux théorèmes permettant d'assurer l'existence d'un point fixe sous des conditions différentes, voire d'en donner une définition constructive.

```
1  (ℓ0) x = 1; (ℓ1)
2
3  while (ℓ2) (x < 10) {
4      (ℓ3) x = x + 1; (ℓ4)
5  } (ℓ5)
```

PROGRAMME 2.4 – Une simple boucle aux points de contrôle explicités.

$$\begin{cases} \mathcal{X}_0 = \mathcal{E} \\ \mathcal{X}_1 = [\![x \leftarrow 1]\!]\mathcal{X}_0 \\ \mathcal{X}_2 = \mathcal{X}_1 \cup \mathcal{X}_4 \\ \mathcal{X}_3 = [\![x < 10]\!]\mathcal{X}_2 \\ \mathcal{X}_4 = [\![x \leftarrow x + 1]\!]\mathcal{X}_3 \\ \mathcal{X}_5 = [\![x \geq 10]\!]\mathcal{X}_2 \end{cases}$$

FIGURE 2.7 – Système d'équations décrivant le programme 2.4.

2.7.2 Sémantique sous forme de point fixe

Avant de présenter leur application à la sémantique des programmes, remarquons que les points fixes permettent d'exprimer les solutions des systèmes d'équations mutuellement récursives. Ainsi, les solutions du système suivant :

$$\begin{cases} x_1 = f(x_1, x_2) \\ x_2 = g(x_1, x_2) \end{cases}$$

Sont les points fixes de la fonction $\vec{F}$ suivante :

$$\vec{F}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} f(x_1, x_2) \\ g(x_1, x_2) \end{pmatrix}$$

Par ailleurs, le comportement des programmes peut être décrit à l'aide de ce type de système. Considérons par exemple le programme 2.4 : il s'agit d'une simple boucle, pour laquelle nous avons explicité les *points de contrôle*, c'est-à-dire l'ensemble des positions possibles dans le code du programme considéré.

L'approche équationnelle de la sémantique d'un tel programme consiste à définir, pour chaque point de contrôle (ℓ_i), l'ensemble $\mathcal{X}_i$ des états accessibles. Ces états sont liés entre eux par des équations correspondant à la structure du programme : celles du programme 2.4 sont décrites en figure 2.7. $\mathcal{E}$ désigne l'ensemble des états initiaux. Ces équations font intervenir des *opérateurs*, notés $[\![op]\!]$, qui sont des fonctions sur les ensembles d'états définissant les opérations de base du langage. Ainsi, $[\![x \leftarrow e]\!]\mathcal{X}$ transforme les états de $\mathcal{X}$ en affectant dans chacun la valeur de l'expression e à x, et $[\![a \leq b]\!]\mathcal{X}$ ne garde que les états de $\mathcal{X}$ vérifiant la condition $a \leq b$.

Grâce à ces opérateurs, nous pouvons construire le système d'équations décrivant un programme par induction sur sa structure : par exemple, dans le cas général, une boucle est de cette forme :

```
(ℓ0) while (ℓ1) c { (ℓ2) s (ℓ3) } (ℓ4)
```

Elle génère alors les équations suivantes :

$$\begin{cases} \mathcal{X}_1 = \mathcal{X}_0 \cup \mathcal{X}_3 \\ \mathcal{X}_2 = [\![c]\!]\mathcal{X}_1 \\ \mathcal{X}_3 = [\![s]\!]\mathcal{X}_2 \\ \mathcal{X}_4 = [\![\neg c]\!]\mathcal{X}_1 \end{cases}$$

Remarquons que la structure du système correspond au graphe de contrôle du programme, où les nœuds sont les points de contrôle (ℓ_i) et les arêtes reliant deux nœuds sont étiquetées par l'opérateur correspondant. De la même façon qu'on construit ce graphe, il est alors possible d'obtenir une définition des opérateurs et du système à l'aide d'une sémantique opérationnelle comme celle du chapitre 1.

On obtient finalement une description de la sémantique d'un programme sous la forme d'un système d'équations $\vec{\mathcal{X}} = \vec{F}(\vec{\mathcal{X}})$. Les solutions du système sont alors par définition des points fixes de $\vec{F}$. Parmi ceux-ci, on cherche à calculer le plus petit, $\operatorname{lfp} \vec{F}$: il correspond exactement aux *états accessibles* du programme. L'ensemble des ensembles d'états étant un treillis complet, le théorème 2.2 (de Tarski) nous garantit l'existence de ce plus petit point fixe. Toutefois, l'ensemble des états accessibles n'est pas calculable dans le cas général : on cherche donc, pour être sûr, à en calculer une sur-approximation $I \supseteq \operatorname{lfp} \vec{F}$, c'est-à-dire un *invariant*.

Au fur et à mesure des sections suivantes, nous verrons comment l'interprétation abstraite nous permet de calculer de tels invariants.

2.8 Correspondances de Galois

Les correspondances de Galois formalisent la notion d'abstraction : elles permettent de relier deux ensembles, le domaine concret et le domaine abstrait, avec des fonctions d'abstraction et de concrétisation. Les propriétés de ces fonctions symbolisent la représentation des éléments concrets par les éléments abstraits.

2.8.1 Définition

Définition 10 (Correspondance de Galois).
Soient deux ensembles ordonnés $(C, \sqsubseteq)$ et $(A, \preccurlyeq)$. Le couple $(\alpha : C \to A, \gamma : A \to C)$ est une correspondance de Galois si et seulement si :

$$\forall a \in A, c \in C, \alpha(c) \preccurlyeq a \iff c \sqsubseteq \gamma(a)$$

On la note $(C, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \preccurlyeq)$. On omet parfois les relations d'ordre pour noter $C \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} A$.
α est l'adjoint supérieur ou abstraction. A est le domaine abstrait.
γ est l'adjoint inférieur ou concrétisation. C est le domaine concret. ♢

Le diagramme de la figure 2.8 représente visuellement une correspondance de Galois.

La proposition 2.2 énonce quelques résultats importants. Nous y ferons référence dans les sections qui suivront.

Proposition 2.2 (Propriétés des correspondances de Galois).
Soit une correspondance de Galois $(C, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \preccurlyeq)$. *Les propriétés suivantes sont vérifiées :*

1. $\gamma \circ \alpha$ *est extensive :* $\forall c \in C, c \sqsubseteq \gamma(\alpha(c))$

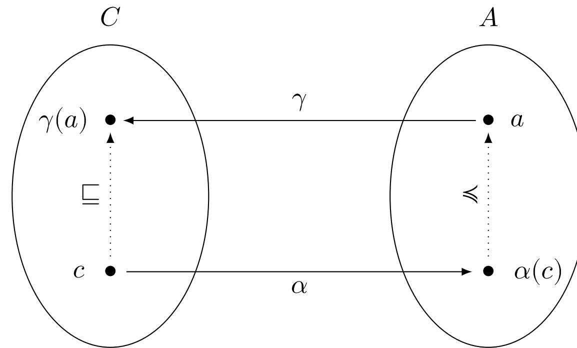


FIGURE 2.8 – Diagramme d'une correspondance de Galois.

2. $\alpha \circ \gamma$ *est réductrice :* $\forall a \in A, \alpha(\gamma(a)) \preccurlyeq a$

3. α *est croissante*

4. γ *est croissante*

5. $\gamma \circ \alpha \circ \gamma = \gamma$

6. $\alpha \circ \gamma \circ \alpha = \alpha$

7. $\alpha \circ \gamma$ *est idempotente :* $\alpha \circ \gamma \circ \alpha \circ \gamma = \alpha \circ \gamma$

8. $\gamma \circ \alpha$ *est idempotente*

2.8.2 Mise en pratique : formalisation du domaine des intervalles

La mise en pratique de ces définitions et propriétés formelles à travers l'exemple des intervalles permettra d'en donner une explication intuitive plus facile à comprendre.

Le domaine des intervalles permet, nous l'avons vu, de représenter de façon abstraite des ensembles d'entiers (par exemple, les valeurs possibles d'une variable). Ainsi, le domaine concret C est ici l'ensemble des ensembles d'entiers (et l'ordre partiel concret l'inclusion ensembliste) :

$$C \triangleq \wp(\mathbb{Z})$$

Le domaine abstrait est l'ensemble des intervalles exprimés comme un couple de bornes :

$$\overline{\mathbb{Z}} \triangleq \mathbb{Z} \cup \{-\infty, +\infty\}$$
$$A \triangleq \overline{\mathbb{Z}} \times \overline{\mathbb{Z}}$$

La concrétisation γ permet de préciser l'ensemble d'entiers décrits par un intervalle donné :

$$\gamma(a, b) = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$$

L'abstraction α quant à elle nous donne la manière la plus précise de représenter un ensemble d'entiers :

$$\alpha(E) = (\min(E), \max(E))$$

La relation d'ordre $\preccurlyeq$ correspond ici à l'inclusion ensembliste des intervalles :

$$(a, b) \preccurlyeq (c, d) \triangleq a \geq c \wedge b \leq d$$

Nous avons vu précédemment que cette relation traduit également la « précision » de l'information encodée dans un élément abstrait. $X \preccurlyeq Y$ signifie que X porte une information plus précise que Y. On peut alors comprendre la croissance de α et γ comme l'expression formelle de cette notion de précision : Y, moins précis que X, représente un ensemble d'entiers plus grand, c'est-à-dire davantage de valeurs potentielles pour la variable ainsi abstraite.

Le même point de vue permet de comprendre l'extensivité de $\gamma \circ \alpha$ comme la « perte d'information » induite par l'abstraction : lorsqu'on abstrait un élément concret S, on obtient un élément abstrait X qui, après concrétisation, représente en réalité un ensemble plus grand que S. La différence entre $\gamma(X)$ et S correspond à l'information qui a été perdue : X ne contient comme information que le minimum et le maximum de S. $\gamma(X)$ contient donc comme valeurs potentielles pour la variable tous les entiers compris entre ces bornes : toute information plus précise sur les entiers intermédiaires réellement possibles a été perdue.

La réductivité de $\alpha \circ \gamma$, quant à elle, exprime le fait que $\alpha(S)$ est toujours l'abstraction la plus précise possible de S dans le domaine considéré. Ainsi, si un élément abstrait X décrit l'élément concret $\gamma(X)$, alors appliquer α à $\gamma(X)$ ne fera pas perdre d'information supplémentaire : on obtient un élément au moins aussi précis que X.

En réalité, $\alpha \circ \gamma$ ne permet pas réellement de « gagner en précision » : le fait qu'on puisse obtenir un élément $Y \preccurlyeq X$ plutôt que X lui-même vient de la non-unicité de la représentation de l'intervalle vide. Ainsi, $[7, 2] \preccurlyeq [6, 4]$ d'après la définition de $\preccurlyeq$, mais ces deux intervalles représentent en réalité le même ensemble vide $\varnothing$. Or $\alpha(\varnothing) =]+\infty, -\infty[= \bigwedge A$: la réductivité est bien respectée.

2.8.3 Composition d'abstractions

Pour construire des abstractions complexes permettant d'exprimer des invariants plus expressifs, il est fréquent de les décomposer en plusieurs niveaux d'abstractions plus simples à définir qui s'enchaînent à l'aide du résultat suivant.

Théorème 2.4 (Composition d'abstractions).
Soient deux correspondances de Galois $(C, \sqsubseteq) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (I, \trianglelefteq)$ *et* $(I, \trianglelefteq) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (A, \preccurlyeq)$.

$(C, \sqsubseteq) \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} (A, \preccurlyeq)$ *est une correspondance de Galois.*

2.8.4 Plongements et isomorphismes de Galois

Pour formaliser des propriétés plus fortes sur l'abstraction et la concrétisation, on peut définir les plongements et les isomorphismes de Galois.

Lemme 2.1.
Soit une correspondance de Galois $(C, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \preccurlyeq)$.

Les propriétés suivantes sont équivalentes :

1. α *est surjective*
2. γ *est injective*
3. $\alpha \circ \gamma = id$

Définition 11.
Un couple (α, γ) vérifiant les conditions du lemme 2.1 est appelé plongement de Galois. On le note :

$$(C, \sqsubseteq) \xleftarrow[\alpha]{\gamma} \twoheadrightarrow (A, \preccurlyeq)$$

Exemple 2.2.
En définissant les intervalles comme des couples de bornes **ordonnées** (a, b) avec $a \leq b$, auxquelles on ajoute l'intervalle vide $\bot$, la correspondance de Galois présentée précédemment devient un plongement de Galois.

Dans un plongement de Galois, « tous les éléments abstraits sont utiles » : ils représentent tous un élément concret (α est surjectif) et deux éléments abstraits différents représentent deux éléments concrets distincts (γ est injective).

Remarque 2.5.
On peut former un plongement de Galois depuis toute correspondance de Galois $C \xleftarrow[\alpha]{\gamma} \rightarrow A$ en quotientant A par la relation d'équivalence $a \equiv a' \overset{\Delta}{\iff} \gamma(a) = \gamma(a')$. C'est ce qui a été réalisé dans l'exemple 2.2 en identifiant tous les intervalles vides à $\bot$. $\lhd$

Lemme 2.2.
Soit une correspondance de Galois $(C, \sqsubseteq) \xleftarrow[\alpha]{\gamma} \rightarrow (A, \preccurlyeq)$.

Les propriétés suivantes sont équivalentes :

1. α *est bijective*
2. γ *est bijective*
3. $\alpha \circ \gamma = \gamma \circ \alpha = id$

Définition 12.
Un couple (α, γ) vérifiant les conditions du lemme 2.2 est appelé isomorphisme de Galois. On le note :

$$(C, \sqsubseteq) \twoheadleftarrow\!\xleftarrow[\alpha]{\gamma} \twoheadrightarrow (A, \preccurlyeq)$$

Les abstractions des isomorphismes de Galois ne perdent aucune information. C'est l'intuition derrière l'injectivité de α : à chaque élément concret, on peut associer un élément abstrait qui lui est propre, toutes les informations présentes dans le concret étant également présentes dans l'abstrait. Elles peuvent toutefois être utiles dans le cadre de l'interprétation abstraite pour représenter sous une forme différente un certain ensemble, afin de pouvoir l'abstraire de façon plus naturelle par la suite en composant les abstractions comme mentionné à la proposition 2.4.

Par exemple, nous définirons pour analyser les programmes que nous étudions un domaine qui consiste simplement à regrouper ensemble les états partageant une caractéristique commune, afin de pouvoir définir plus facilement un traitement spécifique pour chaque groupe. Ce regroupement ne fait que réarranger différemment les mêmes états : il sera donc défini comme un isomorphisme de Galois.

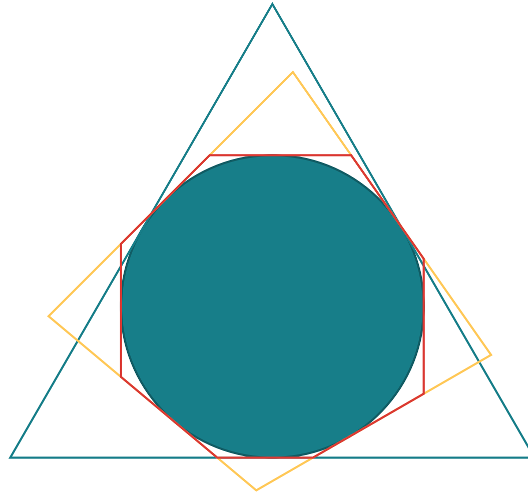


FIGURE 2.9 – Il n'existe pas de sur-approximation polyédrale optimale d'un disque.

2.8.5 Quand α n'existe pas : l'exemple des polyèdres

Les correspondances de Galois sont composées de deux fonctions, une abstraction α et une concrétisation γ. Certains domaines ne permettent cependant pas de définir une fonction d'abstraction α qui respecte les propriétés voulues : c'est par exemple le cas des polyèdres.

En effet, l'intuition derrière cette opération, que nous avons présentée dans le cas des intervalles, est la définition de l'abstraction optimale d'un élément concret donné, c'est-à-dire la façon la plus précise de le représenter dans le domaine abstrait. Or, si notre domaine concret est constitué des ensembles de points en deux dimensions (qui, pour rappel, correspondent à un état de programme à deux variables), il existe des éléments qui n'ont pas de meilleure abstraction dans le domaine des polyèdres : c'est par exemple le cas des disques, illustré en figure 2.9. Il existe en effet une infinité de sur-approximations possibles d'un disque par des polyèdres, mais aucune d'entre elles n'est plus précise que toutes les autres. Cela est dû au fait que les polyèdres ne sont pas clos par intersection infinie, tandis que les éléments d'un domaine abstrait ayant une correspondance de Galois le sont.

Lorsqu'on manipule de tels domaines, on ne définit alors que la concrétisation $\gamma : A \to C$, qui doit être croissante. On parle alors de *domaine défini par concrétisation* (par opposition aux *domaines définis par correspondance de Galois*). Comme nous le verrons dans les sections suivantes, cette définition est suffisante pour établir la sûreté des analyses qui en découlent. En revanche, certains résultats sur la précision des résultats obtenus ne seront pas applicables tels quels.

2.9 Application à l'analyse statique

Dans le cadre de l'analyse statique, les correspondances de Galois nous proposent un cadre formel permettant de calculer une sur-approximation de la sémantique d'un programme sous la forme d'une abstraction. Nous allons présenter dans cette section plusieurs propriétés utiles à cette fin.

Plus précisément, notre objectif est de calculer, dans le domaine abstrait, une abstraction sûre du point fixe de l'application successive d'opérateurs concrets ; ce point fixe concret définissant

la sémantique des programmes à analyser.

Dans toute cette section, on considérera un domaine concret $(C, \sqsubseteq)$ et un domaine abstrait $(A, \preccurlyeq)$.

2.9.1 Opérateurs abstraits

Pour calculer une abstraction sûre d'un résultat concret obtenu par application d'opérateurs de $C \to C$, nous allons définir et calculer des approximations sûres de ces opérateurs, dans $A \to A$.

Domaine défini par concrétisation.

Considérons en premier lieu le cas d'un domaine défini par une fonction de concrétisation croissante $\gamma : A \to C$. Une abstraction *sûre* d'un élément concret c est un élément abstrait qui décrit au moins toute l'information contenue dans c. Par exemple, pour le domaine des intervalles, $[0, 10]$ est une abstraction sûre de $\{1, 2, 5, 6\}$: il existe des abstractions plus précises, mais celle-ci décrit bien toutes les valeurs possibles.

Définition 13 (Abstraction sûre d'un élément concret).
Soit $c \in C$. $a \in A$ est une abstraction sûre de c si et seulement si $c \sqsubseteq \gamma(a)$. $\Diamond$

Considérons maintenant un opérateur concret $f : C \to C$. On cherche à décrire un critère de sûreté pour une abstraction $g : A \to A$ de f, c'est-à-dire une fonction du domaine abstrait permettant de calculer une approximation de f. Considérons un élément abstrait a. a décrit l'élément concret $\gamma(a)$: le résultat de l'application de l'opérateur abstrait doit donc au moins décrire le résultat de l'application de l'opérateur concret à $\gamma(a)$. Par exemple, $\lambda[a, b].] - \infty, +\infty[$ est une abstraction sûre de l'opérateur concret $\lambda X. \{x + 1 \mid x \in X\}$: le résultat d'une application de l'opérateur abstrait couvre bien tous les cas possibles résultant de l'application correspondante de l'opérateur concret.

Définition 14 (Abstraction sûre d'un opérateur concret).
Soit $f : C \to C$. $g : A \to A$ est une abstraction sûre de f si et seulement si :

$$\forall a \in A, (f \circ \gamma)(a) \sqsubseteq (\gamma \circ g)(a)$$

En remplaçant l'ordre $\sqsubseteq$ par l'égalité, on obtient un opérateur exact. Ces opérateurs n'induisent aucune perte de précision : ils calculent un élément abstrait qui décrit exactement le résultat d'une application concrète de l'opérateur. Par exemple, $\lambda[a, b].[a + 1, b + 1]$ est une abstraction exacte de $\lambda X. \{x + 1 \mid x \in X\}$. En revanche, $\lambda[a, b].[2a, 2b]$ n'est pas une abstraction exacte de $\lambda X. \{2x \mid x \in X\}$. En effet, considérons l'intervalle $[0, 1]$. Il décrit l'ensemble $\{0, 1\}$, auquel l'application de l'opérateur concret renvoie $\{0, 2\}$. L'application de l'opérateur abstrait renvoie quant à elle $[0, 2]$, qui décrit l'ensemble $\{0, 1, 2\}$: cet opérateur abstrait a mené à une perte de précision, en calculant comme valeur possible 1, alors que l'application de l'opérateur concret ne le permet pas. Pour autant, on conçoit (cette notion sera définie par la suite) qu'il n'était pas possible de donner une abstraction plus précise de cet opérateur : en effet, $[0, 2]$ est bien l'abstraction la plus petite possible de $\{0, 2\}$. Il n'existe donc pas nécessairement d'abstraction exacte d'un opérateur concret donné.

Définition 15 (Abstraction exacte d'un opérateur concret).
Soit $f : C \to C$. $g : A \to A$ est une abstraction exacte de f si et seulement si :

$$\forall a \in A, (f \circ \gamma)(a) = (\gamma \circ g)(a)$$

Correspondance de Galois et opérateurs optimaux.

Rappelons que, lorsqu'on dispose d'une correspondance de Galois $(C, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \preccurlyeq)$, la fonction d'abstraction α donne l'abstraction « optimale » d'un élément concret c, c'est-à-dire le plus petit élément abstrait a possible qui décrive c ($c \sqsubseteq \gamma(a)$).

α nous permet également de définir l'abstraction optimale d'un opérateur concret $f : C \to C$.

Définition 16 (Abstraction optimale d'un opérateur concret).
Soit une correspondance de Galois $(C, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\twoheadleftarrow\!\!\twoheadrightarrow}} (A, \preccurlyeq)$. Soit $f : C \to C$. L'abstraction optimale g de f est donnée par la formule suivante :

$$g = \alpha \circ f \circ \gamma$$

Ainsi, sur les intervalles, $\lambda[a, b].[2a, 2b]$ est l'abstraction optimale de $\lambda X.\{2x \mid x \in X\}$.

Composition d'opérateurs abstraits.

L'analyse statique nous amène à calculer le résultat de la composition de plusieurs opérateurs concrets successifs. Il est alors utile de disposer des résultats suivants sur la composition des opérateurs abstraits correspondants.

Théorème 2.5 (Composition d'opérateurs abstraits).
Soient deux opérateurs concrets f et f' et deux abstractions sûres de ces opérateurs g et g'.
Alors :

- *Si f est croissante, alors $g \circ g'$ est une abstraction sûre de $f \circ f'$.*
- *Si g et g' sont des abstractions exactes, alors $g \circ g'$ est une abstraction exacte.*

Remarque 2.6.
La composition d'abstractions optimales d'opérateurs n'est pas nécessairement une abstraction optimale de la composition dans le concret des opérateurs.

Considérons par exemple les opérateurs suivants :

$$f = \lambda X.\{x \in X \mid x \leq 1\}$$
$$g = \lambda X.\{2x \mid x \in X\}$$

Leurs abstractions optimales sont :

$$f^\sharp = \lambda[a, b].[a, \min(b, 1)]$$
$$g^\sharp = \lambda[a, b].[2a, 2b]$$

Leur composition donne le résultat suivant :

$$(f^\sharp \circ g^\sharp)([0, 1]) = f^\sharp([0, 2]) = [0, 1]$$

Or l'abstraction optimale de la composition concrète calculerait :

$$\begin{aligned} (\alpha \circ f \circ g \circ \gamma)([0, 1]) &= \\ (\alpha \circ f \circ g)(\{0, 1\}) &= \\ (\alpha \circ f)(\{0, 2\}) &= \\ \alpha(\{0\}) &= \\ [0, 0] & \end{aligned}$$

2.9.2 Abstraction des points fixes

Dans la section 2.7, nous avons vu comment la sémantique des programmes pouvait s'écrire comme le point fixe d'une fonction faisant intervenir les différents opérateurs correspondant aux transitions du programme. Nous allons maintenant montrer comment calculer une abstraction de ce point fixe.

Théorème 2.6 (Approximation de point fixe).
Soient :

- *Un treillis complet* $(C, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$
- *Un ensemble ordonné* $(A, \preccurlyeq)$
- *Une fonction* $\gamma : A \to C$ *croissante*
- *Une fonction* $f : C \to C$ *croissante*
- *Une abstraction sûre* $f^\sharp$ *de* f *:* $\forall x^\sharp \in A, (f \circ \gamma)(x^\sharp) \sqsubseteq (\gamma \circ f^\sharp)(x^\sharp)$
- *Un post-point fixe* $a^\sharp$ *de* $f^\sharp$ *:* $f^\sharp(a^\sharp) \preccurlyeq a^\sharp$

Alors $a^\sharp$ *est une abstraction sûre de lfp*(f) *:*

$$\mathit{lfp}(f) \sqsubseteq \gamma(a^\sharp)$$

Il existe encore une fois d'autres théorèmes d'approximation de points fixes, avec des conditions et des résultats plus ou moins forts. Le théorème 2.6 a l'avantage de ne pas exiger un point fixe abstrait. Lorsque calculer ce point fixe de $f^\sharp$ est difficile, voire impossible (le domaine abstrait ne respectant pas les conditions pour l'existence d'un point fixe), il suffit alors d'en déterminer un post-point fixe $a^\sharp$ pour obtenir une abstraction sûre du point fixe concret qui nous intéresse.

Toutefois, le théorème ne précise pas comment ce post-point fixe de $f^\sharp$ peut être calculé. La section suivante montrera alors un moyen de le construire à l'aide de l'élargissement.

2.9.3 Élargissement

Pour analyser un programme, on cherchera comme expliqué en section 2.7 à obtenir une abstraction sûre du plus petit point fixe de la fonction $\vec{F}$ correspondant au système d'équation qui décrit la sémantique du programme.

Le théorème 2.6 nous donne un moyen de calculer une telle abstraction, en calculant un post-point fixe d'une abstraction sûre $\vec{F}^\sharp$ de $\vec{F}$, elle-même obtenue grâce aux résultats de la section 2.9.1. On notera que $\vec{F}$ est généralement croissante dans le cas de l'analyse statique : si l'ensemble d'états de départ possibles augmente, l'ensemble des états d'arrivée après une transition ne peut pas diminuer.

Cependant, cette condition de chaîne ascendante est très forte, et n'est pas vérifiée dans le cas général : il existe par exemple une infinité de chaînes ascendantes infinies dans le domaine des intervalles. Ainsi :

$$[0,0] \sqsubseteq^\sharp [0,1] \sqsubseteq^\sharp [0,2] \sqsubseteq^\sharp [0,3] \sqsubseteq^\sharp \cdots$$

Nous avons présenté en section 2.3 un outil permettant de résoudre ce problème : il s'agit de l'élargissement ∇ [CC77a]. Nous allons maintenant définir formellement cette notion.

Définition 17 (Opérateur d'élargissement).
Soit un domaine abstrait $(\mathscr{D}^\sharp, \sqsubseteq^\sharp)$ défini par concrétisation γ vers un domaine concret $(\mathscr{D}, \sqsubseteq)$. Un opérateur $\nabla : \mathscr{D}^\sharp \times \mathscr{D}^\sharp \to \mathscr{D}^\sharp$ est un élargissement si et seulement si :

- ∇ calcule un majorant :

$$\forall x^\sharp, y^\sharp \in \mathscr{D}^\sharp, x^\sharp \sqsubseteq^\sharp x^\sharp \nabla y^\sharp \;\wedge\; y^\sharp \sqsubseteq^\sharp x^\sharp \nabla y^\sharp$$

- ∇ assure la terminaison : pour toute suite $(y^\sharp_n)_{n\in\mathbb{N}}$, la suite $(x^\sharp_n)_{n\in\mathbb{N}}$ définie par :

$$\begin{cases} x^\sharp_0 \triangleq y^\sharp_0 \\ x^\sharp_{n+1} \triangleq x^\sharp_i \;\nabla\; y^\sharp_{i+1} \end{cases}$$

se stabilise en temps fini :

$$\exists N \in \mathbb{N}, x^\sharp_{N+1} = x^\sharp_N$$

Les propriétés de l'élargissement vont nous permettre de pouvoir approximer itérativement le point fixe donnant la solution du système suivant :

$$\begin{cases} x^\sharp_1 = \Phi^\sharp_1(x_1, \dots, x_n) \\ \vdots \\ x^\sharp_i = \Phi^\sharp_i(x_1, \dots, x_n) \\ \vdots \\ x^\sharp_n = \Phi^\sharp_n(x_1, \dots, x_n) \end{cases}$$

La méthode itérative standard consiste à itérer le calcul de $\Phi^\sharp$ jusqu'à obtenir une solution. Or, dans le cas général, il n'est pas garanti que la suite des itérations obtenues converge en temps fini : on peut alors appliquer à chaque étape un élargissement. On transforme donc le système précédent vers les équations suivantes :

$$\begin{cases} x^\sharp_1 = x^\sharp_1 \;\nabla\; \Phi^\sharp_1(x_1, \dots, x_n) \\ \vdots \\ x^\sharp_i = x^\sharp_i \;\nabla\; \Phi^\sharp_i(x_1, \dots, x_n) \\ \vdots \\ x^\sharp_n = x^\sharp_n \;\nabla\; \Phi^\sharp_n(x_1, \dots, x_n) \end{cases}$$

Les propriétés de l'élargissement nous garantissent alors que la méthode itérative termine, et trouve un point fixe $\mathcal{I}$ de la fonction abstraite $\lambda\vec{\mathcal{X}}^\sharp.\ \vec{\mathcal{X}}^\sharp \;\nabla\; \vec{\Phi}^\sharp(\vec{\mathcal{X}}^\sharp)$. Or une autre propriété de l'élargissement précise qu'il calcule un majorant, on a donc le résultat suivant :

$$\vec{\Phi}^\sharp(\mathcal{I}) \;\sqsubseteq^\sharp\; \mathcal{I} \;\nabla\; \vec{\Phi}^\sharp(\mathcal{I}) \;=\; \mathcal{I}$$

L'algorithme itératif calcule donc un post-point fixe de $\vec{\Phi}^\sharp$: grâce au théorème 2.6, on sait alors qu'il s'agit d'une abstraction sûre de l'ensemble des états accessibles $\mathrm{lfp}(\vec{\Phi})$. Cette méthode naïve est cependant sujette à des pertes de précision importantes, dues à l'élargissement systématique en chaque équation.

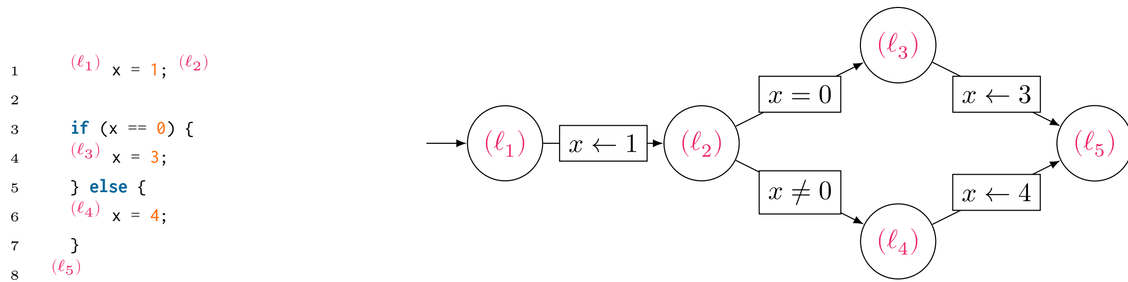


FIGURE 2.10 – Un programme sans boucle et son graphe de contrôle.

Nous avons présenté en section 2.3 plusieurs techniques permettant d'améliorer ce calcul avec une mise en œuvre immédiate :

- L'élargissement retardé : lors du calcul du point fixe, on commence par résoudre le système sans élargissement. Après un nombre arbitrairement choisi d'étapes de calcul, on remplace les équations par celles du système avec élargissement. On peut éventuellement choisir un nombre différent d'itérations standards pour chaque équation, l'important pour assurer la convergence étant de s'assurer qu'après une certaine itération on n'utilise plus que des élargissements (mais on peut les utiliser de manière sporadique avant).

- L'élargissement par seuils : il s'agit simplement d'un opérateur d'élargissement dont l'intuition est de favoriser la convergence par étapes plutôt que brutalement. La section 2.3 présente un moyen de construire de tels opérateurs pour les intervalles. Pour les autres domaines, l'idée générale est la même : les contraintes stables sont conservées, les autres sont relâchées par étapes avant d'être oubliées (ce qui correspond à les « projeter à l'infini »).

- Le rétrécissement : il s'agit de calculer, après stabilisation du système avec élargissement, un point fixe d'un système semblable faisant intervenir un opérateur de rétrécissement [CC77a] qui garantit à la fois la convergence et la sûreté du résultat final. Nous ne détaillerons pas la formalisation d'un tel opérateur, que nous n'utiliserons pas dans la suite de cette thèse.

2.9.4 Stratégie d'itération

Les améliorations présentées dans la section précédente s'attaquent à certains défauts de l'élargissement, mais présentent toujours un défaut majeur : les équations du système modifié font toutes intervenir l'élargissement. Or la section 2.3 montre par l'exemple qu'il n'est pas nécessaire de l'appliquer à chaque point de contrôle : dans le cas du programme 2.2, il suffit d'élargir uniquement en tête de boucle pour une convergence rapide.

Pour élargir ce résultat dans le cas général, il nous faut d'abord définir les *stratégies d'itération*. Rappelons que nous cherchons à résoudre un système de la forme $\vec{X} = \Phi(\vec{X})$, ce qui revient à déterminer le plus petit point fixe de la fonction Φ. La théorie de l'interprétation abstraite nous permet d'en calculer une abstraction sûre en calculant un post-point fixe d'une fonction $\Phi^\sharp$ associée (laquelle peut éventuellement faire intervenir un élargissement ∇), ce qui revient à résoudre un système de la forme $\Phi^\sharp(\vec{X}^\sharp) \sqsubseteq^\sharp \vec{X}^\sharp$.

La solution itérative naïve consiste, à chaque étape, à recalculer toutes les composantes $x_i^\sharp$ de $\vec{X}^\sharp$. Le calcul s'arrête quand un post-point fixe a été trouvé.

L'exemple de la figure 2.10 met en évidence la faiblesse de cette méthode : en calculant à chaque étape la valeur abstraite associée à tous les nœuds, elle demande 4 étapes de calcul

(chacune mettant à jour les nœuds par profondeur croissante), puis une étape pour vérifier la stabilisation, soit 25 calculs abstraits en tout. Or il est possible, en analysant la structure du graphe, de n'en réaliser que 5 : en effet, en remarquant que les sommets peuvent être triés topologiquement, on peut calculer les abstractions associées dans l'ordre correspondant $(\ell_1) - (\ell_2) - (\ell_3) - (\ell_4) - (\ell_5)$ et avoir la garantie d'obtenir directement leurs valeurs finales.

Lorsque ce graphe présente un cycle, il n'existe plus de tri topologique. Il devient alors nécessaire de recalculer certains sommets plusieurs fois, leur valeur dépendant (éventuellement transitivement) d'elle-même. Toutefois, il n'est pas nécessaire de recalculer toutes les équations à toutes les itérations.

François Bourdoncle [Bou93] a alors proposé des méthodes de calcul de stratégies efficaces basées sur une notion d'*ordre topologique faible*, qui étend les ordres topologiques en autorisant les boucles. Adopter de telles stratégies influe beaucoup sur le temps d'analyse, en permettant d'éviter de nombreux calculs d'équations inutiles.

Une stratégie d'itération naturelle consiste à suivre la structure du programme, c'est-à-dire à l'interpréter comme si on l'exécutait concrètement, mais en considérant la sémantique induite par le domaine abstrait choisi (d'où le terme d'*interprétation abstraite*). Ainsi, on exécute les instructions les unes à la suite des autres en calculant les résultats abstraits associés, et on répète les instructions d'une boucle jusqu'à stabilisation (Bourdoncle [Bou93] ayant montré qu'il suffit de vérifier la stabilisation de la tête de boucle).

Par ailleurs, nous avons vu qu'il est parfois nécessaire d'utiliser un élargissement pour accélérer (voire permettre) la convergence. La méthode naïve consiste alors à élargir à chaque nœud : encore une fois, ce n'est pas nécessaire et influe beaucoup sur la précision. Il suffit en effet que chaque boucle de calcul passe par au moins un point d'élargissement. En pratique, on choisit en général les têtes de boucles, l'intuition présentée à la section 2.3 étant que la perte de précision induite est alors en partie limitée par les conditions d'entrée et de sortie de boucle. Les stratégies d'itération proposées par Bourdoncle [Bou93] permettent également de déterminer des ensembles restreints de points d'élargissement.

Pour les analyses définies dans cette thèse, nous considérerons la stratégie d'itération fondée sur la structure du programme lorsque c'est possible. Nous étudierons également un cas où cette stratégie ne s'applique pas, et discuterons alors d'un autre choix. Dans tous les cas, nous supposerons que l'ensemble des points d'élargissements est celui induit par la stratégie d'itération adoptée.

2.10 Conclusion

L'interprétation abstraite offre un cadre formel sûr pour construire des analyses statiques performantes. À travers la notion d'abstraction, formalisée par les correspondances de Galois, elle propose de ne garder des états d'un programme que les informations estimées nécessaires à sa vérification en oubliant le reste. En modélisant la sémantique de ce programme comme le point fixe d'un système d'équations, elle permet alors de l'approximer dans l'abstrait à l'aide d'itérations chaotiques. Un opérateur d'élargissement peut le cas échéant garantir la calculabilité de cette analyse. En modulant le choix du domaine abstrait, on peut par ailleurs atteindre le bon compromis entre précision et performance pour mener à bien une analyse efficace.

Dans cette thèse, nous définirons plusieurs domaines abstraits dédiés à l'analyse de programmes concurrents en mémoire faiblement cohérente. Nous pourrons alors utiliser les résultats définis dans ce chapitre pour construire des analyses de ces programmes que nous présenterons dans les parties qui vont suivre.

Chapitre 3

Mémoire faiblement cohérente

Nous définirons dans ce chapitre le modèle mémoire dans lequel s'exécutent nos programmes cibles. Nous nous placerons pour cela dans le cadre spécifique des programmes concurrents implémentés dans le paradigme de la mémoire partagée. Dans ce cadre, plusieurs processeurs exécutent simultanément les instructions du processus correspondant, lesquelles instructions peuvent accéder à une zone mémoire accessible en lecture et en écriture par tous les processus. Le modèle de cohérence mémoire correspond alors à un ensemble de contraintes sur la visibilité d'une modification de la mémoire par les autres processus.

La figure 1.1 décrit la syntaxe des programmes concurrents que nous considérerons. Elle sera commune aux différents modèles.

3.1 Cohérence séquentielle

Rappelons que, dans le modèle de la cohérence séquentielle, les exécutions valides d'un programme concurrent correspondent aux entrelacements non déterministes des instructions de chacun de ses processus [Lam79].

La figure 3.1 formalise cette sémantique. Elle complète la sémantique des processus isolés de la figure 1.2 (page 14) par la règle d'entrelacement spécifiant les pas d'exécution du programme global.

Remarquons que, dès ce modèle mémoire et pour tous ceux qui suivront, nous considérerons que l'affectation et l'évaluation d'une expression sont atomiques. Ceci signifie qu'aucun processus ne peut prendre la main au milieu de leur exécution et est formalisé par le fait que ces opérations sont définies par un unique pas d'exécution des processus isolés (cf. figure 1.2). Ce n'est pas une perte de généralité : en effet, on peut toujours décomposer une éventuelle instruction non atomique en plusieurs instructions atomiques qui pourront alors s'exécuter selon les règles du modèle considéré.

3.2 Le modèle TSO : *Total Store Ordering*

3.2.1 Un modèle mémoire relâché

Le modèle mémoire qui, en plus des comportements séquentiellement cohérents, autorise les réordonnancements entre une écriture et une lecture à un autre emplacement mémoire qui la suit

$$\textsc{Thread}\ \frac{\langle s, S\rangle \to \langle s', S'\rangle}{\langle \texttt{thread \{ } s \texttt{ \}}, S\rangle \to \langle \texttt{thread \{ } s' \texttt{ \}}, S'\rangle}$$

$$\textsc{Interleaving}\ \frac{\langle T_i, S\rangle \to \langle T_i', S'\rangle}{\langle [T_1, \dots, T_i, \dots, T_N], S\rangle \to \langle [T_1, \dots, T_i', \dots, T_N], S'\rangle}$$

FIGURE 3.1 – Sémantique d'un programme concurrent en cohérence séquentielle.

(cf. Introduction, section 0.2.2, page 6), est appelé *TSO*, pour *Total Store Ordering*. Il est entre autres à la base du modèle de l'architecture x86 [SSO+10].

Dans ce modèle, les modifications de la mémoire ne sont plus totalement ordonnées sur le programme global par l'ordre des instructions correspondantes, mais elles le sont au sein de chaque processus. Si un processus donné effectue deux écritures, les autres processus ne les voient pas forcément immédiatement, mais ils les verront chacune nécessairement tous en même temps, et les deux dans le même ordre.

Exemple 3.1.
Revenons au programme 0.2, page 6. Le comportement contre-intuitif considéré est observable dans TSO si, alors que les deux processus ont bien procédé à leurs écritures respectives dans `x` et `y`, ces événements ne sont pas visibles immédiatement par l'autre processus. Lors des lectures, les deux processus peuvent alors voir la valeur 0 même si les deux écritures ont été effectuées.○

3.2.2 Tampons d'écriture

Pour décrire formellement ce modèle, il est courant de considérer qu'il existe un *tampon d'écriture* (en anglais *store buffer*) par processus. Les modifications de la mémoire effectuées par un processus, au lieu d'affecter immédiatement la mémoire partagée, sont temporairement stockées dans ce tampon. Elles y sont totalement ordonnées selon l'ordre où elles sont effectuées. À n'importe quel point de l'exécution du programme, l'entrée la plus ancienne du tampon peut être transférée dans la mémoire, mettant à jour la valeur de la variable correspondante dans la mémoire partagée. Lorsqu'un processus cherche à lire la valeur d'une variable, il obtient, si elle existe, la valeur correspondant à l'entrée la plus récente pour cette variable dans son propre tampon. S'il n'a aucune entrée pour la variable considérée, il obtient la valeur associée dans la mémoire partagée.

Les comportements possibles d'un programme redeviennent alors les entrelacements des instructions des processus qui le composent avec des transferts en mémoire pouvant arriver n'importe quand.

La figure 3.2 représente graphiquement l'état d'un programme dans ce modèle. Si le processus 0 lit la valeur de x, il obtiendra 1. Le processus 1, quant à lui, lira 4. Le processus 0 lira également 2 pour la variable y ; en revanche, le processus 1 n'a aucune entrée pour y dans son tampon. Il lira alors 0. Si le processus 0 transfert l'entrée la plus ancienne de son tampon en mémoire, x vaudra alors 42 en mémoire. Toutefois, le processus 1 continuera à lire la valeur inscrite dans son tampon (soit 4) tant que celle-ci n'est pas elle-même transférée.

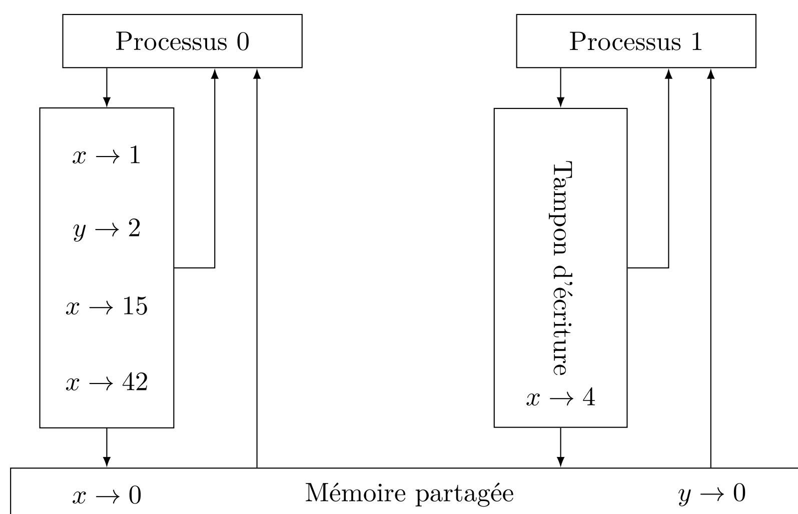


FIGURE 3.2 – L'état d'un programme dans le modèle TSO.

Remarque 3.1.
Les modèles mémoire, en particulier celui que nous décrivons dans cette section, ne cherchent pas à donner une description exhaustive du fonctionnement du système décrit mais à le représenter par un modèle formel équivalent du point de vue de l'exécution des programmes. On peut alors raisonner sur ces exécutions en considérant cette formalisation plutôt que les détails d'implémentation du système original.

Il est alors possible que ces modèles décrivent plus de choses que la réalité : ainsi, certains comportements peuvent être autorisés par les règles d'exécution de TSO sans qu'on ait réussi à les observer en pratique sur un processeur x86. Par exemple, les tampons du modèle théorique ne sont pas bornés, mais des tests en conditions réelles ne permettraient pas d'observer des comportements différant un nombre arbitrairement grand d'écritures.

L'important est que cette description soit sûre : une exécution observée non conforme au modèle est un bug d'implémentation du processeur. $\triangleleft$

3.2.3 Barrière

Afin de permettre au programmeur de retrouver les garanties offertes par la cohérence séquentielle lorsque c'est nécessaire, ce modèle ajoute aux instructions $\langle stmt \rangle$ du langage de la figure 1.1 une **barrière mémoire** `mfence` (pour *memory fence*).

Cette instruction empêche de réordonner les opérations mémoire qui la précèdent avec celles qui la suivent. En plaçant une telle instruction après chaque écriture, on peut donc retrouver un comportement séquentiellement cohérent. Cependant, ce n'est pas souhaitable dans le cas général : les comportements supplémentaires étant dus à des optimisations, il est préférable de ne pas les bloquer lorsqu'ils ne rendent pas le programme incorrect.

Dans le modèle avec tampons d'écriture, cette instruction force le transfert en mémoire de tout le contenu du tampon du processus qui l'exécute. Une définition équivalente la fait bloquer l'exécution tant que ce transfert n'a pas été effectué.

Remarque 3.2.
Le modèle x86-TSO définit également un préfixe `lock` applicable aux instructions d'assemblage effectuant une séquence *read-modify-write*. Celui-ci a pour effet de forcer l'atomicité de l'instruc-

tion préfixée : par exemple, `lock incr [x]` lit la valeur rangée à l'adresse `x`, l'incrémente et range le résultat à cette même adresse, sans que la valeur rangée n'ait pu changer entre temps. De plus, ce préfixe force le transfert en mémoire du contenu du tampon avant et après l'exécution de l'instruction.

Nous nous concentrerons dans cette thèse sur les aspects proprement faiblement cohérents du modèle et ne traiterons pas de l'atomicité des instructions préfixées de `lock` (le double transfert étant quant à lui équivalent à l'instruction de deux instructions `mfence`). Nous expliquerons au chapitre 15, section 15.2, comment l'implémenter par la suite dans les analyses obtenues. $\triangleleft$

3.2.4 Sémantique opérationnelle

Pour formaliser la sémantique opérationnelle dans TSO, nous commencerons par séparer les variables de *Var* en deux ensembles distincts, *Shared* et *Local*. *Shared* est l'ensemble des variables partagées par les différents processus, qui sont donc concernées par le stockage temporaire dans les tampons. *Local* est un ensemble de variables locales à un processus donné, similaire aux registres de l'assembleur : elles ne sont visibles et modifiables que par le processus auxquelles elles appartiennent, et à ce titre, elles sont simplement stockées dans une mémoire dédiée. Les variables locales servent notamment de destination aux lectures depuis la mémoire (ou le tampon le cas échéant) ou à des calculs internes au fonctionnement d'un processus qui n'ont pas vocation à être visibles de l'extérieur.

Pour simplifier la sémantique des programmes, nous ajouterons alors des restrictions syntaxiques à notre langage :

- Une expression affectée à une variable partagée ne peut elle-même contenir aucune variable partagée ;
- Une expression affectée à une variable locale ou présente dans le test d'une conditionnelle ou d'une boucle peut contenir au plus une occurrence d'une seule variable partagée.

Nous ne considérerons donc que des programmes respectant ces conditions, vérifiables dans notre langage simplifié par une analyse purement syntaxique. Elles nous permettront alors de lever syntaxiquement toute ambiguïté sémantique sur l'atomicité des pas d'exécution. Remarquons ainsi qu'il reste possible de considérer des expressions complexes contenant plusieurs variables partagées, en les décomposant en plusieurs expressions simples que l'on affecte à des variables locales temporaires. Par ailleurs, remarquons également que cette restriction correspond à celle de l'instruction `mov` (entre autres) de l'assembleur x86, que nous prenons pour cible.

Pour représenter l'état S d'un programme dans TSO, nous le décomposerons maintenant en deux parties : la mémoire partagée et locale Σ et, pour chaque processus, le tampon d'écriture σ. Un tampon σ est un mot fini sur l'alphabet des entrées, qui sont de la forme $x \mapsto v$ avec $x \in$ *Shared* et $v \in \mathbb{V}$. Si $x \mapsto v$ est la première entrée (dans l'ordre « de gauche à droite », donc la plus récente « chronologiquement ») de la forme $x \mapsto ?$ dans le mot σ, on note $\sigma(x) \triangleq v$. On note $\sigma(x) \triangleq \bot$ s'il n'existe aucune entrée de la forme $x \mapsto ?$ dans σ. $\varnothing$ dénote un tampon vide, et on dispose d'un opérateur $\cdot$ de concaténation.

La figure 3.3 recourt à ces notations pour remanier certaines règles des figures 1.2 (page 14) et 3.1 (page 44), qui définissaient ensemble la sémantique séquentiellement cohérente, pour obtenir la sémantique opérationnelle dans TSO.

Nous commençons par modifier les règles concernant les opérations d'accès aux variables partagées de la figure 1.2a. Les nouvelles règles d'écriture et de lecture sont en figure 3.3a.

$$\textsc{LocalGet}\ \frac{x \in \mathit{Local} \qquad \Sigma(x) = v}{\langle x, (\Sigma, \sigma)\rangle \Rightarrow v} \qquad \textsc{LocalAssign}\ \frac{\langle e, S\rangle \Rightarrow v \qquad x \in \mathit{Local}}{\langle x \texttt{ = } e, (\Sigma, \sigma)\rangle \rightarrow (\Sigma[x \mapsto n], \sigma)}$$

$$\textsc{ReadBuf}\ \frac{x \in \mathit{Shared} \qquad \sigma(x) = v}{\langle x, (\Sigma, \sigma)\rangle \Rightarrow v} \qquad \textsc{ReadMem}\ \frac{x \in \mathit{Shared} \qquad \sigma(x) = \bot \qquad \Sigma(x) = v}{\langle x, (\Sigma, \sigma)\rangle \Rightarrow v}$$

$$\textsc{Write}\ \frac{x \in \mathit{Shared} \qquad \langle e, (\Sigma, \sigma)\rangle \Rightarrow v}{\langle x \texttt{ = } e, \Sigma\rangle \rightarrow (\Sigma, (x \mapsto v) \cdot \sigma)}$$

(a) Écriture et lecture en mémoire.

$$\textsc{Flush}\ \frac{}{\langle s, (\Sigma, \sigma' \cdot (x \mapsto v))\rangle \rightarrow \langle s, (\Sigma[x \mapsto v], \sigma')\rangle} \qquad \textsc{MFence}\ \frac{}{\langle \texttt{mfence}, (\Sigma, \varnothing)\rangle \rightarrow (\Sigma, \varnothing)}$$

(b) Transfert et barrière.

$$\textsc{Interleaving}\ \frac{\langle T_i, (\Sigma, \sigma_i)\rangle \rightarrow \langle T_i', (\Sigma', \sigma_i')\rangle}{\langle [T_1, \dots, T_i, \dots, T_N], (\Sigma, [\sigma_1, \dots, \sigma_i, \dots, \sigma_N])\rangle \rightarrow \langle [T_1, \dots, T_i', \dots, T_N], (\Sigma', [\sigma_1, \dots, \sigma_i', \dots, \sigma_N])\rangle}$$

(c) Entrelacement des processus.

FIGURE 3.3 – Sémantique d'un programme concurrent dans TSO.

Nous définissons également en figure 3.3b des règles additionnelles d'exécution d'un processus pour le transfert et les barrières mémoire. La règle FLUSH prend l'entrée la plus ancienne dans le tampon d'un processus et met à jour la mémoire partagée avec la valeur correspondante, retirant simultanément l'entrée du tampon. Le non-déterminisme des transferts est encodé par le non-déterminisme de la sélection du prochain processus à être exécuté dans la sémantique du programme global : le transfert d'une entrée étant représenté par un pas d'exécution atomique d'un processus, il peut s'exécuter entre n'importe quelles instructions propres du programme. La règle MFENCE empêche la progression d'un processus dans son exécution tant que son tampon n'est pas vide. Le seul moyen d'en retirer des entrées étant par application de la règle FLUSH, MFENCE force un processus à toutes les transférer avant de passer à l'instruction suivante.

Enfin, nous réécrivons la règle d'entrelacement en figure 3.3c. Elle reste fondamentalement la même mais est ainsi accordée aux nouvelles notations.

3.3 Le modèle PSO : *Partial Store Ordering*

Dans le modèle TSO, toutes les modifications de la mémoire par un processus sont totalement ordonnées. Ainsi, le programme 3.1 ne peut pas terminer dans un état où `r1 = 1 && x = 0` : `T2` ne peut pas voir la modification de `x` après celle de `y`.

Il existe un modèle plus relâché que TSO, appelé PSO pour *Partial Store Ordering*, qui

```
1  /* Initial : x = y = 0 */
2
3  thread /* T1 */ {
4      x = 1;
5      y = 1;
6  }
7
8  thread /* T2 */ {
9      r1 = y;
10     r2 = x;
11 }
```

Programme 3.1 – Écriture et lecture dans deux variables différentes.

autorise un tel comportement. Ce modèle autorise en effet le réordonnancement de deux écritures mémoire à des emplacements différents. Autrement dit, dans ce modèle, les modifications de la mémoire par un processus ne sont que partiellement ordonnées : seul l'ordre entre deux modifications de la même variable est retenu.

Une représentation opérationnelle de ce modèle, similaire à celle de TSO, consiste à affecter à chaque processus un tampon d'écriture par variable (ou emplacement mémoire) ; l'entrée la plus ancienne de n'importe quel tampon pouvant toujours être transférée en mémoire de façon non déterministe.

L'instruction `mfence` existe toujours : elle bloque tout réordonnancement entre les opérations qui la précèdent et celles qui la suivent. Opérationnellement, elle transfère en mémoire toutes les entrées de tous les tampons du processus qui l'exécute — ou alternativement, bloque l'exécution tant que ce transfert n'a pas été effectué.

Il existe dans ce modèle une autre instruction de barrière, `sfence` (*store fence*). Cette instruction interdit le réordonnancement entre les écritures mémoire qui la précèdent et les écritures qui la suivent. Elle permet donc de retrouver le comportement de TSO si elle est insérée entre chaque paire d'écritures à des emplacements différents. Le modèle opérationnel à un tampon par processus et par variable ne permet en revanche pas de donner une sémantique exacte à cette instruction. Il est nécessaire de définir un modèle plus complexe pour pouvoir la prendre en compte, en ajoutant un ordre partiel sur les écritures à des emplacements différents, induit par les instructions `sfence` et restreignant les transferts autorisés.

Alors que TSO est un modèle fondamental ayant de nombreuses applications concrètes (à commencer par la modélisation des processeurs x86), l'intérêt de PSO est principalement théorique (bien qu'il soit utilisé pour les architectures SPARC). Nous nous en servirons principalement comme un modèle strictement plus relâché que TSO, c'est-à-dire autorisant davantage de comportements, tout en étant conceptuellement plus simple par certains aspects (notamment par la possibilité de le modéliser à l'aide de tampons contenant exclusivement des écritures sur une même variable).

Dans ce cadre, ayant pour objectif la vérification de programmes exécutés dans TSO, nous ne chercherons donc pas à définir PSO plus précisément que nécessaire. En particulier, nous ignorerons les instructions `sfence`, qui n'ont pas d'effet dans TSO, bloquant des réordonnancements déjà interdits par ce modèle.

La figure 3.4 redéfinit par rapport à la figure 3.3 (qui formalise TSO) les règles nécessaires pour obtenir une sémantique opérationnelle dans PSO. σ associe maintenant à chaque symbole partagé x la file σ_x des valeurs du tampon correspondant.

$$\textsc{ReadBuf}\quad \frac{x \in \mathit{Shared} \qquad \sigma_x = v \cdot \sigma'_x}{\langle x, (\Sigma, \sigma)\rangle \Rightarrow v} \qquad \textsc{ReadMem}\quad \frac{x \in \mathit{Shared} \qquad \sigma_x = \varnothing \qquad \Sigma(x) = v}{\langle x, (\Sigma, \sigma)\rangle \Rightarrow v}$$

$$\textsc{Write}\quad \frac{\langle e, (\Sigma, \sigma)\rangle \Rightarrow v}{\langle x \texttt{ = } e, \Sigma\rangle \to (\Sigma, \sigma[x \mapsto v \cdot \sigma_x])}$$

$$\textsc{Flush}\quad \frac{\sigma_x = \sigma'_x \cdot v}{\langle s, (\Sigma, \sigma)\rangle \to \langle s, (\Sigma[x \mapsto v], \sigma[x \mapsto \sigma'_x])\rangle} \qquad \textsc{MFence}\quad \frac{\forall x \in \mathit{Shared}, \sigma^{\varnothing}_x = \varnothing}{\langle \texttt{mfence}, (\Sigma, \sigma^{\varnothing})\rangle \to (\Sigma, \sigma^{\varnothing})}$$

FIGURE 3.4 – Sémantique des opérations mémoire dans PSO.

3.4 Autres modèles

Il existe de nombreux autres modèles qui ont été proposées pour modéliser des systèmes aux comportements plus complexes. Si nous ne les traiterons pas dans le cadre de cette thèse, nous les mentionnons brièvement ici pour compléter l'exposé.

Dans le domaine des processeurs, TSO et PSO font partie des modèles les plus stricts, n'autorisant respectivement qu'un ou deux types de réordonnancement parmi tous ceux qui respectent la sémantique séquentielle d'un programme. À l'opposé, des modèles comme POWER [SSA+11], également utilisé pour les processus ARM, sont beaucoup plus relâchés : ils autorisent quasiment tous les réordonnancements possibles, comme une lecture suivie d'une écriture ou deux lectures à des emplacements différents. Ces modèles viennent avec plusieurs instructions de barrière correspondant aux divers réordonnancements autorisés, et sont en général définis par une sémantique axiomatique, c'est-à-dire un ensemble de règles permettant de caractériser les exécutions valides.

À un plus haut niveau que l'assembleur, certains langages de programmation comme C [ISO11] ou Java [MPA05] définissent aussi un modèle de cohérence mémoire précis, plutôt que de considérer que des accès concurrents à la mémoire mènent systématiquement à un comportement indéterminé. Ces modèles spécifient les optimisations autorisées pour le compilateur, qui doit alors assurer que les programmes compilés respectent la sémantique des programmes sources en insérant éventuellement des barrières mémoire à des endroits choisis.

3.5 Conclusion

Nous disposons maintenant de modèles formels définissant la sémantique des programmes concurrents en mémoire faiblement cohérente. Ces modèles permettent de décrire les comportements relâchés dus aux optimisations ou aux détails d'implémentation des processeurs.

Dans les parties qui vont suivre, nous considérerons en particulier les modèles TSO et PSO. Nous définirons des abstractions sûres de la sémantique concrète décrite dans ce chapitre afin de certifier corrects les programmes s'exécutant dans ces modèles, même s'ils présentent des comportements relâchés jugés contre-intuitifs.

Deuxième partie

Interprétation abstraite monolithique

Chapitre 4

Considérations préalables

Avant de définir nos abstractions, nous commencerons par dresser un état de l'art de l'analyse statique dans les modèles mémoire faibles. Nous montrerons ensuite qu'un modèle simplifié qui paraît équivalent est en réalité plus strict, et définirons la construction du graphe de contrôle produit d'un programme concurrent qui nous servira pour calculer nos points fixes.

4.1 Travaux antérieurs

La vérification de programmes dans les modèles mémoire relâchés a été l'objet de plusieurs travaux antérieurs à cette thèse. Nombre d'entre eux se sont limités à des programmes à espace d'état fini, en utilisant des techniques comme la vérification de modèles (*model checking*) [ABBM10, AKL+11, KVY12, BDM13, AAN15, AAJL16].

L'approche de l'interprétation abstraite est mieux adaptée au cas général de programmes dont les espaces d'états ne sont pas bornés a priori. Dans ce domaine, Alglave [AKL+11] a montré que les analyses par interprétation abstraite **non relationnelle** sûres en cohérence séquentielle restent sûres dans les modèles mémoire faibles. En revanche, les analyses relationnelles ne le sont plus nécessairement : Alglave propose dans le même temps un moyen de les *réparer*, mais cette méthode conduit à des pertes de précision importantes en raison de la perte non contrôlée d'informations de relation.

Kuperstein *et al.* [KVY11] proposent d'utiliser des domaines abstraits pour modéliser des programmes aux tampons éventuellement non bornés. Cependant, leur méthode utilise ces abstractions au niveau de chaque état : si la taille d'un tampon donné dépasse une limite fixée, il est abstrait ; mais les ensembles éventuellement infinis d'états obtenus ne le sont pas. Par conséquent, leur méthode ne permet que d'analyser des programmes ayant un espace d'états fini en cohérence séquentielle, même si ces espaces peuvent devenir infinis dans TSO en raison de tampons non bornés.

Le travail de Dan *et al.* [DMVY14] avec lequel nous nous comparons au chapitre 8 a le problème inverse : il gère les programmes ayant un espace d'états séquentiellement cohérent infini, mais se limite à des tampons de taille bornée, dont une borne supérieure est calculée statiquement.

Par ailleurs, une méthode répandue pour vérifier la correction d'exécutions faiblement cohérentes consiste à réutiliser des analyses préexistantes sûres dans le cadre de la cohérence séquentielle, en effectuant une traduction source à source d'un programme en mémoire faible pour ramener le problème de son analyse à l'analyse d'un programme séquentiellement cohérent ayant la même sémantique [AKNT13, DMVY14, MDVY14]. Cette transformation fait donc apparaître

explicitement, dans le code source, les effets du modèle mémoire (par exemple en insérant des variables et des instructions explicites pour modéliser l'accès mémoire *via* un tampon). Cette approche semble en particulier bien se prêter à la génération automatique de barrières pour retrouver des propriétés qu'on sait valides en cohérence séquentielle. Toutefois, certaines propriétés peuvent être difficiles à exprimer après transformation des programmes, comme « les entrées du tampon sont triées en ordre croissant ». De plus, ces transformations peuvent être insuffisantes pour analyser avec précision le programme cible : à cause de la modélisation explicite et bas-niveau de chaque accès mémoire, elles obscurcissent sa sémantique et rendent difficile le développement d'abstractions dédiées. Cela demande en effet de modifier l'outil d'analyse en cohérence séquentielle pour prendre en compte des informations sur le modèle mémoire originel (par exemple, en considérant que certaines variables proviennent de la modélisation d'un tampon, ce qui n'a a priori pas de sens pour un analyseur séquentiellement cohérent). Enfin, ce manque d'abstraction dédiée peut aussi causer une perte de performances de l'analyseur.

Appliquer l'interprétation abstraite directement au programme source originel fournit en revanche un cadre permettant de concevoir plus aisément des domaines spécifiques, qui représentent des informations précises liées au modèle pour contourner ces deux problèmes.

4.2 L'importance des transferts distincts

Une tentation plusieurs fois observée (lors de discussions informelles) dans la recherche d'analyses de TSO consiste à considérer un modèle plus simple, qui semble équivalent, et dont nous allons montrer ici qu'il ne l'est en réalité pas.

Ce modèle consiste à considérer que les transferts non déterministes du tampon transfèrent tout le tampon atomiquement, plutôt qu'une écriture à la fois. L'intuition sous-jacente veut qu'une exécution dans TSO faisant intervenir des transitions de transfert puisse être remplacée par une autre exécution du même programme, dans le nouveau modèle, avec un entrelacement différent des écritures concernées, qui mène aux mêmes états. Si l'ensemble des traces dans ce nouveau modèle est trivialement restreint, on obtiendrait quand même la même sémantique d'états accessibles.

Sans développer ce qui mène à une impasse, nous observerons simplement que cette modification paraît correcte pour un nombre étonnamment élevé de programmes, ce qui est sans doute à l'origine des tentatives de simplification du modèle.

Le programme 4.1 est toutefois un contre-exemple : il possède strictement plus d'états accessibles dans TSO que dans le modèle simplifié. Considérons ainsi un état final respectant la propriété suivante :

$$r = 0 \land s_0 = 0 \land s_1 = 1$$

Cet état est accessible dans TSO : le processus `T1` exécute tout son code sans transfert, puis le processus `T2` écrit dans `y`, force le transfert par la barrière, et lit en mémoire $x \mapsto 0$ qu'il affecte à `s0`. Le transfert de l'écriture $x \mapsto 1$ est alors effectué par `T1`, et `T2` peut affecter `1` à `s1`.

En revanche, dans le modèle simplifié, il n'existe pas d'exécution menant à cet état final. En effet, la lecture de `0` à l'instruction `r = y` de `T1` force les deux écritures de `x` à être exécutées avant la barrière de `T2`. De la même façon, la lecture de `0` par `s0 = x` exige que ces deux écritures soient toujours dans le tampon de `T1` quand cette instruction s'exécute. Il n'est alors plus possible de transférer uniquement l'écriture $x \mapsto 1$, et `T2` ne peut lire que `0` ou `2` pour affecter `s1`.

Les analyses se basant sur le modèle simplifié n'ont donc aucune garantie de rester correctes pour TSO. Ce résultat est aussi vrai pour PSO (tampon par tampon) : le même programme le met en évidence.

```
/* Initial: x = 0 && y = 0 && s0 = 0 && s1 = 0 */

thread /* T1 */ {
    x = 1;
    x = 2;
    r = y;
}

thread /* T2 */ {
    y = 1;
    mfence;
    s0 = x;
    s1 = x;
}

/* Reachable: r = 0 && s0 = 0 && s1 = 1 ? */
```

PROGRAMME 4.1 – Contre-exemple de la simplification de la sémantique.

4.3 Graphe de contrôle produit

Nous avons vu en section 2.9.4 que les équations décrivant la sémantique d'un programme séquentiel correspondent à son graphe de contrôle. Ce graphe permet alors d'établir des stratégies d'itération pour le calcul du point fixe associé. Une de ces stratégies consiste à calculer les équations dans l'ordre du programme, qu'on peut retrouver à l'aide de la structure du graphe.

Dans le cas des programmes concurrents, l'équivalent du graphe de contrôle est donné par le produit des graphes de contrôle de chaque processus, comme illustré en figure 4.1. Soient n graphes $G_1, \dots, G_n$ donnés par leurs sommets, arêtes et sommets initiaux (V_i, E_i, I_i) (avec $I_i \in V_i$). Leur produit $G_\square = G_1 \square \cdots \square G_n$ est défini de la façon suivante :

- $V_\square \triangleq V_1 \times \cdots \times V_n$
- $E_\square \triangleq \left\{ (v_1, \dots, v_i, \dots, v_n) \xrightarrow{e_i} (v_1, \dots, v_i', \dots, v_n) \;\middle|\; v_i \xrightarrow{e} v_i' \in E_i \right\}$
- $I_\square \triangleq (I_1, \dots, I_n)$

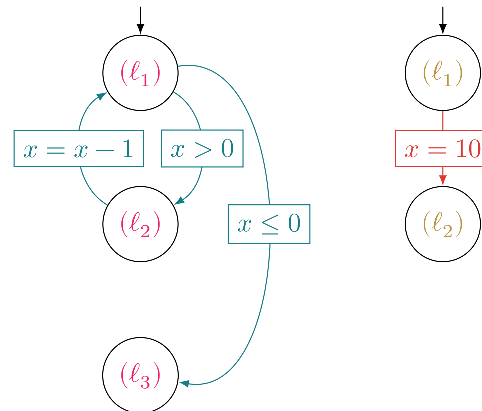
Les graphes-produits ainsi obtenus n'ont plus les mêmes propriétés structurelles que ceux des programmes séquentiels : en particulier, ils ne sont plus réductibles, c'est-à-dire qu'il n'est plus possible de « retrouver la structure du programme » en réduisant successivement les intervalles du graphe. Cette perte de propriété est due aux entrelacements non déterministes des processus qui brisent les intervalles correspondant aux structures de contrôle des processus.

Bourdoncle [Bou93] a toutefois proposé d'autres algorithmes permettant d'établir des stratégies d'itération qui, si elles ne sont pas optimales dans le cas de graphes bien structurés, donnent toutefois de bons résultats dans le cas général.

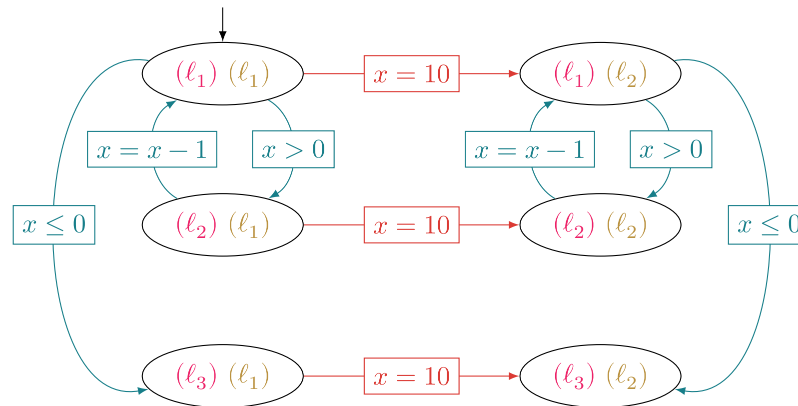
Dans une première approche, nous utiliserons donc cette construction de graphe produit pour l'interprétation abstraite des programmes concurrents. Une fois ce graphe construit et la stratégie d'itération établie, le calcul du point fixe se fait alors de façon identique aux programmes séquentiels : cette méthode a pour elle l'avantage de la simplicité.

```
1   thread {
2       (ℓ1) while (x > 0) {
3           (ℓ2) x = x - 1;
4       } (ℓ3)
5   }
6
7   thread {
8       (ℓ1) x = 10; (ℓ2)
9   }
```

(a) Programme à deux processus.



(b) Graphes de contrôle des processus.



(c) Graphe de contrôle produit.

FIGURE 4.1 – Le graphe de contrôle d'un programme est le produit des graphes des processus.

Chapitre 5

Sémantique et opérateurs concrets

Nous avons présenté une sémantique opérationnelle à petits pas de TSO dans les figures 3.1 et 3.3. Cette présentation, en détaillant les pas d'exécutions comme des transitions, a l'avantage d'être proche du modèle opérationnel lui-même (et donc plus aisée à comprendre). Pour l'analyse statique et l'interprétation abstraite, on préférera donner une sémantique sous une forme plus équationnelle, fondée sur le point fixe du système d'équations correspondant au graphe de contrôle (cf. chapitre 2, section 2.7.2, page 30). Elle restera à relativement petits pas : nous garderons les résultats de tous les nœuds intermédiaires du graphe.

5.1 Domaine concret

La figure 5.1 définit le domaine concret correspondant au modèle PSO. Rappelons que nous avons établi en section 3.3 que, bien que TSO soit une cible plus intéressante car plus proche des systèmes réels, PSO est strictement plus relâché. Ainsi, une analyse sûre pour PSO est a fortiori sûre pour TSO.

Notations

$\mathbb{V}$ est l'ensemble de définition des variables, par exemple $\mathbb{Z} \cup \mathbb{B}$.

Les états mémoire concrets sont définis comme un produit d'un état mémoire partagé, d'un état local et de l'état des différents tampons ; ces états étant représentés par des fonctions aux domaines de définition distincts. $\mathscr{S}$ est donc isomorphe à l'ensemble des fonctions depuis l'union de ces ensembles de définition vers $\mathbb{V}$. Pour simplifier les notations, on utilisera alternativement l'une ou l'autre de ces représentations équivalentes.

Dans toute la suite, à moins que le contraire ne soit explicitement précisé, on utilisera les lettres x, y, z pour les variables de *Shared* et les lettres r, s, t pour les variables de *Local*.

De plus, les symboles de *Shared* sont utilisés à la fois pour la mémoire partagée *Mem* et les tampons Buf_x^T. Ce dernier usage les indexe comme x_n^T, il n'y a donc pas de confusion possible. Cependant, pour plus de clarté, nous utiliserons la notation explicite x^{mem} pour désigner les variables dans la mémoire partagée.

Enfin, pour un état $S \in \mathscr{S}$, la taille du tampon du processus $T \in Thread$ et de la variable $x \in Shared$ sera notée $\#_x^T S$.

$$\begin{aligned}
Mem &\triangleq Shared \to \mathbb{V} && \text{Mémoire partagée}\\
TLS &\triangleq Local \to \mathbb{V} && \text{Variables locales } \textit{(Thread Local Storage)}\\
\forall x \in Shared, Buf_x^T &\triangleq \bigcup_{N \in \mathbb{N}} (\{x_1^T, \dots, x_N^T\} \to \mathbb{V}) && \text{Tampons}\\
\mathscr{S} &\triangleq Mem \times TLS \times \prod_{\substack{x \in Shared\\ T \in Thread}} Buf_x^T && \text{États mémoire}\\
\mathscr{D} &\triangleq \wp(\mathscr{S}) && \text{Domaine concret : ensembles d'états mémoire}
\end{aligned}$$

FIGURE 5.1 – Domaine concret pour les programmes dans PSO.

$$[\![.]\!] : \mathscr{D} \to \mathscr{D}$$

$$[\![a \leftarrow b]\!]X \triangleq \begin{cases} \{S[a \mapsto b] \mid S \in X, a \in \mathfrak{D}_S\} & \text{si } b \in \mathbb{V} \\ \{S[a \mapsto S(b)] \mid S \in X, a \in \mathfrak{D}_S, b \in \mathfrak{D}_S\} & \text{si } b \in Var \end{cases}$$

FIGURE 5.2 – Opérateur d'affectation fondamental sur les états concrets.

Ce domaine concret représente un tampon d'écriture pour le processus T et la variable x par N variables $x_1^T, \dots, x_N^T$, où N est le nombre d'entrées du tampon. Il s'agit de la *représentation explicite* des tableaux dans un domaine numérique concret, c'est-à-dire un ensemble de variables (ou dimensions) : nous l'adaptons ici à des tampons de longueur variable.

Afin de pouvoir traiter cette taille variable, nous aurons donc besoin de deux opérations de $\mathscr{D} \to \mathscr{D}$, $[\![\text{add}\, v]\!]$ et $[\![\text{drop}\, v]\!]$, qui respectivement ajoutent et suppriment une variable v à tous les états d'un ensemble donné. La variable ajoutée par $[\![\text{add}\, .]\!]$ n'est pas initialisée : on peut rajouter une valeur $\top$ à $\mathbb{V}$ à cette fin. En pratique, ce problème ne se posera pas : les variables que nous ajouterons seront initialisées immédiatement après.

Nous aurons également besoin d'un opérateur permettant d'accéder aux variables de ces états numériques et de les modifier. On notera ainsi $[\![a \leftarrow b]\!]$ l'opérateur de $\mathscr{D} \to \mathscr{D}$ affectant, dans tous les états S d'un ensemble donné où a est définie, la valeur de b à cette variable a. La figure 5.2 explicite cette opération standard. On note $\mathfrak{D}_S$ l'ensemble de définition de S vu comme une fonction de valuation de ses variables.

5.2 Opérateurs concrets

Une fois le domaine concret établi, il nous faut définir les opérateurs concrets donnant la sémantique du langage à analyser dans une forme adaptée. Notre analyse sera basée sur les équations décrivant le graphe de contrôle du programme : ainsi, les opérateurs que nous devons définir sont ceux correspondant aux arêtes de ce graphe. Il s'agit donc de l'affectation d'une variable, de l'évaluation d'une condition et des barrières. À ceux-ci nous ajouterons un opérateur de transfert : pour une variable et un processus donné, cet opérateur effectue le transfert de la

$$
\begin{aligned}
\{\!|.|\!\}_T &: \mathscr{S} \to \mathscr{S} \\
\{\!|r|\!\}_T S &\triangleq S(r) && r \in \mathit{Local} \\
\{\!|x|\!\}_T S &\triangleq \begin{cases} S(x^{mem}) & \text{si } \#_x^T S = 0 \\ S(x_1^T) & \text{si } \#_x^T S \geq 1 \end{cases} && x \in \mathit{Shared} \\
\{\!|v|\!\}_T S &\triangleq v && v \in \mathbb{V} \\
\{\!|\star e|\!\}_T S &\triangleq \star\,(\{\!|e|\!\} S) && \star \in \text{Unop} \\
\{\!|e_1 \dagger e_2|\!\}_T S &\triangleq (\{\!|e_1|\!\} S)\ \dagger\ (\{\!|e_2|\!\} S) && \dagger \in \text{Binop}
\end{aligned}
$$

FIGURE 5.3 – Évaluation d'une expression dans le domaine concret.

dernière entrée du tampon correspondant vers la mémoire partagée, si celui-ci n'est pas vide. En première approche, nous pouvons alors ajouter à chaque nœud du graphe de contrôle une boucle par variable et par processus ayant pour étiquette le transfert correspondant, modélisant ainsi le non-déterminisme de cette opération.

La figure 5.4 définit ces opérateurs à partir de l'opérateur fondamental de la figure 5.2 et d'une fonction $\{\!|.|\!\}$ d'évaluation des expressions, elle-même définie en figure 5.3 pour le domaine $\mathscr{D}$. La notation $[\![op]\!]_T X$ désigne l'évaluation de l'opérateur *op* par le processus T sur l'ensemble d'états X. Par commodité, plusieurs opérateurs $[\![.]\!]$ sont définis sur des singletons de $\mathscr{D}$ avant d'être étendus aux ensembles quelconques d'états.

5.3 Commutation des transferts

Nous avons vu que la méthode directe d'encodage du non-déterminisme des transferts consiste à ajouter des boucles de transfert à chaque nœud. Lors du calcul par itérations de la sémantique du programme, cette méthode conduit, à chaque calcul de la valeur à un point de contrôle donné, à calculer le point fixe de tous les transferts possibles pour une variable et un processus quelconques.

Nous allons maintenant établir un lemme grâce auquel nous disposerons d'une méthode de calcul plus efficace. Ce lemme stipule qu'il est équivalent d'effectuer les transferts d'une variable x avant ou après l'application d'un opérateur non lié à x : on obtient les mêmes états.

Lemme 5.1 (Commutation des transferts).
Soit $x \in \mathit{Shared}$ et $[\![op_{\not x}]\!]$ un opérateur qui ne lit ni ne modifie x, c'est-à-dire $[\![\texttt{mfence}]\!]$, $[\![y \leftarrow expr]\!]$, $[\![r \leftarrow expr]\!]$ ou $[\![condition]\!]$, où $x \neq y$ et x n'apparaît ni dans expr, ni dans condition.

Alors :

$$\forall X \in \mathscr{D}, \forall T \in \mathit{Thread}, [\![\text{flush}\, x]\!]_T \circ [\![op_{\not x}]\!] X = [\![op_{\not x}]\!] \circ [\![\text{flush}\, x]\!]_T X$$

PREUVE. Commençons par prouver le résultat pour $X = \{S\}$ avec $S \in \mathscr{S}$, par disjonction des cas sur $\#_x^T S$.

- $\#_x^T S = 0$
  $[\![\text{flush}\, x]\!]_T \{S\} = \varnothing$, donc $[\![op_{\not x}]\!]([\![\text{flush}\, x]\!]_T \{S\}) = \varnothing$.

$$
\begin{aligned}
[\![.]\!]_T &: \mathscr{D} \to \mathscr{D} \\
[\![r \leftarrow e]\!]_T\{S\} &\triangleq [\![r \leftarrow \{\![e]\!\}_T S]\!] S \\
[\![x \leftarrow e]\!]_T\{S\} &\triangleq [\![x_1^T \leftarrow \{\![e]\!\}_T S]\!] \circ \\
&\quad [\![x_2^T \leftarrow x_1^T]\!] \circ \\
&\quad \vdots \\
&\quad [\![x^T_{\#^T_x S+1} \leftarrow x^T_{\#^T_x S}]\!] \circ \\
&\quad [\![\text{add}\, x^T_{\#^T_x S+1}]\!]\ \{S\} \\
[\![condition]\!]_T X &\triangleq \{S \in X \mid \{\![condition]\!\}_T S = \mathtt{true}\} \\
[\![\mathtt{mfence}]\!]_T\{S\} &\triangleq \begin{cases} S & \text{si } \forall x \in \mathit{Var}, \#^T_x S = 0 \\ \varnothing & \text{sinon} \end{cases} \\
[\![\text{flush}\, x]\!]_T\{S\} &\triangleq \begin{cases} \varnothing & \text{si } \#^T_x S = 0 \\ [\![\text{drop}\, x^T_{\#^T_x S}]\!] \circ [\![x^{mem} \leftarrow x^T_{\#^T_x S}]\!]\{S\} & \text{si } \#^T_x S \geq 1 \end{cases} \\
\forall X \in \mathscr{D}, [\![op]\!]_T X &\triangleq \bigcup_{S \in X} [\![op]\!]_T\{S\}
\end{aligned}
$$

FIGURE 5.4 – Opérateurs concrets dans PSO.

Par ailleurs, $[\![op_{\not x}]\!]$ n'ajoute aucune entrée au tampon de x et T, $[\![x \leftarrow e]\!]$ étant le seul opérateur ayant cet effet. Par conséquent, $\forall S' \in [\![op_{\not x}]\!]\{S\}, \#^T_x S' = 0$, d'où $[\![\text{flush}\, x]\!]_T([\![op_{\not x}]\!]\{S\}) = \varnothing$.

- $\#^T_x S > 0$
  $[\![op_{\not x}]\!]$ ne modifie pas la valeur de $x^T_{\#^T_x S}$, et n'utilise pas la valeur de x^{mem}. Par conséquent $[\![x^{mem} \leftarrow x^T_{\#^T_x S}]\!]$ commute avec $[\![op_{\not x}]\!]$. $[\![op_{\not x}]\!]$ n'utilise pas non plus la valeur de $x^T_{\#^T_x S}$, donc $[\![op_{\not x}]\!]$ commute également avec $[\![\text{drop}\, x^T_{\#^T_x S}]\!]$. Par succession des deux commutations, $[\![op_{\not x}]\!]$ commute avec $[\![\text{flush}\, x]\!]_T$.

Pour $X \in \mathscr{D}$ quelconque, le lemme se déduit en appliquant ce résultat état par état :

$$
\begin{aligned}
[\![\text{flush}\, x]\!]_T \circ [\![op_{\not x}]\!] X &= \bigcup_{S \in X} [\![\text{flush}\, x]\!]_T \circ [\![op_{\not x}]\!]\{S\} \\
&= \bigcup_{S \in X} [\![op_{\not x}]\!] \circ [\![\text{flush}\, x]\!]_T\{S\} = [\![op_{\not x}]\!] \circ [\![\text{flush}\, x]\!]_T X
\end{aligned}
$$

□

Considérons alors un élément $X \in \mathscr{D}$ où tous les transferts possibles de x ont été effectués. Après application d'un opérateur $[\![op_{\not x}]\!]$ à X, ce lemme nous indique qu'il est inutile de recalculer les transferts de x : puisqu'ils ont déjà été réalisés avant l'application de l'opérateur, on obtiendrait des états déjà présents dans X.

Grâce à ce résultat, plutôt que de calculer le point fixe du transfert de chaque variable par chaque processus à chaque sommet du graphe, nous pourrons donc ne calculer que le transfert des variables liées aux opérateurs menant à ce sommet. Plus précisément, lorsque nous calculerons le résultat d'une arête dont l'élément d'origine est X et l'étiquette $[\![op_x]\!]_T$ (où x est une

variable partagée et $[\![op_x]\!]$ est un opérateur lisant ou écrivant dedans [1]), nous ne calculerons pas simplement $[\![op_x]\!]_T X$ comme pour la méthode standard, mais l'élément suivant :

$$[\![\text{flush}\, x]\!]^* \circ [\![op_x]\!]_T X$$
où :
$$[\![\text{flush}\, x]\!]^* X \triangleq \text{lfp}\left(\lambda Y.X \cup \bigcup_{T \in Thread} [\![\text{flush}\, x]\!]_T Y\right)$$

Cet élément est la *clôture par transfert* du résultat de l'opérateur. On remarquera que les transferts sont calculés depuis tous les processus, pas uniquement celui qui exécute $[\![op_x]\!]$.

Exemple 5.1.
Considérons l'état suivant :

$$x^{mem} \mapsto 0,\ y^{mem} \mapsto 0,\ x_1^1 \mapsto 1,\ y_1^2 \mapsto 2$$

Le plus petit ensemble clos par transfert qui le contient est le suivant :

$$\begin{array}{l} x^{mem} \mapsto 0,\ y^{mem} \mapsto 0,\ x_1^1 \mapsto 1,\ y_1^2 \mapsto 2 \\ x^{mem} \mapsto 1,\ y^{mem} \mapsto 0,\ \qquad\qquad y_1^2 \mapsto 2 \\ x^{mem} \mapsto 0,\ y^{mem} \mapsto 2,\ x_1^1 \mapsto 1 \\ x^{mem} \mapsto 1,\ y^{mem} \mapsto 2 \end{array}$$

Effectuons maintenant l'opération $[\![x \leftarrow 3]\!]_2$. Avant clôture par transfert, elle ajoute simplement une entrée $x_1^2 \mapsto 3$:

$$\begin{array}{l} x^{mem} \mapsto 0,\ y^{mem} \mapsto 0,\ x_1^1 \mapsto 1,\ y_1^2 \mapsto 2,\ x_1^2 \mapsto 3 \\ x^{mem} \mapsto 1,\ y^{mem} \mapsto 0,\ \qquad\qquad y_1^2 \mapsto 2,\ x_1^2 \mapsto 3 \\ x^{mem} \mapsto 0,\ y^{mem} \mapsto 2,\ x_1^1 \mapsto 1,\ \qquad\qquad x_1^2 \mapsto 3 \\ x^{mem} \mapsto 1,\ y^{mem} \mapsto 2,\ \qquad\qquad\qquad\qquad x_1^2 \mapsto 3 \end{array} \tag{5.1}$$

Pour clore cet ensemble, il faut maintenant calculer tous les transferts possibles de x. Ceux-ci incluent notamment les transferts de l'entrée qui vient d'être produite, qui génèrent les états suivants :

$$\begin{array}{l} x^{mem} \mapsto 3,\ y^{mem} \mapsto 0,\ x_1^1 \mapsto 1,\ y_1^2 \mapsto 2 \\ x^{mem} \mapsto 3,\ y^{mem} \mapsto 0,\ \qquad\qquad y_1^2 \mapsto 2 \\ x^{mem} \mapsto 3,\ y^{mem} \mapsto 2,\ x_1^1 \mapsto 1 \\ x^{mem} \mapsto 3,\ y^{mem} \mapsto 2 \end{array} \tag{5.2}$$

Il faut également y inclure les transferts de l'entrée $x_1^1 \mapsto 1$, même si elle ne correspond pas au processus venant d'exécuter l'opération. En effet, ces transferts ne commutent pas avec celui de ladite opération. Ils permettent de générer les états suivants, absents de l'ensemble partiellement clos décrit par les équations (5.1) et (5.2) :

$$\begin{array}{l} x^{mem} \mapsto 1,\ y^{mem} \mapsto 0,\ y_1^2 \mapsto 2 \\ x^{mem} \mapsto 1,\ y^{mem} \mapsto 2 \end{array} \tag{5.3}$$

1. Rappelons que les restrictions de syntaxe introduites en section 3.2.4 nous garantissent qu'un opérateur ne peut lire ou écrire que dans une seule variable partagée au plus.

En revanche, il n'est pas nécessaire de calculer à nouveau les transferts de y_1^2 : ils ne génèrent que des états déjà présents dans l'ensemble calculé. Par exemple, considérons cet état présent dans l'équation (5.3), donc obtenu après transferts successifs de x_1^2 puis x_1^1 :

$$x^{mem} \mapsto 1, y^{mem} \mapsto 0, y_1^2 \mapsto 2$$

Depuis cet état, le transfert de y_1^2 génère l'état suivant :

$$x^{mem} \mapsto 1, y^{mem} \mapsto 2$$

Or cet état n'est pas nouveau : il était déjà décrit par l'équation (5.3). Il est donc bien déjà présent dans le résultat du calcul de la clôture de l'opération. Ce résultat est dû à l'équivalence entre les séquences de transfert $[x_1^2, x_1^1, y_1^2]$ et $[y_1^2, x_1^2, x_1^1]$. ○

Exemple 5.2.
Mettons maintenant en évidence la sûreté du calcul de la clôture par transfert, par opposition à la sémantique simplifiée de la section 4.2 du chapitre 4, en reprenant l'exécution du programme 4.1 (cf. page 55). Considérons donc l'état suivant, faisant partie de l'ensemble (clos par transfert) accessible après exécution complète du processus `T1` puis exécution de `T2` jusqu'à la barrière incluse :

$$x^{mem} \mapsto 0,\ x_1^{\texttt{T1}} \mapsto 2,\ x_2^{\texttt{T1}} \mapsto 1,\ r \mapsto 0,\ y^{mem} \mapsto 1,\ s_0 \mapsto 0,\ s_1 \mapsto 0$$

La lecture `s0 = x` depuis cet état ne le modifie pas par elle-même. En revanche, il s'agit d'une lecture d'une variable partagée : pour calculer les états accessibles après son exécution, il faut procéder à une clôture par transfert. Depuis cet état, on obtient alors les états suivants :

$$\begin{array}{ll}
x^{mem} \mapsto 0,\ x_1^{\texttt{T1}} \mapsto 2,\ x_2^{\texttt{T1}} \mapsto 1, & r \mapsto 0,\ y^{mem} \mapsto 1,\ s_0 \mapsto 0,\ s_1 \mapsto 0 \\
x^{mem} \mapsto 1,\ x_1^{\texttt{T1}} \mapsto 2, & r \mapsto 0,\ y^{mem} \mapsto 1,\ s_0 \mapsto 0,\ s_1 \mapsto 0 \\
x^{mem} \mapsto 2, & r \mapsto 0,\ y^{mem} \mapsto 1,\ s_0 \mapsto 0,\ s_1 \mapsto 0
\end{array}$$

La lecture `s1 = x` par `T2`, avant clôture par transfert, conduit alors aux états suivants :

$$\begin{array}{ll}
x^{mem} \mapsto 0,\ x_1^{\texttt{T1}} \mapsto 2,\ x_2^{\texttt{T1}} \mapsto 1, & r \mapsto 0,\ y^{mem} \mapsto 1,\ s_0 \mapsto 0,\ s_1 \mapsto 0 \\
x^{mem} \mapsto 1,\ x_1^{\texttt{T1}} \mapsto 2, & r \mapsto 0,\ y^{mem} \mapsto 1,\ s_0 \mapsto 0,\ s_1 \mapsto 1 \\
x^{mem} \mapsto 2, & r \mapsto 0,\ y^{mem} \mapsto 1,\ s_0 \mapsto 0,\ s_1 \mapsto 2
\end{array}$$

Il existe alors bien un état accessible, calculé par cette méthode, qui vérifie la propriété $r = 0 \wedge s_0 = 0 \wedge s_1 = 1$. ○

Pour formaliser l'analyse monolithique, nous pouvons alors définir, pour chaque opérateur $[\![op]\!]$ du langage, l'*opérateur clos par transfert* associé $\overline{[\![op]\!]}$:

$$\overline{[\![op]\!]}_T : \mathscr{D} \to \mathscr{D} \triangleq \lambda X. \begin{cases} [\![\text{flush}\, x]\!]^* \circ [\![op]\!]_T X & \text{si } [\![op]\!] \text{ lit ou écrit dans } x \in \mathit{Shared} \\ [\![op]\!]_T X & \text{sinon} \end{cases}$$

Soit $\mathscr{C}$ l'ensemble des états de contrôle, c'est-à-dire les sommets du graphe de contrôle produit. Soit $R_0 : \mathscr{C} \to \mathscr{D}$ la fonction donnant les résultats initiaux avant toute analyse ; c'est-à-dire

les états accessibles en chaque point de contrôle avant que le programme ne s'exécute : habituellement, $\top$ à l'état initial et $\bot$ partout ailleurs. Le résultat $\mathcal{R}$ de l'analyse peut finalement être exprimé sous la forme habituelle d'un point fixe sur ce graphe, en remplaçant simplement les opérateurs par leur clôture :

$$\mathcal{R} : \mathscr{C} \to \mathscr{D} \triangleq \text{lfp } \lambda R.R_0 \cup \left(\lambda c. \bigcup_{\text{arêtes } c' \xrightarrow{op_T} c} \overline{[\![op]\!]}_T R(c') \right)$$

Remarque 5.1.
Dans le domaine concret, cette méthode de calcul n'a aucun impact sur la précision : elle calcule exactement les mêmes états en chaque point de contrôle. En revanche, ceci ne sera plus nécessairement vrai après abstraction, et nous avons effectivement parfois observé qu'elle engendrait une analyse plus précise (par rapport au calcul systématique de tous les transferts) : l'opérateur de transfert abstrait pouvant mener à des pertes de précision, en calculer moins peut en effet permettre de limiter ces pertes. $\triangleleft$

Nous avons donc défini dans ce chapitre une sémantique concrète dans PSO sous la forme d'un système d'équations, et formulé un résultat permettant de limiter les calculs lors de l'itération du point fixe. Nous pouvons maintenant abstraire cette sémantique, pour en obtenir une approximation calculable, à l'aide de domaines dédiés.

Chapitre 6

Partitionnement

Notre première étape d'abstraction consiste à partitionner les états [CC79] selon une information partielle sur la taille des tampons.

Dans le cas général, le partitionnement des états regroupe ensemble les états selon un critère donné, afin d'abstraire par la suite séparément les différentes partitions. Habituellement, il sert à améliorer la précision d'une analyse existante : en effet, en distinguant les différentes partitions plutôt qu'en représentant leur ensemble par sa borne supérieure, il permet de limiter les unions abstraites. Or celles-ci peuvent causer d'importantes pertes de précision, particulièrement dans les domaines convexes comme les polyèdres, les octogones ou les intervalles.

Dans notre cas, il permettra également de simplifier la construction de l'analyse : en regroupant ensemble les états ayant des tampons de longueur similaire (d'après un critère que nous définirons par la suite), il permettra d'obtenir des partitions avec des environnements numériques semblables sur lesquelles les opérations du langage ont des comportements homogènes (par exemple, la lecture d'une variable diffère selon qu'il existe une entrée ou non dans le tampon). Il sera alors plus facile à la fois d'abstraire les partitions dans les domaines qui suivront et de formuler les opérateurs abstraits. C'est pourquoi, plutôt que de présenter le partitionnement comme un raffinement après les abstractions spécifiques des chapitres suivants, nous l'abordons dès maintenant.

6.1 Domaines de partitionnement des états

Commençons par définir, dans le cas général, les domaines de partitionnement des états.

Définition 18 (Domaine de partitionnement des états).
Considérons un ensemble d'états concrets $\mathscr{S}$ et le domaine concret $\mathscr{D} = \wp(\mathscr{S})$.

Un domaine de partitionnement des états $\mathscr{D}^\sharp_P$ est défini par :

- Un ensemble fini de propriétés $\mathbb{P}$
- Un critère de partitionnement $\delta : \mathscr{S} \to \mathbb{P}$
- Une abstraction $\mathscr{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathscr{D}^\sharp$

$\mathscr{D}_P^\sharp$ est alors défini de la façon suivante :

$$\mathscr{D}_P^\sharp \triangleq \mathbb{P} \to \mathscr{D}^\sharp$$
$$\sqsubseteq_P^\sharp : \mathscr{D}_P^\sharp \times \mathscr{D}_P^\sharp$$
$$X_P^\sharp \sqsubseteq_P^\sharp Y_P^\sharp \stackrel{\Delta}{\equiv} \forall P \in \mathbb{P}, X_P^\sharp(P) \sqsubseteq^\sharp Y_P^\sharp(P)$$
$$(\mathscr{D}, \subseteq) \underset{\alpha_P}{\overset{\gamma_P}{\leftrightarrows}} (\mathscr{D}_P^\sharp, \sqsubseteq_P^\sharp)$$
$$\gamma_P(X_P^\sharp) \triangleq \left\{ S \in \mathscr{S} \;\middle|\; S \in \gamma(X_P^\sharp(\delta(S))) \right\}$$
$$\alpha_P(X) = \lambda P.\alpha\left(\{S \in X \mid \delta(S) = P\}\right)$$

$\mathscr{D}^\sharp$ peut également être défini simplement par une concrétisation γ. $\mathscr{D}_P^\sharp$ est alors lui-même défini par la concrétisation γ_P. ◊

Preuve. Montrons que $\mathscr{D} \underset{\alpha_P}{\overset{\gamma_P}{\leftrightarrows}} \mathscr{D}_P^\sharp$ est bien une correspondance de Galois.

$\alpha_P(X) \sqsubseteq_P^\sharp Y_P^\sharp$	
$\iff \lambda P.\alpha\left(\{S \in X \mid \delta(S) = P\}\right) \sqsubseteq_P^\sharp Y_P^\sharp$	*Définition de* $\alpha_P^\sharp$
$\iff \forall P \in \mathbb{P}, \alpha\left(\{S \in X \mid \delta(S) = P\}\right) \sqsubseteq_P^\sharp Y_P^\sharp(P)$	*Définition de* $\sqsubseteq_P^\sharp$
$\iff \forall P \in \mathbb{P}, \{S \in X \mid \delta(S) = P\} \subseteq \gamma(Y_P^\sharp(P))$	$\mathscr{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathscr{D}^\sharp$
$\iff \forall P \in \mathbb{P}, \forall S \in X, \delta(S) = P \implies S \in \gamma(Y_P^\sharp(P))$	*Définition de* $\subseteq$
$\iff \forall P \in \mathbb{P}, \forall S \in X, S \in \gamma(Y_P^\sharp(\delta(S)))$	*Simplification de* $\implies$
$\iff \forall S \in X, S \in \gamma(Y_P^\sharp(\delta(S)))$	P *n'intervient plus*
$\iff X \subseteq \left\{ S \in \mathscr{S} \;\middle|\; S \in \gamma(Y_P^\sharp(\delta(S))) \right\}$	*Définition de* $\subseteq$
$\iff X \subseteq \gamma_P^\sharp(Y_P^\sharp)$	*Définition de* $\gamma_P^\sharp$

□

Une instance typique de ce genre de domaine est obtenue, pour un critère δ quelconque, en choisissant $\mathscr{D}^\sharp = \mathscr{D}$. On obtient alors un domaine qui consiste simplement à partitionner les états, sans appliquer aucune abstraction supplémentaire. La correspondance de Galois associée est alors un isomorphisme $\mathscr{D} \underset{\alpha_P}{\overset{\gamma_P}{\leftleftarrows\!\!\twoheadrightarrow}} \mathscr{D}_P^\sharp$.

6.2 Partitionnement selon la longueur des tampons

Pour partitionner les états concrets dans PSO, nous utiliserons un critère partiel δ sur la taille des tampons, défini en figure 6.1. Ce critère consiste à classer les tampons selon trois « tailles abstraites » : ils peuvent être vides (noté $\widetilde{0}$), contenir exactement un élément (noté $\widetilde{1}$), ou en contenir strictement plus (noté $\widetilde{\omega}$). L'ensemble des critères pour chaque tampon (c'est-à-dire une taille abstraite par processus et par variable partagée) caractérise la partition à laquelle appartient un état donné.

Le domaine qui en découle est récapitulé en figure 6.2. Aucune abstraction supplémentaire n'étant pour l'instant appliquée aux partitions, il s'agit d'un isomorphisme de Galois contenant toute l'information initiale, qui est simplement représentée sous une forme différente.

$$\mathscr{B}^\flat \triangleq \mathit{Shared} \times \mathit{Thread} \to \{\tilde{0}, \tilde{1}, \tilde{\omega}\}$$
$$\delta : \mathscr{S} \to \mathscr{B}^\flat$$
$$\delta(S) \triangleq \lambda(x, T). \begin{cases} \tilde{0} & \text{si } \#_x^T S = 0 \\ \tilde{1} & \text{si } \#_x^T S = 1 \\ \tilde{\omega} & \text{si } \#_x^T S > 1 \end{cases}$$

FIGURE 6.1 – Critère de partitionnement sur les tampons.

$$\mathscr{D}^\flat \triangleq \mathscr{B}^\flat \to \mathscr{D}$$
$$\mathscr{D} \underset{\alpha_P}{\overset{\gamma_P}{\leftleftarrows\!\!\rightrightarrows}} \mathscr{D}^\flat$$
$$\alpha_P(X) \triangleq \lambda b^\flat . \left\{ S \in X \;\middle|\; \delta(S) = b^\flat \right\}$$
$$\gamma_P(X^\flat) \triangleq \left\{ S \in \mathscr{S} \;\middle|\; S \in X^\flat(\delta(S)) \right\}$$

FIGURE 6.2 – Récapitulatif du domaine de partitionnement des tampons.

Pour comprendre ce choix de partitionnement, remarquons que ces trois tailles abstraites correspondent à des catégories de tampons ayant des effets bien distincts sur le comportement vis-à-vis de la mémoire. Un tampon vide ne contient aucune entrée : la mémoire partagée sert donc pour la lecture de la variable correspondante. En revanche, un tampon contenant une entrée est prioritaire sur la mémoire : ces deux cas mettent donc en œuvre un mécanisme fondamentalement différent quant à la lecture. La distinction du cas $\tilde{\omega}$ sera mieux justifiée par la suite, quand nous développerons les abstractions appliquées aux partitions. En première approche, remarquons simplement que les tampons correspondants contiennent des entrées inaccessibles : elles ne sont pas lues par le processus en question (qui ne voit que la plus récente). Pourtant ces entrées sont importantes : une fois transférées en mémoire, elles deviendront visibles par les autres processus. La seule différence entre ces entrées, inaccessibles en lecture directe mais qui seront utilisées pour des mises à jour, est l'ordre dans lequel cette mise à jour sera effectuée. Nous considérerons donc que des tampons à 2 ou 10 états sont suffisamment similaires quant aux opérations qu'ils proposent pour être regroupés ensemble.

Exemple 6.1.
Considérons le programme suivant :

```
1 thread {
2     x = 1;
3 }
4
5 thread {
6     y = 2;
7     y = 3;
8     y = 4;
9 }
```

Les états accessibles en fin d'exécution sont les suivants :

$$\begin{array}{llll}
x^{mem} \mapsto 0,\ y^{mem} \mapsto 0,\ x_1^1 \mapsto 1,\ y_1^2 \mapsto 4,\ y_2^2 \mapsto 3,\ y_3^2 \mapsto 2 \\
x^{mem} \mapsto 1,\ y^{mem} \mapsto 0,\ \qquad y_1^2 \mapsto 4,\ y_2^2 \mapsto 3,\ y_3^2 \mapsto 2 \\
x^{mem} \mapsto 0,\ y^{mem} \mapsto 2,\ x_1^1 \mapsto 1,\ y_1^2 \mapsto 4,\ y_2^2 \mapsto 3 \\
x^{mem} \mapsto 1,\ y^{mem} \mapsto 2,\ \qquad y_1^2 \mapsto 4,\ y_2^2 \mapsto 3 \\
x^{mem} \mapsto 0,\ y^{mem} \mapsto 3,\ x_1^1 \mapsto 1,\ y_1^2 \mapsto 4 \\
x^{mem} \mapsto 1,\ y^{mem} \mapsto 3,\ \qquad y_1^2 \mapsto 4 \\
x^{mem} \mapsto 0,\ y^{mem} \mapsto 4,\ x_1^1 \mapsto 1 \\
x^{mem} \mapsto 1,\ y^{mem} \mapsto 4
\end{array}$$

Ces états forment 6 partitions :

$$\begin{array}{lcl}
\left\langle x^1 \mapsto \tilde{0}, y^1 \mapsto \tilde{0}, x^2 \mapsto \tilde{0}, y^2 \mapsto \tilde{0} \right\rangle & : & x^{mem} \mapsto 1, y^{mem} \mapsto 4 \\
\left\langle x^1 \mapsto \tilde{1}, y^1 \mapsto \tilde{0}, x^2 \mapsto \tilde{0}, y^2 \mapsto \tilde{0} \right\rangle & : & x^{mem} \mapsto 0, y^{mem} \mapsto 4, x_1^1 \mapsto 1 \\
\left\langle x^1 \mapsto \tilde{0}, y^1 \mapsto \tilde{0}, x^2 \mapsto \tilde{0}, y^2 \mapsto \tilde{1} \right\rangle & : & x^{mem} \mapsto 1, y^{mem} \mapsto 3, y_1^2 \mapsto 4 \\
\left\langle x^1 \mapsto \tilde{1}, y^1 \mapsto \tilde{0}, x^2 \mapsto \tilde{0}, y^2 \mapsto \tilde{1} \right\rangle & : & x^{mem} \mapsto 0, y^{mem} \mapsto 3, x_1^1 \mapsto 1, y_1^2 \mapsto 4 \\
\left\langle x^1 \mapsto \tilde{0}, y^1 \mapsto \tilde{0}, x^2 \mapsto \tilde{0}, y^2 \mapsto \tilde{\omega} \right\rangle & : & \begin{cases} x^{mem} \mapsto 1, y^{mem} \mapsto 2, y_1^2 \mapsto 4, y_2^2 \mapsto 3 \\ x^{mem} \mapsto 1, y^{mem} \mapsto 0, y_1^2 \mapsto 4, y_2^2 \mapsto 3, y_3^2 \mapsto 2 \end{cases} \\
\left\langle x^1 \mapsto \tilde{1}, y^1 \mapsto \tilde{0}, x^2 \mapsto \tilde{0}, y^2 \mapsto \tilde{\omega} \right\rangle & : & \begin{cases} x^{mem} \mapsto 0, y^{mem} \mapsto 2, x_1^1 \mapsto 1, y_1^2 \mapsto 4, y_2^2 \mapsto 3 \\ x^{mem} \mapsto 0, y^{mem} \mapsto 0, x_1^1 \mapsto 1, y_1^2 \mapsto 4, y_2^2 \mapsto 3, y_3^2 \mapsto 2 \end{cases}
\end{array}$$

Calculer le résultat de $[\![r \leftarrow y]\!]_2$ peut maintenant se faire partition par partition : par exemple, si $y^2 \mapsto \tilde{0}$, on applique $[\![r \leftarrow y^{mem}]\!]$ à tous les états de la partition. Sinon, on leur applique $[\![r \leftarrow y_1^2]\!]$. Chaque opérateur abstrait se traduit ainsi par les mêmes opérations élémentaires sur tous les états d'une même partition.

De plus, dans le chapitre suivant, nous regrouperons, dans les partitions où $y^2 \mapsto \tilde{\omega}$, les variables y_2^2 et y_3^2 dans un même symbole y_ω^2 : dans chaque partition, les variables seront alors définies de façon homogène, ce qui permettra de leur appliquer facilement une abstraction numérique. ○

Nous avons donc, comme première étape de notre abstraction, partitionné les états selon un critère de taille sur leurs tampons. Nous allons maintenant pouvoir abstraire chaque partition en regroupant les entrées anciennes de chaque tampon. Grâce au partitionnement, nous obtiendrons des états numériques homogènes : nous pourrons alors les abstraire dans les domaines numériques usuels. De plus, cette homogénéité simplifiera grandement l'expression des opérateurs abstraits.

Chapitre 7

Abstraction des tampons

La présence des tampons constitue le principal obstacle à l'abstraction dans les modèles TSO ou PSO. Les principales difficultés proviennent de leur taille non bornée, qui impose une construction abstraite dédiée pour pouvoir les représenter de façon calculable, et des variations dynamiques et non déterministes de cette taille, que la représentation devra prendre en compte.

Toutefois, nous avons montré précédemment que l'aspect non déterministe est déjà correctement pris en charge dans le domaine concret, par la construction du graphe produit et à l'aide du lemme 5.1. Ce non-déterminisme est réduit à l'expression de l'opérateur $[\![flush]\!]$ et à son application choisie sur certaines arêtes.

Il ne nous reste donc plus qu'à donner une représentation abstraite des tampons qui permette de tenir compte du caractère à la fois non borné et dynamique de leur taille. Pour cela, nous nous baserons sur des domaines déjà proposés pour représenter des structures de données, en particulier des tableaux, que nous adapterons aux spécificités des tampons.

7.1 Condensé des tampons

La première abstraction que nous développerons sera la condensation des tampons. Cette technique (en anglais *summarisation*) a pour origine la représentation de tableaux (éventuellement alloués dynamiquement) dans des abstractions non relationnelles en rassemblant leurs cases dans un même symbole sur lequel les affectations se traduisent en *mises à jour faibles* [CWZ90, BCC+02, RSW04, GRS05]. Gopan *et al.* [GDD+04] ont par la suite formalisé ce concept pour représenter des paquets non bornés de variables y compris dans des domaines relationnels, en donnant une expression sûre des opérateurs abstraits induits.

7.1.1 Domaines non relationnels et mises à jour faibles

Dans le cas de l'analyse non relationnelle, la condensation peut se décrire de façon simple et intuitive par une distinction entre mise à jour forte (*strong update*) et mise à jour faible (*weak update*) [CWZ90, BCC+02, RSW04, GRS05].

Considérons par exemple les deux états suivants :

$$x \mapsto 0, y \mapsto 1, t_1 \mapsto 3, t_2 \mapsto 4$$
$$x \mapsto 0, y \mapsto 2, t_2 \mapsto 5, t_2 \mapsto 6$$

La représentation non relationnelle la plus concrète consiste simplement à associer, à chaque variable, l'ensemble des valeurs qu'elle peut prendre :

$$x \mapsto \{0\}, y \mapsto \{1,2\}, t_1 \mapsto \{3,5\}, t_2 \mapsto \{4,6\} \tag{7.1}$$

Les domaines non relationnels permettent par la suite de représenter chacun de ces ensembles de valeurs de façon bornée. Nous nous contenterons, pour cet exemple, de l'abstraction « non relationnelle concrète ».

Supposons maintenant que les variables t_1 et t_2 représentent deux cases d'un tableau. Pour réduire la taille de la représentation, plutôt que de distinguer ces deux cases, nous pouvons les regrouper (ou les *condenser*) en un seul symbole t auquel nous associons l'ensemble des valeurs présentes dans le tableau dans un des états considérés :

$$x \mapsto \{0\}, y \mapsto \{1,2\}, t \mapsto \{3,4,5,6\}$$

Lorsqu'un programme exécute une instruction `x = 3` depuis les états ainsi décrits, il affecte depuis l'état de départ `3` à `x` sans modifier le reste :

$$x \mapsto \{3\}, y \mapsto \{1,2\}, t \mapsto \{3,4,5,6\}$$

Lorsque l'instruction exécutée est de la forme `x = y`, x peut alors prendre dans les états résultants toutes les valeurs que y prenait dans les états d'origine :

$$x \mapsto \{1,2\}, y \mapsto \{1,2\}, t \mapsto \{3,4,5,6\}$$

Enfin, l'opération est la même lorsque l'instruction est `x = t[1]` : toutes les valeurs possibles de t_1 deviennent des valeurs possibles de x. Dans l'élément abstrait, on ne distingue plus t_1 et t_2 : on affecte alors à x l'ensemble des valeurs de la variable de condensation t.

$$x \mapsto \{3,4,5,6\}, y \mapsto \{1,2\}, t \mapsto \{3,4,5,6\}$$

On dit que ces opérations se traduisent par des *mises à jour fortes* de x : lorsque cette variable se voit affecter une expression, on remplace simplement dans l'élément initial ses valeurs possibles par les valeurs possibles de l'expression. Ces mises à jour fortes permettent d'obtenir une abstraction sûre des résultats concrets de l'opération d'affectation lorsque la variable correspondante ne représente qu'elle-même.

En revanche, lorsque la variable affectée est condensée, les mises à jour fortes ne sont plus correctes. Par exemple, exécuter depuis l'élément de l'équation (7.1) l'instruction `t[1] = 7` produit le résultat suivant :

$$x \mapsto \{0\}, y \mapsto \{1,2\}, t_1 \mapsto \{7\}, t_2 \mapsto \{4,6\}$$

Cet élément se condense ainsi :

$$x \mapsto \{0\}, y \mapsto \{1,2\}, t \mapsto \{4,6,7\}$$

Or la mise à jour forte $t \leftarrow \{7\}$ conduirait à cet élément condensé :

$$x \mapsto \{0\}, y \mapsto \{1,2\}, t \mapsto \{7\}$$

Il manque donc des valeurs possibles pour t. En effet, la mise à jour forte oublie qu'à t étaient associées non seulement les valeurs concrètes de `t[1]`, qui vient de se voir affecter `7`, mais aussi celles de `t[2]`, qui n'ont pas été modifiées par l'instruction. Comme t ne fait plus de distinction entre les deux variables, sa mise à jour ne peut simplement remplacer ses valeurs possibles par celle de l'expression considérée, mais doit les ajouter aux valeurs initialement associées à ce symbole, considérant qu'il pouvait s'agir de valeurs de `t[2]` dans le concret.

Ainsi, la traduction d'une expression de la forme `t[i] = e` se traduit par une *mise à jour faible* de la variable condensée t représentant `t[i]`, qui peut elle-même s'écrire à l'aide d'une union entre anciennes et nouvelles valeurs : $t \leftarrow t \cup e$.

7.1.2 Extension aux domaines relationnels

Dans le cas général des domaines relationnels, la condensation consiste toujours à regrouper plusieurs variables dans un seul symbole, réduisant ainsi d'autant la dimension du domaine numérique sous-jacent. Considérons par exemple deux états accessibles dans un domaine de dimension 3 :

$$(x, y, z) \in \{(1, 2, 3); (4, 5, 6)\}$$

Nous pouvons regrouper les deux variables x et y en un seul symbole v_{xy}. Cette pseudo-variable (ou variable condensée) aura alors pour valeurs possibles, pour chaque état, la valeur de x ou celle de y. Nous obtenons alors les états condensés suivants :

$$(v_{xy}, z) \in \{(1, 3); (2, 3); (4, 6); (5, 6)\}$$

Avec la réduction de la dimension du domaine vient encore une fois une inévitable perte de précision : v_{xy} ne fait plus de différence entre x et y dans son ensemble de valeurs. Les états suivants, absents de l'ensemble concret de départ, sont alors aussi décrits par la représentation condensée :

$$(x, y, z) \in \{(1, 1, 3); (2, 2, 3); (2, 1, 3); (4, 4, 6); (5, 5, 6); (5, 4, 6)\}$$

Lors de l'abstraction des opérateurs concrets, la traduction directe des opérations d'affectation vers leur équivalent condensé, même en produisant éventuellement des mises à jour faibles, ne suffit plus. En effet, appliquons l'opérateur $[\![z \leftarrow y]\!]$ à notre exemple concret de départ. Un des états résultants est $(x, y, z) = (1, 2, 2)$, qui après condensation produit en particulier l'état abstrait $(v_{xy}, z) = (1, 2)$.

Or l'application directe de l'opération $[\![z \leftarrow v_{xy}]\!]$ à l'état condensé initial produit le résultat suivant :

$$(v_{xy}, z) \in \{(1, 1); (2, 2); (4, 4); (5, 5)\}$$

Cet ensemble ne contient pas l'état condensé $(1, 2)$: la traduction immédiate de l'affectation n'est pas sûre. Elle oublie que v_{xy} ne représente qu'une des valeurs possibles des variables initiales x et y, mais que celles-ci ne sont pas nécessairement toutes identiques. v_{xy} n'est donc a priori pas égal à z après application de l'opérateur.

Remarquons que z ne fait pas partie des variables condensées : ici, il ne s'agit donc pas d'un problème de mise à jour forte ou faible. Ainsi, l'union entre l'élément condensé initial et son résultat après affectation ne contient toujours pas l'état $(1, 2)$ considéré.

Remarquons également que ce problème n'apparaissait pas avec l'abstraction non relationnelle, parce qu'on ne considérait que l'ensemble des valeurs possibles pour chaque variable, sans relation entre ces ensembles. On n'inférait donc pas l'invariant (incorrect) $v_{xy} = z$ après l'affectation, ne gardant que l'information que ces deux variables pouvaient prendre leurs valeurs dans le même ensemble (ce qui est correct).

Formalisation

Pour formaliser sûrement ces domaines et les opérateurs abstraits associés, Gopan *et al.* [GDD+04] ont alors proposé d'ajouter aux opérations usuelles deux opérateurs, *fold* et *expand*.

$[\![\text{fold}\, x, y]\!]X$ est l'opération de condensation à proprement parler : y est supprimé des dimensions de l'élément concret X, et pour chaque état de X, sa valeur est ajoutée comme valeur possible de x (le reste de l'état n'étant pas modifié).

$[\![\text{expand}\, x, y]\!]X$ est l'opération duale qui ajoute une nouvelle variable y à X, dont les valeurs possibles sont, pour chaque état, les valeurs possibles de x lorsque le reste de l'état ne change pas. Autrement dit, y hérite de toutes les contraintes de x, excepté la contrainte d'être égal à

$$\mathscr{D} \triangleq \wp(Var \to \mathbb{V})$$
$$\mathscr{D}' \triangleq \wp(Var \setminus \{y\} \to \mathbb{V})$$

$$[\![\text{fold}\, x, y]\!] : \mathscr{D} \to \mathscr{D}'$$
$$[\![\text{fold}\, x, y]\!]X \triangleq \left\{ S' \in \mathscr{D}' \;\middle|\; \exists S \in X, \begin{pmatrix} (S'(x) = S(x) \vee S'(x) = S(y)) \\ \wedge \forall z \neq x, y, S'(z) = S(z) \end{pmatrix} \right\}$$

$$[\![\text{expand}\, x, y]\!] : \mathscr{D}' \to \mathscr{D}$$
$$[\![\text{expand}\, x, y]\!]X' \triangleq \left\{ S \in \mathscr{D} \;\middle|\; \exists S'_1, S'_2 \in X', \begin{pmatrix} S(x) = S'_1(x) \wedge S(y) = S'_2(y) \\ \wedge \forall z \neq x, S(z) = S'_1(z) = S'_2(z) \end{pmatrix} \right\}$$

FIGURE 7.1 – Opérateurs *fold* et *expand* pour la condensation.

x. Ainsi, dans notre exemple de condensation, l'ensemble des états décrits par l'état abstrait est obtenu par l'opération $[\![\text{expand}\, v_{xy}, y]\!]$ (puis renommage de v_{xy} en x).

La figure 7.1 formalise la définition de ces opérateurs depuis et vers un domaine concret. On notera Var un ensemble quelconque de variables dont x et y.

À partir de *fold* et *expand*, Gopan *et al.* [GDD+04] formulent alors des constructions génériques des opérateurs intervenant sur des variables éventuellement condensées. Nous ne donnerons pas leur formulation générale ici, nous contentant de notre cas d'application particulier.

Enfin, les mêmes auteurs montrent que *fold* et *expand*, pour deux variables données, forment un plongement de Galois :

$$\mathscr{D} \underset{[\![\text{fold}\, x,y]\!]}{\overset{[\![\text{expand}\, x,y]\!]}{\longleftrightarrow}} \mathscr{D}'$$

7.1.3 Application aux tampons

Considérons maintenant un ensemble d'états concrets de PSO, partitionné selon le critère δ défini en section 6.2. Dans chaque partition où celles-ci sont définies, c'est-à-dire les partitions pour lesquelles $\delta(.)(x, T) = \tilde{\omega}$, nous regrouperons les variables $x^T_2, \dots, x^T_N$ en une variable condensée x^T_ω.

Remarquons que l'entrée la plus récente des tampons est gardée séparément dans un symbole non condensé : puisqu'elle est utilisée pour la lecture de la variable correspondante, il est nécessaire de la distinguer pour la précision de cette opération.

La figure 7.2 formalise le domaine $\mathscr{D}^{\mathfrak{sum}}$ correspondant à cette condensation. Suivant la formalisation de Gopan *et al.* [GDD+04], nous procédons en plusieurs étapes : après avoir défini les ensembles d'états et le domaine condensés sous-jacents, nous spécifions la condensation choisie par une fonction de condensation $\Phi_{\mathfrak{sum}}$ associant les variables condensées aux variables concrètes. De cette fonction dérive un ensemble $\Pi_{\Phi_{\mathfrak{sum}}}$ de projections orthogonales, qui transforment les états concrets en états condensés. Ces projections permettent à leur tour de définir le plongement de Galois $\mathscr{D} \underset{\alpha_{\mathfrak{sum}}}{\overset{\gamma_{\mathfrak{sum}}}{\longleftrightarrow}} \mathscr{D}^{\mathfrak{sum}}$. $\uplus$ note l'union disjointe de deux ensembles.

Dans le domaine partitionné, la condensation se fait partition par partition : le domaine résultant $\mathscr{D}^\sharp$ est formalisé en figure 7.3.

$$
\begin{aligned}
\mathit{Var} &\triangleq \mathit{Shared} \uplus \mathit{Local} \uplus \left\{ x_n^T \;\middle|\; x \in \mathit{Shared}, T \in \mathit{Thread}, n \in \mathbb{N}^* \right\} \\
\mathit{Var}^{\mathfrak{sum}} &\triangleq \mathit{Shared} \uplus \mathit{Local} \uplus \left\{ x_1^T, x_\omega^T \;\middle|\; x \in \mathit{Shared}, T \in \mathit{Thread} \right\} \\
\mathscr{S}^{\mathfrak{sum}} &\triangleq \mathit{Var}^{\mathfrak{sum}} \to \mathbb{V} \\
\mathscr{D}^{\mathfrak{sum}} &\triangleq \wp(\mathscr{S}^{\mathfrak{sum}})
\end{aligned}
$$

(a) États et domaine condensés.

$$
\Phi_{\mathfrak{sum}} : \mathit{Var} \to \mathit{Var}^{\mathfrak{sum}}
$$

$$
\Phi_{\mathfrak{sum}}(v) \triangleq \begin{cases} v & \text{si } v \in \mathit{Shared} \uplus \mathit{Local} \\ v & \text{si } \exists x \in \mathit{Shared}, T \in \mathit{Thread}, v = x_1^T \\ x_\omega^T & \text{si } \exists x \in \mathit{Shared}, T \in \mathit{Thread}, n > 1, v = x_n^T \end{cases}
$$

$$
\Pi_{\Phi_{\mathfrak{sum}}} \triangleq \left\{ \pi : \mathscr{S} \to \mathscr{S}^{\mathfrak{sum}} \;\middle|\; \forall v^{\mathfrak{sum}} \in \mathit{Var}^{\mathfrak{sum}}, \exists v \in \Phi_{\mathfrak{sum}}^{-1}(v^{\mathfrak{sum}}), \forall S \in \mathscr{S}, v \in \mathfrak{D}_S \implies \pi(S)(v^{\mathfrak{sum}}) = S(v) \right\}
$$

(b) Fonction de condensation et projections orthogonales.

$$
\mathscr{D} \underset{\alpha_{\mathfrak{sum}}}{\overset{\gamma_{\mathfrak{sum}}}{\leftrightarrows}} \mathscr{D}^{\mathfrak{sum}}
$$

$$
\begin{aligned}
\gamma_{\mathfrak{sum}}(X^{\mathfrak{sum}}) &= \left\{ S \in \mathscr{D} \;\middle|\; \forall \pi \in \Pi_{\Phi_{\mathfrak{sum}}}, \pi(S) \in X^{\mathfrak{sum}} \right\} \\
\alpha_{\mathfrak{sum}}(X) &= \left\{ \pi(S) \;\middle|\; S \in X, \pi \in \Pi_{\Phi_{\mathfrak{sum}}} \right\}
\end{aligned}
$$

(c) Plongement de Galois.

FIGURE 7.2 – Condensation des tampons pour une représentation bornée.

$$\mathscr{D}^\sharp \triangleq \mathscr{B}^\flat \to \mathscr{D}^{\mathtt{sum}}$$

$$\mathscr{D}^\flat \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} \mathscr{D}^\sharp$$
$$\gamma_S(X^\sharp) \triangleq \lambda b^\flat.\gamma_{\mathtt{sum}}(X^\sharp(b^\flat))$$
$$\alpha_S(X) \triangleq \lambda b^\flat.\alpha_{\mathtt{sum}}(X(b^\flat))$$

$$\mathscr{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathscr{D}^\sharp$$

$$\gamma \triangleq \gamma_P \circ \gamma_S$$
$$\gamma(X^\sharp) = \left\{S \in \mathscr{S} \mid S \in \gamma_{\mathtt{sum}}(X^\sharp(\delta(S)))\right\}$$

$$\alpha \triangleq \alpha_S \circ \alpha_P$$
$$\alpha(X) = \lambda b^\flat.\alpha_{\mathtt{sum}}\left(\left\{S \in X \mid \delta(S) = b^\flat\right\}\right)$$

FIGURE 7.3 – Domaine avec condensation et partitionnement.

Les opérateurs abstraits sur $\mathscr{D}^\sharp$ sont exprimés en figure 7.4. Pour les définir, nous recourrons à une construction intermédiaire $(\!|.|\!)$, évaluée dans une partition donnée, qui renvoie des couples composés d'une partition de destination dans $\mathscr{B}^\flat$ et d'une image propre dans $\mathscr{D}^{\mathtt{sum}}$. Les opérateurs abstraits complets appliquent alors cette construction partition par partition, unissant les images. La construction des images dans $\mathscr{D}^{\mathtt{sum}}$ par $(\!|.|\!)$ provient des opérateurs abstraits formulés par Gopan *et al.* sur les domaines avec condensation.

Exemple 7.1.
Montrons comment calculer le résultat de quelques opérateurs depuis l'état initial simple suivant :

$$x^{mem} \mapsto 0$$

Nous omettrons les index des partitions pour plus de clarté. Après avoir exécuté $[\![x \leftarrow 1]\!]_1$, nous obtenons l'élément suivant :

$$x^{mem} \mapsto 0, x_1^1 \mapsto 1$$

Il n'y a là pas de différence avec l'opérateur concret : la condensation n'intervient pas. Exécutons maintenant $[\![x \leftarrow 2]\!]_1$: l'ancienne entrée x_1^1 va former une entrée x_ω^1

$$x^{mem} \mapsto 0, x_1^1 \mapsto 2, x_\omega^1 \mapsto 1$$

Le calcul de cet état ne faisait pas encore appel à la condensation à proprement parler : x_ω^1 ne représente qu'une seule variable concrète. Toutefois, une fois l'état calculé, cette information est perdue : l'élément abstrait obtenu décrit par exemple aussi l'état suivant, qui n'est pas accessible dans le concret avec la suite d'instructions effectuée.

$$x^{mem} \mapsto 0, x_1^1 \mapsto 2, x_2^1 \mapsto 1, x_3^1 \mapsto 1$$

$$\forall T \in \mathit{Thread}, \llparenthesis . \rrparenthesis_T : (\mathscr{B}^\flat \times \mathscr{D}^{\mathtt{sum}}) \to \wp(\mathscr{B}^\flat \times \mathscr{D}^{\mathtt{sum}})$$

$$\llparenthesis r \leftarrow e \rrparenthesis^\sharp_T(b^\flat, X^{\mathtt{sum}}) = \left\{b^\flat, \llbracket r \leftarrow \{\!|e|\!\} X^{\mathtt{sum}} \rrbracket X^{\mathtt{sum}}\right\}$$

$$\llparenthesis x \leftarrow e \rrparenthesis^\sharp_T(b^\flat, X^{\mathtt{sum}}) = \begin{cases} \left\{ \begin{array}{l} b^\flat[x^T \mapsto \tilde{1}], \\ \quad \llbracket x^T_1 \leftarrow \{\!|e|\!\} X^{\mathtt{sum}} \rrbracket \circ \\ \quad \llbracket \mathrm{add}\, x^T_1 \rrbracket X^{\mathtt{sum}} \end{array} \right\} & \text{si } b^\flat(x^T) = \tilde{0} \\ \left\{ \begin{array}{l} b^\flat[x^T \mapsto \tilde{\omega}], \\ \quad \llbracket x^T_1 \leftarrow \{\!|e|\!\} X^{\mathtt{sum}} \rrbracket \circ \\ \quad \llbracket x^T_\omega \leftarrow x^T_1 \rrbracket \circ \\ \quad \llbracket \mathrm{add}\, x^T_\omega \rrbracket X^{\mathtt{sum}} \end{array} \right\} & \text{si } b^\flat(x^T) = \tilde{1} \\ \left\{ \begin{array}{l} b^\flat, \\ \quad \llbracket x^T_1 \leftarrow \{\!|e|\!\} X^{\mathtt{sum}} \rrbracket \circ \\ \quad \llbracket \mathrm{fold}\, x^T_\omega, x^{(temp)}_2 \rrbracket \circ \\ \quad \llbracket x^{(temp)}_2 \leftarrow x^T_1 \rrbracket \circ \\ \quad \llbracket \mathrm{add}\, x^{(temp)}_2 \rrbracket X^{\mathtt{sum}} \end{array} \right\} & \text{si } b^\flat(x^T) = \tilde{\omega} \end{cases}$$

$$\llparenthesis \mathtt{mfence} \rrparenthesis^\sharp_T(b^\flat, X^{\mathtt{sum}}) = \begin{cases} \left\{b^\flat, X^{\mathtt{sum}}\right\} & \text{si } \forall x \in \mathit{Shared}, b^\flat(x^T) = \tilde{0} \\ \varnothing & \text{sinon} \end{cases}$$

$$\llparenthesis \mathrm{flush}\, x \rrparenthesis^\sharp_T(b^\flat, X^{\mathtt{sum}}) = \begin{cases} \varnothing & \text{si } b^\flat(x^T) = \tilde{0} \\ \left\{ \begin{array}{l} b^\flat[x^T \mapsto \tilde{0}], \\ \quad \llbracket \mathrm{drop}\, x^T_1 \rrbracket \circ \\ \quad \llbracket x^{mem} \leftarrow x^T_1 \rrbracket X^{\mathtt{sum}} \end{array} \right\} & \text{si } b^\flat(x^T) = \tilde{1} \\ \left\{ \begin{array}{l} b^\flat[x^T \mapsto \tilde{1}], \\ \quad \llbracket \mathrm{drop}\, x^T_\omega \rrbracket \circ \\ \quad \llbracket x^{mem} \leftarrow x^T_\omega \rrbracket X^{\mathtt{sum}}; \\ b^\flat, \\ \quad \llbracket \mathrm{drop}\, x^{(temp)} \rrbracket \circ \\ \quad \llbracket x^{mem} \leftarrow x^{(temp)} \rrbracket \circ \\ \quad \llbracket \mathrm{expand}\, x^T_\omega, x^{(temp)} \rrbracket X^{\mathtt{sum}} \end{array} \right\} & \text{si } b^\flat(x^T) = \tilde{\omega} \end{cases}$$

$$\forall T \in \mathit{Thread}, \llbracket . \rrbracket^\sharp_T : \mathscr{D}^\sharp \to \mathscr{D}^\sharp$$

$$\llbracket ins \rrbracket^\sharp_T X^\sharp = \lambda b^\flat . \bigsqcup^{\mathtt{sum}}_{\substack{\exists b'^\flat \in \mathscr{B}^\flat, \\ (b^\flat, Y^{\mathtt{sum}}) \in \\ \llparenthesis ins \rrparenthesis^\sharp_T(b'^\flat, X^\sharp(b'^\flat))}} Y^{\mathtt{sum}}$$

FIGURE 7.4 – Sémantique abstraite avec condensés.

Si nous exécutons alors $[\![x \leftarrow 3]\!]_1$, la condensation intervient : x^1_ω peut prendre deux valeurs possibles. On obtient alors un ensemble de deux états.

$$x^{mem} \mapsto 0, x^1_1 \mapsto 3, x^1_\omega \mapsto 2$$
$$x^{mem} \mapsto 0, x^1_1 \mapsto 3, x^1_\omega \mapsto 1$$

Ces deux états ne sont pas totalement indépendants : la fraction non condensée coïncide. Ils décrivent alors des états concrets où x^1_ω se concrétise en plusieurs variables de tampons avec des valeurs différentes, par exemple :

$$x^{mem} \mapsto 0, x^1_1 \mapsto 3, x^1_2 \mapsto 1, x^1_3 \mapsto 2, x^1_4 \mapsto 1$$

Le transfert $[\![\text{flush}\, x]\!]_1$ doit alors considérer toutes les options possibles. Tout d'abord, il est possible que, dans chacun des états, x^1_ω ne décrive qu'une variable concrète x^1_2. Le résultat s'obtient alors simplement en la transférant dans x^{mem}.

$$x^{mem} \mapsto 2, x^1_1 \mapsto 3$$
$$x^{mem} \mapsto 1, x^1_1 \mapsto 3$$

L'opérateur doit également prendre en compte la possibilité d'états concrets où plus de deux variables étaient condensées. Dans ces états, l'entrée la plus ancienne est transférée : elle peut valoir 1 ou 2. Les autres entrées peuvent également valoir 1 ou 2, et ce indépendamment de celle qui est transférée, comme nous l'avons vu. On obtient donc les états suivants :

$$x^{mem} \mapsto 1, x^1_1 \mapsto 3, x^1_\omega \mapsto 1$$
$$x^{mem} \mapsto 1, x^1_1 \mapsto 3, x^1_\omega \mapsto 2$$
$$x^{mem} \mapsto 2, x^1_1 \mapsto 3, x^1_\omega \mapsto 1$$
$$x^{mem} \mapsto 2, x^1_1 \mapsto 3, x^1_\omega \mapsto 2$$

Nous avons montré dans cet exemple les cas les plus complexes pour le calcul des opérateurs abstraits. Les autres formules de la figure 7.4 sont plus directes. Enfin, la preuve du théorème de sûreté détaillera plus formellement ces cas pour montrer leur correction. ○

7.2 Abstraction numérique

La dernière étape consiste à abstraire le contenu des partitions, de $\mathscr{D}^{\mathtt{sum}}$, à l'aide de domaines numériques. Le partitionnement prend alors tout son sens : chaque partition est en effet un ensemble d'états condensés de mêmes dimensions (les mêmes variables sont définies dans chaque état d'une partition donnée). Nous pouvons alors abstraire directement ces états par les domaines numériques usuels.

Formellement, nous remplaçons $\mathscr{D}^{\mathtt{sum}}$ par un domaine numérique condensé $\mathscr{D}^{\mathtt{num}}$. Gopan *et al.* [GDD+04] montrent qu'un domaine doit simplement implémenter les opérations *fold* et *expand* pour être compatible avec la condensation. Ils proposent également des implémentations pour des domaines numériques usuels : intervalles, polyèdres et zones.

Siegel et Simon [SS12] ont par la suite étendu ce travail sur la condensation et proposé une construction générique de *fold* et *expand* fondée sur des opérations standard sur les domaines numériques. Cette construction est présentée en figure 7.5. $[\![\text{swap}\, x, y]\!]$ échange les valeurs des variables x et y.

$$[\![\text{fold}\, x, y]\!]X = [\![\text{drop}\, y]\!](X \sqcup [\![x \leftarrow y]\!]X)$$
$$[\![\text{expand}\, x, y]\!]X = [\![\text{add}\, y]\!]X \sqcap ([\![\text{swap}\, x, y]\!] \circ [\![\text{add}\, y]\!]X)$$

FIGURE 7.5 – Expression de *fold* et *expand* à partir d'opérations de base.

Grâce à cette construction alternative, nous disposons d'un vaste choix d'abstractions pour cette dernière étape. Nous obtenons alors une abstraction finale paramétrée par le choix de notre domaine numérique : nous pouvons ainsi moduler ce paramètre pour obtenir la puissance nécessaire pour exprimer les invariants à inférer. Là où la condensation nous permettait de représenter des états individuels de taille non bornée, nous pouvons maintenant également représenter des ensembles d'états dont le cardinal est lui-même éventuellement infini. Les opérateurs abstraits gardent la même formulation qu'en figure 7.4.

7.3 Sûreté

Théorème 7.1.
Les opérateurs abstraits définis en figure 7.4 sont sûrs :

$$\forall X^\sharp \in \mathscr{D}^\sharp, [\![ins]\!]_T \circ \gamma(X^\sharp) \subseteq \gamma \circ [\![ins]\!]^\sharp_T(X^\sharp)$$

PREUVE.
Notre domaine est construit en assemblant deux abstractions : le partitionnement d'états et la condensation. Par construction du partitionnement, la sûreté des opérateurs découle de la sûreté de la construction des images condensées et de leur classement dans les bonnes partitions.

La sûreté de la construction des images se déduit dans la plupart des cas directement des résultats de Gopan *et al.* [GDD+04]. Nous montrerons spécifiquement la correction des cas restants. Il nous reste alors à prouver que $(\!|.|\!)^\sharp_T$ partitionne correctement les images ainsi obtenues.

L'opérateur de lecture n'implique pas de variable condensée : sa sémantique est directe et quasiment identique à la sémantique concrète. Les tampons concrets n'étant pas modifiés, l'indice de la partition ne l'est pas non plus. L'opérateur $[\![\texttt{mfence}]\!]^\sharp$ est également une traduction directe de l'opérateur concret.

L'opérateur d'écriture en mémoire partagée distingue deux cas : si la variable concernée est absente du buffer correspondant, la condensation n'intervient pas. Si elle est présente exactement une fois, la variable x^T_ω apparaît, mais il n'y a toujours pas de condensation en jeu : cette variable représente ici exactement la variable concrète x^T_2. Dans ces deux cas, l'opérateur abstrait est semblable à sa version concrète, et la seule partition de destination de l'image est obtenue en ajoutant 1 au tampon de x^T, dont la longueur abstraite passe alors à $\tilde{\omega}$.

Dans le dernier cas, la variable est présente plus d'une fois. La condensation entre alors en jeu. L'image numérique est obtenue en traduisant chaque étape de l'équation concrète de la figure 5.4, d'après les résultats de Gopan *et al.* [GDD+04]. Les opérations $[\![x^T_1 \leftarrow (\!|e|\!)_T]\!]$, $[\![x^T_2 \leftarrow x^T_1]\!]$ and $[\![\text{add}\, x^T_{\#^T_x S+1}]\!]$ sont directement traduites vers les opérations abstraites correspondantes, condensation inclue. x^T_ω étant déjà présente dans X^{sum}, $[\![\text{add}\, x^T_{\#^T_x S+1}]\!]$ devient l'identité. Considérons maintenant les opérations de la forme $[\![x^T_{i+1} \leftarrow x^T_i]\!]$, pour $i \geq 2$. Elles se traduisent par condensation en

$$[\![\text{drop}\, x']\!] \circ [\![\text{fold}\, x^T_\omega, x'']\!] \circ [\![x'' \leftarrow x']\!] \circ [\![\text{add}\, x'']\!] \circ [\![\text{expand}\, x^T_\omega, x']\!]$$

Après application de $[\![x'' \leftarrow x']\!]$, x'' et x' sont interchangeables : ils partagent exactement les mêmes contraintes. Nous pouvons alors réécrire la formule précédente en :

$$[\![\mathrm{drop}\, x'']\!] \circ [\![\mathrm{fold}\, x^T_\omega, x']\!] \circ [\![x'' \leftarrow x']\!] \circ [\![\mathrm{add}\, x'']\!] \circ [\![\mathrm{expand}\, x^T_\omega, x']\!]$$

$[\![\mathrm{drop}\, x'']\!]$ et $[\![\mathrm{fold}\, x^T_\omega, x']\!]$ étant indépendants, nous pouvons alors les faire commuter, obtenant :

$$[\![\mathrm{fold}\, x^T_\omega, x']\!] \circ [\![\mathrm{drop}\, x'']\!] \circ [\![x'' \leftarrow x']\!] \circ [\![\mathrm{add}\, x'']\!] \circ [\![\mathrm{expand}\, x^T_\omega, x']\!]$$

Or $[\![\mathrm{drop}\, x'']\!] \circ [\![x'' \leftarrow x']\!] \circ [\![\mathrm{add}\, x'']\!]$ se réduit trivialement en l'identité. Notre formule devient alors $[\![\mathrm{fold}\, x^T_\omega, x']\!] \circ [\![\mathrm{expand}\, x^T_\omega, x']\!]$, dont Siegel et Simon [SS12] montrent également qu'elle équivaut à l'identité. Toutes les opérations de la forme $[\![x^T_{i+1} \leftarrow x^T_i]\!]$ pour $i \geq 2$ étant ainsi réduites, nous obtenons bien finalement l'image formulée en figure 7.4.

Quant à la partition, elle n'est pas modifiée : ajouter un élément à un tampon qui en contient au moins deux conduit toujours à un tampon qui en contient au moins deux.

L'opération de transfert est la plus complexe. Sur les partitions pour lesquelles la variable n'est présente qu'une fois dans le tampon, la traduction est directe et sans condensation, comme pour les opérations précédentes.

En revanche, lorsqu'un transfert est effectué depuis une partition où la variable est présente plus d'une fois dans le tampon, l'élément abstrait source représente à la fois des états concrets où cette variable est présente deux fois (donc une fois dans les éléments concrets après transfert) et plus de deux fois (donc plus d'une fois après transfert).

Par conséquent, dans le cas $\tilde{\omega}$, l'opérateur de transfert abstrait doit construire deux images, en considérant les deux possibilités respectives, et envoyer ces résultats dans deux partitions différentes. Le contenu de chacune de ces partitions tel que formulé en figure 7.4 s'ensuit alors directement des résultats de Gopan *et al.* [GDD+04]. □

7.4 Conclusion

L'abstraction proprement dite des tampons constitue le cœur de notre contribution à l'interprétation abstraite dans les modèles mémoire faibles. Elle se fait partition par partition, en condensant les entrées anciennes en un seul symbole, puis en appliquant un domaine numérique aux ensembles d'états obtenus.

Il est possible d'augmenter le nombre n d'entrées distinguées en tête de chaque tampon. x_ω condensera alors les entrées $x_{n+1}, x_{n+2}, \ldots$ L'expression des opérateurs abstraits ne change quasiment pas. On gagne ainsi en précision ; en revanche, on augmente le nombre de partitions et la taille de chacune, ce qui risque d'avoir un impact non négligeable sur les performances. Les résultats expérimentaux présentés au chapitre suivant semblent montrer que la précision additionnelle n'est pas nécessaire sur les exemples typiques, ce pourquoi nous ne la développerons pas davantage. Le lecteur pourra toutefois garder à l'esprit ce paramétrage possible de l'analyse ; applicable par exemple à un point de contrôle où une meilleure précision est nécessaire et où on sait pouvoir borner à n le nombre d'écritures en attente.

Chapitre 8

Résultats expérimentaux

La pertinence d'une méthode d'analyse statique ne peut pas se justifier uniquement par des considérations théoriques : elle doit être validée par l'expérience, à la fois en termes de précision sur des cas pratiques et de performances réelles. Pour valider notre analyse, nous l'avons donc implémentée dans un analyseur expérimental dédié.

8.1 Contexte et objectifs

8.1.1 Implémentation de l'analyseur

Notre analyseur a été écrit en OCaml. Son code est disponible à l'adresse suivante :

`https://github.com/thizanne/cormoran`

Le calcul de point fixe est programmé à l'aide de la bibliothèque ocamlgraph [CFS07], à laquelle nous avons ajouté deux modules pour calculer des ordres topologiques faibles et pour mettre en œuvre des stratégies d'itération basées sur ces structures. La stratégie d'itération choisie pour l'analyse monolithique du graphe-produit correspond à la méthode générale décrite par Bourdoncle [Bou93] : on procède par décompositions successives du graphe en composantes fortement connexes, en retirant à chaque étape la tête des composants obtenus.

Les abstractions (logico-)numériques pour $\mathscr{D}^{\mathtt{sum}}$ sont implémentées à l'aide des domaines relationnels fournis par les bibliothèques Apron [JM09] et Bddapron [Jea09]. Nous avons utilisé l'implémentation des polyèdres et des octogones par Apron, et des diagrammes de décision binaires composés de ces deux mêmes abstractions numériques par Bddapron (cf. sections 2.4 et 2.5).

Lorsque ces opérations n'étaient pas fournies, nous avons utilisé les formules de Siegel et Simon [SS12] pour implémenter *fold* et *expand*.

8.1.2 Expériences et objectifs

Pour valider les méthodes qui sous-tendent notre analyseur, nous souhaitons tester à la fois sa précision et ses performances. Nous l'avons à cette fin comparé sur ces deux critères à des résultats récents de Dan *et al.* [DMVY14], qui s'attaquent au problème de la suppression de barrières.

8.1.3 Suppression de barrières

Rappelons qu'il suffit, pour n'importe quel programme exécuté dans TSO (ou PSO), d'ajouter une barrière après chaque instruction d'écriture en mémoire pour retrouver sa sémantique séquentiellement cohérente. Ces barrières ayant un coût à l'exécution (par les optimisations qu'elles interdisent), la suppression de barrières vise à en retirer un maximum, tout en préservant la validation d'une propriété cible vérifiée par le programme dans un modèle séquentiellement cohérent.

Étant donné un analyseur sûr, une méthode naïve mais correcte de suppression de barrières consiste simplement à retirer des barrières jusqu'à ce que l'analyse ne parvienne plus à vérifier la propriété cible (en essayant éventuellement d'autres combinaisons pour éviter de rester dans un minimum local). Il existe bien sûr des méthodes de recherche dédiées mieux adaptées à ce problème, parmi lesquelles celle de Dan *et al.* [DMVY14]

Leur outil est fondé sur une traduction source-à-source d'un programme dans TSO (ou PSO) vers un programme ayant une sémantique équivalente lorsqu'il est exécuté en cohérence séquentielle. Cette traduction fait apparaître explicitement les tampons (à travers des variables semblables à nos x_i^T) dans le code source. Les auteurs branchent alors sur ce programme image un système dédié à la suppression de barrières [MDVY14] basé sur l'analyseur statique ConcurInterproc [Jea13] et le solveur SMT Z3 [DMB08] afin de déterminer le nombre minimal de barrières nécessaires pour vérifier la propriété voulue.

8.1.4 Cibles des analyses

Nous disposons pour nos mesures d'une suite de programmes tests déjà utilisés pour l'évaluation de l'implémentation de Dan *et al.* [DMVY14].

Il s'agit de petits programmes à deux processus, dont les codes sources sont fournis en annexe, implémentant pour la plupart des algorithmes standards de programmation concurrente comme l'exclusion mutuelle ou les systèmes producteur-consommateur. La correction de ces programmes est exprimée par une propriété de sûreté, c'est-à-dire une condition qui doit être vérifiée à certains états de contrôle du programme (c'est-à-dire des produits de points de contrôles des différents processus). Par exemple, vérifier les algorithmes d'exclusion mutuelle revient prouver que la condition `false` est vérifiée à l'état de contrôle correspondant à l'entrée en section critique des deux processus ; ou de façon équivalente, que l'abstraction calculée à cette endroit est $\bot$.

Parce qu'ils mettent en œuvre des algorithmes de communication entre les processus, ces programmes vont fréquemment utiliser les opérations au comportement relâché du modèle mémoire. Il s'agit donc de bons tests de nos abstractions dédiées, qui devront modéliser avec suffisamment de précision ces opérations pour pouvoir vérifier que leurs usages sont corrects. Pour chacun, nous mesurons le temps, et la mémoire et le nombre de barrières nécessaires à sa vérification.

Notre analyseur n'implémentant aucune stratégie particulière de suppression de barrières, nous ne mesurons que les performances de l'analyse capable de vérifier un programme test avec leur sous-ensemble minimal. Ces barrières ont donc été placées manuellement. En revanche, l'outil de Gopan *et al.* détermine ces barrières minimales automatiquement par un algorithme spécifique : les chiffres de performance en temps et mémoire ne peuvent donc être comparés directement, leur outil effectuant un travail plus important de façon optimisée. Toutefois, considérant la méthode naïve qui se contente de recommencer l'analyse de zéro pour chaque ensemble de barrières, et en raison du faible nombre de combinaisons possibles sur les programmes de test, ces chiffres donnent un ordre de grandeur satisfaisant des performances obtenues.

Enfin, notre analyseur étant paramétré par le domaine numérique final, nous avons mesuré l'influence de ce paramètre en observant les performances relatives des différentes implémentations disponibles.

8.2 Résultats

Le Tableau de la figure 8.1 présente les résultats des différentes analyses.

Les mesures de notre analyseur ont été réalisées sur une machine disposant d'un processeur `Intel(R) Core(TM) i7-3612QM CPU @ 2.10GHz` et de 4Go de RAM. Dan *et al.* [DMVY14] ont utilisé une machine aux performances similaires pour leurs propres tests.

Le temps et la mémoire (ligne `Average resident set size`) ont été mesurés par la commande Unix `time`. L'entrée *Domaine* correspond aux paramètres suivants :

Oct : Octogones d'Apron

Poly : Polyèdres d'Apron

BddOct : Diagrammes de décision binaires et octogones de Bddapron

BddPoly : Diagrammes de décision binaires et polyèdres de Bddapron

AGT : *Abstraction Guided Translation* : meilleur résultat obtenu sur ce programme, dans PSO, par Dan *et al.* [DMVY14]

Lorsqu'aucun résultat n'est indiqué, le domaine correspondant ne permettait pas de mener à bien la vérification de la correction du test considéré. En particulier, les domaines purement numériques d'Apron ne permettant pas d'abstraire les calculs sur les booléens, ils sont insuffisants pour les programmes qui utilisent ces opérations. Nous avons essayé d'encoder les booléens dans les entiers, mais les encodages simples conduisent à des pertes de précision importantes (en particulier dues à l'élargissement) : la complexité additionnelle nécessaire pour une implémentation efficace n'était alors pas rentable par rapport à l'utilisation des pendants logico-numériques de Bddapron.

Enfin, certains programmes vérifiés par Dan *et al.* [DMVY14] sont absents du tableau : ils requièrent des constructions que notre implémentation ne gère pas, notamment le modèle d'ordonnancement coopératif des processus.

8.3 Interprétation

8.3.1 Comparaison avec l'état de l'art

Performances. Comme nous l'avons expliqué, obtenir une comparaison pertinente des performances relatives de notre méthode et de celle de Dan *et al.* [DMVY14] demanderait d'implémenter leur méthode évoluée de synthèse de barrières, ce qui sortait du cadre de notre étude. Les chiffres que nous obtenons montrent tout de même que notre analyseur est compétitif selon ce critère : nous analysons le programme avec le nombre minimal de barrières en un temps et avec une mémoire très inférieurs à ceux nécessaires pour que leur outil le détermine. Nous pouvons donc estimer que les performances de l'analyse proprement dite sont du même ordre de grandeur.

Précision. Sur ces exemples, la précision de l'analyse se mesure au nombre minimal de barrières nécessaires pour que l'outil parvienne à vérifier la correction du programme. Sur la plupart des tests, nous obtenons les mêmes résultats. Il s'agit alors probablement du nombre minimal de barrières pour que le programme soit effectivement correct dans le concret.

Pour deux d'entre eux (*Loop2 TLM* et *Queue*), notre analyse est cependant strictement plus précise : nous parvenons en effet à analyser ces algorithmes sans insérer aucune barrière, plutôt que respectivement 2 et 1. *Queue* par exemple correspond au programme 8.1 : en plus

Algorithme	**Domaine**	**Barr.**	**Temps**	**Mem**	**Domaine**	**Barr.**	**Temps**	**Mem**
Abp	Oct	-	-	-	Bdd+Oct	0	0.3	32
	Poly	-	-	-	Bdd+Poly	0	0.3	32
	AGT	0	6	167				
Bakery	Oct	-	-	-	Bdd+Oct	-	-	-
	Poly	-	-	-	Bdd+Poly	-	-	-
	AGT	4	3429	10951				
Concloop	Oct	2	0.19	38	Bdd+Oct	2	0.29	34
	Poly	2	0.24	37	Bdd+Poly	2	0.29	34
	AGT	2	6	504				
Dekker	Oct	4	23	52	Bdd+Oct	4	62	42
	Poly	4	22	50	Bdd+Poly	4	66	43
	AGT	4	121	1580				
Kessel	Oct	-	-	-	Bdd+Oct	4	4	33
	Poly	-	-	-	Bdd+Poly	4	4	34
	AGT	4	6	198				
Loop2 TLM	Oct	-	-	-	Bdd+Oct	**0**	4.3	34
	Poly	-	-	-	Bdd+Poly	**0**	4.2	34
	AGT	2	36	1650				
Peterson	Oct	4	1.53	39	Bdd+Oct	4	2.77	32
	Poly	4	1.53	39	Bdd+Poly	4	2.94	33
	AGT	4	20	901				
Queue	Oct	**0**	0.15	36.9	Bdd+Oct	**0**	0.70	34.3
	Poly	**0**	0.2	34.7	Bdd+Poly	**0**	0.31	33.3
	AGT	1	1	108				

FIGURE 8.1 – Résultats expérimentaux. Temps en sec, Mem en Mo.

```
1  /* Propriété à vérifier: aux labels (bp0; bp1), tail < h1 */
2
3  // Shared: head
4
5  thread /* ENQUEUE */ {
6      // Local : h1;
7
8      while true {
9              h1 = head;
10         bp0:
11             h1 = h1 + 1;
12             head = h1;
13             // mfence;
14         }
15 }
16
17 thread /* DEQUEUE */ {
18     // Local : tail, h2;
19
20     tail = head;
21     while true {
22             h2 = head;
23             while (tail >= h2) {
24                 h2 = head;
25             }
26         bp1:
27             tail = tail + 1;
28         }
29 }
```

Programme 8.1 – Tampons et espace de valeurs non bornés.

d'un espace d'états accessibles infini en cohérence séquentielle (que la méthode de Dan et *al.* ne permet pas de vérifier), il met en œuvre des tampons non bornés. Notre méthode permet malgré ces difficultés de vérifier le programme en représentant précisément ces tampons, alors que Dan *et al.* requièrent une borne supérieure finie et calculable pour leurs tailles (sans quoi l'analyse échoue). Leur analyse nécessite donc d'insérer une barrière supplémentaire pour limiter la quantité d'écritures que le processus `/* ENQUEUE */` peut émettre dans son tampon.

En revanche, contrairement à leur analyseur, notre prototype d'implémentation ne permet pas de vérifier l'algorithme *Bakery* de Lamport [Lam74] en un temps raisonnable : nous avons dû l'arrêter après plusieurs jours d'analyse.

8.3.2 Influence du domaine numérique

Les octogones et les polyèdres montrent des performances similaires, ainsi qu'une précision semblable pour la plupart des tests. Les octogones sont toutefois légèrement plus rapides, ce qui tend à les désigner comme un choix par défaut raisonnable.

Lorsque les programmes à analyser utilisent de nombreuses expressions logiques, les domaines intégrant les booléens de Bddapron offrent de bons résultats de précision et d'efficacité. Pour les analyses où ce gain d'expressivité n'est pas nécessaire, le coût additionnel induit par rapport aux domaines purement numériques n'est toutefois pas négligeable : jusqu'au triple du temps écoulé

et au double de mémoire utilisée. Il est donc préférable de choisir explicitement ces domaines lorsqu'on manipule des programmes utilisant des constructions logiques complexes plutôt que de les considérer comme un choix par défaut.

8.3.3 Impact du partitionnement

En distinguant plusieurs ensembles d'états, le partitionnement permet d'améliorer la précision d'une analyse (en plus d'aider à la construire dans notre cas). Cette amélioration a un coût : les opérateurs abstraits doivent être calculés sur chaque partition plutôt que sur un seul élément (qui serait alors leur borne supérieure).

Il est donc intéressant de connaître la proportion moyenne du nombre de partitions effectivement présentes dans les abstractions inférées. Leur nombre théorique maximal est $3^{\text{nb_processus} \times \text{nb_vars}}$, mais nous avons observé qu'une grande majorité d'entre elles est vide en pratique. Dans l'exemple de l'algorithme d'exclusion mutuelle de Peterson [Pet81], comprenant 3 variables et 2 processus, seul un état de contrôle présente 9 partitions non vides. La plupart n'en ont que 4 ou moins, en général 2.

Notre implémentation du partitionnement est creuse : les partitions vides ne sont pas représentées du tout. Ainsi, nous profitons grandement du faible nombre de partitions non vides. Le partitionnement des états semble donc effectivement être une abstraction intéressante, qui mène à une précision sensiblement meilleure tout en ayant un faible impact estimé sur les performances.

8.4 Conclusion

Dans cette partie, nous avons présenté une première méthode d'analyse de programmes concurrents dans des modèles comme TSO ou PSO. Cette méthode fonctionne par analyse monolithique du graphe de contrôle produit. En adaptant des abstractions de tableaux, nous avons montré comment construire des abstractions précises et robustes, paramétrées par un domaine numérique, capable de modéliser des tampons de taille non bornée.

Nous avons formalisé ces domaines dans le cas particulier d'une abstraction fondée sur la condensation. Nous avons implémenté cette méthode particulière, et nos résultats expérimentaux ont mis en évidence des performances et une précision satisfaisantes en comparaison avec l'état de l'art. Grâce à la représentation finie de tampons non bornés, nous sommes également capables de vérifier certains programmes avec strictement moins de barrières que les travaux précédents.

Toutefois, cette méthode monolithique souffrira des mêmes problèmes de passage à l'échelle que les travaux précédents, en raison de son caractère monolithique qui supportera mal l'augmentation du nombre de processus. La partie suivante proposera des techniques d'analyse modulaire permettant de s'affranchir notamment de l'augmentation exponentielle de la taille du graphe produit.

Enfin, nous proposerons dans une dernière partie des améliorations de notre domaine basées sur d'autres abstractions de tableaux, qui permettront de gagner en précision. Nous discuterons également de l'extension de ces abstractions pour gérer des modèles plus relâchés et plus complexes, comme POWER ou C.

Troisième partie

Interprétation abstraite modulaire

Chapitre 9

Introduction à l'analyse modulaire

Nous présenterons dans ce chapitre le principe général des analyses modulaires, puis détaillerons le fonctionnement du système formel des interférences, simples puis relationnelles. Nous dresserons enfin un état de l'art de l'analyse modulaire en mémoire faiblement cohérente.

9.1 Passage à l'échelle d'une analyse statique

L'analyse monolithique présentée en partie II ne présente pas de difficulté conceptuelle particulière par rapport à l'analyse d'un programme séquentiel. Nous avons vu qu'elle donnait des résultats satisfaisants sur des petits programmes, en termes de performances comme de précision.

Toutefois, elle souffre par construction d'un défaut majeur de passage à l'échelle : la taille du graphe produit augmente linéairement avec la taille de chaque processus, et exponentiellement avec le nombre de processus. Calculer un point fixe sur ce graphe devient alors impossible dans un temps raisonnable : il est nécessaire de recourir à une autre méthode.

Les méthodes modulaires cherchent à résoudre ce problème en analysant chaque processus séparément, plutôt que tous ensemble dans une analyse globale. Ainsi, idéalement, l'ajout d'un processus ne demanderait comme travail supplémentaire que l'analyse de ce processus : on obtiendrait alors une complexité linéaire en le nombre de processus.

Toutefois, ce modèle idéal ne s'applique pas dans notre cas : les processus interagissant entre eux, il est nécessaire, pour obtenir une analyse sûre, de prendre en compte ces interactions. Or nos analyses sont constructives : elles établissent les invariants du programme *ex nihilo*, par opposition à des outils de vérification qui prendraient en paramètre une description des effets des processus afin de vérifier qu'ils sont bien corrects (et vérifient d'éventuelles propriétés recherchées). Ajouter un processus demande alors non seulement de calculer son effet sur l'exécution du reste du programme, à l'aide d'un point fixe, mais aussi de reprendre l'analyse des autres processus pour y incorporer ces effets.

Les analyses de ce type sont donc modulaires dans le sens où elles analysent le code de chaque processus séparément, indépendamment du code des autres processus. L'objectif est par ailleurs d'arriver à une stabilisation rapide de la découverte des effets de chaque processus, ce qui limite le nombre d'analyses à réitérer.

L'analyse statique modulaire que nous allons présenter est basée sur la notion d'interférence définie par Antoine Miné [Min11, Min14], que nous allons présenter informellement dans les

```
1  /* Initial: x = 1 */
2
3  thread /* 0 */ {
4      while true {
5          (ℓ0) while x != 0 {}
6          /* Critical section start */
7          ...
8          /* Critical section end */
9          (ℓ1) x = 1; (ℓ2)
10     }
11 }
12
13 thread /* 1 */ {
14     while true {
15         (ℓ0) while x != 1 {}
16         /* Critical section start */
17         ...
18         /* Critical section end */
19         (ℓ1) x = 0; (ℓ2)
20     }
21 }
```

Programme 9.1 – Algorithme du tourniquet pour l'exclusion mutuelle.

sections qui suivent.

9.2 Interférences simples

On appelle interférences les effets d'un processus sur l'exécution du reste du programme. Dans une première approche, nous nous intéresserons aux *interférences simples*, qui indiquent qu'un processus peut affecter une valeur donnée à une variable partagée. Considérons par exemple le programme 9.1, dans un environnement séquentiellement cohérent.

Ce programme implémente une version simple de l'algorithme du tourniquet (*round-robin*) pour l'exclusion mutuelle. Plusieurs processus (deux pour le programme 9.1) partagent une ressource à laquelle ils accèdent dans une *section critique*. L'accès simultané de deux processus à la ressource est interdit : un seul processus à la fois doit pouvoir se trouver en section critique. L'algorithme du tourniquet répartit les accès à la ressource à l'aide d'un jeton matérialisé par une variable `x`. Chaque processus attend activement que son tour vienne en bouclant tant que la valeur de `x` ne correspond pas à son numéro. Quand son tour vient, il exécute la portion de code en section critique, puis passe le jeton au processus suivant. Les accès à la ressource se font donc à tour de rôle et en boucle, dans l'ordre des processus.

Déroulement d'une analyse par interférences.

1. Notre méthode commence par considérer le processus 0 comme un programme isolé. `x` vaut initialement `1`, le processus ne peut donc pas entrer en section critique. Aucune interférence venant d'un autre processus n'ayant pour l'instant été calculée, cette analyse intermédiaire s'arrête là.

2. Nous passons alors à l'analyse, toujours en isolation, du processus 1.

- Il quitte immédiatement sa boucle d'attente et entre en section critique, à la sortie de laquelle il affecte `0` à `x`.
- Cette affectation génère une interférence simple $P1 : x \mapsto 0$, indiquant que le processus 1 peut affecter `0` à `x`.
- Toute lecture de `x` par un autre processus peut maintenant renvoyer `0` au lieu de la dernière valeur stockée dans `x` par le processus en question.
- L'analyse du processus 1 s'arrête alors : il ne peut plus entrer en section critique, `x` étant maintenant égal à `0` et aucune interférence ne permettant d'y lire une autre valeur.

3. De retour au processus 0, l'analyse prend en compte l'interférence précédemment générée pour valider la sortie de la boucle d'attente en lisant `1` dans `x`. Le processus 0 peut alors entrer en section critique et, de façon similaire au processus 1, générer l'interférence $P0 : x \mapsto 1$. L'analyse calculera par ailleurs que ce processus peut maintenant entrer en section critique plusieurs fois, en réappliquant l'interférence générée processus 1.

4. L'analyse consécutive du processus 1 conclura également qu'il peut entrer plusieurs fois en section critique. Aucune interférence supplémentaire n'étant générée, l'analyse du programme est terminée.

Pour mener à bien une analyse complète, il n'a donc fallu procéder au total qu'à quatre interprétations abstraites de processus aux codes isolés.

En augmentant le nombre de processus, on obtiendrait un résultat similaire : chacun ne devrait être analysé que deux fois avant que la stabilisation des interférences ne soit établie.

Remarque 9.1.
Ce résultat optimal dépend de l'ordre d'interprétation abstraite des processus : s'ils sont analysés dans l'ordre inverse d'entrée en section critique, il sera nécessaire de procéder à un plus grand nombre d'analyses individuelles. Toutefois un grand nombre de ces analyses termineront immédiatement (le processus considéré ne pouvant pas entrer en section critique), les performances globales seront donc sensiblement les mêmes.

Ce résultat est le pendant modulaire du choix de la stratégie d'itération pour l'analyse d'un programme séquentiel. Cependant, il n'existe pas d'outil semblable au flot de contrôle pour précalculer les dépendances entre les processus afin de les interpréter dans un ordre optimal : nous les analyserons donc dans l'ordre du programme. Une interprétation abstraite préalable avec des domaines dédiés permettrait sans doute d'approximer un meilleur ordre, mais cela sort du cadre de cette thèse. $\triangleleft$

Si elles conduisent à des analyses rapides et sûres [Min11], les interférences simples sont toutefois peu précises : par exemple, elles ne permettent pas de prouver l'exclusion mutuelle dans le programme 9.1. Elles sont en effet indépendantes du flot de contrôle (*flow-insensitive*), et ne gardent aucune relation, ni entre les variables, ni entre les valeurs d'une même variable avant et après affectation.

9.3 Interférences relationnelles

Pour résoudre les problèmes de précision posés par les interférences simples, Antoine Miné [Min14] a proposé d'utiliser les *interférences relationnelles*. Ces interférences permettent de représenter les relations introduites dans la section précédente :

- Entre une variable affectée et les autres variables, permettant ainsi de représenter une condition sur l'état du programme pour qu'une interférence soit applicable ;

- Entre l'ancienne et la nouvelle valeur d'une variable affectée, renforçant ces conditions.

- Elles peuvent également être sensibles au flot de contrôle, en permettant de représenter des informations sur le point de contrôle où doivent se trouver les différents processus pour qu'elles soient applicables.

Ajoutons par exemple le premier type de relations à l'analyse de la section précédente. Au lieu de générer l'interférence $P1 : x \mapsto 0$, l'interprétation abstraite du processus 1 générera une interférence de la forme $P1 : x = 1 \vdash x \mapsto 0$, indiquant que ce processus peut affecter `0` à `x` si `x` contient initialement `1`.

Lors de l'interprétation abstraite du processus 0, cette même interférence pourra être exploitée pour valider l'entrée en section critique, et de même, une interférence $P0 : x = 0 \vdash x \mapsto 1$ sera générée par la suite. En revanche, une fois revenue à l'entrée de la boucle d'attente, l'interprétation abstraite a inféré que `x` vaut maintenant `1`. Il n'est alors plus possible d'appliquer l'interférence générée par le processus 1, et l'analyse du processus 0 s'arrête là, ayant prouvé que les entrées successives dans la section critique sont en réalité impossible. Il en va de même pour la deuxième analyse du processus 1, qui stabilise les interférences sur ce résultat.

La précision supérieure des interférences relationnelles leur permet ainsi d'inférer des propriétés autrement inaccessibles. Toutefois, l'analyse que nous venons de dérouler ne permet toujours pas de prouver la correction du programme : ni les interférences, ni le domaine utilisé pour l'interprétation abstraite de chaque processus ne sont en effet suffisamment expressifs pour pouvoir l'établir. En effet, il s'agit par essence d'une propriété sensible au flot de contrôle, puisqu'elle porte précisément sur un état de contrôle précis du programme.

Il est donc nécessaire d'incorporer des informations de flot à nos interférences. Elles seront alors de la forme $P1 : P0 \triangleright (\ell_1) \wedge P1 \triangleright (\ell_1) \wedge x = 1 \vdash x \mapsto 0 \wedge P1 \triangleright (\ell_2)$. Cette notation signifie que *si P0 est au point de contrôle* (ℓ_1) *et P1 au point de contrôle* (ℓ_1)*, et si* `x` *vaut* `1`*, alors P1 peut affecter* `0` *à* `x` *et passera alors au point de contrôle* (ℓ_2).

Pour pouvoir profiter au sein de l'interprétation abstraite d'un processus T de ces nouvelles informations, nous lui ajouterons une variable auxiliaire $\mathfrak{pc}_{T'}$ par processus $T' \neq T$. Cette variable enregistrera le point de contrôle du processus T'. On notera $\mathfrak{PC}_T \triangleq \{\mathfrak{pc}_{T'} \mid T' \neq T\}$.

L'ajout de ces informations de contrôle nous oblige à générer un nouveau type d'interférence, qui ne correspond à la mise à jour d'aucune variable mais à un changement du point de contrôle d'un processus. Ainsi, une interférence générée par le processus 0 pour symboliser le retour en tête de la boucle externe `while true` pourrait être de la forme $P0 : P0 \triangleright (\ell_2) \wedge P1 \triangleright (\ell_1) \wedge x = 1 \vdash P0 \triangleright (\ell_0)$. Ces interférences, générées systématiquement, sont le seul moyen pour un processus T de mettre à jour ses informations sur le contrôle des autres processus T'.

L'interprétation abstraite du programme 9.1 est alors sensiblement plus lourde à détailler pas-à-pas en raison de la génération et de l'application systématique à chaque étape de ces interférences. Sans la détailler, nous observerons tout de même que les seules interférences générées par le processus 0 avec pour point de contrôle de destination (ℓ_1) ont également pour condition initiale $x = 0$, en raison de la condition de la boucle d'attente (qui domine (ℓ_1) : c'est un point de passage obligatoire). Or, quand l'interprétation abstraite du processus 1 arrive à (ℓ_1), `x` vaut `1`. Il est donc impossible d'appliquer les interférences qui font passer $\mathfrak{pc}_0$ à (ℓ_1) : l'exclusion mutuelle est alors vérifiée.

Sûreté, complétude et abstraction des interférences.

Antoine Miné [Min14] a montré que, dans le concret, ces analyses basées sur les interférences relationnelles et sensibles au flot de contrôle sont non seulement sûres, mais également complètes. En effet, mémoriser dans l'interprétation de chaque processus le point de contrôle exact des autres processus revient en fait à reconstruire le graphe de contrôle produit au sein de chaque analyse individuelle.

Sûre et complète, cette sémantique n'est donc pas calculable : il est nécessaire de recourir à l'abstraction pour pouvoir l'exploiter dans des analyses statiques. Remarquons que les interférences simples, que nous avons étudiées en introduction, constituent déjà une abstraction des interférences relationnelles. Celle-ci consiste à oublier toutes les informations de contrôle (les variables $\mathfrak{pc}_{T'}$ deviennent alors inutiles, l'information qu'elles portent ne pouvant jamais être précisée), ainsi que les informations sur l'état du programme avant génération de l'interférence. De même, les interférences relationnelles mais insensibles au flot de contrôle sont une abstraction des interférences totalement relationnelles, laquelle est strictement plus précise que les interférences simples.

Remarquons que même les interférences simples, pourtant incomplètes comme nous l'avons montré, ne sont généralement pas calculables : en effet, un processus qui incrémenterait en boucle infinie une variable partagée en générerait un ensemble infini. Il est donc nécessaire de les abstraire pour pouvoir représenter cet ensemble infini et, le cas échéant, de recourir à un élargissement dédié afin de stabiliser en un temps fini cette abstraction.

9.4 État de l'art

Si l'analyse modulaire a été l'objet de plusieurs travaux [GBCS07, Min14, KW16, HMVW17, MM17, MPS+17, BGOS18], peu d'entre eux l'ont appliquée aux modèles mémoire faibles. Il a toutefois été démontré [AKL+11, Min11] que les analyses non relationnelles sûres en cohérence séquentielle le restaient dans un modèle faiblement cohérent, permettant ainsi d'adapter certains de ces travaux si la perte de précision induite est acceptable. En particulier, Antoine Miné [Min11] a proposé une analyse, non relationnelle et insensible au flot de contrôle, correcte dans un modèle relâché.

Ridge [Rid10] a par ailleurs formalisé une logique *rely-guarantee* pour x86-TSO, permettant de raisonner sur des programmes de façon modulaire. Son travail est en revanche axé sur les systèmes de preuve plus que sur l'analyse statique : il propose donc une approche très expressive pour la construction des invariants. Si cette approche offre un avantage certain pour l'écriture de preuves puissantes, elle est en revanche moins adaptée à la conception d'un analyseur statique qui cherche à abstraire les détails d'expressivité pour obtenir un procédé calculable.

Kusano *et al.* [KW17] proposent quant à eux une analyse modulaire précise pour les modèles mémoire faibles, en particulier TSO et PSO. Celle-ci consiste à générer rapidement une sur-approximation imprécise des interférences et à tirer profit d'un solveur Datalog pour supprimer celles qui peuvent être prouvées impossibles. Néanmoins, ces interférences ne sont pas relationnelles : elles ne maintiennent ni information de contrôle, ni lien entre la modification d'une variable et son ancienne valeur. Leur méthode souffre donc des mêmes limitations que l'analyse non relationnelle d'Antoine Miné [Min11].

9.5 Conclusion

L'analyse modulaire s'impose comme le seul moyen de passer à l'échelle lorsque le nombre de processus ou la taille du programme augmente. Le concept d'interférences permet de mettre en place cette modularité dans le cadre de l'interprétation abstraite. Il consiste à analyser chaque processus indépendamment comme un programme séquentiel, mais en incorporant aux résultats obtenus les effets que les autres processus peuvent avoir sur son exécution. Ce procédé infère *ex nihilo* ces effets : il est donc nécessaire de reprendre l'analyse des processus lorsque de nouvelles interférences ont été découvertes, jusqu'à stabilisation du résultat.

Dans les chapitres qui suivent, nous allons formaliser la sémantique concrète modulaire, puis proposer de nouvelles abstractions pour chacun de ses composants, états locaux et interférences. Nous validerons expérimentalement la méthode ainsi conçue en la comparant à l'analyse monolithique de la partie précédente.

Chapitre 10

Sémantique concrète modulaire

Pour formaliser la sémantique modulaire des programmes concurrents, nous aurons recours aux systèmes de transition étiquetés. Après avoir défini ce concept, nous définirons la sémantique monolithique dans le cadre qu'il propose puis en proposerons une forme modulaire.

10.1 Sémantiques transitionnelles

10.1.1 Systèmes de transition étiquetés

Le formalisme des systèmes de transition a de nombreuses applications en informatique, comme les automates finis ou les machines de Turing. Dans le cadre de la sémantique de programmes concurrents, nous nous intéresserons plus particulièrement à leur version étiquetée, dont les transitions sont enrichies de labels permettant d'en distinguer des classes différentes.

Définition 19 (Système de transition étiqueté).
Un système de transition étiqueté est un quadruplet $(\Sigma, \Lambda, E, \rightarrow)$ où :

- Σ est l'ensemble des *états*.
- Λ est l'ensemble des *étiquettes* (en anglais *labels*, aussi parfois appelées *actions*).
- $\mathcal{E} \subseteq \Sigma$ est l'ensemble des *états initiaux* ou *états d'entrée*.
- $\longrightarrow \subseteq \Sigma \times \Lambda \times \Sigma$ est la *relation de transition*. On notera $\sigma \xrightarrow{\varepsilon} \sigma'$ pour $(\sigma, \varepsilon, \sigma') \in \longrightarrow$. $\lozenge$

Appliqués à la sémantique de programmes [CC77a], les systèmes de transition se représentent une forme générale de sémantique à petits pas, permettant de raisonner indépendamment du langage de programmation choisi.

10.1.2 Sémantique monolithique

États et points de contrôle.

Rappelons que $\mathscr{C}$ représente l'ensemble des états de contrôle, c'est-à-dire l'ensemble des étiquettes du graphe de contrôle. Plus précisément, nous noterons $\mathbb{L}$ l'ensemble des *labels*[1] ou *points de contrôle*, soit les différentes positions possibles dans le code d'un processus. Un état de

1. À ne pas confondre avec les *labels* des systèmes de transition.

contrôle est alors un produit de points de contrôle, ou de façon équivalente, la donnée d'un point de contrôle pour chaque processus : $\mathscr{C} = \mathit{Thread} \to \mathbb{L}$. Nous noterons (ℓ_E^T) le point de contrôle initial d'un processus T.

Système de transition.

Le système de transition encodant la sémantique d'un programme est défini de la façon suivante :

- $\Sigma \triangleq \mathscr{C} \times \mathscr{S}$: un état du système est le produit d'un état de contrôle et d'un état mémoire.
- $\Lambda \triangleq \mathit{Thread}$: les transitions sont étiquetées par le processus qui les génère.
- $\mathcal{E} \triangleq \{(\lambda T.(\ell_E^T),\ \lambda x.0)\}$: dans l'unique état initial, les processus se trouvent à leur point d'entrée, et les variables sont initialisées à zéro (sauf précision contraire, qui sera alors explicite).
- $\longrightarrow$ modélise un pas atomique d'exécution du programme, où on choisit un processus T à exécuter, mettant à jour son point de contrôle et l'état global et laissant inchangés les points de contrôle des autres processus :

$$C_1, S_1 \xrightarrow{T} C_2, S_2 \triangleq (C_1(T), S_1) \rightharpoonup_T (C_2(T), S_2) \land \forall T' \neq T, C_1(T') = C_2(T')$$

où :

$$(\ell_1), S_1 \rightharpoonup_T (\ell_2), S_2 \triangleq$$

$$(((\ell_1), S_1), ((\ell_2), S_2)) \in \overrightarrow{[\![\mathit{code}(T)]\!]}_T \lor (\ell_1) = (\ell_2) \land \exists x \in \mathit{Shared}, S_2 \in [\![\text{flush } x]\!]_T\{S_1\}$$

Les transitions $\overrightarrow{[\![^{(\ell)}\mathit{stmt}^{(\ell')}]\!]}$, présentées en figure 10.1, sont celles que génèrent les instructions d'un processus. Elles sont construites par induction sur la syntaxe du code de ce processus, sur lequel les points de contrôle ont été explicités. Certaines de ces constructions réutilisent les opérateurs de la sémantique opérationnelle de la figure 5.4 (dont l'opérateur de transfert est également repris pour la définition des transitions de transfert). L'évaluation des expressions $\{\!|.|\!\}$ reste également identique à celle de la figure 5.3.

Remarque 10.1.
Les transitions du système construit depuis un programme donné ne sont pas toutes nécessairement accessibles lors d'exécutions réelles du programme : par exemple, le code `if false {`$^{(\ell_1)}$ `x = 1` $^{(\ell_2)}$`}` génère entre autres la transition $(\ell_1), \{x \mapsto 42\} \xrightarrow{T} (\ell_2), \{x \mapsto 1\}$, même si l'état de départ ne sera jamais atteint et l'instruction correspondante jamais exécutée (même depuis d'autres états). $\lhd$

Sémantique de traces et d'états.

La sémantique d'un système de transition, c'est-à-dire la donnée des exécutions possibles du programme associé, est fondée sur les *traces d'exécution*. Une trace d'exécution est une séquence finie ou infinie d'états de Σ entre lesquels sont intercalées des étiquettes de Λ. Elle est notée $\sigma_0 \overset{\varepsilon_1}{\rightsquigarrow} \sigma_1 \overset{\varepsilon_2}{\rightsquigarrow} \cdots$. On assimilera également un état σ à une trace de longueur 1.

La *sémantique de traces* d'un système rassemble alors les traces d'exécution correspondant aux comportements réels du programme, c'est-à-dire celles qui commencent dans un état initial et dont deux états consécutifs sont reliés par une transition. On s'intéressera en particulier

$$
\begin{aligned}
\overrightarrow{[\![^{(\ell)}\textit{<stmt>}^{(\ell')}]\!]}_T &\subseteq (\mathbb{L} \times \mathscr{S}) \times (\mathbb{L} \times \mathscr{S}) \\
\overrightarrow{[\![^{(\ell_1)} v \texttt{ := } e^{(\ell_2)}]\!]}_T &\triangleq \{((\ell_1), S), ((\ell_2), S') \mid S \in \mathscr{S}, S' \in [\![v \leftarrow e]\!]_T\{S\}\} \\
\overrightarrow{[\![^{(\ell_1)}\texttt{mfence}^{(\ell_2)}]\!]}_T &\triangleq \{((\ell_1), S), ((\ell_2), S') \mid S \in \mathscr{S}, S' \in [\![\texttt{mfence}]\!]_T\{S\}\} \\
\overrightarrow{[\![^{(\ell_1)} s;^{(\ell_2)} s'^{(\ell_3)}]\!]}_T &\triangleq \overrightarrow{[\![^{(\ell_1)} s^{(\ell_2)}]\!]}_T \cup \overrightarrow{[\![^{(\ell_2)} s'^{(\ell_3)}]\!]}_T \\
\overrightarrow{[\![^{(\ell_1)}\texttt{if } e \texttt{ \{}^{(\ell_2)} s^{(\ell_3)}\texttt{\}}^{(\ell_4)}]\!]}_T &\triangleq \overrightarrow{[\![^{(\ell_2)} s^{(\ell_3)}]\!]}_T \cup \\
&\quad \{((\ell_1), S), ((\ell_2), S) \mid S \in \mathscr{S}, \{\!|e|\!\}_T S = \texttt{true}\} \cup \\
&\quad \{((\ell_1), S), ((\ell_4), S) \mid S \in \mathscr{S}, \{\!|e|\!\}_T S = \texttt{false}\} \cup \\
&\quad \{((\ell_3), S), ((\ell_4), S) \mid S \in \mathscr{S}\} \\
\overrightarrow{[\![^{(\ell_1)}\texttt{if } e \texttt{ \{}^{(\ell_2)} s^{(\ell_3)}\texttt{\} else \{}^{(\ell_4)} s'^{(\ell_5)}\texttt{\}}^{(\ell_6)}]\!]}_T &\triangleq \overrightarrow{[\![^{(\ell_2)} s^{(\ell_3)}]\!]}_T \cup \overrightarrow{[\![^{(\ell_4)} s'^{(\ell_5)}]\!]}_T \cup \\
&\quad \{((\ell_1), S), ((\ell_2), S) \mid S \in \mathscr{S}, \{\!|e|\!\}_T S = \texttt{true}\} \cup \\
&\quad \{((\ell_1), S), ((\ell_4), S) \mid S \in \mathscr{S}, \{\!|e|\!\}_T S = \texttt{false}\} \cup \\
&\quad \{((\ell_3), S), ((\ell_6), S) \mid S \in \mathscr{S}\} \cup \\
&\quad \{((\ell_5), S), ((\ell_6), S) \mid S \in \mathscr{S}\} \\
\overrightarrow{[\![^{(\ell_1)}\texttt{while }^{(\ell_2)} e \texttt{ \{}^{(\ell_3)} s^{(\ell_4)}\texttt{\}}^{(\ell_5)}]\!]}_T &\triangleq \overrightarrow{[\![^{(\ell_3)} s^{(\ell_4)}]\!]}_T \cup \\
&\quad \{((\ell_2), S), ((\ell_3), S) \mid S \in \mathscr{S}, \{\!|e|\!\}_T S = \texttt{true}\} \cup \\
&\quad \{((\ell_2), S), ((\ell_5), S) \mid S \in \mathscr{S}, \{\!|e|\!\}_T S = \texttt{false}\} \cup \\
&\quad \{((\ell_1), S), ((\ell_2), S) \mid S \in \mathscr{S}\} \cup \\
&\quad \{((\ell_4), S), ((\ell_2), S) \mid S \in \mathscr{S}\}
\end{aligned}
$$

FIGURE 10.1 – Transitions générées par les instructions d'un processus.

à la sémantique de traces partielles, rassemblant les préfixes finis des traces d'exécution du programme. Ces préfixes finis sont en effet suffisants pour vérifier des propriétés de sûreté, qui stipulent qu'un état (ou un ensemble d'états) d'erreur donné est inaccessible, par définition en temps fini. Cette sémantique peut être définie par un point fixe $\mathcal{T}$:

$$\mathcal{T} \triangleq \text{lfp } \lambda\tau.\mathcal{E} \cup \left\{ \sigma_0 \overset{T_1}{\rightsquigarrow} \cdots \overset{T_i}{\rightsquigarrow} \sigma_i \overset{T_{i+1}}{\rightsquigarrow} \sigma_{i+1} \;\middle|\; \sigma_0 \overset{T_1}{\rightsquigarrow} \cdots \overset{T_i}{\rightsquigarrow} \sigma_i \in \tau \wedge \sigma_i \xrightarrow{T_{i+1}} \sigma_{i+1} \right\} \tag{10.1}$$

La sémantique d'états accessibles, c'est-à-dire l'ensemble $\mathcal{R}$ d'états accessibles dans les traces du programme, peut également s'écrire comme un point fixe :

$$\begin{gathered} \mathcal{R} \triangleq \text{lfp } R \\ \text{où :} \\ R \triangleq \lambda\rho.\mathcal{E} \cup \left\{ \sigma \in \Sigma \;\middle|\; \exists \sigma' \in \rho, T \in \Lambda, \sigma' \xrightarrow{T} \sigma \right\} \end{gathered} \tag{10.2}$$

La sémantique d'états peut également être vue comme une abstraction de la sémantique de traces qui oublie l'ordre des états :

$$\begin{gathered} \mathcal{R} = \alpha^{reach}(\mathcal{T}) \\ \text{où :} \\ \alpha^{reach}(\tau) = \left\{ \sigma \;\middle|\; \sigma_0 \overset{T_1}{\rightsquigarrow} \cdots \sigma \cdots \overset{T_n}{\rightsquigarrow} \sigma_n \in \tau \right\} \end{gathered}$$

Lien avec les systèmes d'équations.

Par définition, $\mathcal{R}$ est la plus petite solution de l'équation $\rho = R(\rho)$. La correspondance avec la définition équationnelle de la sémantique des programmes, sur laquelle se fonde la partie précédente, s'établit alors en partitionnant $\mathcal{R}$ par l'information de contrôle présente dans les états.

On peut alors voir un ensemble d'états $\rho \in \wp(\Sigma)$ du système de transition comme la donnée, pour chaque point de contrôle $C \in \mathscr{C}$, d'un ensemble d'états mémoire $\mathcal{X}_C \in \mathscr{D} = \wp(\mathscr{S})$. L'équation $\rho = R(\rho)$ peut alors se décomposer en un ensemble d'équations portant sur les $\mathcal{X}_C$, pour un certain ensemble de fonctions Ψ_C définies par R :

$$\forall C \in \mathscr{C}, \mathcal{X}_C = \Psi_C\left(\mathcal{X}_1, \ldots, \mathcal{X}_{\text{card}(\mathscr{C})}\right)$$

On retrouve alors la forme habituelle des systèmes d'équations définissant la sémantique d'un programme. Ce lien nous fournit donc une méthode de sur-approximation de la sémantique transitionnelle, laquelle consiste à employer les méthodes usuelles d'interprétation abstraite fondées sur le calcul d'un point fixe sur le graphe associé. Une fois le système résolu et un post-point fixe calculé, on retrouve alors $\mathcal{R}$ en réassemblant les partitions :

$$\mathcal{R} = \{(C, \mathcal{S}_C) \mid C \in \mathscr{C}, \mathcal{S}_C \in \mathcal{X}_C\}$$

On retrouve alors l'analyse monolithique de la section précédente. Le problème de passage à l'échelle s'explique alors par la taille exponentielle de $\mathscr{C}$, qui mène à un système d'équations trop grand.

10.2 Sémantique modulaire

Pour passer à l'échelle, nous chercherons donc à analyser chaque processus séparément, réduisant ainsi la taille du système d'équations induit.

$$\mathscr{S}_T \triangleq \mathscr{S} \times \prod_{T' \neq T} (\mathfrak{pc}_{T'} \to \mathbb{V})$$
$$\Sigma_T \triangleq \mathbb{L} \times \mathscr{S}_T$$
$$\mathscr{D}_T \triangleq \wp(\mathscr{S}_T)$$

FIGURE 10.2 – États et domaines locaux d'un processus T.

10.2.1 États locaux

Pour nous ramener à l'analyse d'un processus T en isolation, nous définirons ainsi ses **états locaux** Σ_T en projetant la fraction de contrôle qui compose un état global de Σ au seul point de contrôle dans $\mathbb{L}$ de T. La structure du système d'équations obtenu par la correspondance de la section précédente correspondra alors au graphe de contrôle de ce processus.

Pour ne pas perdre l'information de contrôle des autres processus, nous la mémorisons dans une variable $\mathfrak{pc}_{T'}$ par processus $T' \neq T$. Nous noterons $\mathscr{S}_T$ les états de $\mathscr{S}$ enrichis de ces variables. La figure 10.2 définit ces ensembles, ainsi que le domaine concret local $\mathscr{D}_T$.

Les états locaux accessibles $\mathcal{R}_{\mathcal{L}}(T) \in \wp(\Sigma_T)$ se définissent alors par projection des états accessibles $\mathcal{R}$ (cf. équation 10.2 page 96). $\overset{+}{\longmapsto}$ note l'ajout d'une variable au domaine de définition d'une fonction et définit son image.

$$\mathcal{R}_{\mathcal{L}}(T) \triangleq \pi_T(\mathcal{R})$$
où :
$$\pi_T : \Sigma \to \Sigma_T \triangleq \lambda(C,S).C(T), S\left[\forall T' \neq T : \mathfrak{pc}_{T'} \overset{+}{\longmapsto} C(T')\right]$$
est étendu point par point aux ensembles $\rho \subseteq \Sigma$:
$$\pi_T : \wp(\Sigma) \to \wp(\Sigma_T) \triangleq \lambda\rho.\,\{\pi_T(\sigma) \mid \sigma \in \rho\}$$

Grâce aux variables $\mathfrak{pc}_{T'}$, aucune information n'est perdue : π_T est bijective. Ce résultat permettra de montrer la complétude de la sémantique modulaire concrète.

L'interprétation individuelle de chaque processus permet alors de calculer les états locaux accessibles par les transitions générées par le même processus. Pour que l'analyse globale soit correcte, il faut incorporer à ces analyses individuelles les transitions générées par les autres processus : ce sont les interférences.

10.2.2 Interférences

Les interférences générées par un processus sont les transitions engendrées par ce processus dans la sémantique de traces $\mathcal{T}$ (il s'agit donc des transitions effectivement accessibles lors de l'exécution du programme). Elles sont ainsi notées $\mathcal{I}$: *Thread* $\to \wp(\Sigma \times \Sigma)$ et s'expriment formellement ainsi (rappelons que $\mathcal{T}$ est défini par l'équation 10.1 page 96) :

$$\mathcal{I}(T) \triangleq \alpha_T^{itf}(\mathcal{T})$$
où :
$$\alpha_T^{itf}(\tau) \triangleq \left\{\langle\sigma_i, \sigma_{i+1}\rangle \;\middle|\; \exists \sigma_0 \overset{T_1}{\leadsto} \cdots \overset{T_n}{\leadsto} \sigma_n \in \tau, T_{i+1} = T\right\}$$

$$\mathcal{R}_{\mathcal{L}}(T) = \text{lfp } \mathcal{R}_T(\mathcal{I})$$
$$\text{où :}$$
$$\mathcal{R}_T(I) \triangleq \lambda\, \rho_T. \begin{pmatrix} \pi_T(\mathcal{E}) \\ \cup\ \{\pi_T(\sigma') \mid \exists \sigma \in \Sigma, \pi_T(\sigma) \in \rho_T \wedge \sigma \xrightarrow{T} \sigma'\} \\ \cup\ \{\pi_T(\sigma') \mid \exists \sigma \in \Sigma, T' \in \mathit{Thread} \setminus \{T\}, \pi_T(\sigma) \in \rho_T \wedge \langle \sigma, \sigma' \rangle \in I(T')\} \end{pmatrix}$$

(a) États locaux.

$$\mathcal{I}(T) = \mathit{Collect}(\mathcal{R}_{\mathcal{L}})(T)$$
$$\text{où :}$$
$$\mathit{Collect} : \prod_{T \in \mathit{Thread}} \wp(\Sigma_T) \to (\mathit{Thread} \to \wp(\Sigma \times \Sigma))$$
$$\mathit{Collect}(\rho)(T) \triangleq \left\{ \langle \sigma, \sigma' \rangle \;\middle|\; \pi_T(\sigma) \in \rho(T) \wedge \sigma \xrightarrow{T} \sigma' \right\}$$

(b) Interférences.

FIGURE 10.3 – Expression des états locaux et interférences comme des points fixes interdépendants.

Autrement dit, les interférences générées par T forment le sous-ensemble de la relation de transition $\xrightarrow{T}$ restreint aux transitions intervenant dans les exécutions du programme. L'interprétation abstraite individuelle du processus T permet de déterminer ces transitions : d'après la correspondance entre les sémantiques équationnelle et transitionnelle, ce calcul est équivalent à celui des états locaux accessibles. Ces interférences peuvent alors, à leur tour, intervenir dans le calcul des états locaux d'un autre processus T' : en les appliquant aux états déjà découverts, on obtient de nouveaux états accessibles qui à leur tour peuvent mener par des transitions de T' à des états locaux supplémentaires, générant ainsi de nouvelles interférences.

10.2.3 Formulation et méthode de calcul modulaires

Points fixes imbriqués.

La figure 10.3 formule les états locaux accessibles $\mathcal{R}_{\mathcal{L}}$ et les interférences $\mathcal{I}$ comme les points fixes correspondant à la méthode de calcul que nous avons présentée.

Nous exprimons d'abord les états locaux d'un processus T comme un point fixe dépendant des interférences $\mathcal{I}$: les états accessibles sont explorés en entrelaçant des étapes de la relation de transition restreinte au processus T et des interférences générées par les autres processus. Le résultat est présenté en figure 10.3a. Les interférences $\mathcal{I}(T)$ sont ensuite formulées en figure 10.3b en joignant toutes les transitions produites par T depuis un état accessible de $\mathcal{R}_{\mathcal{L}}(T)$.

Les équations exprimant $\mathcal{R}_T$ et $\mathcal{I}$ sont mutuellement dépendantes : pour pouvoir construire leur solution, nous l'exprimerons comme un point fixe imbriqué en englobant les équations de la figure 10.3 dans un point fixe externe formulé par le théorème 10.1.

Théorème 10.1 (Sémantique modulaire par points fixes imbriqués).

$$\mathcal{R}_{\mathcal{L}} = \mathit{lfp}\ \Omega$$
$$\text{où :}$$
$$\Omega \triangleq \lambda\rho.\, \lambda T.\, \mathit{lfp}\, \mathcal{R}_T(\mathit{Collect}(\rho))$$

PREUVE. Par Miné [Min12]. □

Schéma d'itération.

La méthode de calcul du point fixe imbriqué du théorème 10.1 met en œuvre deux itérations également imbriquées :

- L'itération interne calcule, pour un sous-ensemble donné des interférences, les états locaux accessibles d'un processus. Elle produit également les interférences générées par le même processus. Ω correspond à la répétition de cette itération pour tous les processus, un à la fois.
- L'itération externe calcule pleinement Ω, c'est-à-dire répète le calcul du point fixe interne en réutilisant les interférences générées par une analyse comme paramètre de l'analyse suivante. Ce calcul se répète jusqu'à la stabilisation de l'ensemble global des interférences, et donc également des états accessibles.

Ce calcul de la sémantique est modulaire parce qu'il analyse chaque processus isolément par rapport au code des autres processus. Toutefois, comme précisé dans l'introduction de cette partie, la prise en compte les interférences des autres processus et le fait que cette méthode soit constructive et les infère *ex nihilo* nécessite d'itérer le calcul point fixe extérieur plutôt que d'analyser une seule fois chaque processus.

Cette sémantique est également complète : les projections π_T étant bijectives, il suffit, pour n'importe quel processus, de calculer $\pi_T^{-1}(\mathcal{R}_{\mathcal{L}}(T))$ pour retrouver $\mathcal{R}$. Elle n'est donc pas calculable en un temps fini : l'interprétation abstraite nous permettra par la suite de la sur-approximer sûrement.

Détermination du point fixe interne.

Si la procédure de calcul du point fixe externe consiste simplement à itérer le calcul interne jusqu'à stabilisation des interférences, ce dernier présente en revanche quelques particularités : s'il s'agit en première approche d'un calcul similaire à l'analyse d'un processus individuel, il faut toutefois y intégrer la génération et l'application des interférences.

Rappelons que $\mathcal{R}_T$, dont le point fixe définit la sémantique locale d'un processus (cf. théorème 10.1), correspond structurellement à l'interprétation abstraite du processus vu comme un programme unique : on retrouve la structure du graphe de contrôle du processus en partitionnant Σ_T par $\mathbb{L}$. De plus, la découverte des états locaux, après initialisation, se décompose en deux parties (cf. figure 10.3a) : à partir de chaque état déjà obtenu, on applique d'une part les transitions locales générées par le code du processus, d'autre part les transitions obtenues depuis l'ensemble d'interférences qui paramètre l'analyse.

Rappelons également que le schéma d'itération classique consiste d'abord à déterminer une stratégie d'itération chaotique, laquelle peut ici être fondée sur la structure du programme : s'agissant d'un seul processus (dans un langage structuré sans `goto`), le graphe de contrôle est

réductible [Bou93]. Une fois cette stratégie fixée, les états accessibles dans chaque sommet sont obtenus en joignant le résultat de l'opération associée aux arêtes menant à ce sommet, appliquée au sommet d'origine. Dans le cas présent, ce schéma correspond à l'énumération des transitions locales.

La phase de génération des interférences s'intègre naturellement dans cet algorithme : à chaque calcul d'un nouvel état local accessible, une interférence est générée entre l'état d'origine et ce nouvel état.

L'application des interférences extérieures se fait quant à elle après avoir déterminé l'ensemble d'états mémoire associé à un point de contrôle [Min14, MM17]. Les interférences extérieures ne modifiant que le point de contrôle du processus qui les a générées, le résultat de leur application reste associé au point de contrôle courant du processus local. Parmi les interférences, l'analyseur sélectionne celles compatibles avec l'ensemble d'états mémoire local calculé (dans le concret, leur origine lui appartient) et lui ajoute leurs états de destination.

Exemple 10.1.
Supposons que l'analyse locale du processus T_1 ait calculé, pour le point de contrôle (ℓ_1), l'ensemble d'états suivant :

$$x \mapsto 1, y \mapsto 2, \mathfrak{pc}_{T_2} \mapsto (\ell_2)$$
$$x \mapsto 3, y \mapsto 2, \mathfrak{pc}_{T_2} \mapsto (\ell_3)$$

Supposons également qu'elle dispose des interférences suivantes générées par T_2 :

$$\left\langle x \mapsto 1, y \mapsto 2, \mathfrak{pc}_{T_1} \mapsto (\ell_1), \mathfrak{pc}_{T_2} \mapsto (\ell_2) \right\rangle \rightsquigarrow \left\langle x \mapsto 1, y \mapsto 5, \mathfrak{pc}_{T_1} \mapsto (\ell_1), \mathfrak{pc}_{T_2} \mapsto (\ell_3) \right\rangle$$
$$\left\langle x \mapsto 3, y \mapsto 2, \mathfrak{pc}_{T_1} \mapsto (\ell_6), \mathfrak{pc}_{T_2} \mapsto (\ell_3) \right\rangle \rightsquigarrow \left\langle x \mapsto 5, y \mapsto 2, \mathfrak{pc}_{T_1} \mapsto (\ell_6), \mathfrak{pc}_{T_2} \mapsto (\ell_0) \right\rangle$$
$$\left\langle x \mapsto 1, y \mapsto 2, \mathfrak{pc}_{T_1} \mapsto (\ell_1), \mathfrak{pc}_{T_2} \mapsto (\ell_3) \right\rangle \rightsquigarrow \left\langle x \mapsto 5, y \mapsto 2, \mathfrak{pc}_{T_1} \mapsto (\ell_1), \mathfrak{pc}_{T_2} \mapsto (\ell_0) \right\rangle$$

Alors seule la première est applicable : la deuxième ne correspond pas au bon point de contrôle pour T_1, et l'état initial de la dernière n'appartient pas à l'ensemble accessible calculé par l'analyse locale. Après application, l'ensemble des états locaux accessibles en (ℓ_1) est alors le suivant :

$$x \mapsto 1, y \mapsto 2, \mathfrak{pc}_{T_2} \mapsto (\ell_2)$$
$$x \mapsto 3, y \mapsto 2, \mathfrak{pc}_{T_2} \mapsto (\ell_3)$$
$$x \mapsto 1, y \mapsto 5, \mathfrak{pc}_{T_2} \mapsto (\ell_3)$$

○

10.2.4 Clôture par transfert

La méthode de calcul par points fixes imbriqués décrite dans la section précédente n'est pas nouvelle : elle s'appliquait déjà en cohérence séquentielle [Min14, MM17]. En revanche, dans notre modèle relâché, le non-déterminisme des transferts exige des calculs supplémentaires.

Le modèle standard de ces transferts, qui fait intervenir des boucles de transfert sur chaque sommet du graphe de contrôle, conduit ainsi à calculer un point fixe en chaque point de contrôle, en bouclant jusqu'à stabilisation. Or la phase d'application des interférences intervient à chaque calcul d'un état local : si ceux-ci sont répétés dans un point fixe, il faut donc dans l'analyse modulaire calculer le point fixe d'un entrelacement d'opérations de transfert et d'application d'interférence. De fait, les transferts générant de nouveaux états locaux, de nouvelles interférences

peuvent éventuellement s'appliquer ; inversement, l'application d'interférences qui ne commutent pas avec un transfert donné demande de calculer ce transfert après cette application pour rester clos. Il est donc bien nécessaire d'entrelacer les deux opérations dans un point fixe : on ne peut pas les ordonner afin de les exécuter chacune une seule fois. Par ailleurs, les transferts générant eux-mêmes des interférences, elles doivent être prises en compte dans le résultat de l'analyse d'un processus : contrairement à un modèle séquentiellement cohérent où les deux phases sont en général bien séparées, ici la phase d'application d'interférences externes (qui inclut des transferts) génère également des interférences internes.

Ceci étant, le lemme 5.1 de commutation des transferts s'applique toujours dans le cas de l'analyse modulaire. En premier lieu, sans adaptation, il permet de n'appliquer après calcul d'une transition locale que les transferts qui ne commutent pas avec cette transition ; c'est-à-dire les transferts de la variable x si la transition correspond à une instruction qui fait intervenir x, et aucun transfert sinon. Il est également adaptable aux interférences : après application d'une interférence qui commute avec un transfert de x, il n'est pas nécessaire de calculer ce transfert.

Or le fait qu'une interférence commute ou non avec le transfert de x dépend de l'instruction ayant généré cette interférence : pour s'en convaincre, on peut revenir à la sémantique monolithique et observer que les interférences correspondent à l'exécution par un autre processus d'une instruction, laquelle peut ou non commuter avec un transfert de x par le processus en cours selon qu'elle fait intervenir ou non x. Si on sait, étant donnée une interférence, avec quels transferts elle commute, il devient alors possible de calculer le point fixe des transferts et applications d'interférences avec moins d'opérations que la méthode naïve qui considère les boucles de transfert brutes.

Afin de disposer de cette information, nous pouvons alors étiqueter les interférences avec la variable qu'elles font éventuellement intervenir. Rappelons que les opérations du langage, et par conséquent les interférences qu'elles génèrent, ne peuvent manipuler qu'au plus une variable partagée (cf. chapitre 3, section 3.2.4, page 46).

$$\begin{aligned} \mathit{Shared}^{?} &\triangleq \mathit{Shared} \uplus \{\bot^{?}\} \\ \mathcal{I} \in \mathscr{I} &\triangleq \mathit{Thread} \rightarrow \wp(\mathit{Shared}^{?} \times \Sigma \times \Sigma) \end{aligned}$$

Par rapport aux fonctions de la figure 10.3, $\mathcal{R}_T(I)$ (en particulier la dernière ligne, soit l'application des interférences) ne change fondamentalement pas : les variables dans $\mathit{Shared}^{?}$ sont simplement ignorées. $\mathit{Collect}(\mathcal{R}_L)(T)$ ne s'écrit plus aussi naturellement, mais la méthode de calcul des interférences que nous venons d'exposer fondée sur les stratégies d'itération chaotique s'adapte plus aisément : considérons ainsi une interférence de la sémantique originale. Dans l'itération locale, elle a été générée en calculant le résultat de l'opération associée à une arête, appliquée à l'état associé à l'origine de cette arête. Pour étiqueter cette interférence, nous lui associons simplement x si ladite opération lit ou modifie $x \in \mathit{Shared}$ (cf. lemme 5.1), et $\bot^{?}$ si elle ne lit ni ne modifie aucune variable partagée.

Une fois ces interférences étiquetées, nous pouvons mettre en œuvre une procédure de clôture par transferts. Remarquons tout d'abord que seuls les transferts locaux sont concernés : les transferts effectués par d'autres processus sont intégrés dans leurs interférences et donc pris en compte lors de la phase d'application. Nous pouvons alors adapter la clôture par transfert et les

opérateurs clos monolithiques de la section 5.3 en formulant leur équivalent local modulaire :

$$[\![\text{flush}\, x]\!]_T^* X \triangleq \text{lfp}\ \lambda Y . X \cup [\![\text{flush}\, x]\!]_T Y$$

$$\overline{[\![op]\!]}_T : \mathscr{D}_T \to \mathscr{D}_T$$

$$\overline{[\![op]\!]}_T(X) \triangleq \begin{cases} [\![\text{flush}\, x]\!]_T^* \circ [\![op]\!]_T X & \text{si } [\![op]\!] \text{ est lié à } x \in \mathit{Shared} \\ [\![op]\!]_T X & \text{sinon} \end{cases}$$

De même, nous pouvons définir un opérateur clos d'application d'une interférence à un ensemble d'états locaux, qui, si l'interférence est applicable, en ajoute le résultat à l'ensemble d'états avant de clore le tout par transfert si nécessaire.

$$\overline{\text{apply}}_\ell : \mathit{Shared}^? \times \Sigma \times \Sigma \to \mathscr{D}_T \to \mathscr{D}_T$$

$$\overline{\text{apply}}_\ell(x^?, \langle \sigma, \sigma' \rangle)(X_T) \triangleq \texttt{let} \begin{cases} \ell_\sigma, S \ = \pi_T(\sigma) \\ \ell_\sigma, S' = \pi_T(\sigma') \\ X'_T = X_T \cup \{S'\} \end{cases}$$

$$\texttt{in} \begin{cases} \begin{cases} X'_T & \text{si } x^? = \bot^? \\ [\![\text{flush}\, x^?]\!]_T^* X'_T & \text{si } x^? \in \mathit{Shared} \end{cases} & \text{et si } \ell_\sigma = \ell \wedge S \in X_T \\ X_T & \text{sinon} \end{cases}$$

Cet opérateur s'étend point par point à l'ensemble des interférences.

$$\overline{\text{apply}}_\ell : \mathscr{I} \to \mathscr{D}_T \to \mathscr{D}_T$$

$$\overline{\text{apply}}_\ell(I)(X_T) \triangleq \bigcup_{\substack{T' \neq T \\ \iota \in I(T')}} \overline{\text{apply}}_\ell(\iota)(X_T)$$

Lors du calcul du point fixe interne, par itération sur le graphe de contrôle comme pour un programme séquentiel, l'étape d'analyse d'un sommet consiste donc à joindre le résultat de chaque arête locale menant à ce sommet, puis à clore le tout par application d'interférences et transferts. Cette dernière phase pouvant faire que de nouvelles interférences soient applicables, elle est répétée jusqu'à stabilisation du point fixe correspondant.

$$\mathit{analyse}_\ell : \mathscr{I} \to (\mathbb{L} \to \mathscr{D}_T) \to \mathscr{D}_T$$

$$\mathit{analyse}_\ell(I)(\mathcal{R}_T) \triangleq \text{lfp}\ \lambda X_T . \bigcup_{\ell' \xrightarrow[\text{arêtes}]{op} \ell} \overline{[\![op]\!]}_T(\mathcal{R}_T(\ell')) \ \cup \ \overline{\text{apply}}_\ell(I)(X_T)$$

Exemple 10.2.
Supposons l'état local suivant (nous omettrons le contrôle pour simplifier) :

$$x^{mem} \mapsto 2, x_1^1 \mapsto 1, r \mapsto 0$$

Sa clôture par transfert aboutit alors à l'ensemble suivant :

$$x^{mem} \mapsto 2, x_1^1 \mapsto 1, r \mapsto 0$$
$$x^{mem} \mapsto 1, r \mapsto 0$$

Supposons maintenant qu'il existe l'interférence suivante, qui correspond à une lecture de x :

$$x : \langle x^{mem} \mapsto 2, r \mapsto 0\rangle \leadsto \langle x^{mem} \mapsto 2, r \mapsto 2\rangle$$

Elle est applicable et générera cet état local :

$$x^{mem} \mapsto 2, x_1^1 \mapsto 1, r \mapsto 2$$

Il est alors nécessaire de clore par transfert à nouveau, ce qui générera ce nouvel état :

$$x^{mem} \mapsto 1, r \mapsto 2$$

Le point fixe est alors atteint. Toutefois, il se peut dans le cas général que cette clôture découvre encore de nouvelles interférences applicables : on répéterait alors la procédure. ○

On peut alors procéder au calcul de $\mathcal{R}_T$, après partitionnement par $\mathbb{L}$, de la façon classique de calcul d'un point fixe selon une stratégie d'itération déterminée au préalable.

$$\mathcal{R}_T : \mathscr{I} \to (\mathbb{L} \to \mathscr{D}_T)$$
$$\mathcal{R}_T(I) \triangleq \text{lfp } \lambda\rho_T.\ \pi_T(\mathcal{E}) \cup \lambda\ell.\ \mathit{analyse}_\ell(I)(\rho_T)$$

L'expression de la sémantique globale ne change pas : on répète l'analyse individuelle de chaque processus jusqu'à stabilisation des interférences.

Conclusion.

Dans le système des interférences, nous avons donc défini une sémantique concrète modulaire pour nos programmes et y avons intégré un algorithme de clôture par transfert optimisé. Il nous faut maintenant abstraire les états locaux et les interférences pour pouvoir calculer une approximation de cette sémantique.

Chapitre 11

Domaines et sémantique abstraits

La sémantique concrète modulaire étant complète, elle n'est pas calculable. Nous abstrairons donc à la fois les états locaux et les interférences pour en construire une sur-approximation sûre.

11.1 Domaine des états locaux

Un état local possède deux composants : la mémoire et les informations de contrôle des autres processus. Nous présenterons leurs deux abstractions séparément avant de les unifier en un seul domaine local.

11.1.1 Abstraction de mémoire

Le domaine mémoire des états locaux d'un processus T est composé d'une projection et d'une abstraction semblable au domaine monolithique de la partie II.

La projection consiste à oublier, dans les états de $\mathscr{S}_T$, les variables représentant les tampons des autres processus. La raison intuitive de cette abstraction est l'impossibilité par le processus courant T d'observer directement ces entrées ; il sera en revanche en mesure de les lire une fois transférées dans la mémoire partagée. Les variables locales des autres processus sont en revanche conservées : elles encodent l'état de ces processus (en particulier la valeur des dernières variables partagées qu'ils ont lues) et sont utiles pour une application précise des interférences. Il en va de même pour les variables $\mathtt{pc}_{T'}$ portant les informations de contrôle, qui ne sont pas concernées par la projection. La figure 11.1 définit ce domaine.

Les états locaux sont ensuite partitionnés selon un critère semblable à celui de l'analyse monolithique (cf. chapitre 6), que nous avons restreint aux seuls tampons du processus local (les entrées des tampons des autres processus étant absentes des états locaux après projection). Ce critère et le domaine résultant sont définis en figure 11.2. Rappelons que le partitionnement lui-même ne perd aucune information : $\underset{\alpha_P^T}{\overset{\gamma_P^T}{\twoheadleftarrow\!\!\twoheadrightarrow}}$ est un isomorphisme de Galois.

Dans chaque partition, nous appliquons alors la technique de condensation monolithique déjà présentée au chapitre 7, page 69, toujours adaptée aux états locaux. Dans chaque partition, nous regroupons les variables définies $x_2^T, \dots, x_\infty^T$ de chaque état dans une variable condensée x_ω^T. Cette abstraction est appliquée partition par partition. La figure 11.3 formalise le domaine ainsi obtenu. Il s'agit d'un rappel de la formalisation déjà présentée à la figure 7.2, page 73.

Enfin, le domaine monolithique incluait une abstraction numérique. Si cette étape est toujours présente dans notre domaine local, elle nous permettra d'unifier l'abstraction mémoire et

$$\mathscr{S}_T^\natural \triangleq (\mathfrak{PC}_T \to \mathbb{L}) \times Mem \times TLS \times \prod_{x \in Shared} Buf_x^T$$
$$\mathscr{D}_T^\natural \triangleq \wp(\mathscr{S}_T^\natural)$$
$$\mathscr{D}_T \xleftarrow[\alpha_{Buf}^T]{\gamma_{Buf}^T} \mathscr{D}_T^\natural$$
$$\gamma_{Buf}^T(X_T^\natural) \triangleq \left\{ PC, M, S, (T, x) \mapsto B_x^T \in \mathscr{S} \;\middle|\; PC, M, S, x \mapsto B_x^T \in X_T^\natural \right\}$$
$$\alpha_{Buf}^T(X) \triangleq \left\{ PC, M, S, x \mapsto B_x^T \in \mathscr{D}_T^\natural \;\middle|\; PC, M, S, (T, x) \mapsto B_x^T \in X \right\}$$

FIGURE 11.1 – Première étape d'abstraction modulaire : oubli des tampons externes.

$$\mathscr{B}^\flat \triangleq Shared \to \{\tilde{0}, \tilde{1}, \tilde{\omega}\}$$
$$\forall T \in Thread, \delta_T : \mathscr{S}_T^\natural \to \mathscr{B}^\flat$$
$$\delta_T(S_T^\natural) \triangleq \lambda x. \begin{cases} \tilde{0} & \text{if } \#_{S_T^\natural}(x) = 0 \\ \tilde{1} & \text{if } \#_{S_T^\natural}(x) = 1 \\ \tilde{\omega} & \text{if } \#_{S_T^\natural}(x) > 1 \end{cases}$$

(a) Critère local de partitionnement.

$$\mathscr{D}_T^\flat \triangleq \mathscr{B}^\flat \to \mathscr{D}^\natural$$
$$\mathscr{D}_T^\natural \underset{\alpha_P^T}{\overset{\gamma_P^T}{\leftleftarrows\!\!\rightrightarrows}} \mathscr{D}_T^\flat$$
$$\alpha_P^T(X_T^\natural) \triangleq \lambda b^\flat. \left\{ S_T^\natural \in X_T^\natural \;\middle|\; \delta_T(S_T^\natural) = b^\flat \right\}$$
$$\gamma_P^T(X_T^\flat) \triangleq \left\{ S_T^\natural \in \mathscr{S}_T^\natural \;\middle|\; S_T^\natural \in X_T^\flat(\delta_T(S_T^\natural)) \right\}$$

(b) Domaine partitionné.

FIGURE 11.2 – Partitionnement des états locaux selon la longueur abstraite des tampons.

$$\begin{aligned}
Var_T &\triangleq \mathfrak{PC}_T \uplus Shared \uplus Local \uplus \left\{x_n^T \;\middle|\; x \in Shared, n \in \mathbb{N}^*\right\} \\
Var_T^{\mathtt{sum}} &\triangleq \mathfrak{PC}_T \uplus Shared \uplus Local \uplus \left\{x_1^T, x_\omega^T \;\middle|\; x \in Shared\right\} \\
\mathscr{S}_T^{\mathtt{sum}} &\triangleq Var_T^{\mathtt{sum}} \to \mathbb{V} \\
\mathscr{D}_T^{\mathtt{sum}} &\triangleq \wp(\mathscr{S}_T^{\mathtt{sum}})
\end{aligned}$$

(a) États et domaine condensés.

$$\Phi_{\mathtt{sum}}^T : Var_T \to Var_T^{\mathtt{sum}}$$

$$\Phi_{\mathtt{sum}}^T(v) \triangleq \begin{cases} v & \text{si } v \in \mathfrak{PC}_T \uplus Shared \uplus Local \\ v & \text{si } \exists x \in Shared, v = x_1^T \\ x_\omega^T & \text{si } \exists x \in Shared, n > 1, v = x_n^T \end{cases}$$

$$\Pi_{\Phi_{\mathtt{sum}}^T}^T \triangleq \left\{\pi : \mathscr{S}_T \to \mathscr{S}_T^{\mathtt{sum}} \;\middle|\; \forall v^{\mathtt{sum}} \in Var_T^{\mathtt{sum}}, \exists v \in \Phi_{\mathtt{sum}}^{T^{-1}}(v^{\mathtt{sum}}), \forall S \in \mathscr{S}_T, \pi(S)(v^{\mathtt{sum}}) = S(v)\right\}$$

(b) Fonction de condensation et projections orthogonales.

$$\models_T^{\mathtt{sum}} : \mathscr{D}_T^{\mathtt{sum}} \times \mathscr{S}_T^\natural$$

$$X_T^{\mathtt{sum}} \models_T^{\mathtt{sum}} S_T^\natural \overset{\triangle}{\equiv} \left\{\pi(S_T^\natural) \;\middle|\; \pi \in \Pi_{\Phi_{\mathtt{sum}}^T}^T\right\} \subseteq X_T^{\mathtt{sum}}$$

$$\mathscr{D}_T^\natural \underset{\alpha_{\mathtt{sum}}^T}{\overset{\gamma_{\mathtt{sum}}^T}{\leftrightarrows}} \mathscr{D}_T^{\mathtt{sum}}$$

$$\begin{aligned}
\gamma_{\mathtt{sum}}^T(X_T^{\mathtt{sum}}) &= \left\{S_T^\natural \in \mathscr{D}_T^\natural \;\middle|\; X_T^{\mathtt{sum}} \models_T^{\mathtt{sum}} S_T^\natural\right\} \\
\alpha_{\mathtt{sum}}^T(X_T^\natural) &= \left\{\pi(S_T^\natural) \;\middle|\; S_T^\natural \in X_T^\natural, \pi \in \Pi_{\Phi_{\mathtt{sum}}^T}^T\right\}
\end{aligned}$$

(c) Relation de représentation et plongement de Galois.

$$\mathscr{D}_T^{\maltese} \triangleq \mathscr{B}^\flat \to \mathscr{D}_T^{\mathtt{sum}}$$

$$\mathscr{D}_T^\flat \underset{\alpha_S^T}{\overset{\gamma_S^T}{\leftrightarrows}} \mathscr{D}_T^{\maltese}$$

$$\begin{aligned}
\gamma_S(X_T^{\maltese}) &\triangleq \lambda b^\flat.\gamma_{\mathtt{sum}}^T(X_T^{\maltese}(b^\flat)) \\
\alpha_S(X_T^\flat) &\triangleq \lambda b^\flat.\alpha_{\mathtt{sum}}^T(X_T^\flat(b^\flat))
\end{aligned}$$

(d) Condensation partition par partition.

FIGURE 11.3 – Condensation du tampon dans le domaine local. Cf. figure 7.2 page 73.

l'abstraction de contrôle, et sera donc présentée en section 11.1.3.

11.1.2 Abstraction de contrôle

En plus de l'abstraction de la fraction mémoire des états locaux, similaire au domaine monolithique, nous devons développer une abstraction de contrôle pour les variables $\mathfrak{pc}_T$ qui sont une nouveauté de l'analyse modulaire.

Rappelons que ces variables sont la traduction dans les états locaux d'une information qui était portée par la structure du graphe de contrôle produit. La sémantique modulaire étant complète, il est ainsi possible grâce à ces variables de simuler l'analyse monolithique par entrelacements. Notons toutefois que le résultat sera a priori moins performant, ces variables ne pouvant pas être prises en compte pour le calcul d'une stratégie d'itération optimale (contrairement à l'analyse monolithique où ce calcul est effectué sur la structure globale du programme). Il est également possible d'abstraire cette information : en modulant le choix du domaine sous-jacent, il devient alors possible de paramétrer l'analyse modulaire pour obtenir le compromis souhaité entre performance et précision.

Nous avons donc implémenté et adapté à notre analyse plusieurs domaines proposés par d'autres auteurs dans des travaux antérieurs [Min11, MM17] afin d'évaluer leurs performances comparées.

Abstraction identité.

La représentation la plus immédiate consiste simplement à garder les variables $\mathfrak{pc}_T$ telles quelles, sans abstraction aucune. Comme nous l'avons fait remarquer plus haut, cela revient à simuler une analyse par entrelacements, avec toutefois une stratégie d'itération globale moins adaptée. Si la précision est maximale, il n'est en général pas nécessaire de garder autant d'informations sur les autres processus lors des analyses locales, et les mauvaises performances contre-indiquent cette option comme choix par défaut.

Insensibilité au flot de contrôle.

Cette abstraction [Min11] consiste à projeter les états locaux en oubliant toute information de contrôle des autres processus. Alternativement, le domaine des variables $\mathfrak{pc}_T$ est $\{\top\}$. L'analyse locale d'un processus reste évidemment sensible au flot de contrôle de ce même processus, qui structure la stratégie d'itération pour le calcul du point fixe. Remarquons que ce domaine consiste à revenir au niveau du domaine local à l'analyse par interférences simples de la section 9.2.

Cette abstraction conduit à une analyse très rapides, mais qui s'est avérée trop imprécise sur nos tests : elle n'a pas permis de vérifier un nombre important de programmes. Intuitivement, cela s'explique en partie par le fait que les propriétés que nous cherchons à vérifier sont souvent intimement liées à la structure de contrôle du programme global : par exemple, une exclusion mutuelle s'exprime sur les points de contrôle de tous les processus, nécessitant ainsi une analyse suffisamment expressive pour pouvoir exprimer des relations entre eux.

Partitionnement du contrôle.

Cette technique a été étudiée dans le cadre de la cohérence séquentielle par Monat et Miné [MM17] et consiste à partitionner en des ensembles de points adjacents le contrôle des processus externes. Pour cela, on définit pour chaque processus un ensemble de *labels abstraits*, et on associe à chaque point de contrôle concret du processus un de ces labels. On réduit ainsi la taille de l'espace de contrôle des autres processus. Insistons sur l'unicité du partitionnement en labels

abstraits des points de contrôle d'un processus T donné : ce partitionnement est défini pour tout le programme et n'est pas spécifique à chaque processus $T' \neq T$.

Il s'agit d'une abstraction flexible : il est possible de moduler sa précision en raffinant ou en élargissant le partitionnement choisi (ce qui conduit à augmenter ou diminuer le nombre de labels abstraits). En particulier, il est possible d'exprimer sous cette forme les deux domaines précédents : la représentation précise consiste à choisir l'ensemble des points de contrôle comme labels abstraits et à associer chaque point de contrôle concret à lui-même, et l'abstraction insensible au flot de contrôle revient à choisir un unique label abstrait $\top$.

Formellement, on définit pour chaque processus T une partition $\mathbb{L}^\sharp_T$ de ses points de contrôle dans $\mathbb{L}$. On note alors $\dot{\alpha}_{\mathbb{L}_T} : \mathbb{L} \to \mathbb{L}^\sharp_T$ la correspondance entre un point de contrôle concret et sa classe dans $\mathbb{L}^\sharp_T$.

Pour l'abstraction identité, on a ainsi $\mathbb{L}^\sharp_T = \mathbb{L}$ et $\dot{\alpha}_{\mathbb{L}_T} = \lambda\ell.\ell$. L'abstraction insensible au flot de contrôle correspond à $\mathbb{L}^\sharp_T = \{\top\}$ et $\dot{\alpha}_{\mathbb{L}_T} = \lambda\ell.\top$.

Enfin, considérons, dans le cas du programme 9.1 implémentant un tourniquet, l'abstraction qui regroupe d'une part les points de contrôle jusqu'à la section critique incluse et de l'autre ceux qui suivent la section critique. Elle est définie par :

$$\mathbb{L}^\sharp_T = \{[\ell_\mathcal{E} \mathinner{..} \ell_1], [\ell_2 \mathinner{..} end]\}$$

Et on aura par exemple :

$$\dot{\alpha}_{\mathbb{L}_{\ell_0}} = [\ell_\mathcal{E} \mathinner{..} \ell_1]$$

Remarquons que le partitionnement ne regroupe pas nécessairement des points de contrôle adjacents. Il s'agit souvent du choix le plus intuitif, mais il est en effet possible sur certains programmes de considérer qu'un regroupement de points de contrôle distants a plus de sens. Par exemple, pour analyser un programme exécutant une suite d'opérations `lock` et `unlock` verrouillant et déverrouillant un mécanisme d'exclusion mutuelle, il peut être raisonnable de regrouper ensemble d'un côté les points de contrôle d'un processus correspondant à la section critique, où il possède le verrou, et de l'autre ceux où il cherche à l'acquérir ou procède à d'autres opérations pour lesquelles la synchronisation avec les autres processus n'est pas demandée. C'est notamment le choix effectué par l'analyseur AstréeA [Min14].

En raison de son caractère modulaire et du bon compromis entre performances et précision qu'elle permet d'obtenir avec un partitionnement bien choisi, le partitionnement en labels abstraits sera notre abstraction par défaut pour l'analyse de nos programmes cibles. Nous discuterons au chapitre 12 sa définition précise pour un programme donné.

11.1.3 Domaine numérique final

Après avoir abstrait les fractions mémoire et contrôle des états locaux, nous les regroupons dans un même domaine numérique. L'information de contrôle, quelle que soit l'abstraction choisie, peut être représentée par un entier (borné) : cette étape est donc fortement similaire à celle de l'analyse monolithique. Le partitionnement par rapport aux tampons a encore une fois été choisi de telle sorte que les mêmes variables (éventuellement condensées) soient définies dans les états d'une même partition, permettant ainsi d'appliquer directement l'abstraction numérique partition par partition. Le domaine correspondant sera noté $\mathscr{D}^{\mathsf{num}}$ et défini par sa concrétisation :

$$\gamma^T_{\mathsf{num}} : \mathscr{D}^{\mathsf{num}} \to \mathscr{D}^{\mathsf{sum}}_T$$

Notre analyse est toujours paramétrique par rapport au domaine numérique choisi. Puisque par ailleurs l'abstraction du contrôle a a priori été choisie de sorte qu'elle permette d'exprimer

$$\mathscr{D}^\sharp_T \triangleq \mathscr{B}^\flat \to \mathscr{D}^{\mathtt{num}}$$
$$\gamma_T : \mathscr{D}^\sharp_T \to \mathscr{D}_T$$
$$\gamma_T(X^\sharp_T) \triangleq \gamma^T_{Buf}\left(\left\{S^\natural_T \in \mathscr{S}^\natural_T \;\middle|\; S^\natural_T \in \gamma^T_{\mathfrak{sum}} \circ \gamma^T_{\mathtt{num}}(X^\sharp_T(\delta_T(S^\natural)))\right\}\right)$$

FIGURE 11.4 – Domaine final des états locaux.

précisément les invariants sur lesquels le programme fonde sa correction, l'intuition conduit à garder cette abstraction intacte à travers le domaine numérique, c'est-à-dire à ne pas appliquer aux variables de contrôle d'abstraction supplémentaire. Il est toutefois préférable, pour des raisons d'implémentation, de considérer que ces variables font partie du domaine numérique, qui peut alors choisir une représentation adaptée pour tout l'état local. En pratique, nous avons déjà établi que l'information de contrôle pouvait s'encoder dans des entiers bornés : les domaines logico-numériques présentés en section 2.5 permettant de représenter précisément les valeurs d'un type fini, ils représentent un candidat idéal à ce stade.

Le domaine numérique intégrant alors pleinement les points de contrôle abstraits, il devient alors formellement plus pratique, sans perte de généralité, d'intégrer l'abstraction du contrôle directement dans le domaine numérique.

La figure 11.4 récapitule le domaine final $\mathscr{D}^\sharp_T$ des états locaux, défini par sa concrétisation γ_T.

11.2 Interférences abstraites

Rappelons que les interférences concrètes appartiennent à $\mathscr{I} = \mathit{Thread} \to \wp(\mathit{Shared}^? \times \Sigma \times \Sigma)$. Elles sont partitionnées par le processus qui les a émises, et composées d'un état (global) d'origine, d'un état de destination et, si elles proviennent d'une écriture ou d'une lecture de la mémoire partagée, de la variable correspondante.

Le partitionnement par le processus émetteur est essentiel pour éviter d'appliquer à un processus des interférences qu'il aurait lui-même générées, ce qui mène à des calculs inutiles dans le concret et à des pertes de précision supplémentaires dans l'abstrait. Il sera donc conservé tel quel, et nous abstrairons donc les interférences processus par processus, c'est-à-dire des éléments de $\wp(\mathit{Shared}^? \times \Sigma \times \Sigma)$.

Dans les états de départ et de destination des interférences, nous projetterons la fraction de contrôle de $\mathscr{C}$ dans la fraction mémoire de $\mathscr{S}$ de façon similaire aux états locaux, à l'aide de variables $\mathfrak{pc}_T$:

$$\Sigma_{\mathscr{I}} \triangleq \mathscr{S} \times \prod_{T \in \mathit{Thread}} (\mathfrak{pc}_T \to \mathbb{V})$$
$$\pi_{\mathscr{I}} : \Sigma \to \Sigma_{\mathscr{I}} \triangleq \lambda(C, S).S\left[\forall T \in \mathit{Thread} : \mathfrak{pc}_T \overset{+}{\longmapsto} C(T)\right]$$

Avec extension point par point :

$$\pi_{\mathscr{I}} : \wp(\Sigma) \to \wp(\Sigma_{\mathscr{I}}) \triangleq \lambda\rho.\,\{\pi_{\mathscr{I}}(\sigma) \mid \sigma \in \rho\}$$

Cette projection est encore une fois bijective et revient simplement à une représentation différente des mêmes objets. Nous chercherons donc à abstraire les interférences concrètes générées par un processus, soit des éléments de $\mathit{Shared}^? \times \Sigma_{\mathscr{I}} \times \Sigma_{\mathscr{I}}$.

Pour cela, nous utiliserons des abstractions semblables à celles des états locaux. Dans les états d'origine et de destination, nous oublierons les variables de tampon de tous les processus, et appliquerons à chaque variable $\mathfrak{pc}_T$ la même abstraction que dans les états locaux, à savoir le partitionnement en labels abstraits. Insistons sur l'oubli de tous les tampons dans une interférence, y compris ceux du processus qui la génère : la justification intuitive est qu'ils sont absents des états locaux des processus auxquels l'interférence sera *appliquée*, ne permettant donc pas de gain de précision significatif.

Les interférences sont alors partitionnées par la variable partagée qui les étiquette : comme discuté en section 10.2.3, cela nous permettra de clore par transfert après application des interférences en ne calculant que les transferts des variables affectées.

Nous abstrairons enfin les partitions de $\Sigma_{\mathscr{I}} \times \Sigma_{\mathscr{I}}$ dans un domaine numérique. Pour y représenter des paires d'états, nous les regrouperons dans un même environnement numérique : les variables de l'état d'origine seront gardées telles quelles, et nous utiliserons une version *primée* v' de chaque variable v pour représenter l'état de destination [1]. Ainsi, le couple d'états suivant :

$$\langle \mathfrak{pc}_1 \mapsto 1, x \mapsto 2\rangle, \langle \mathfrak{pc}_1 \mapsto 3, x \mapsto 4\rangle$$

Sera représenté par l'élément suivant :

$$\langle \mathfrak{pc}_1 \mapsto 1, x \mapsto 2, \mathfrak{pc}_1' \mapsto 3, x' \mapsto 4\rangle$$

Nous noterons merge la fonction qui fusionne deux états numériques en un seul. Cette technique, qui utilise une représentation classique [CC77b] des relations d'entrée-sortie sur des ensembles de variables (comme les appels de fonctions), nous permet d'obtenir un unique ensemble numérique représentant les interférences [MM17]. Nous pouvons alors l'abstraire directement à l'aide des domaines numériques usuels.

S'il est théoriquement possible d'utiliser n'importe quelle abstraction numérique à ce stade, il s'est avéré plus utile d'utiliser le même domaine que pour les états locaux. En effet, l'application et la génération d'interférences font intervenir des opérations qui manipulent des éléments des deux ensembles : par exemple, les interférences sont générées à partir d'opérations sur les états locaux puis jointes à des interférences déjà existantes, et l'application commence par comparer les états locaux avec les états d'origine des interférences pour sélectionner celles qui sont applicables. Il est alors plus aisé d'utiliser des opérations, comme l'union abstraite, qui sont nativement définies (parce que les deux éléments appartiennent au même domaine), plutôt que de déterminer des opérateurs équivalents fonctionnant sur deux éléments de type différent.

La figure 11.5 formalise le domaine des interférences abstraites et sa concrétisation. Nous omettrons les définitions exactes de $\gamma^{\mathscr{I}}_{\mathsf{num}}$ et $\gamma^{\mathscr{I}}_{Buf}$ par concision : elles sont similaires à leurs équivalents locaux.

11.3 Opérateurs abstraits

En plus des opérateurs locaux de calcul des états utilisés pour l'analyse interne de chaque processus, les opérateurs de l'analyse modulaire incluent la génération des interférences (par les instructions locales d'un processus) et leur application.

L'abstraction locale étant très semblable à l'abstraction monolithique, les opérateurs locaux sont eux aussi quasiment identiques. Nous référons donc le lecteur à la figure 7.4, page 75, pour leur formulation.

1. Considérons pour simplifier que les variables originales ne peuvent pas finir par une apostrophe, ce qui garantit l'unicité d'une variable dans l'environnement regroupé.

$$\begin{gathered}
\mathscr{I} = \mathit{Thread} \to \wp\left(\mathit{Shared}^? \times \Sigma \times \Sigma\right) \\
\mathscr{I}^\sharp \triangleq \mathit{Thread} \to \mathit{Shared}^? \to \mathscr{D}^{\mathtt{num}} \\
\gamma_{\mathscr{I}} : \mathscr{I}^\sharp \to \mathscr{I} \\
\gamma_{\mathscr{I}}(I^\sharp) \triangleq \lambda T.\left\{ \left\langle x^?, \pi_{\mathscr{I}}^{-1}(\gamma_{\mathit{Buf}}^{\mathscr{I}}(\sigma)), \pi_{\mathscr{I}}^{-1}(\gamma_{\mathit{Buf}}^{\mathscr{I}}(\sigma')) \right\rangle \;\middle|\; \begin{array}{c} x^? \in \mathit{Shared}^?, \\ \langle \sigma, \sigma' \rangle \in \mathrm{merge}^{-1}(\gamma_{\mathtt{num}}^{\mathscr{I}}(I^\sharp(T)(x^?))) \end{array} \right\}
\end{gathered}$$

FIGURE 11.5 – Domaine des interférences abstraites.

Les opérateurs de génération des interférences sont quant à eux formalisés en figure 11.6. $\triangleright$ est une notation postfixe pour l'application de fonction : $x \triangleright f \triangleright g$ vaut $g(f(x))$. ${}^{(\ell_1)}[\![op]\!]_T{}^{(\ell_2)}$ calcule les interférences générées par le processus T lorsqu'il effectue l'opération $[\![op]\!]$ entre les points de contrôle (concrets) (ℓ_1) et (ℓ_2). Pour définir cette opération, nous avons recours à une opération intermédiaire $(\!|.|\!)$ agissant sur une partition donnée, et à une autre fonction $(\!|.|\!)$ renvoyant la variable qui doit étiqueter l'interférence générée.

Nous attirons en particulier l'attention du lecteur sur les interférences générées par l'opérateur $[\![x \leftarrow e]\!]$, dont le seul effet est de mettre à jour le point de contrôle de T. En effet, en raison de la mise en tampon de l'écriture effectuée par le processus, elle n'est pas immédiatement visible en mémoire et donc présente dans les interférences (qui, rappelons le, ne contiennent pas d'information sur les tampons). L'écriture effective en mémoire ne se fera qu'après un transfert : elle sera donc présente dans l'interférence correspondante, qui sera elle-même générée lors de la clôture par transferts.

La figure 11.7 définit l'opérateur clos abstrait d'application des interférences. On l'obtient par adaptation automatique de l'opérateur concret aux partitionnements et représentations abstraites.

11.3.1 Sûreté

La sûreté de l'analyse modulaire se fonde sur celle des deux constructions qu'elle met en œuvre : d'une part, l'abstraction monolithique prouvée correcte par le théorème 7.1 page 77, d'autre part l'analyse modulaire concrète traitée par Antoine Miné [Min14]. Nous n'en donnerons donc pas une preuve détaillée mais en énoncerons les idées directrices.

Premièrement, les opérateurs internes de calcul des états locaux étant quasiment identiques à leurs équivalents modulaires, leur preuve de correction ne change essentiellement pas.

Toujours dans le calcul des états locaux, l'opérateur d'application des interférences est obtenu, comme nous l'avons fait remarquer, par une traduction automatique de l'opérateur concret par composition d'opérations abstraites déjà connues comme sûres. Il reste donc sûr par construction.

Considérons enfin la génération des interférences. L'étiquetage par la variable partagée impliquée correspond au classement donné par le lemme 5.1 de commutation des transferts. Quant à la génération de l'interférence elle-même, on observera que, pour chaque opération, la valeur des variables non primées n'est pas modifiée : l'état d'origine est l'état depuis lequel l'interférence est produite. L'information de contrôle du processus actif est bien affectée dans les états d'origine et de destination. Quant aux variables primées, on remarquera que leur valeur leur est affectée de manière identique au résultat des opérateurs abstraits monolithiques de la figure 7.4 (page 75), modulo suppression des tampons (ce qui conduit par ailleurs à simplifier quelques expressions faisant intervenir des variables condensées). Les interférences générées sont donc des

$$Sym \triangleq Var \uplus \{\mathfrak{pc}_T \mid T \in Thread\}$$

$$\text{extend}_T \triangleq \begin{cases} \mathscr{D}^{\text{num}} \to \mathscr{D}^{\text{num}} \\ \quad X \mapsto add(X, \{v' \mid v \in Sym\} \uplus \{\mathfrak{pc}_T\}) \end{cases}$$

$$(\!|.|\!) : Shared^?$$
$$(\!|.|\!)_T : \mathscr{B}^\flat \times \mathscr{D}^{\text{num}} \to \mathscr{D}^{\text{num}}$$

$$(\!|r \leftarrow e|\!) = \begin{cases} x & \text{si } x \in e \\ \bot^? & \text{sinon} \end{cases} \qquad ^{(\ell_1)}(\!|r \leftarrow e|\!)_T{}^{(\ell_2)}(b^\flat, X) = \begin{cases} \text{extend}_T(X) \\ \rhd [\![\forall v \in Sym, v' \leftarrow v]\!] \\ \rhd [\![\mathfrak{pc}_T \leftarrow \dot{\alpha}_{\mathbb{L}_T}(\ell_1)]\!] \circ [\![\mathfrak{pc}'_T \leftarrow \dot{\alpha}_{\mathbb{L}_T}(\ell_2)]\!] \\ \rhd [\![r' \leftarrow \{\!|e|\!\}_T X]\!] \\ \rhd [\![\forall y \in Shared, drop\ \{y_1^T, y_\omega^T\}]\!] \end{cases}$$

$$(\!|\texttt{mfence}|\!) = \bot^? \qquad ^{(\ell_1)}(\!|\texttt{mfence}|\!)_T{}^{(\ell_2)}(b^\flat, X) = \begin{cases} \bot^{\text{num}} & \text{si } \exists b^\flat(x) \neq \tilde{0} \\ \text{extend}_T(X) \\ \rhd [\![\forall v \in Sym, v' \leftarrow v]\!] & \text{sinon} \\ \rhd [\![\mathfrak{pc}_T \leftarrow \dot{\alpha}_{\mathbb{L}_T}(\ell_1)]\!] \circ [\![\mathfrak{pc}'_T \leftarrow \dot{\alpha}_{\mathbb{L}_T}(\ell_2)]\!] \\ \rhd [\![\forall y \in Shared, drop\ \{y_1^T, y_\omega^T\}]\!] \end{cases}$$

$$(\!|x \leftarrow e|\!) = \bot^? \qquad ^{(\ell_1)}(\!|x \leftarrow e|\!)_T{}^{(\ell_2)}(b^\flat, X) = \begin{cases} \text{extend}_T(X) \\ \rhd [\![\forall v \in Sym, v' \leftarrow v]\!] \\ \rhd [\![\mathfrak{pc}_T \leftarrow \dot{\alpha}_{\mathbb{L}_T}(\ell_1)]\!] \circ [\![\mathfrak{pc}'_T \leftarrow \dot{\alpha}_{\mathbb{L}_T}(\ell_2)]\!] \\ \rhd [\![\forall y \in Shared, drop\ \{y_1^T, y_\omega^T\}]\!] \end{cases}$$

$$(\!|\text{flush}\, x|\!) = x \qquad ^{(\ell_1)}(\!|\text{flush}\, x|\!)_T{}^{(\ell_1)}(b^\flat, X) = \begin{cases} \bot^{\text{num}} & \text{si } b^\flat(x) = \tilde{0} \\ \text{extend}_T(X) \\ \rhd [\![\forall v \in Sym, v' \leftarrow v]\!] \\ \rhd [\![\mathfrak{pc}_T \leftarrow \dot{\alpha}_{\mathbb{L}_T}(\ell_1)]\!] \circ [\![\mathfrak{pc}'_T \leftarrow \dot{\alpha}_{\mathbb{L}_T}(\ell_1)]\!] & \text{sinon} \\ \rhd [\![x^{mem\prime} \leftarrow x_1^T \texttt{ if } b^\flat(x) = \tilde{1} \texttt{ else } x_\omega^T]\!] \\ \rhd [\![\forall y \in Shared, drop\ \{y_1^T, y_\omega^T\}]\!] \end{cases}$$

$$[\![.]\!]_T^\sharp : \mathscr{D}_T^\sharp \to \mathscr{I}^\sharp$$
$$^{(\ell_1)}[\![op]\!]_T^\sharp{}^{(\ell_2)}(X^\sharp) = \lambda T'.\lambda x^?. \begin{cases} \displaystyle\bigsqcup_{b^\flat \in \mathscr{B}^\flat}^{sum} {}^{(\ell_1)}(\!|ins|\!)_T^\sharp(b^\flat, X^\sharp(b^\flat))^{(\ell_2)} & \text{si } x^? = (\!|op|\!) \wedge T' = T \\ \bot^{\text{num}} & \text{sinon} \end{cases}$$

FIGURE 11.6 – Opérateurs abstraits pour la génération des interférences.

$$Sym_T \triangleq Var \uplus \mathfrak{PC}_T$$

$$\mathrm{img}_T : \mathscr{D}^{\mathtt{num}} \to \mathscr{D}^{\mathtt{num}}$$
$$\mathrm{img}_T(X^{\mathtt{num}}) = [\![\forall v \in Sym_T, rename(v' \to v)]\!] \left([\![\mathrm{drop}\ \ Sym_T]\!](X^{\mathtt{num}})\right)$$

$$\mathrm{apply}^{\mathtt{num}}_\ell : \mathscr{D}^{\mathtt{num}} \to \mathscr{D}^{\mathtt{num}} \to \mathscr{D}^{\mathtt{num}}$$
$$\mathrm{apply}^{\mathtt{num}}_\ell(I^{\mathtt{num}})(X^{\mathtt{num}}) = X^{\mathtt{num}} \sqcup^{\mathtt{num}} \mathrm{img}_T(\mathrm{extend}_T(X^{\mathtt{num}}) \sqcap^{\mathtt{num}} I^{\mathtt{num}})$$

$$\mathrm{apply}^1_\ell : Shared^? \times \mathscr{D}^{\mathtt{num}} \to \mathscr{D}^\sharp_T \to \mathscr{D}^\sharp_T$$
$$\mathrm{apply}^1_\ell(x^?, I^{\mathtt{num}})(X^\sharp_T) \triangleq \mathtt{let}\ Y^\sharp_T = \lambda b^\flat.\, \mathrm{apply}^{\mathtt{num}}_\ell(I^{\mathtt{num}})(X^{\mathtt{num}}(b^\flat))$$
$$\mathtt{in} \begin{cases} Y^\sharp_T & \text{si } x^? = \bot \\ [\![\mathrm{flush}\, x]\!]^{\sharp *}_T Y^\sharp_T & \text{sinon} \end{cases}$$

$$\mathrm{apply}^\sharp_\ell : \mathscr{I}^\sharp \to \mathscr{D}^\sharp_T \to \mathscr{D}^\sharp_T$$
$$\mathrm{apply}^\sharp_\ell(I^\sharp)(X_T) = \bigsqcup^\sharp_{x^? \in Shared^?} \mathrm{apply}^1_\ell(x^?, I^\sharp(x^?))(X^\sharp_T)$$

FIGURE 11.7 – Application des interférences abstraites.

représentations correctes des transitions rassemblant l'état de départ et un état d'arrivée atteint par application de l'opérateur correspondant.

11.3.2 Conclusion

Pour abstraire les états locaux, nous commençons par oublier les tampons des autres processus. Nous construisons ensuite un domaine similaire à celui de l'interprétation abstraite monolithique : les états sont partitionnés selon la taille des tampons, puis les entrées anciennes de ces derniers sont condensées et une abstraction numérique est appliquée au résultat. La fraction de contrôle des processus externes, spécifique à l'analyse modulaire, est abstraite en partitionnant l'ensemble des points de contrôle de chaque processus en un ensemble réduit de points de contrôle abstraits, dans lequel les variables auxiliaires de contrôle prendront leurs valeurs.

L'abstraction des interférences est similaire : elle ne garde que les informations communes à tous les états locaux. Ainsi, tous les points de contrôle sont abstraits et tous les tampons sont oubliés. En utilisant une technique de dédoublement des variables, nous obtenons alors des états numériques que nous pouvons abstraire avec le même domaine que les états locaux. Pour faciliter le calcul de la clôture par transfert, les interférences sont finalement étiquetées avec la variable partagée éventuelle dont elles représentent une lecture ou une écriture.

Nous obtenons alors une analyse sûre calculable par l'itération abstraite du double point fixe. Il nous faut maintenant la valider par l'expérience, ce que nous ferons dans le chapitre suivant.

Chapitre 12

Expérimentations

12.1 Implémentation de l'analyse modulaire

Pour valider la pertinence de l'analyse modulaire dans les environnements faiblement cohérents, nous l'avons implémentée dans notre analyseur de test. Nous utilisons toujours la bibliothèque Bddapron pour les domaines logico-numériques, en particulier pour pouvoir représenter les labels abstraits comme des types finis. Nous rappelons que le code est disponible à l'adresse suivante :

`https://github.com/thizanne/cormoran`

L'analyse interne des processus réutilise en grande partie le moteur d'analyse monolithique déjà développé. Nous l'avons adapté pour y intégrer l'application des interférences après calcul d'un état, et la génération des interférences dans un ensemble global. Le point fixe externe itère l'analyse interne jusqu'à stabilisation des interférences générées.

Abstraction du contrôle.

Nous n'avons implémenté aucune méthode de partitionnement automatique du contrôle des processus : l'utilisateur doit le spécifier explicitement en ajoutant des notations `label` à des points de séparation choisis (les labels abstraits regroupant alors tous les points de contrôle situés entre deux séparations adjacentes).

Toutefois, nous avons observé que, dans nos tests, il était toujours suffisant d'insérer des séparations en tête de boucle et aux points du programme où une propriété doit être vérifiée. De plus Monat et Miné [MM17] utilisent dans un cadre séquentiellement cohérent une méthode similaire et parviennent à de bons résultats. Nous pensons donc qu'une telle heuristique peut conduire à un partitionnement automatique fiable.

Les mesures présentées dans cette section, analyses monolithiques comprises, ont été faites sur une machine disposant d'un processeur `Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz` et de 8 Go de RAM.

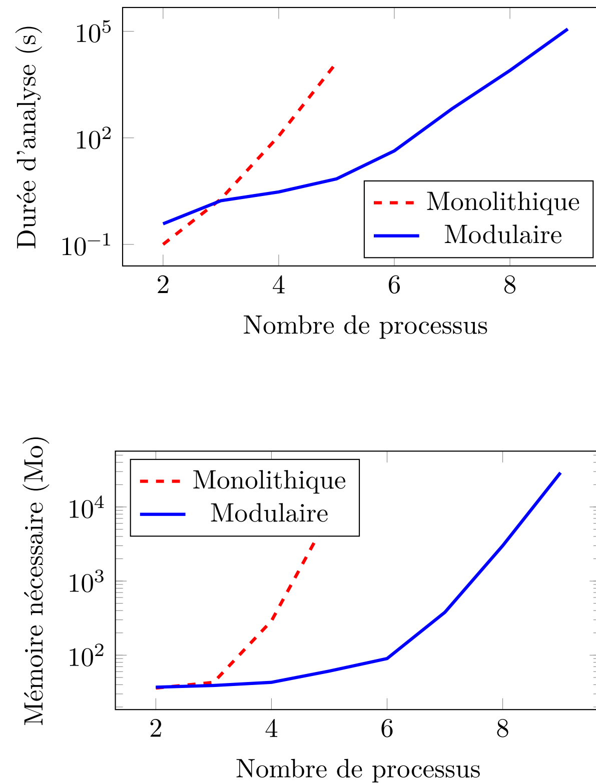


FIGURE 12.1 – Passage à l'échelle en temps et en mémoire.

12.1.1 Nombre d'itérations

Avant les mesures proprement dites des performances de notre analyseur, nous avons déjà observé que le nombre d'itérations externes avant stabilisation des interférences est en général très faible : moins de 5 tours ont été nécessaires pour la quasi-totalité des programmes. Nous avons toutefois noté que ce chiffre était très sensible aux paramètres d'élargissement, rendant difficile son estimation pertinente.

12.2 Passage à l'échelle

Pour vérifier le passage à l'échelle de l'analyse modulaire, nous avons utilisé la version à N processus du programme 9.1 implémentant un tourniquet, et avons mesuré le temps d'exécution et la mémoire nécessaires pour mener à bien les analyses monolithique et modulaire quand N augmente. Le programme 12.1 correspond au cas $N = 4$: on observe que les processus se passent le jeton à tour de rôle. Les résultats sont présentés en figure 12.1.

Ces graphes montrent que l'analyse modulaire passe effectivement mieux à l'échelle : le rapport de performances (sur les deux mesures) entre les deux méthodes est exponentiel. Toutefois, elle reste exponentiellement lente et donc nettement moins efficace qu'en cohérence séquentielle, où elle permet d'analyser plusieurs centaines de processus en quelques heures [MM17].

Plusieurs difficultés d'analyse spécifiquement liées au modèle mémoire nous semblent pouvoir expliquer cette différence. Remarquons tout d'abord que la taille globale du problème augmente

```
/* Check: forall i ≠ j, at critical_i × critical_j, false holds */

/* Initial: x = 0 */

thread /* 0 */ {
    while true {
        while x != 0 {}
        label critical_0;
        x = 1;
    }
}

thread /* 1 */ {
    while true {
        while x != 1 {}
        label critical_1;
        x = 2;
    }
}

thread /* 2 */ {
    while true {
        while x != 2 {}
        label critical_2;
        x = 3;
    }
}

thread /* 3 */ {
    while true {
        while x != 3 {}
        label critical_3;
        x = 0;
    }
}
```

PROGRAMME 12.1 – Algorithme du tourniquet à 4 processus.

Test	Abp	Concloop	Kessel	Dekker	Peterson	Queue	Bakery
Monolithique	✓	✓	✓	✓	✓	✓	⌛
Modulaire	✓	✓	✗	✓	✓	✓	✗

FIGURE 12.2 – Précision de l'analyse sur des programmes tests.

sensiblement plus vite dans notre modèle mémoire. Considérons en effet un programme, comme le tourniquet, où le nombre de variables est toujours le même. En cohérence séquentielle, un processus supplémentaire n'ajoute que du code à analyser et un point de contrôle en plus. Dans notre modèle, il vient en revanche avec ses tampons, soit des dimensions supplémentaires à l'état global alors même que le nombre de variables du programme n'a pas augmenté. Toutefois, notre abstraction ne représente jamais ces dimensions supplémentaires simultanément : l'augmentation de la taille de l'état global n'a donc pas d'influence directe sur la taille des domaines manipulés (mais ajouter un processus reste plus coûteux).

Le profilage de l'analyse a semblé par ailleurs montrer que l'opération la plus coûteuse correspondait à l'application des interférences, et plus particulièrement à la clôture par transfert. Considérons en effet un tourniquet à N processus, et appliquons les interférences au premier. En cohérence séquentielle, il suffit d'appliquer une fois les interférences générées par les $N - 1$ suivants. Mais nous devons également, dans notre modèle, clore par transfert après ces applications. Cette clôture est calculée pour chaque interférence : elle doit donc être répétée N fois. Or nous avons vu qu'une autre phase d'application est nécessaire après clôture, car de nouvelles interférences sont peut-être valides : on répète donc cette étape, ainsi que la clôture qui suit, jusqu'à atteindre un point fixe. S'il est possible d'améliorer cette méthode naïve en regroupant certains calculs, on ne peut toutefois échapper à une complexité largement supérieure à celle de la cohérence séquentielle, qu'on ne peut atteindre sans perte significative de précision.

12.3 Test de précision

L'efficacité de l'analyse modulaire a un coût : les abstractions supplémentaires, par exemple des états de contrôle, peuvent conduire à des pertes de précision. Pour évaluer cette régression potentielle, nous l'avons comparée avec les résultats déjà présentés de l'analyse monolithique en figure 12.2. Rappelons que l'analyse de ces programmes doit vérifier des propriétés de sûreté exprimées par des invariants numériques, comme l'exclusion mutuelle.

On observe que, dans la plupart des cas, la méthode modulaire parvient à analyser le programme avec une précision suffisante pour prouver sa correction, malgré l'abstraction supplémentaire. Toutefois, elle échoue sur deux tests, `kessel` et `bakery`. Nous pensons qu'elle parviendrait à les passer avec un partitionnement du contrôle mieux choisi, mais nos heuristiques manuelles n'ont pas permis de le déterminer.

Par ailleurs, `bakery` est un programme sensiblement plus gros que les autres. Notre analyse n'a certes pas permis de le vérifier, mais elle a terminé (en quelques minutes avec une abstraction agressive), alors que l'analyse monolithique a dû être interrompue au bout de plusieurs jours. On ne peut pas considérer cela comme un véritable passage à l'échelle, en raison de l'échec (il est trivial d'écrire une analyse qui termine immédiatement en échouant), mais c'est un indice supplémentaire des avantages de la méthode.

12.3.1 TSO et PSO

Notre analyse suppose que les programmes à analyser tournent dans le modèle PSO. Cependant certains programmes peuvent être corrects dans TSO, plus strict, et incorrects dans PSO : par exemple, il est parfois possible de supprimer des barrières entre des écritures de variables différentes et obtenir ainsi un programme qui n'est plus correct dans PSO mais le restera dans TSO. Notre prototype, sûr vis-à-vis de PSO, ne pourra pas vérifier ces programmes.

Or TSO est notre cible prioritaire : il s'agit donc d'une perte de précision. Cependant, nous avons observé que tous les tests de régression, excepté `peterson`, ont été vérifiés avec le nombre minimal de barrières pour qu'ils soient déjà corrects dans TSO. Cette observation valide ainsi notre choix d'abstraction, même avec TSO pour cible.

Par ailleurs, parmi les pistes d'amélioration de ce travail présentées en partie IV, certaines permettront de retrouver une partie de la précision de TSO. Elles permettront alors éventuellement de vérifier certains programmes dans ce modèle avec un nombre encore réduit de barrières.

12.4 Conclusion

Les abstractions que nous avons définies engendrent une analyse aux résultats prometteurs : sur les exemples exploitant les comportements relâchés, l'abstraction supplémentaire garde quasi-systématiquement suffisamment de précision pour pouvoir vérifier les propriétés voulues. De plus, ses performances sont bien meilleures lorsque le nombre de processus d'un programme simple augmente.

Toutefois, ce passage à l'échelle de notre prototype reste limité par rapport aux analyses en cohérence séquentielle. La complexité supplémentaire des états de notre modèle et la nécessité de clore par transfert permettent sans doute d'expliquer ces difficultés, mais il reste à mettre en œuvre les mesures permettant de les contourner. Nous discuterons dans la partie suivante des pistes d'amélioration, à la fois de la précision et des performances de l'analyse, pour obtenir un analyseur plus complet.

Quatrième partie

Approfondissement et perspectives

Chapitre 13

Ordre inter-variables dans TSO

Nous avons fait remarquer en section 12.3.1 que les abstractions que nous avons présentées jusque-là oublient totalement l'ordre des entrées concernant différentes variables dans les tampons. Nous proposerons dans ce chapitre un domaine permettant de retrouver en partie cet ordre, afin d'obtenir une analyse plus précise dans TSO (qui ne sera évidemment plus correcte dans PSO).

Ce domaine consiste à ordonner entre elles les variables de la forme x_1^T et y_ω^T, représentant des entrées d'un tampon condensé. Comme les y_ω^T représentent chacun plusieurs entrées, il n'est pas toujours possible de les ordonner totalement : nous définirons donc un ordre partiel. Ainsi une variable condensée x_ω^T est antérieure à une autre y_ω^T si et seulement si toutes les entrées de x sont antérieures à toutes celles de y (excepté éventuellement la première, qui correspond à y_1^T). Cet ordre est donc un relâchement de l'« ordre total » qui définit TSO (qui n'est pas représentable, étant de taille non borné).

Pour construire ce domaine, nous aurons recours à une méthode appelée produit réduit [CC79], que nous commencerons par présenter. Nous définirons ensuite le domaine de l'ordre, puis son produit avec les domaines présentés dans les parties précédentes pour établir l'abstraction finale.

Comme pour les chapitres qui suivront dans cette partie IV, cette abstraction fait partie des pistes d'amélioration de la précision que nous avons pu étudier partiellement sans avoir le temps de les implémenter dans notre prototype. Ces chapitres exploreront donc les idées sous-jacentes sans les formaliser ni les évaluer expérimentalement dans tous leurs détails.

13.1 Produit de domaines

Le produit réduit [CC79] est une technique permettant de combiner deux domaines abstraits afin d'obtenir une analyse plus précise.

Nous le présenterons à travers des analyses du programme 13.1, en montrant comment différentes constructions de *domaines produits* permettent d'obtenir des analyses plus précises que celles qui n'utilisent que les domaines de base.

Dans le concret, la boucle d'incrémentation de ce programme parcourt les nombres impairs jusqu'à atteindre $x = 11$ et terminer. La branche du `if` n'est pas empruntée, et x reste égal à 11 à la fin de l'exécution.

```
1  x = 1;
2  while (x <= 10) {
3      x = x + 2;
4  }
5
6  if (x >= 12) {
7      x = 0;
8  }
9  /* assert x = 11 */
```

Programme 13.1 – Incrémentation de 2 en 2.

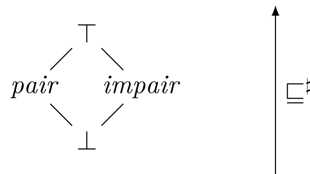


Figure 13.1 – Diagramme de Hasse du domaine des parités.

13.1.1 Analyses séparées

En première approche, nous effectuerons deux analyses successives du programme 13.1, à l'aide des domaines des intervalles puis des parités.

Intervalles.

Analysons d'abord ce programme à l'aide des intervalles. Ce domaine stabilise l'analyse de la boucle en inférant l'invariant $x = [1, 12]$ en tête de boucle, soit $x = [11, 12]$ à la sortie.

La branche `if` peut alors être empruntée si $x = 12$, générant l'élément abstrait $[0, 0]$, ou ne pas l'être si $x = 11$, générant l'élément $[11, 11]$. L'invariant final en fin de programme est la borne supérieure des différents chemins possibles, soit $x = [0, 11]$. La propriété de la ligne 9 n'est pas vérifiée.

Parités.

Le domaine des parités, introduit par Patrick et Radhia Cousot [CC79], est un domaine non relationnel qui ne garde pour chaque variable que l'information de sa parité arithmétique. Son diagramme de Hasse est représenté en figure 13.1, et nous ne le présenterons pas plus en détail, considérant qu'il s'agit d'un domaine simple qui ne nous servira que pour présenter les domaines produits. Nous mentionnerons toutefois le domaine des congruences [Gra89], qui en est une généralisation inférant des invariants de la forme $x \in a\mathbb{Z} + b$, et qui est notamment utile pour inférer des propriétés sur l'alignement des pointeurs et les accès en mémoire.

Remarquons tout d'abord que la propriété de la ligne 9 n'est pas exprimable précisément dans le domaine de la parité : on peut donc déjà s'attendre à ce que l'analyse ne soit pas en mesure de la vérifier.

De fait, l'invariant inféré en fin de boucle est x *est impair*. Le test `x >= 12` se traduit dans le domaine des parités par l'identité, tout comme sa négation : ainsi, la branche conditionnelle peut être empruntée (rendant alors x pair) ou non (laissant x impair). En fin d'exécution du

$$\begin{aligned}
\mathscr{D}_{1\times 2} &\triangleq \mathscr{D}_1 \times \mathscr{D}_2 \\
\gamma_{1\times 2}(A_1, A_2) &\triangleq \gamma_1(A_1) \cap \gamma_2(A_2) \\
\alpha_{1\times 2}(X) &= (\alpha_1(X), \alpha_2(X)) \qquad \text{(Si } \alpha_1 \text{ et } \alpha_2 \text{ existent)} \\
(A_1, A_2) \sqsubseteq_{1\times 2} (B_1, B_2) &\stackrel{\triangle}{\equiv} A_1 \sqsubseteq_1 B_1 \wedge A_2 \sqsubseteq B_2 \\
\bot_{1\times 2} &\triangleq (\bot_1, \bot_2) \\
\top_{1\times 2} &\triangleq (\top_1, \top_2) \\
(A_1, A_2) \sqcup_{1\times 2} (B_1, B_2) &\stackrel{\triangle}{\equiv} (A_1 \sqcup_1 B_1), (A_2 \sqcup_2 B_2) \\
(A_1, A_2) \sqcap_{1\times 2} (B_1, B_2) &\stackrel{\triangle}{\equiv} (A_1 \sqcap_1 B_1), (A_2 \sqcap_2 B_2) \\
(A_1, A_2) \nabla_{1\times 2} (B_1, B_2) &\stackrel{\triangle}{\equiv} (A_1 \nabla_1 B_1), (A_2 \nabla_2 B_2) \\
[\![op]\!]_{1\times 2}(A_1, A_2) &\triangleq ([\![op]\!]_1 A_1, [\![op]\!]_2 A_2)
\end{aligned}$$

FIGURE 13.2 – Produit direct de deux domaines.

programme, l'information de parité est alors perdue : l'invariant inféré est $\top$. La propriété de la ligne 9 n'est effectivement pas vérifiée.

13.1.2 Produit direct

Si aucun des deux domaines, pris séparément, n'est capable d'inférer la propriété $x = 11$ en fin de programme, on remarque que leur combinaison, intuitivement, le permet : en effet, le domaine des intervalles infère $x \in [11, 12]$ en sortie de boucle, et le domaine des parités permet de montrer *x est impair*. Or 11 est le seul entier impair dans l'intervalle $[11, 12]$. Il s'ensuit que la conditionnelle ne peut pas être empruntée, et que $x = 11$ reste vrai en fin de programme.

Pour formaliser cette combinaison, une première approche consiste à construire le *produit direct* des domaines des intervalles et des parités. Les éléments du produit direct sont les paires d'éléments de chacun des deux domaines, et ils représentent la conjonction des deux invariants correspondants. On calcule les opérateurs abstraits sur le domaine produit en les appliquant composant par composant. Ceci revient à mener en parallèle les deux analyses.

La figure 13.2 définit le produit direct $\mathscr{D}_{1\times 2}$ de deux domaines $\mathscr{D}_1$ et $\mathscr{D}_2$. La conjonction des invariants est formalisée par une intersection lors de la concrétisation.

Cependant, appliqué au programme 13.1, ce domaine est toujours insuffisant pour prouver sa correction : en effet, l'invariant inféré en fin de programme est $([0, 11], \top)$. L'invariant de parité n'apporte ici aucune information supplémentaire : la conjonction des deux propriétés ne permet toujours de prouver que $x \in [0, 11]$.

13.1.3 Produit réduit

La conjonction des invariants apportée par le produit direct n'est donc pas suffisante pour prouver le programme 13.1. Pourtant, comme nous l'avons montré intuitivement dans le concret, le domaine sous-jacent possède toute l'information nécessaire : en effet, en sortie de boucle, elle infère l'invariant $([11, 12], \textit{impair})$. Or 11 est le seul entier impair dans l'intervalle $[11, 12]$, et si $x = 11$ en sortie de boucle, alors la branche conditionnelle n'est pas empruntée et ce même invariant reste vrai en fin de programme.

Cette imprécision est due au fait que les opérateurs sont appliqués en parallèle sur les deux composants du domaine, sans qu'il y ait échange d'information. Autrement dit, même si l'opérateur abstrait sur chaque composant est optimal, l'opérateur produit tel qu'il est défini en figure 13.2 n'est pas optimal vis-à-vis de la correspondance de Galois du domaine produit (définie sur la même figure). Ainsi, le test d'entrée $[\![x \geq 12]\!]$ dans la boucle depuis l'invariant produit $([11, 12], \mathit{impair})$ génère l'élément abstrait $([12, 12], \mathit{impair})$. La concrétisation de cet élément abstrait est $\bot$, mais sans communication entre ses deux composants, l'analyse n'est pas en mesure de l'inférer : elle suppose alors qu'il est possible d'entrer dans la conditionnelle, avec le résultat que nous avons vu.

Pour lever cette imprécision, il faut donc faire passer des informations entre les composants au cours de l'analyse plutôt que mener les deux interprétations abstraites en parallèle. Cet échange se fait *via* une fonction de réduction $\rho : \mathscr{D}_{1\times 2} \to \mathscr{D}_{1\times 2}$, qui restreint chaque composant du produit en utilisant les informations de l'autre composant pour en éliminer les états impossibles. Cette réduction est effectuée après le calcul de chaque opérateur abstrait pour en préciser le résultat. Le domaine ainsi obtenu est appelé produit réduit des domaines initiaux [CC79].

Par exemple, dans l'analyse par produit direct du programme 13.1, nous avons rencontré l'élément abstrait $([11, 12], \mathit{impair})$ en sortie de boucle. Sur cet élément, la fonction de réduction peut utiliser l'information *impair* pour restreindre le premier composant à $[11, 11]$. L'invariant réduit est alors $([11, 11], \mathit{impair})$. L'image directe de la condition d'entrée dans la conditionnelle est alors $(\bot, \mathit{impair})$, qui devient après réduction $(\bot, \bot)$. La branche n'est pas empruntée : l'invariant en fin d'exécution reste $([11, 11], \mathit{impair})$, ce qui permet de prouver la propriété recherchée.

Lorsque les domaines $\mathscr{D}_1$ et $\mathscr{D}_2$ sont en correspondance de Galois avec le domaine concret, il est possible de définir une réduction optimale à l'aide des fonctions d'abstraction α_1 et α_2 correspondantes. Toutefois, lorsque ce n'est pas le cas, comme pour les opérateurs abstraits, il est possible de se contenter d'une fonction de réduction partielle, qui doit alors respecter deux contraintes : d'une part, son application doit rester sûre (elle ne supprime pas d'états concrets après concrétisation par $\gamma_{1\times 2}$), d'autre part, elle doit renforcer les invariants représentés dans chaque composant. La figure 13.3 récapitule ces définitions, ainsi que le calcul des opérateurs dans le produit réduit (qu'on enchaîne avec une réduction afin d'améliorer la précision). Le lecteur remarquera qu'on n'effectue pas de réduction après application d'un opérateur d'élargissement : on perdrait en effet la garantie de terminaison des séquences élargies, qui sont stationnaires pour chaque composant (donc pour leur produit) si l'élargissement, après un certain rang, est appliqué à chaque étape sans entrelacement avec une réduction.

Exemple 13.1 (Intervalles et parité).
Les domaines des intervalles et des parités sont tous deux en correspondance de Galois avec les domaines numériques concrets. Il existe donc une réduction optimale. Celle-ci commence par raffiner les bornes des intervalles en utilisant l'information de parité. Ensuite, si l'intervalle obtenu est $\bot$ ou un singleton $[a', a']$, la parité peut elle-même être raffinée respectivement en $\bot$ ou en la parité de a' (qui est en fait l'abstraction $\alpha(\{a'\})$ du domaine des parités). La fonction ainsi obtenue est la suivante :

$$\rho([a, b], \bot) \triangleq (\bot, \bot)$$

$$\rho([a, b], \top) \triangleq \begin{cases} (\bot, \bot) & \text{si } a > b \\ ([a, a], \mathit{parit\acute{e}}(a)) & \text{si } a = b \\ ([a, b], \top) & \text{sinon} \end{cases}$$

$$\rho([a, b], \mathit{pair}) \triangleq \begin{cases} (\bot, \bot) & \text{si } a' > b' \\ ([a', b'], \mathit{pair}) & \text{sinon} \end{cases} \qquad \textit{(Idem pour impair)}$$

$$\mathscr{D} \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \mathscr{D}_1$$
$$\mathscr{D} \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \mathscr{D}_2$$
$$\rho : \mathscr{D}_{1\times 2} \to \mathscr{D}_{1\times 2}$$
$$\rho(X) \triangleq (\alpha_1(\gamma_{1\times 2}(X)), \alpha_2(\gamma_{1\times 2}(X)))$$

(a) Réduction complète en présence de correspondances de Galois.

$$\gamma_1 : \mathscr{D}_1 \to \mathscr{D}$$
$$\gamma_2 : \mathscr{D}_2 \to \mathscr{D}$$
$$\begin{aligned}(Y_1, Y_2) = \rho(X_1, X_2) \implies\ & \gamma_{1\times 2}(Y_1, Y_2) = \gamma_{1\times 2}(X_1, X_2) && \textit{(Sûreté)}\\ & \wedge \gamma_1(Y_1) \subseteq \gamma_1(X_1) && \textit{(Renforcement de } X_1\textit{)}\\ & \wedge \gamma_2(Y_2) \subseteq \gamma_2(X_2) && \textit{(Renforcement de } X_2\textit{)}\end{aligned}$$

(b) Propriété caractéristique d'une réduction partielle ρ.

$$(A_1, A_2) \sqcup_{1\times 2} (B_1, B_2) \stackrel{\triangle}{\equiv} \rho(A_1 \sqcup_1 B_1, A_2 \sqcup_2 B_2)$$
$$(A_1, A_2) \sqcap_{1\times 2} (B_1, B_2) \stackrel{\triangle}{\equiv} \rho(A_1 \sqcap_1 B_1, A_2 \sqcap_2 B_2)$$
$$(A_1, A_2) \nabla_{1\times 2} (B_1, B_2) \stackrel{\triangle}{\equiv} (A_1 \nabla_1 B_1, A_2 \nabla_2 B_2)$$
$$[\![op]\!]_{1\times 2}(A_1, A_2) \triangleq \rho([\![op]\!]_1 A_1, [\![op]\!]_2 A_2)$$

(c) Définition des opérateurs abstraits sur un produit réduit.

FIGURE 13.3 – Produit réduit de deux domaines.

```
/* initial flag = 0 && turn = 0 */

thread /* T0 */ {
    flag = 1;
    turn = 1;
}

thread /* T1 */ {
    /* local f, t */
    t = turn;
    f = flag;
    /* assert not (f = 0 && t = 1) */
}
```

Programme 13.2 – Extrait simplifié de l'algorithme de Peterson

où :

$$a' = \min \{x \geq a \mid x \text{ } \textit{est pair}\}$$
$$b' = \max \{x \leq b \mid x \text{ } \textit{est pair}\}$$

Cette réduction est appliquée dimension par dimension si l'environnement comprend plus d'une variable. ○

Exemple 13.2 (Réduction simple).
La fonction identité $\rho = \lambda x.x$ est une réduction partielle triviale : elle est toujours sûre, et ne renforce ni ne relâche les invariants respectifs. Elle revient à considérer le produit direct.

Une autre réduction simple, un peu plus précise, consiste à ne fusionner que les plus petits éléments de chaque domaine :

$$\rho(X_1, X_2) \triangleq \begin{cases} (\bot_1, \bot_2) & \text{si } X_1 = \bot_1 \vee X_2 = \bot_2 \\ (X_1, X_2) & \text{sinon} \end{cases}$$

Cette réduction permet de propager une information d'inaccessibilité d'un composant sur l'autre. Nous utiliserons une réduction similaire sur les tampons.

13.2 Considérations informelles sur l'ordre des variables

Nous présenterons maintenant informellement le domaine de l'ordre en détaillant son application sur plusieurs exemples choisis. Ce domaine, qui abstrait l'ordre des variables dans les tampons, a vocation à être combiné par produit réduit avec les domaines des parties précédentes qui abstraient leur contenu.

13.2.1 Ordonnancement des entrées

Considérons en premier lieu le programme 13.2. Il s'agit d'un extrait simplifié de l'algorithme d'exclusion mutuelle de Peterson [Pet81], présenté en annexe à la page 168.

Dans ce programme, un processus écrit successivement dans deux variables `flag` et `turn`, qui sont lues par un autre processus. La propriété de correction de l'extrait demande à ce que les

```
1 thread {
2     while(true) {
3         x = 1;
4         y = 1;
5     }
6 }
```

PROGRAMME 13.3 – Écriture en continu dans deux variables.

écritures éventuelles par le premier processus soient lues dans le même ordre par le second : dans le programme complet, le comportement contraire pourrait mener les deux processus à entrer simultanément en section critique. Cette propriété est bien vérifiée dans TSO : en effet, si `T1` lit `1` dans `turn`, alors l'écriture antérieure de `flag` lui est également visible et il ne peut donc pas y lire `0`. En revanche, dans PSO, ces écritures ne sont pas ordonnées, et le comportement interdit est possible. Les analyses des parties précédentes ne permettront donc pas de prouver la correction de ce programme dans TSO.

Pour résoudre ce problème, nous ajouterons au domaine des états une relation d'ordre strict $\prec$ entre les écritures de différentes variables. Cette information permettra d'interdire le transfert d'une entrée du tampon dont on sait qu'elle est précédée par des entrées antérieures.

Déroulons l'interprétation abstraite modulaire de la partie III sur le programme 13.2. Après exécution de la deuxième instruction du processus `T0`, et avant clôture par transfert, son état local contiendra les partitions suivantes :

$$flag^{mem} = 0 \wedge turn^{mem} = 0 \wedge flag_1 = turn_1 = 1 \wedge flag_1 \prec turn_1$$
$$flag^{mem} = 1 \wedge turn^{mem} = 0 \wedge turn_1 = 1$$

Procédons maintenant à la clôture par transfert. Nous savons qu'il suffit de calculer les éventuels transferts de *turn*, l'instruction que nous venons d'exécuter étant une écriture dans cette variable. Dans la première partition, l'information $flag_1 \prec turn_1$ permet d'interdire ce transfert : il existe une entrée antérieure, qui devra elle-même être transférée au préalable. Le seul transfert à effectuer vient donc de la deuxième partition, et l'interférence générée est la suivante (en oubliant pour l'alléger les informations de contrôle) :

$$flag^{mem} = flag^{mem'} = 1 \wedge turn^{mem} = 0 \wedge turn^{mem'} = 1$$

Cette interférence précise que le processus `T0` ne peut écrire `1` dans `turn` que si `flag` vaut aussi `1` en mémoire partagée. Lors de l'analyse du processus `T1`, il s'agit de la seule interférence permettant de mettre à jour `turn` avant sa lecture. Dans tous les états où cette lecture aura renvoyé `1`, pourvu que le domaine numérique soit suffisamment précis (ce qui est le cas de tous les domaines relationnels que nous avons utilisés), l'analyse sera donc capable d'inférer que, `flag` valant alors également `1`, la lecture qui s'ensuit renverra la même valeur. La propriété est vérifiée.

Il est possible d'ordonner naturellement plusieurs variables à l'aide de cette information. $\prec$ sera alors une relation d'ordre strict, représentant l'ordre des écritures en mémoire.

13.2.2 Condensation de l'ordre

Pour qu'elle reste calculable, nous souhaitons obtenir une représentation de taille bornée pour $\prec$. Considérons par exemple le processus du programme 13.3, qui écrit simplement en continu et alternativement dans des variables `x` et `y`.

Dans le concret, la relation d'ordre correspondante après n tours de boucle serait alors :

$$x_n \prec y_n \prec x_{n-1} \prec \cdots \prec x_1 \prec y_1$$

La taille de cette relation n'est donc pas bornée, et après interprétation complète du processus, elle est même infinie. Pour obtenir une représentation bornée à coup sûr de relations d'ordre qui peuvent être de taille infinie (ce qui est nécessaire dans le cadre d'une analyse calculable), il est donc nécessaire d'abstraire.

Un choix naturel d'abstraction consiste alors à appliquer à ces relations d'ordre une transformation similaire à la condensation des tampons que nous avons déjà utilisée : plutôt que de porter sur toutes les entrées des tampons, on considérera la relation d'ordre abstraite entre les entrées condensées. Ainsi, dans l'exemple du programme 13.3, à la fin d'un tour de boucle, l'entrée la plus récente de y est postérieure à l'entrée la plus récente de x :

$$x_1 \prec y_1$$

De plus, l'entrée la plus récente de x est elle-même postérieure à toutes les autres entrées de y :

$$y_\omega \prec x_1$$

On remarquera que, pour n'importe quelle variable z, la relation $z_\omega \prec z_1$ est toujours vérifiée. Par transitivité, $x_1 \prec y_1$ implique donc $x_\omega \prec y_1$.

En revanche, il n'est pas possible d'ordonner x_ω et y_ω : les entrées correspondantes sont entrelacées. L'ordre obtenu est donc partiel, et peut se résumer par :

$$y_\omega \prec x_1 \prec y_1$$

À la prochaine itération de la boucle, le processus écrira dans x. L'entrée actuelle x_1 rejoindra alors x_ω, sur lequel aucune information n'est disponible : elle disparaîtra alors de l'ordre partiel. Le résultat obtenu sera alors :

$$y_\omega \prec y_1 \prec x_1$$

Après réécriture de y, on obtient alors de nouveau l'ordre précédemment établi en fin de tour de boucle.

13.2.3 Partitionnement et treillis des ordres

Partitionnement.

De la même façon que les états numériques, le lecteur aura remarqué que les ordres condensés peuvent être définis sur des ensembles de variables différents. Il est alors raisonnable de leur appliquer le même traitement, soit un partitionnement selon cet ensemble de variables.

Considérons par exemple les ordres concrets obtenus après exécution du programme 13.4. Après clôture par transfert, les partitions suivantes sont accessibles ($\varnothing$ étant le seul ordre strict existant sur les ensembles d'au plus un élément) :

$$\begin{aligned}
&x \mapsto \tilde{0}, y \mapsto \tilde{0}, z \mapsto \tilde{0} : \{\varnothing\}\\
&x \mapsto \tilde{0}, y \mapsto \tilde{1}, z \mapsto \tilde{0} : \{\varnothing\}\\
&x \mapsto \tilde{1}, y \mapsto \tilde{0}, z \mapsto \tilde{0} : \{\varnothing\}\\
&x \mapsto \tilde{1}, y \mapsto \tilde{1}, z \mapsto \tilde{0} : \{x_1 \prec y_1, y_1 \prec z_1\}\\
&x \mapsto \tilde{1}, y \mapsto \tilde{1}, z \mapsto \tilde{1} : \{z_1 \prec x_1 \prec y_1, z_1 \prec y_1 \prec x_1\}
\end{aligned}$$

```
thread {
    z = 1;
    if (random) {
        x = 1;
        y = 2;
    } else {
        y = 1;
        x = 2;
    }
}
```

Programme 13.4 – Plusieurs ordres d'écriture possibles.

À chaque partition est associé un ensemble d'ordres concrets. Dans ce cas précis, selon la branche du `if` empruntée, x peut notamment être antérieur ou postérieur à y dans les états où les deux sont présents.

Après condensation, l'ensemble des ordres possibles dans chaque partition est borné. On peut donc le représenter exactement sans perdre la calculabilité. Dans chaque partition, le domaine de l'ordre est alors défini par les ensembles d'ordre, muni des opérations habituelles : la borne supérieure est l'union ensembliste, la borne inférieure l'intersection, l'élément minimal l'ensemble vide et l'élément maximal l'ensemble de tous les ordres possibles sur les variables de la partition.

Treillis des ordres.

Pour réduire la taille de la représentation, il est aussi possible d'abstraire chaque ensemble d'ordres par l'ordre qui ne contient que les relations vérifiées par tous les ordres de l'ensemble.

Considérons par exemple la partition $\{z_1 \prec x_1 \prec y_1, z_1 \prec y_1 \prec x_1\}$ de l'exemple précédent. Elle contient deux éléments, qui diffèrent spécifiquement sur l'ordre de x_1 et y_1. Il est possible de l'abstraire en un seul élément qui ne contient que les relations certaines :

$$z_1 \prec x_1 \wedge z_1 \prec y_1$$

Sur cet exemple, l'abstraction obtenue décrit exactement l'ensemble de départ. Ce n'est pas toujours le cas. Considérons par exemple la partition suivante :

$$x \mapsto \tilde{1}, y \mapsto \tilde{1}, z \mapsto \tilde{1} : \{x_1 \prec y_1 \prec z_1, y_1 \prec z_1 \prec x_1\}$$

L'abstraction ne garde comme information certaine que l'ordre entre y et z :

$$x \mapsto \tilde{1}, y \mapsto \tilde{1}, z \mapsto \tilde{1} : y_1 \prec z_1$$

Cette abstraction décrit des états absents de l'ensemble d'ordres de départ, par exemple :

$$y_1 \prec x_1 \prec z_1$$

Le fait que cette abstraction puisse mener à une perte de précision était prévisible : le cas contraire signifierait qu'il existe une surjection depuis l'ensemble des ordres vers l'ensemble de ses parties, ce qui est impossible.

Cette abstraction mène à considérer la définition d'un treillis sur les ordres abstraits.

La relation d'ordre $\sqsubseteq^{\prec}$ entre les ordres traduira l'information « est plus précis que ». Dans le cadre des ordres partiels, un ordre $\prec_1$ est plus précis qu'un ordre $\prec_2$ si tous les couples d'éléments

ordonnés présents dans $\prec_2$ sont également présents dans $\prec_1$ (qui peut en contenir d'autres, précisant ainsi des relations qui étaient inconnues dans $\prec_2$). Dans le point de vue des ordres vus comme des ensembles de couples d'éléments ordonnés, $\sqsubseteq^{\prec}$ correspond donc à la réciproque de l'inclusion $\supseteq$. Ainsi, la relation suivante est vérifiée :

$$x_1 \prec y_1 \prec z_1 \sqsubseteq^{\prec} x_1 \prec z_1 \wedge y_1 \prec z_1$$

Ce qui correspond à l'inclusion réciproque :

$$\{(x_1, y_1), (x_1, z_1), (y_1, z_1)\} \supseteq \{(x_1, z_1), (y_1, z_1)\}$$

De même, la borne supérieure $\sqcup^{\prec}$ correspond à l'intersection ensembliste $\cap$: l'ordre le plus précis qui est moins précis que $\prec_1$ et $\prec_2$ contient exactement toutes les relations qui sont présentes dans $\prec_1$ et dans $\prec_2$. Reprenons l'exemple vu plus haut :

$$z_1 \prec x_1 \prec y_1 \sqcup^{\prec} z_1 \prec y_1 \prec x_1 = z_1 \prec x_1 \wedge z_1 \prec y_1$$

De façon duale, la définition de la borne inférieure correspond intuitivement à l'union ensembliste. Cependant, l'ensemble des relations d'ordre strict n'est pas clos par union : cette opération ne maintient ni la transitivité, ni l'asymétrie.

Considérons par exemple les deux ordres suivants :

$$x_1 \prec_1 y_1 ; y_1 \prec_2 z_1$$

Leur union ensembliste n'est pas transitive :

$$x_1 \,(\prec_1 \cup \prec_2)\, y_1 \wedge y_1 \,(\prec_1 \cup \prec_2)\, z_1 \wedge \neg\, (x_1 \,(\prec_1 \cup \prec_2)\, z_1)$$

De même, soient les ordres suivants :

$$x_1 \prec_1 y_1 ; y_1 \prec_2 x_1$$

Leur union ensembliste n'est pas asymétrique :

$$x_1(\prec_1 \cup \prec_2)\, y_1 \wedge y_1(\prec_1 \cup \prec_2)\, x_1$$

La transitivité s'obtient simplement en calculant la clôture transitive $(\prec_1 \cup \prec_2)^+$ de l'union des ordres initiaux.

Pour l'asymétrie, nous compléterons les ordres partiels par une valeur $\perp^{\prec}$, par définition inférieure à tous les autres ordres partiels. La borne inférieure de deux ordres dont l'intersection n'est pas asymétrique (parce qu'ils contiennent des relations incompatibles car réciproques) vaudra alors $\perp^{\prec}$. Autrement dit, $\perp^{\prec}$ représente un ensemble vide d'ordres, c'est-à-dire un état inaccessible.

Dualement, il existe pour chaque partition d'index $b^\flat$ un élément maximal $\top^{\prec}(b^\flat)$: c'est la borne supérieure de tous les ordres partiels. Pour les partitions $b^\flat$ qui contiennent des variables x_ω, $\top^{\prec}(b^\flat)$ ne contient que les relations de la forme $x_\omega \prec x_1$. Pour les autres, c'est l'ordre vide : il ne contient aucune information connue.

13.2.4 Opérations sur les ordres abstraits

La lecture d'une variable partagée et le test d'une condition ne modifient pas les tampons : dans le domaine de l'ordre abstrait, elles se traduisent par l'identité. Les barrières agissent comme un filtre, ne gardant que les états aux tampons vides : sa formulation sur les ordres ne posera pas de difficulté particulière, ne gardant que la partition éventuelle où tous les tampons sont vides. Il nous reste alors à formuler les deux opérations qui peuvent modifier les tampons : l'écriture d'une variable partagée et le transfert vers la mémoire. Nous présenterons ici leurs particularités respectives.

Écriture.

Dans le domaine des ordres, quel que soit l'ordre initial, après une écriture de x, x_1 sera toujours l'entrée la plus récente de l'ordre résultat. Les relations $y_i \prec z_j$, pour y et z distincts de x, ne sont quant à elles pas modifiées. Il s'agit des deux seules informations vraies dans tous les cas : le reste des relations dépend de la présence initiale de x.

Si x est initialement absent du tampon, il n'existe alors aucune autre relation : ces deux informations définissent donc totalement le résultat de l'écriture. Toutes les relations sont gardées, et on ajoute une relation $y_i \prec x_1$ pour toute entrée du tampon (y compris celles sur lesquelles aucune information d'ordre n'était disponible).

Ainsi, si l'ordre suivant est défini sur les entrées $\{y_1, y_\omega, z_1\}$:

$$y_\omega \prec z_1$$

Alors le résultat d'une écriture de x sera l'ordre suivant :

$$y_\omega \prec z_1 \prec x_1 \land y_1 \prec x_1$$

Si x est initialement présent une fois dans le tampon, alors l'entrée concrète x_1 correspondante deviendra x_2 après écriture. Après abstraction, elle devient x_ω, qui ne représente alors qu'une variable (il n'y a pas de condensation à proprement parler). Toutes les relations initialement valables sur x_1 restent alors vérifiées en remplaçant x_1 par x_ω.

Ainsi, depuis l'ordre suivant :

$$x_1 \prec y_\omega \prec y_1$$

On obtient le résultat suivant :

$$x_\omega \prec y_\omega \prec y_1 \prec x_1$$

Dans le dernier cas où x est présent plus d'une fois, il n'est pas toujours possible de garder les relations vérifiées par x_ω. Considérons en effet l'ordre de départ suivant :

$$x_\omega \prec y_1 \prec x_1$$

Il décrit par exemple cet ordre concret :

$$x_2 \prec y_1 \prec x_1$$

Après écriture dans x, l'ordre concret obtenu est alors :

$$x_3 \prec y_1 \prec x_2 \prec x_1$$

Il s'abstrait ainsi :

$$y_1 \prec x_1$$

L'entrée de x_ω a disparu : en effet, après une nouvelle écriture de x, l'ancienne entrée x_1 l'a rejointe. y_1 s'est alors retrouvé intercalé entre deux entrées correspondant à x_ω : l'information d'ordre est alors perdue.

De façon générale, il ne faut garder dans le nouvel ordre, pour x_ω, que les relations qui étaient vérifiées à la fois par x_1 et par x_ω dans l'ordre originel. Puisqu'on sait que par ailleurs la relation $x_\omega \prec x_1$ est toujours vérifiée, cela revient à garder toutes les relations de la forme $y_i \prec x_\omega$ et à remplacer x_1 par x_ω dans toutes les relations de la forme $x_1 \prec y_i$.

Transfert.

Notre domaine trouve son intérêt principal dans l'amélioration de la précision de l'opérateur de transfert' : en effet, comme nous l'avons vu sur les premiers exemples, il nous permet de montrer que certains transferts sont impossibles.

Ainsi, le transfert de x depuis une partition $(b^\flat, \prec)$ où $x_{b^\flat(x)}$ n'est pas le premier élément de $\prec$ renverra $\bot_\prec$. C'est par exemple le cas pour l'ordre suivant :

$$y_1 \prec x_\omega \prec x_1$$

Cet ordre garantit qu'il existe au moins, dans les états concrets, une entrée de y antérieure à toutes les entrées de x. Le transfert de x est impossible.

Si en revanche $b^\flat(x) = \tilde{1}$ et x_1 est le plus petit élément de $\prec$, alors le transfert est possible, et l'ordre qui en résulte est simplement $\prec$ auquel on aura retiré x_1. Ainsi, transférer x depuis la partition suivante :

$$x \mapsto \tilde{1}, y \mapsto \tilde{\omega}, z \mapsto \tilde{1} : x_1 \prec y_\omega \prec z_1 \prec y_1$$

Renverra cette nouvelle partition :

$$x \mapsto \tilde{0}, y \mapsto \tilde{\omega}, z \mapsto \tilde{1} : y_\omega \prec z_1 \prec y_1$$

S'il existe déjà une partition ayant cet index, l'ordre obtenu $y_\omega \prec z_1 \prec y_1$ sera joint à l'ordre déjà présent par l'opération $\sqcup^\prec$.

Le cas où $b^\flat(x) = \tilde{\omega}$ présente la même particularité que pour les états numériques : ne sachant pas si les tampons concrets contiennent deux ou plus de deux entrées pour x, il est nécessaire de calculer les deux résultats possibles. Les images obtenues atteindront alors deux partitions différentes. La première partition correspond au cas où x était présent deux fois, il est donc présent une fois dans l'image, et x_ω est retiré de l'ordre initial. La deuxième partition correspond au cas où x était présent strictement plus de deux fois : il est alors toujours présent au moins deux fois dans l'image initiale, et x_ω reste à la même position dans l'ordre des entrées. En effet, si toutes les entrées concrètes $x_2, \dots, x_n$ correspondant à x_ω respectaient un certain ordre vis-à-vis des autres variables, alors *a fortiori* les entrées $x_2, \dots, x_{n-1}$ respectent cet ordre.

Ainsi, depuis la partition suivante :

$$x \mapsto \tilde{\omega}, y \mapsto \tilde{\omega} : x_\omega \prec x_1 \prec y_\omega \prec y_1$$

Les partitions images du transfert de x seront :

$$x \mapsto \tilde{1}, y \mapsto \tilde{\omega} : x_1 \prec y_\omega \prec y_1$$
$$x \mapsto \tilde{\omega}, y \mapsto \tilde{\omega} : x_\omega \prec x_1 \prec y_\omega \prec y_1$$

13.3 Formalisation

13.3.1 Domaine de l'ordre

La figure 13.4 définit le treillis des ordres sur un ensemble de symboles $\mathcal{S}(b^\flat)$ donné. Nous y définissons la relation d'ordre $\sqsubseteq$ sur les ordres, ainsi que les opérations de borne supérieure $\sqcup$ et de borne inférieure $\sqcap$. Ces définitions correspondent à leurs présentations informelles de la section précédente.

La figure 13.5 définit quant à elle le domaine local de l'ordre. Il s'agit comme nous l'avons présenté informellement du domaine des ensembles d'ordres, partitionné par les ensembles de

$$\mathscr{B}^\flat = \mathit{Shared} \rightarrow \left\{\tilde{0}, \tilde{1}, \tilde{\omega}\right\}$$
$$\mathcal{S}(b^\flat : \mathscr{B}^\flat) \triangleq \left\{x_1 \;\middle|\; x \in \mathit{Shared}, b^\flat(x) = \tilde{1} \vee b^\flat(x) = \tilde{\omega}\right\} \cup \left\{x_\omega \;\middle|\; x \in \mathit{Shared}, b^\flat(x) = \tilde{\omega}\right\}$$
$$\mathscr{O}(b^\flat) \triangleq \left\{ \prec \in \mathcal{S}(b^\flat) \times \mathcal{S}(b^\flat) \;\middle|\; \begin{array}{c} \prec \text{ est un ordre partiel strict et :} \\ \forall x \in \mathit{Shared}, b^\flat(x) = \tilde{\omega} \implies x_\omega \prec x_1 \end{array} \right\} \cup \left\{\bot^\prec\right\}$$

$$\forall b^\flat \in \mathscr{B}^\flat, \sqsubseteq : \mathscr{O}(b^\flat) \times \mathscr{O}(b^\flat)$$
$$\prec_1 \sqsubseteq \prec_2 \;\equiv\; \prec_1 = \bot^\prec \vee \prec_2 \subseteq \prec_1$$

$$\forall b \in \mathscr{B}^\flat, \sqcup : \mathscr{O}(b^\flat) \times \mathscr{O}(b^\flat) \rightarrow \mathscr{O}(b^\flat)$$
$$\prec_1 \sqcup \prec 2 \triangleq \begin{cases} \prec_2 & \text{si } \prec_1 = \bot^\prec \\ \prec_1 & \text{si } \prec_2 = \bot^\prec \\ \prec_1 \cap \prec_2 & \text{sinon} \end{cases}$$

$$\forall b \in \mathscr{B}^\flat, \sqcap : \mathscr{O}(b^\flat) \times \mathscr{O}(b^\flat) \rightarrow \mathscr{O}(b^\flat)$$
$$\prec_1 \sqcap \prec_2 \triangleq \begin{cases} \bot^\prec & \text{si } \prec_1 = \bot^\prec \vee \prec_2 = \bot^\prec \\ \bot^\prec & \text{si } \exists x_a, y_b,\; x_a \prec_1 y_b \wedge y_b \prec_2 x_a \\ (\prec_1 \cup \prec_2)^+ & \text{sinon} \end{cases}$$

FIGURE 13.4 – Treillis des ordres.

$$\prec \dashv\Vdash \ll_T \;\equiv\; \mathfrak{D}_\prec \subseteq \left\{\Phi^T_{\mathfrak{sum}}(x_i) \;\middle|\; x_i \in \mathfrak{D}_\ll\right\} \wedge \forall x_i, y_i \in \mathfrak{D}_\ll, \Phi^T_{\mathfrak{sum}}(x_i) \prec \Phi^T_{\mathfrak{sum}}(y_i) \implies x_i \ll_T y_i$$

$$\mathscr{D}^\prec_T \triangleq \prod_{b^\flat : \mathscr{B}^\flat} \mathscr{O}(b^\flat)$$

$$\mathscr{D}^{TSO}_T \xleftarrow[\alpha]{\gamma} \mathscr{D}^\prec_T$$

$$\gamma(X^\prec) \triangleq \left\{(S_T, \ll) \in \mathscr{S}^{TSO}_T \;\middle|\; X^\prec(\delta_T(S_T)) \dashv\Vdash \ll_T\right\}$$

$$\alpha(X^{TSO}_T) = \lambda b^\flat . \boxed{\prec} \left\{\prec \in \mathscr{O}(b^\flat) \;\middle|\; \exists (S, \ll) \in X^{TSO}_T, \delta_T(S_T) = b^\flat \wedge \prec \dashv\Vdash \ll_T\right\}$$

FIGURE 13.5 – Domaine de l'ordre.

variables sur lesquels ils sont définis, où chaque partition a été abstraite par sa borne supérieure. Autrement dit, il s'agit d'une fonction qui à chaque élément de $\mathscr{B}^\flat$, représentant une information de présence abstraite dans les tampons, associe un ordre sur les entrées condensées associées. Comme pour l'abstraction des états locaux de la partie III, l'abstraction locale de l'ordre pour un processus T oublie les ordres des entrées dans les tampons des autres processus.

La correspondance de Galois est définie à l'aide d'une relation $\dashv\Vdash$ qui formalise la représentation d'un ordre total concret par un ordre partiel condensé. Elle suppose l'existence d'un ensemble d'états concrets locaux $\mathscr{S}^{TSO}_T$ et d'un domaine concret local $\mathscr{D}^{TSO}_T \triangleq \wp(\mathscr{S}^{TSO}_T)$. Ce travail sur les ordres étant toujours exploratoire, nous ne donnerons pas ici de formulation mathématique de ces domaines (en particulier, nous ne formaliserons pas les opérateurs concrets). Considérons simplement que les états de TSO peuvent se définir en complétant les états concrets de PSO, pour chaque processus T, par une relation d'ordre strict totale $\ll_T$ sur les variables de tampon définissant leur ordre chronologique dans l'unique tampon du processus T. Pour la complétude, les états concrets locaux de TSO gardent la relation d'ordre de tous les processus.

La formalisation de ce domaine réutilise également les fonctions δ_T et $\Phi^T_{\mathfrak{sum}}$. Rappelons que la première est la fonction-critère de partitionnement des états locaux et que la seconde associe une variable condensée à une variable concrète. Nous référons le lecteur aux figures 11.2a (page 106) et 11.3b (page 107) de la partie III pour leur formalisation dans PSO, leur définition dans TSO étant semblable.

13.3.2 Sémantique de l'ordre

La figure 13.6 présente les opérateurs locaux sur le domaine de l'ordre. Ils formalisent leur présentation en section 13.2.4 et sont d'abord définis partition par partition.

La figure 13.7 définit quant à elle les opérateurs pour la génération des interférences, depuis chaque partition. Rappelons que les abstractions modulaires de la partie III oublient le contenu des tampons dans les interférences. Nous avons fait le même choix pour les interférences des ordres. Or les ordres ne concernent que les tampons : il ne reste donc que deux types d'interférences abstraites possibles, $\top$ et $\bot$ (selon que des interférences peuvent être générées depuis un état donné ou pas). Nous oublierons également les informations de contrôle, parce qu'elles sont déjà présentes dans les abstractions de la partie III et les ordres n'apportent pas d'information supplémentaire dessus. Pour simplifier la formulation de l'opérateur final du domaine produit, nous considérerons que les interférences générées depuis une partition sont dans le sous-ensemble $\{\top^{\mathtt{num}}, \bot^{\mathtt{num}}\}$ de $\mathscr{D}^{\mathtt{num}}$.

$$\llparenthesis . \rrparenthesis : \sum_{b^\flat : \mathscr{B}^\flat} \mathscr{O}(b^\flat) \to \wp\left(\sum_{b^\flat : \mathscr{B}^\flat} \mathscr{O}(b^\flat)\right)$$

$$\llparenthesis r \leftarrow e \rrparenthesis^{\prec}(b^\flat, \prec) = \left\{b^\flat, \prec\right\}$$

$$\llparenthesis x \leftarrow e \rrparenthesis^{\prec}(b^\flat, \prec) = \begin{cases} \left\{\begin{array}{l} b^\flat[x \mapsto \tilde{1}], \\ \qquad \prec \sqcap \left\{y_{b^\flat(y)}, x_1 \;\middle|\; y \in \mathit{Shared}, b^\flat(y) \in \left\{\tilde{1}, \tilde{\omega}\right\}\right\} \end{array}\right\} & \text{si } b^\flat(x) = \tilde{0} \\ \left\{\begin{array}{l} b^\flat[x \mapsto \tilde{\omega}], \\ \qquad \left\{y_{b^\flat(y)}, x_1 \;\middle|\; y \in \mathit{Shared}, b^\flat(y) \in \left\{\tilde{1}, \tilde{\omega}\right\}\right\} \\ \qquad \sqcap \left\{y_i, z_j \;\middle|\; y \neq x \wedge y_i \prec z_j\right\} \\ \qquad \sqcap \left\{x_\omega, y_i \;\middle|\; x_1 \prec y_i\right\} \\ \qquad \sqcap \left\{y_i, x_\omega \;\middle|\; y_i \prec x_1\right\} \end{array}\right\} & \text{si } b^\flat(x) = \tilde{1} \\ \left\{\begin{array}{l} b^\flat, \\ \qquad \left\{y_{b^\flat(y)}, x_1 \;\middle|\; y \in \mathit{Shared}, b^\flat(y) \in \left\{\tilde{1}, \tilde{\omega}\right\}\right\} \\ \qquad \sqcap \left\{y_i, z_j \;\middle|\; y \neq x \wedge y_i \prec z_j\right\} \\ \qquad \sqcap \left\{x_\omega, y_i \;\middle|\; x_1 \prec y_i\right\} \end{array}\right\} & \text{si } b^\flat(x) = \tilde{\omega} \end{cases}$$

$$\llparenthesis \mathtt{mfence} \rrparenthesis^{\prec}(b^\flat, \prec) = \begin{cases} \left\{b^\flat, \varnothing\right\} & \text{si } \forall x \in \mathit{Shared}, b^\flat(x) = \tilde{0} \\ \varnothing & \text{sinon} \end{cases}$$

$$\llparenthesis \text{flush } x \rrparenthesis^{\prec}(b^\flat, \prec) = \begin{cases} \varnothing & \text{si } b^\flat(x) = \tilde{0} \vee \exists y_i, y_i \prec x_{b^\flat(x)} \\ \left\{b^\flat[x \mapsto \tilde{0}], \left\{y_i, z_j \;\middle|\; y \neq x \wedge z \neq x \wedge y_i \prec z_j\right\}\right\} & \text{sinon si } b^\flat(x) = \tilde{1} \\ \left\{\begin{array}{l} b^\flat[x \mapsto \tilde{1}], \left\{y_i, z_j \;\middle|\; y_i \neq x_\omega \wedge z_j \neq x_\omega \wedge y_i \prec z_j\right\} \\ \qquad b^\flat, \prec \end{array}\right\} & \text{sinon si } b^\flat(x) = \tilde{\omega} \end{cases}$$

$$[\![.]\!]^{\prec} : \mathscr{D}^{\prec} \to \mathscr{D}^{\prec}$$

$$[\![\mathit{ins}]\!]^{\prec} X^{\prec} = \lambda b^\flat . \lfloor\prec\rfloor \left\{\prec \;\middle|\; \exists b'^\flat \in \mathscr{B}^\flat, (b^\flat, \prec) \in \llparenthesis \mathit{ins} \rrparenthesis^{\prec}(b'^\flat, X^{\prec}(b'^\flat))\right\}$$

FIGURE 13.6 – Sémantique locale de l'ordre inter-variables.

$$\llparenthesis . \rrparenthesis^{\prec}_{\mathscr{I}} : \sum_{b^\flat : \mathscr{B}^\flat} \mathscr{O}(b^\flat) \to \{\bot^{\mathtt{num}}, \top^{\mathtt{num}}\}$$

$$\llparenthesis r \leftarrow e \rrparenthesis^{\prec}_{\mathscr{I}}(b^\flat, \prec) = \top^{\mathtt{num}}$$

$$\llparenthesis \mathtt{mfence} \rrparenthesis^{\prec}_{\mathscr{I}}(b^\flat, \prec) = \top^{\mathtt{num}}$$

$$\llparenthesis x \leftarrow e \rrparenthesis^{\prec}_{\mathscr{I}}(b^\flat, \prec) = \top^{\mathtt{num}}$$

$$\llparenthesis \text{flush } x \rrparenthesis^{\prec}_{\mathscr{I}}(b^\flat, \prec) = \begin{cases} \bot^{\mathtt{num}} & \text{si } b^\flat(x) = \tilde{0} \vee \exists y_i, y_i \prec x_{b^\flat(x)} \\ \top^{\mathtt{num}} & \text{sinon} \end{cases}$$

FIGURE 13.7 – Génération des interférences depuis une partition du domaine de l'ordre.

$$\mathscr{D}_T^\times \triangleq \mathscr{D}^\prec \times \mathscr{D}_T^\sharp$$
$$\rho : \mathscr{D}_T^\times \to \mathscr{D}_T^\times$$
$$\rho(X^\prec, X_T^\sharp) \triangleq \lambda b^\flat. \begin{cases} \bot^\prec & \text{si } X_T^\sharp(b^\flat) = \bot^{\mathtt{num}} \\ X^\prec(b^\flat) & \text{sinon} \end{cases}, \ \lambda b^\flat. \begin{cases} \bot^{\mathtt{num}} & \text{si } X^\prec(b^\flat) = \bot^\prec \\ X_T^\sharp(b^\flat) & \text{sinon} \end{cases}$$

(a) Domaine produit et fonction de réduction.

$$[\![.]\!]_{\mathscr{I}}^\times : \mathscr{D}_T^\times \to \mathscr{I}^\sharp$$
$$^{(\ell_1)}[\![op]\!]_{\mathscr{I}}^{\times\,(\ell_2)}(X^\prec, X_T^\sharp) = \lambda T'.\lambda x^?. \begin{cases} \bigsqcup^{\mathtt{sum}}_{b^\flat \in \mathscr{B}^\flat} \begin{pmatrix} ^{(\ell_1)}(\!|ins|\!)_T^\sharp(b'^\flat, X_T^\sharp(b'^\flat))^{(\ell_2)} \\ \sqcap^{\mathtt{num}} \\ (\!|ins|\!)_{\mathscr{I}}^\prec(b^\flat, X^\prec(b^\flat)) \end{pmatrix} & \text{si } \begin{cases} x^? = (\!|op|\!) \\ T' = T \end{cases} \\ \bot^{\mathtt{num}} & \text{sinon} \end{cases}$$

(b) Génération des interférences.

FIGURE 13.8 – Produit réduit des domaines des tampons condensés et de l'ordre.

13.3.3 Domaine final

Pour obtenir le domaine final d'analyse des programmes dans TSO, nous effectuerons le produit réduit du domaine des états locaux de la partie III et du domaine de l'ordre. La fonction de réduction est définie en figure 13.8a. Elle correspond à l'application de la réduction simple de l'exemple 13.2 partition par partition. On montre ainsi facilement que cette réduction est sûre. De plus, sans l'avoir prouvé, nous pensons qu'il s'agit de la réduction optimale. L'argument est que les deux domaines portent sur des composants trop différents de l'état concret pour pouvoir échanger plus d'information : un invariant sur l'ordre des variables ne permet pas de préciser leur contenu, et réciproquement (sauf bien sûr si l'un de ces invariants est $\bot$).

Les opérateurs abstraits se calculent de la façon habituelle, définie en figure 13.3 (page 129) : on réduit simplement le résultat après leur application aux composants de l'ordre et du contenu.

La figure 13.8b définit également le calcul des interférences depuis le domaine produit. Les interférences générées par le domaine de l'ordre ayant par commodité été définies dans $\{\bot^{\mathtt{num}}, \top^{\mathtt{num}}\}$, il suffit de calculer leur intersection avec celle de l'interférence générée par le contenu. Comme pour le calcul des états locaux par réduction, l'effet du domaine de l'ordre est alors simplement d'inférer que certaines interférences, générées par les transferts, sont impossibles, en raison de l'existence d'une variable plus ancienne dans les tampons.

Une fois ces opérations définies, le domaine produit de l'ordre et du contenu des tampons peut se brancher à l'analyse modulaire de la partie III. Il induira une analyse plus précise et sûre dans TSO, permettant par exemple de vérifier sans barrière superflue, comme nous l'avons montré en introduction de ce chapitre, l'algorithme de Peterson présenté en annexe page 168 et dont le programme 13.2 est un extrait.

Remarque 13.1.
Un ensemble de contraintes de la forme $x_i \prec y_j$ peut se représenter comme un graphe ordonné, où les sommets sont les $x_i, y_j, \ldots$ et les arêtes relient deux variables ordonnées entre elles.

Cette représentation est semblable à celle du domaine relationnel des zones [Min01], qui

encode des invariants de la forme $x - y \leq \alpha$ (les octogones en forment donc un raffinement). En restreignant α à $\{0, +\infty\}$, on retrouve en effet exactement les ordres inter-variables.

Les opérations comme l'union et l'intersection sont alors similaires, et la clôture transitive correspond à la mise sous une forme canonique des zones, qu'on applique pour garantir la précision de certaines opérations (comme l'union ou l'oubli d'une dimension). $\triangleleft$

13.4 Conclusion

L'analyse modulaire telle qu'elle a été définie en partie III est sûre dans PSO. Elle induit donc, lorsqu'on considère TSO comme modèle concret cible, des imprécisions sur les programmes à plusieurs variables qui peuvent demander de rajouter des barrières supplémentaires pour vérifier les programmes, ralentissant leur exécution.

En définissant un domaine qui abstrait les relations d'ordre chronologique d'entrée dans le tampon des différentes variables correspondantes, nous avons réduit cette imprécision. Ce domaine s'utilise en conjonction avec le domaine de la partie III à l'aide d'un produit réduit, ce qui permet de se concentrer sur sa définition en déléguant à cette construction le soin de propager les informations d'accessibilité des transferts.

Pour des travaux futurs, il reste à implémenter ce domaine dans un analyseur pour évaluer ses performances et sa précision dans le cas général. Cette validation expérimentale doit aussi confirmer (ou non) le choix de ce domaine comme un bon compromis général entre performances et précision, notamment en comparaison avec l'abstraction qui consiste à garder tous les ordres possibles (voire à partitionner les contenus selon l'ordre correspondant).

Chapitre 14

Abstractions non uniformes

Les domaines développés en parties II et III, ainsi qu'au chapitre 13, ont ceci en commun qu'ils représentent toutes les entrées d'un tampon (excepté la plus récente) de façon *uniforme* : après abstraction, il n'existe aucun moyen de distinguer deux entrées différentes condensées dans une variable x^T_ω.

Dans ce chapitre, nous explorerons quelques abstractions non uniformes : nous montrerons comment elles bénéficient d'informations leur permettant de distinguer des entrées condensées pour vérifier avec précision de nouvelles classes de programmes.

14.1 Tampons triés

Le programme 14.1 implémente un mécanisme d'horloge à deux processus. Le processus horloge incrémente en boucle un compteur `t`, et le processus client lit en boucle l'heure courante. Pour les besoins de la démonstration, le client effectue deux lectures par tour de boucle. La propriété de correction spécifie que la deuxième lecture engendre une valeur supérieure ou égale à la première : le temps est croissant. Notons que le client ne lira pas forcément tous les instants : il peut en sauter certains, de manière non déterministe. De même, il peut éventuellement lire deux ou plusieurs fois de suite la même heure courante. Notons également que ce programme ne comporte qu'une seule variable partagée : il a donc exactement la même sémantique dans TSO et PSO.

Dans le concret, la propriété est vérifiée parce que les écritures en mémoire par l'horloge doivent être lues dans le même ordre par le client. Nous allons montrer que les abstractions des parties précédentes ne permettent cependant pas de le prouver, et proposer une abstraction qui maintient le caractère trié des tampons afin de retrouver la précision nécessaire. Nous formaliserons le domaine ainsi obtenu et décrirons informellement la construction des opérateurs abstraits.

14.1.1 Limitation de l'abstraction uniforme

Considérons une analyse modulaire du programme 14.1 avec les méthodes et domaines développés en partie III.

Après un tour de boucle, l'analyse de l'horloge aura généré une écriture dans t_1, qui elle-même aura produit par transfert une interférence qui peut stocker 1 dans t^{mem} si sa valeur est initialement 0. Remarquons que l'application de cette interférence par le client ne compromet

```
/* Shared : int t */

thread /* Horloge */ {
    r = 0;
    while (true) {
        r = r + 1;
        t = r;
    }
}

thread /* Client */ {
    while (true) {
        r1 = t;
        r2 = t;
        /* assert r2 >= r1 */
    }
}
```

Programme 14.1 – Horloge et client.

pas encore la propriété recherchée : en effet, s'il l'applique pour lire 1 dans r_1, il ne peut plus remettre la valeur de t^{mem} à 0 pour la lecture de r_2.

Après deux tours, les deux écritures du processus horloge auront atteint t_ω. Leurs transferts auront généré deux interférences, qui écrivent respectivement 2 et 1 dans t^{mem} si sa valeur est respectivement 1 ou 0. Ils en auront également généré une troisième par clôture par transfert depuis l'état suivant :

$$t^{mem} \mapsto 0, t_1 \mapsto 2, t_\omega \mapsto 1$$

À partir de cet état, le transfert de t génère en particulier l'état suivant :

$$t^{mem} \mapsto 1, t_1 \mapsto 2, t_\omega \mapsto 1$$

On peut alors réitérer le transfert de t. L'état produit sera le même, mais on générera au passage une interférence qui écrit 1 dans t^{mem} si sa valeur est égale à 1.

Cette interférence correspond donc au chemin d'exécution abstraite où l'horloge place ses deux écritures dans son tampon, transfère la première en supposant qu'il existe encore des entrées (qui sont toujours égales à 1 qui est alors la seule valeur possible de t_ω), puis procède à nouveau au même transfert. Il n'existe en revanche pas d'interférence permettant de transférer 1 si t^{mem} vaut 2 : la propriété n'est toujours pas compromise.

En revanche, après trois tours, la condensation dans t_ω devient effective et cette variable a deux valeurs possibles, 1 et 2, correspondant à deux entrées successives dans le tampon qu'il est impossible de distinguer. Il existe donc une exécution abstraite qui transfère 2, considère qu'il existe encore au moins une entrée dans t_ω qui n'est donc pas supprimée de l'environnement, puis transfère à nouveau 1 qui est une valeur possible de cette variable. Le client peut alors appliquer la première interférence avant de lire dans r_1 puis la deuxième avant de lire dans r_2, ce qui invalide la propriété recherchée.

Ce problème se présenterait aussi avec l'analyse monolithique : il provient du fait que la correction de la propriété se fonde sur une relation entre deux entrées successives dans le tampon du processus horloge, alors que la condensation efface toute distinction entre lesdites entrées. Il est donc nécessaire de recourir à une abstraction non uniforme pour retrouver la précision nécessaire.

14.1.2 Domaine des tampons triés

Complétons donc les tampons des états locaux par une annotation choisie parmi deux niveaux d'information possibles : soit le tampon est trié à coup sûr (en ordre croissant de l'entrée la plus ancienne à la plus récente), ce que nous noterons $\nearrow$, soit nous ne disposons d'aucune information de cette sorte, ce que nous noterons $\top$. Remarquons qu'il est suffisant de faire porter cette information sur les variables condensées dans t_ω (à l'exclusion donc de t_1) : en effet, la propriété de tri du tampon complet peut alors s'écrire comme « les entrées condensées sont triées et $t_1 \geq t_\omega$ ».

Il existe une annotation de cette sorte par tampon condensé, donc par variable partagée. Considérons par exemple l'état suivant :

$$x_1 \mapsto 1, x_2 \mapsto 20, x_3 \mapsto 10, y_1 \mapsto 0, y_2 \mapsto 1, y_3 \mapsto 2, z_1 \mapsto 1, z_2 \mapsto 3$$

Le tampon de x est trié (x_1 exclu), celui de y ne l'est pas. Celui de z n'a qu'une entrée z_2 : il est trivialement trié (et dans l'abstrait, cette information de taille sera perdue). Il correspond donc à l'état abstrait suivant :

$$x : \nearrow, x_1 \mapsto 1, x_\omega \mapsto \{10, 20\}, y : \top, y_1 \mapsto 0, y_\omega \mapsto \{1, 2\}, z : \nearrow, z_1 \mapsto 1, z_\omega \mapsto \{3\}$$

Remarque 14.1.
Il est possible de considérer qu'un tampon condensé vide (donc correspondant à un tampon concret d'exactement une entrée) est trié. On peut aussi choisir de ne pas lui attacher d'information. Considérant qu'il s'agit essentiellement d'un détail de la formalisation et que cette présentation est principalement exploratoire, nous utiliserons selon la situation la représentation la plus pratique sans chercher à la fixer formellement. $\triangleleft$

Nous pouvons ensuite abstraire les états locaux comme nous l'avons fait en partie III. L'information de tri des tampons de chaque variable est alors regroupée au niveau des partitions, en considérant la borne supérieure des informations individuelles de chaque état, comme nous l'avons fait pour l'ordre des variables au chapitre 13. Le domaine résultant est défini en figure 14.1. Nous suivons le schéma de l'abstraction du chapitre 13 en définissant d'abord les informations de tri, puis une relation caractérisant les tampons triés. Ces informations sont ensuite partitionnées selon le critère habituel, et le domaine final est défini comme un produit entre les informations de tri et les abstractions du contenu (cf. figure 13.2, chapitre 13, page 127).

Comme pour l'ordre inter-variables, l'information de tri sera absente des interférences : elle porte encore une fois sur des tampons que nous avons choisi d'oublier. Leur définition n'est donc pas modifiée, mais nous montrerons encore une fois que ce domaine permet de les générer avec plus de précision.

14.1.3 Construction des opérateurs abstraits

Notre domaine ainsi défini, nous reprendrons maintenant l'exemple de l'analyse du programme 14.1 pour montrer comment s'obtiennent les opérateurs abstraits. En observant plusieurs étapes de cette analyse, nous détaillerons informellement leur construction qui doit être suffisamment précise pour vérifier la propriété recherchée.

Écriture.

Après deux tours, et sans transfert, on atteint l'état suivant en fin du corps de la boucle :

$$r \mapsto 1, t_1 \mapsto 1, t^{mem} \mapsto 0$$

$$\mathscr{T} \triangleq \{\nearrow, \top\}$$

(a) L'information de tri : croissant ou inconnu.

$$\dashv\Vdash : (\mathit{Shared} \to \mathscr{T}) \times S_T$$
$$\theta \dashv\Vdash S_T \stackrel{\triangle}{\equiv} \forall x \in \mathit{Shared}, \theta(x) = \nearrow \implies \forall i > 1, x_{i+1} \in \mathfrak{D}_{S_T} \implies S_T(x_{i+1}) \leq S_T(x_i)$$

(b) Caractérisation des états avec des tampons triés.

$$\mathscr{D}^{\nearrow} \triangleq \mathscr{B}^{\flat} \to (\mathit{Shared} \to \mathscr{T})$$
$$\gamma_{\nearrow} : \mathscr{D}^{\nearrow} \to \mathscr{D}_T$$
$$\gamma_{\nearrow}(\Theta) \triangleq \{S_T \in \mathscr{S}_T \mid \Theta(\delta_T(S_T)) \dashv\Vdash S_T\}$$

(c) Domaine du tri partitionné.

$$\mathscr{D}_T^{\times} \triangleq \mathscr{D}^{\nearrow} \times \mathscr{D}_T^{\sharp}$$

(d) Domaine final construit par produit.

Figure 14.1 – Domaine des états locaux avec tri des tampons.

Le tour de boucle suivant génère un état où t_ω apparaît. Lors du calcul de cet état, on sait que cette variable ne représente qu'une entrée : elle est alors triée.

$$t : \nearrow, r \mapsto 2, t_1 \mapsto 2, t_\omega \mapsto 1, t^{mem} \mapsto 0$$

Lors du troisième tour, qui procède à une nouvelle écriture dans t, la valeur de t_1 sera condensée avec t_ω. L'opérateur d'écriture sur le domaine du contenu conduira alors t_ω à prendre ses valeurs dans l'ensemble $\{1, 2\}$. Remarquons que les opérateurs obtenus par produit réduit du tri et du contenu, s'ils restent sûrs par construction, ne sont pas suffisants pour propager l'information qui nous intéresse ici : en effet, la réduction depuis $t_\omega \in \{1, 2\}$ ne nous donne aucune information sur la croissance ou non des entrées correspondantes. Un opérateur abstrait qui génère avec précision chaque composant du produit aura donc besoin d'émettre des requêtes à l'autre composant pour calculer son résultat [CCF+06].

En l'occurrence, l'opérateur calculant le contenu du tampon après écriture ne profite pas de l'information de tri : il condense simplement t_1 dans t_ω. En revanche, la propriété de tri du tampon après écriture est équivalente à une décomposition en deux propositions :

1. Le tampon était trié avant l'écriture ;
2. La nouvelle écriture préserve le tri.

La première sous-propriété s'obtient par l'information initiale de tri du tampon, qui en l'occurrence est trié. La deuxième peut quant à elle se réécrire en « la nouvelle entrée du tampon condensé est supérieure à toutes les précédentes ». L'opérateur calculant l'information de tri émet

alors une requête au domaine du contenu : il s'assure que la propriété $t_1 \geq t_\omega$ est bien valide. Dans notre exemple, elle l'est : le tampon condensé reste alors trié.

$$t : \nearrow, r \mapsto 3, t_1 \mapsto 3, t_\omega \mapsto \{1, 2\}, t^{mem} \mapsto 0$$

Les écritures suivantes seront semblables, la propriété $t_1 \geq t_\omega$ étant valide dans tous les états générés. Remarquons simplement que le domaine numérique choisi devra permettre de prouver cette propriété afin de pouvoir maintenir l'information de tri. En particulier, une abstraction non relationnelle comme les intervalles sera insuffisante : après N tours, l'invariant obtenu en fin de boucle sera de la forme $t_1 \in [1, N] \wedge t_\omega \in [1, N-1]$, ce qui ne permet pas de prouver $t_1 \geq t_\omega$.

Transfert.

Lorsque le transfert se fait depuis t_1, l'éventuelle annotation de tri n'apporte aucune information pertinente. État local et interférence sont alors générés comme si elle n'existait pas.

Nous nous intéresserons donc aux transferts depuis t_ω. Considérons par exemple l'invariant condensé correspondant à exactement trois tours de boucle du programme 14.1.

$$t : \nearrow, r = 3, t_1 = 3, t_\omega \in \{1, 2\}, t^{mem} = 0$$

Nous avons vu que le transfert depuis t_ω générait des états locaux dans deux partitions : la première suppose que t_ω ne contenait qu'une entrée. Cette variable est alors supprimée de l'environnement, et l'information de tri n'a alors plus lieu d'être. Les états ainsi produits respectent l'invariant suivant :

$$r = 3, t_1 = 3, t^{mem} \in \{1, 2\}$$

Et les interférences générées ont la forme suivante :

$$r = r' = 3, t^{mem} = 0, t^{mem\prime} \in \{1, 2\}$$

Le tri n'apporte ici aucune précision supplémentaire par rapport à l'analyse de la partie III : en effet, si on considère que t_ω ne correspond qu'à une entrée, elle est alors trivialement triée et cette information était en quelque sorte déjà disponible.

Le deuxième résultat du transfert correspond au cas où t_ω condensait plus d'une entrée. En oubliant un instant le tri, rappelons que l'invariant mémoire obtenu est alors le suivant :

$$r = 3, t_1 = 3, t_\omega \in \{1, 2\}, t^{mem} \in \{1, 2\}$$

Rappelons également qu'en raison de la condensation, il n'existe aucune autre relation liant t_ω à t^{mem} après transfert. En particulier, $t_\omega = 1 \wedge t^{mem} = 2$ fait partie des états concrets décrits par cet invariant, même s'il n'est en réalité pas accessible comme nous l'avons vu au début de ce chapitre.

Considérons maintenant l'information de tri de t_ω. Remarquons tout d'abord que, si elle est vérifiée dans l'état initial, elle l'est *a fortiori* après un transfert : une sous-séquence d'une séquence triée est nécessairement triée. Plus important encore, elle nous permet de contraindre les valeurs transférées : puisque t_ω est trié dans l'état initial, alors l'entrée la plus ancienne est par définition la plus petite. Après transfert, c'est cette entrée qui atteindra t^{mem} : la propriété $t^{mem} \leq t_\omega$ est donc vérifiée. L'invariant local obtenu est alors le suivant :

$$t : \nearrow, r = 3, t_1 = 3, t_\omega \in \{1, 2\}, t^{mem} \in \{1, 2\}, t^{mem} \leq t_\omega$$

Remarquons que cet invariant permet en particulier de montrer que les états vérifiant $t_\omega = 1 \wedge t^{mem} = 2$ sont inaccessibles.

Remarque 14.2.
L'opérateur local de transfert a un comportement dual de celui de l'écriture dans un tampon : alors que ce dernier faisait appel au domaine du contenu pour générer l'information de tri, celui-là requiert l'information de tri pour préciser l'invariant sur le contenu. ◁

Les interférences générées par ce transfert ont pour l'instant toujours la même forme :

$$r = r' = 3, t^{mem} = 0, t^{mem'} \in \{1, 2\}$$

Remarquons que les interférences de cette forme, lors de l'analyse du client, ne permettront effectivement pas de lire successivement deux valeurs décroissantes : t^{mem} ne peut que croître lors de leur application. Toutefois, l'information de tri n'a pour l'instant rien apporté de nouveau pour la vérification de cette propriété.

Rappelons cependant que, pour obtenir un ensemble d'états locaux clos, il est nécessaire de répéter l'opération de transfert jusqu'à atteindre un point fixe, chacune de ces étapes générant ses propres interférences. Considérons l'invariant que nous venons d'obtenir :

$$t : \nearrow, r = 3, t_1 = 3, t_\omega \in \{1, 2\}, t^{mem} \in \{1, 2\}, t^{mem} \leq t_\omega$$

Un nouveau transfert laissera cet invariant inchangé : le point fixe sera alors atteint. En revanche, une nouvelle interférence sera générée. L'information de tri ne nous est toujours pas directement utile pour la produire, mais sa présence à l'étape précédente nous avait permis de renforcer l'invariant local avec la propriété $t^{mem} \leq t_\omega$. Le transfert se faisant depuis t_ω vers t^{mem}, cette propriété permet d'assurer qu'il ne peut que faire croître la valeur de la variable de destination. Dans l'interférence, ce résultat s'obtient grâce à l'opération $t^{mem'} \leftarrow t_\omega$ par laquelle elle est générée (cf. figure 11.6, chapitre 11, page 113). Cette opération transfère alors sur $t^{mem'}$ les contraintes initiales sur t_ω, générant l'interférence suivante :

$$r = r' = 3, t^{mem} \in \{1, 2\}, t^{mem'} \in \{1, 2\}, t^{mem} \leq t^{mem'}$$

Cette interférence ne permet toujours pas de réduire la valeur de t^{mem} : son application maintient la correction de la propriété recherchée. Il en sera de même pour toutes les interférences générées dans le cadre de l'analyse du programme 14.1, qui pourra ainsi être vérifié grâce à cette nouvelle information.

Les autres opérations ne présentent pas de difficulté particulière : comme pour l'ordre inter-variables, elles n'interagissent pas avec la partie condensée des tampons, et ne modifient donc ni n'utilisent leur caractère trié.

14.1.4 Conclusion sur le tri

Dans cette section, nous proposons d'ajouter au domaine des états locaux, pour tous les tampons condensés, une annotation permettant d'indiquer que les entrées du tampon, dans l'ordre chronologique, sont croissantes.

Si le domaine induit s'écrit comme un produit d'abstractions, il est cette fois insuffisant de se contenter de la technique du produit réduit pour obtenir des opérateurs abstraits précis : il est nécessaire de faire communiquer les composants du domaine produit au sein même de ces opérateurs. Ainsi, l'écriture dans le tampon utilise les informations de valeur pour déterminer l'état trié ou non du tampon, alors que l'opérateur local de transfert utilise l'information de tri

```
1  /* Shared int x */
2
3  thread /* Compteur */ {
4      /* Local int n */
5      n = 1;
6      while (true) {
7          n = 3 * n + 2;
8          x = n;
9      }
10 }
11
12 thread /* Client */ {
13     /* Local int r1, r2 */
14     r1 = x;
15     r2 = x;
16     /* Assert r2 = r1 || r2 >= 3 * r1 + 2 */
17 }
```

Programme 14.2 – Relation complexe entre deux entrées successives.

pour préciser les invariants des états de destination. Grâce à ces mêmes invariants, la génération des interférences de transfert peut par la suite préciser la croissance de la variable de destination, permettant ainsi lors de l'application de garantir que les valeurs lues successivement par un autre processus ne peuvent qu'augmenter.

Nous sommes donc en mesure de vérifier avec précision des programmes auparavant inaccessibles. Ce domaine peut en outre se combiner par exemple avec celui du chapitre 13 pour une analyse plus précise en ciblant TSO. L'annotation de tri induite par le produit de ces deux abstractions portera alors sur chaque variable : en conjonction, avec une information sur l'ordre inter-variables, elle permettra d'obtenir des informations de tri globales sur le tampon d'un processus.

Enfin, nous n'avons considéré que les tampons triés en ordre croissant, mais ce domaine s'adapte trivialement aux tampons décroissants. Il est également possible de représenter les deux informations simultanément, soit par un produit réduit, soit en adaptant le domaine avec un treillis de tri à quatre éléments $\{=, \nearrow, \searrow, \top\}$. La section 14.2 proposera une généralisation de ces abstractions.

14.2 Variables auxiliaires supplémentaires

Il existe encore de nombreuses façons d'abstraire une collection de variables de façon non uniforme pour pouvoir exprimer des propriétés plus précises sur leurs valeurs respectives [Cou03, HP08, CCL11, MG16]. Parmi elles, les domaines de segmentation paramétriques [CCL11] sont des domaines permettant entre autres de généraliser le domaine de tri de la section précédente, ainsi que d'exprimer des propriétés précises sur le contenu de tableaux.

Sans développer ces domaines dans leur généralité, nous présenterons dans cette section une instance de ces abstractions qui généralise entre autres le domaine de tri de la section précédente ; montrant en quoi cette technique nous semble profitable pour la représentation des tampons.

14.2.1 Variables auxiliaires supplémentaires pour le tampon

Considérons le programme 14.2. Il s'agit d'une légère variation du programme 14.1, où le compteur n'incrémente plus de 1 la valeur écrite à chaque tour de boucle, mais la multiplie par 3 puis l'incrémente de 2. La propriété recherchée sur deux lectures successives par le client est alors vérifiée par croissance de la fonction $n \mapsto 3n + 2$ sur $[1, +\infty[$.

Pour prouver cette propriété, nous pourrions recourir au même type d'abstraction qu'à la section précédente, à savoir une information de la forme « deux entrées successives sont liées par l'équation $x_i = 3 \times x_{i+1} + 2$ ». Les domaines de ce type, composés d'une information représentable par un booléen muni de fonctions de transfert dédiées, font partie des *abstractions par prédicats paramétriques* [Cou03].

Munies de fonctions de transfert dédiées, ces abstractions par prédicats permettraient donc de construire un domaine *ad hoc* pour la vérification du programme 14.2. Toutefois, ce domaine s'adapterait assez mal à de légères variations du programme, et demande donc une aide de l'utilisateur (ou une heuristique syntaxique, souvent peu robuste [CCL11]) pour formuler l'invariant à maintenir.

Pour représenter de façon plus générale des relations entre deux entrées successives dans le tampon, il est alors possible d'utiliser une représentation plus explicite en rajoutant deux variables auxiliaires pour la représentation du tampon [CCL11].

Notons $x_{\omega+1}$ et x_∞ ces nouvelles variables. Dans chaque état condensé, si x_ω représente une valeur possible d'une certaine variable concrète x_i, $x_{\omega+1}$ représente la valeur de x_{i+1} dans le même état concret et x_∞ représente l'entrée la plus ancienne du tampon (il ne s'agit donc pas d'une variable condensée). Considérons par exemple le tampon concret suivant, atteint après quelques itérations du programme 14.2 :

$$x_2 \mapsto 53, x_3 \mapsto 17, x_4 \mapsto 5, x_5 \mapsto 1$$

Les états condensés qui le décrivent sont exactement les états suivants :

$$\begin{array}{lll} x_\omega \mapsto 53, & x_{\omega+1} \mapsto 17, & x_\infty \mapsto 1 \\ x_\omega \mapsto 17, & x_{\omega+1} \mapsto 5, & x_\infty \mapsto 1 \\ x_\omega \mapsto 5, & x_{\omega+1} \mapsto 1, & x_\infty \mapsto 1 \\ x_\omega \mapsto 1, & & x_\infty \mapsto 1 \end{array}$$

Si $x_{\omega+1}$ n'est pas défini dans le dernier état, parce qu'alors x_ω représente l'entrée la plus ancienne du tampon concret, il reste sûr de supposer qu'il peut exister d'autres entrées et de généraliser leur propriété en abstrayant l'état concret initial par l'invariant suivant :

$$x_\omega \in [1, 53] \ \wedge \ x_\omega = 3 \times x_{\omega+1} + 2 \ \wedge \ x_\infty = 1$$

Cet invariant (dont on remarque qu'il peut s'exprimer naturellement dans les domaines numériques relationnels comme les octogones ou les polyèdres) spécifie que les valeurs du tampon vivent dans l'intervalle $[1, 53]$, que deux valeurs successives sont reliées par l'expression $3 \times . + 2$, et que la dernière entrée vaut 1.

Remarquons que cette abstraction est strictement plus précise que celles que nous avons vu jusque-là en parties II et III et en section 14.1 :

- Les abstractions uniformes des parties précédentes ne permettent d'exprimer que $x_\omega \in [1, 53]$.
- Le domaine des tableaux triés peut exprimer que deux entrées consécutives sont ordonnées (en ordre croissant ou décroissant), mais pas préciser leur rapport.

Ce domaine permet également de représenter précisément la dernière entrée du tampon. Sans la relation entre deux éléments successifs, cette information a peu de valeur : en effet, dans la représentation uniforme, après un transfert, x_∞ peut prendre n'importe quelle valeur de x_ω : la distinction entre ces deux variables n'apporte plus aucune information. Toutefois, la propriété supplémentaire représentée par la relation entre x_ω et $x_{\omega+1}$ restera vraie entre x_∞ et x^{mem} après un transfert : il est donc possible de garantir une précision supplémentaire lorsqu'on considère plusieurs transferts successifs, comme la section qui suit va le montrer.

14.2.2 Application

Appliquons maintenant pleinement cette abstraction au programme 14.2. Elle permettrait d'exprimer la propriété suivante en fin d'un tour de boucle du compteur, avant tout transfert :

$$x_1 \geq 3 \times x_\omega + 2 \wedge x_\omega \in [5, +\infty[\wedge x_\omega \geq 3 \times x_{\omega+1} + 2 \wedge x_\infty = 5$$

Montrons que cette propriété est bien un invariant. Lors du tour de boucle suivant, le compteur incrémentera x_1 (*via* n) : $x'_1 = 3 \times x_1 + 2$. L'ancienne valeur de x_1 rejoindra la variable condensée x_ω pour former la nouvelle valeur x'_ω. La relation entre x'_1 et x'_ω s'obtient alors par conjonction entre les relations $\langle x'_1, x_1\rangle$ et $\langle x_1, x_\omega\rangle$:

$$x'_1 = 3 \times x_1 + 2 \wedge x_1 \geq 3 \times x_\omega + 2 \wedge (x'_\omega = x_\omega \vee x'_\omega = x_1) \implies x'_1 \geq 3 \times x'_\omega + 2$$

De même, on peut obtenir la nouvelle relation $\langle x'_{\omega+1}, x'_\omega\rangle$ comme la borne supérieure des relations $\langle x_1, x_\omega\rangle$ (représentant le nouveau couple d'entrées successives dans le tampon condensé) et $\langle x_\omega, x_{\omega+1}\rangle$ (représentant la relation valable pour toutes les entrées déjà présentes) :

$$x'_\omega \geq 3 \times x'_{\omega+1} + 2$$

La valeur de x_∞ ne change quant à elle pas : écrire dans le tampon n'affecte pas l'entrée la plus ancienne. Enfin, la relation $x_1 \geq 3 \times x_\omega + 2$ est maintenue par croissance de $n \mapsto 3n + 2$ sur l'intervalle considéré (les domaines numériques usuels sont par ailleurs capables d'inférer une telle implication).

On retrouve finalement l'invariant de l'équation (14.2.2) : il est stable par exécution de la boucle (il s'agit donc d'un invariant inductif).

Cet invariant permet d'assurer que le premier transfert vers x^{mem} portera la valeur 5 : il s'agit de l'entrée la plus récente (qui a pu être initialisée lors de la première écriture par le compteur). L'invariant fort sur x_∞ est alors perdu : à cause de la condensation, l'entrée la plus ancienne du tampon peut maintenant être n'importe quelle entrée de x_ω. Toutefois, on sait que, dans le concret, il s'agissait de l'avant-dernière entrée du tampon avant transfert. Or le transfert a porté sur la dernière entrée du tampon, et nous connaissons une relation entre deux entrées successives. Cette relation est donc vérifiée, après transfert, entre la nouvelle dernière entrée du tampon et l'entrée qui vient d'être transférée :

$$x_\infty \geq 3 \times x^{mem} + 2$$

C'est cet invariant, maintenu au cours des transferts, qui nous permet alors de vérifier le programme 14.2 en assurant que chaque interférence écrit en mémoire une valeur $x^{mem\prime}$ supérieure ou égale à $3 \times x^{mem} + 2$. Deux lectures successives par un autre processus peuvent alors soit se produire entre ces interférences, lisant alors la même valeur, soient être intercalées par au moins un transfert, lisant alors des valeurs qui croissent selon la propriété spécifiée.

14.2.3 Conclusion sur les variables auxiliaires

Grâce à des variables auxiliaires supplémentaires, ce nouveau domaine est capable d'exprimer des relations plus complexes entre les entrées successives du tampon. Il est également capable d'inférer des propriétés comme le tri du tampon, en ordre croissant ($x_{\omega+1} \leq x_\omega$) ou décroissant ($x_\omega \leq x_{\omega+1}$), sans que l'utilisateur ait besoin de sélectionner explicitement une abstraction par prédicat pour exprimer la propriété recherchée.

Il est possible d'étendre ce domaine à volonté pour exprimer des relations plus fortes, par exemple sur plus de deux entrées successives. Il peut également être souhaitable d'inclure la relation $\langle x_1, x_2 \rangle$ dans les relations abstraites (ce qui n'est pas le cas dans la représentation qui utilise x_ω et $x_{\omega+1}$) : ceci peut permettre le maintien d'une propriété plus forte lorsque x_1 rejoint x_ω après une écriture (par exemple, dans l'exemple du programme 14.2, $x_\omega = 3 \times x_{\omega+1} + 2$ plutôt qu'une inégalité).

Toutefois, ce domaine rajoute un nombre considérable de dimensions par rapport à celui de la partie III : avec deux symboles supplémentaires par variable partagée ($x_{\omega+1}$ et x_∞), on passe de deux symboles de tampon (x_1 et x_ω) par variable partagée à quatre. Sans abstraction supplémentaire pour réduire la taille des domaines numériques utilisés, on atteindra donc rapidement une limitation d'échelle.

14.3 Conclusion

Afin de vérifier des programmes dont la correction se fonde sur une relation entre les entrées d'un même tampon, nous avons développé dans cette partie des domaines non uniformes. Ceux-ci permettent de représenter des propriétés qui relient ces différentes entrées, gagnant ainsi une précision qui n'était pas accessible pour les domaines uniformes qui ne faisaient aucune distinction entre deux variables concrètes dès lors qu'elles étaient condensées vers le même symbole.

Si ces domaines sont plus précis, ils sont aussi plus complexes, et rajoutent aux éléments abstraits des informations ou dimensions supplémentaires qui ont nécessairement un effet négatif sur les performances. Il faudra donc les évaluer expérimentalement sur des programmes cibles plus ou moins complexes avant de pouvoir valider définitivement nos choix d'abstractions. Nous pensons par ailleurs que ces abstractions plus riches bénéficieraient d'autant plus des techniques d'amélioration des performances qu'un analyseur complet mettra en œuvre, comme le *packing* [CCF+05] pour réduire les dimensions effectives des domaines numériques utilisés (et contrebalancer l'augmentation de la taille totale des éléments abstraits).

S'il existe de nombreux autres domaines non uniformes que nous n'avons pas traité, remarquons enfin que tous ne s'adaptent pas facilement aux abstractions des tampons. Par exemple, une abstraction usuelle consiste à représenter un tableau t à l'aide de deux variables auxiliaires $(i, t[i])$, permettant ainsi d'exprimer une relation entre la valeur d'une case du tableau et son indice. Or nos tampons ont une taille dynamique et non bornée : dans le cas général, après plusieurs étapes d'écriture/transfert et éventuellement d'élargissement, il n'est pas possible d'obtenir une information précise sur la taille finale. Par conséquent, on ne peut pas non plus préciser les valeurs de la dernière case du tampon, qui sert à l'opération de transfert qu'on aurait souhaité rendre plus précise : la relation index/valeur perd alors de son intérêt. De même, la segmentation dynamique du tampon en plusieurs parties [CCL11] n'offre pas d'avantage significatif en raison de l'absence de mise à jour forte de ses entrées (excepté la plus récente) : l'imprécision des parties condensées se propagerait donc vers les éventuelles entrées non condensées sans jamais être réduite.

Chapitre 15

Recherches futures

Les méthodes et domaines présentés dans cette thèse posent les bases d'une interprétation abstraite en mémoire faible. Il reste encore des travaux à effectuer pour bâtir sur ces fondations un analyseur statique robuste. En plus des domaines précis présentés aux chapitres précédents, nous décrirons ici des pistes de recherche pour d'autres futures améliorations possibles.

15.1 Analyse en parallèle des processus

L'analyse modulaire, comme nous l'avons vu, se divise en deux itérations imbriquées : le point fixe interne calcule les états accessibles d'un processus et les interférences qu'il génère, et le point fixe externe réinjecte ces interférences pour le calcul interne des processus suivants.

Il existe alors deux stratégies principales de mise à jour des interférences, qui s'apparentent à des stratégies d'itération chaotique à l'échelle des processus (pour trouver un point fixe des interférences). La première consiste, une fois l'analyse du processus T terminée, à utiliser immédiatement les interférences générées pour l'analyse du processus $T + 1$. Il s'agit d'une stratégie que nous qualifierons de *séquentielle*. La stratégie *parallèle* consiste quant à elle à analyser tous les processus avec le même ensemble d'interférences, avant de collecter ensemble celles qu'ils auront générées et de les passer en paramètre du tour d'analyse suivant. Ces deux stratégies sont analogues aux méthodes de Gauss-Seidel et Jacobi de résolution itérative d'un système linéaire [CC92].

Notons d'abord qu'il n'existe a priori pas de relation de précision entre les différentes méthodes : comme pour toute stratégie d'itération menant à des calculs abstraits exécutés dans un ordre différent, il est probable que l'on puisse construire un programme vérifiable uniquement avec la stratégie parallèle, et *vice versa*.

Toutefois, on peut s'attendre à ce que la stratégie séquentielle converge plus rapidement : en effet, en intégrant immédiatement les interférences générées dans la suite de l'analyse, elle ne retarde pas leur prise en compte, contrairement à la stratégie parallèle qui lancera donc une analyse interne supplémentaire des processus suivants. Notons qu'il s'agit encore une fois d'une heuristique, et que la présence d'un élargissement peut conduire à des observations contraires.

Cependant, la méthode parallèle conduit naturellement à considérer la parallélisation du calcul lui-même : à chaque tour, l'analyse de chaque processus étant indépendante, rien n'empêche de les calculer chacune sur un processus ou un cœur dédié. Si cette technique n'a pas de raison d'être un ordonnancement optimal, elle présente toutefois l'avantage d'être plus facilement mise en œuvre, ne demandant aucun autre changement dans la structure de l'implémentation que le calcul en parallèle plutôt que séquentiellement d'une liste connue d'opérations indépendantes.

Le langage OCaml, avec lequel nous avons programmé notre prototype, ne permet pas à l'heure actuelle d'écrire nativement des programmes bénéficiant d'un processeur multi-cœurs. Toutefois, une bibliothèque comme Parmap [DDC12] pourrait suffire pour l'effort minimal de parallélisation du paragraphe précédent : en transformant l'ensemble d'analyses internes des processus en un `map` parallèle, on devrait déjà obtenir une analyse générale plus rapide.

Nous avons cependant observé que, parmi les calculs internes successifs, le calcul au dernier tour du dernier processus était en général sensiblement plus long que tous les autres. Nous nous attendons donc à un gain de performance qui, bien que réel, resterait limité.

Une véritable réflexion sur la parallélisation de l'analyse pourrait en revanche conduire à de meilleurs algorithmes d'ordonnancement (qui, notamment, n'attendent pas la fin de l'analyse de chaque processus pour lancer le tour suivant), qui pourraient le cas échéant conduire à des gains plus intéressants. Notons par ailleurs qu'un travail similaire a déjà été effectué avec des résultats positifs sur l'analyseur de programmes séquentiels Astrée [Mon05].

Remarquons que, bien que cet aspect de l'implémentation semble indépendant de la cohérence faible, on n'observerait pas nécessairement les mêmes effets que pour l'analyse en cohérence séquentielle : il serait donc nécessaire de valider expérimentalement d'éventuels résultats dans le modèle mémoire choisi.

15.2 Mise en production dans un analyseur réel

Nous avons vu au chapitre 12 que, si l'analyse modulaire permettait d'obtenir des performances sensiblement meilleures quand le nombre de processus augmentait, le temps de l'analyse restait exponentiel dans notre modèle mémoire. Le passage à l'échelle d'un analyseur réaliste en production nécessitera donc sans doute de coupler cette analyse modulaire avec une analyse moins précise et plus performante, par exemple non relationnelle ou insensible au flot de contrôle [KW17, Min11]. L'analyse précise pourra alors être utilisée sur des fragments restreints et difficiles à vérifier du programme, comme ceux qui correspondent à des opérations de synchronisation, où il est nécessaire de modéliser avec précision l'interaction entre les processus. À l'inverse, l'analyse plus performante pourra servir sur des parties du programme où cette précision n'est pas requise, par exemple lorsque les processus effectuent des calculs importants en interagissant peu. Pour passer à l'échelle, il est de toute façon nécessaire qu'une analyse de programmes concurrents soit modulaire : nous pensons donc que la méthode développée en partie III pose de meilleures bases pour ce type de couplage qu'une analyse monolithique.

Les programmes réels mettent également en œuvre des traits additionnels qui ne font pas partie de la sémantique que nous avons considérée. Pour la gestion des pointeurs et des structures de données, il faudrait ainsi y intégrer des analyses de pointeurs, des abstractions de tas ou de l'analyse de forme selon les objectifs. D'autres traits sont liés au modèle de parallélisme : c'est notamment le cas d'opérations atomiques comme `compare-and-swap` et des instructions de verrou `lock`. Les premières peuvent être ajoutées facilement à notre analyse : il s'agit d'évaluer la condition d'exécution, le cas échéant d'effectuer l'opération, et de transférer le tampon en mémoire (comme pour $[\![\texttt{mfence}]\!]$) sans générer ni appliquer d'interférence entre ces étapes. Les secondes sont également compatibles avec notre sémantique, mais demanderaient légèrement plus de travail : l'idée serait de générer une interférence globale pour un bloc `lock/unlock` plutôt que des transitions individuelles pour chaque opération du bloc.

Enfin, pour pouvoir vérifier des programmes réalistes, il serait souhaitable d'implémenter des techniques d'analyse connues qui permettent généralement d'améliorer sensiblement les performances ou la précision d'une interprétation abstraite. Citons par exemple le partitionnement de traces [RM07], qui permet de distinguer localement des branches d'exécution pour améliorer la

précision par rapport aux unions aux points de jonction. Citons également le *packing* [CCF+05], qui consiste à regrouper les variables par paquets, chaque paquet étant abstrait dans un domaine relationnel distinct. Nos domaines font intervenir des domaines numériques intégrant un grand nombre de variables : cette réduction des dimensions pourrait donc conduire à une amélioration non négligeable des performances.

15.3 Vers d'autres modèles

Notre cible principale, dans cette thèse, était le modèle TSO. Bien qu'il s'agisse d'un modèle réaliste, notamment à la base des architectures x86, c'est un modèle très peu relâché : une conséquence en est que sa sémantique peut s'écrire facilement sous une forme opérationnelle.

Ce n'est plus le cas des autres modèles, comme POWER (qui est également à la base des architectures ARM), ou comme les modèles de plus haut niveau de langages comme C ou Java. Ceux-ci sont plus souvent décrits sous une forme axiomatique : plutôt que de spécifier les exécutions possibles étape par étape à partir d'un point donné, ils fournissent un ensemble de règles plus ou moins complexes permettant de décider si une exécution donnée est valide ou non pour un programme. Autrement dit, plutôt que de construire explicitement toutes les exécutions valides d'un programme (ce qui est un problème difficile lorsque le modèle est suffisamment complexe), les règles de ces modèles permettent seulement de savoir si une exécution donnée leur appartient.

Or l'interprétation abstraite est fondée sur la construction pas à pas des exécutions possibles d'un programme : les modèles opérationnels (ou les formes équationnelles, similaires en ce point de vue) sont donc particulièrement adaptés à la mise en place de cette technique. À l'inverse, les modèles axiomatiques ne permettent pas dans le cas général de construire mécaniquement une abstraction, c'est-à-dire une sur-approximation, des comportements d'un programme.

Avant de pouvoir analyser des programmes dans de tels modèles, il faut donc commencer par donner une description constructive de leur sémantique. Lorsqu'on en cherche une définition précise, il s'agit en général d'un problème difficile. De plus, des modèles comme C [KHL+17] et Java [ŠA08] sont passés par de nombreuses révisions et font encore l'objet de recherches pour en améliorer certains aspects peu convaincants, rendant ce travail d'autant plus délicat. Remarquons cependant que l'interprétation abstraite, par construction, calculera une abstraction sûre des comportements exacts des programmes : il est donc suffisant de donner une sémantique opérationnelle qui elle-même les sur-approxime dans le modèle recherché.

Nous pensons par ailleurs que l'approche modulaire, en plus de permettre un meilleur passage à l'échelle, est mieux indiquée pour la définition de telles sémantiques. Elle permet en effet de distinguer la sémantique locale d'un processus, qui reste semblable dans les modèles faiblement cohérents à sa définition séquentielle, de la communication entre les processus (*via* la mémoire partagée), qui fait intervenir les effets relâchés.

Conclusion

Réalisations

Écrire un programme correct, ou vérifier qu'il l'est, est une tâche difficile, chronophage et sujette à de nombreuses erreurs. La concurrence et les modèles mémoire faiblement cohérents la compliquent davantage, rendant primordiaux les outils de vérification automatique conçus à cette fin. Dans cette thèse, nous avons conçu et implémenté des méthodes d'analyse statique, par interprétation abstraite, de programmes concurrents dans les modèles mémoire TSO et PSO.

Notre première approche a été une méthode monolithique fondée sur le calcul standard d'un point fixe sur le graphe produit du programme. Pour cela, nous avons proposé un domaine abstrait pour représenter les états dans PSO par composition du partitionnement des états selon la taille de leur tampon, de la condensation des anciennes entrées de chaque tampon vers un symbole unique, puis d'une abstraction numérique paramétrique. Par l'implémentation de ces méthodes dans un prototype d'analyseur, nous avons montré expérimentalement à la fois l'efficacité et la précision de cette méthode : l'ordre de grandeur des performances est correct, et contrairement à l'état de l'art, nos domaines nous permettent d'analyser de façon précise des programmes comportant un nombre non borné d'états accessibles, qui peuvent de plus exhiber des tampons eux-mêmes non bornés.

Pour résoudre les problèmes de passage à l'échelle de cette analyse, nous avons par la suite montré comment le cadre formel des interférences pouvait nous permettre de concevoir une analyse modulaire. Nous avons ainsi défini des abstractions des états locaux et des interférences semblables aux domaines monolithiques, ainsi qu'une abstraction du contrôle des processus permettant de réduire sensiblement la taille de l'espace de contrôle à explorer. Une contribution spécifique de cette analyse modulaire dans TSO a été de montrer comment un théorème de commutation des transferts, qui se présentait comme une simple optimisation de l'analyse monolithique, fournit dans le cadre de l'analyse modulaire un algorithme de clôture par transfert bien meilleur que l'itération naïve de lourds calculs pour obtenir un point fixe. Les résultats expérimentaux de l'analyse modulaire par notre prototype valident cette approche : les abstractions supplémentaires ne conduisent pas à une perte de précision significative et le passage à l'échelle est exponentiellement meilleur que pour l'analyse monolithique.

Enfin, nous avons proposé plusieurs pistes d'amélioration de l'analyse ainsi obtenue. Pour plus de précision, nous avons défini un domaine permettant de retrouver une partie de l'information d'ordre entre différentes variables pour analyser des programmes dont la correction se base sur des règles de TSO qui ne seraient plus valides dans PSO. Nous avons également montré comment les abstractions non uniformes, pour une complexité réduite, permettaient de distinguer différentes entrées d'un même tampon afin de préciser les valeurs transférées en mémoire partagée. Nous avons donné l'exemple des tampons triés et montré comment, à l'aide d'une variable supplémentaire, il était possible d'inférer des relations complexes entre ces entrées d'un même tampon sans exiger de l'utilisateur qu'il les spécifie lui-même.

Perspectives futures

Les domaines et les techniques que nous avons proposés ont fait leurs preuves sur l'analyse de programmes simples dans TSO. Par la conception d'abstractions et de méthodes de calcul dédiées, nous pensons donc avoir posé des bases saines pour la conception d'analyses performantes, sûres et précises pour dans les modèles mémoire faiblement cohérents.

Les pistes de travaux futurs construits sur cette thèse s'orientent selon nous sur deux axes : la mise en production d'un analyseur réel et la recherche de domaines permettant d'analyser des modèles mémoire plus riches.

Un passage à l'échelle en production exige dès le départ une analyse modulaire : nous pensons donc que celle que nous avons conçue peut servir de base à des recherches visant à la coupler avec des techniques d'interprétation abstraite avancée visant à analyser efficacement des programmes réels.

Quant aux modèles plus complexes, certains de leurs traits, similaires à ceux de TSO, peuvent sans doute s'abstraire avec des idées semblables. Fondamentalement, nous pensons par ailleurs que l'approche modulaire permettra de considérer ces analyses en se focalisant sur les mécanismes de communication inter-processus, qui sont au cœur de la définition de la cohérence des modèles mémoire. Nous espérons donc que les interférences, en plus des performances qu'elles proposent, permettent une définition élégante des sémantiques abstraites dans les modèles plus complexes, ce qui conduirait à des résultats particulièrement intéressants.

Annexe

Annexe A

Programmes de test

```
/* Specification:
   (0:bp1,1:bp2) 1:lRCnt <= 0:lSCnt && * 0:lSCnt <= 1:lRCnt + 1 */

bool msg, ack;
assert (not msg && not ack); // Initial condition

thread {
    lSSt = false;
    lSCnt = 0;
    while (true) {
        lAck = ack;
        if ((not lAck and not lSSt) or (lAck and lSSt)) {
            lSSt = not lSSt;
            lSCnt = lSCnt + 1;
        bp1:
            msg = lSSt;
            // mfence;
        }
    }
}

thread {
    lRSt = false;
    lRCnt = 0;
    while (true) {
        lMsg = msg;
        if ((lMsg and not lRSt) or (not lMsg and lRSt)) {
            lRSt = lMsg;
            lRCnt = lRCnt + 1;
            ack = lRSt;
            // mfence
        bp2:
        }
    }
}
```

Programme A.1 – Abp

```
1  /* Specification:
2     (0:critical_section,1:critical_section) false
3  */
4
5  int number0, number1, x;
6  bool entering0, entering1;
7
8  // Initial condition
9  assert (number0 = 0 && number1 = 0 && not entering0 && not entering1);
10
11 thread {
12     while (true) {
13         entering0 = true;
14         mfence;
15
16         max = number1;
17         number0 = 1 + max;
18         mfence;
19
20         entering0 = false;
21         mfence;
22
23         e = entering1;
24         while (e) {
25             e = entering1;
26         }
27
28         n1 = number1;
29         n0 = number0;
30
31         while (n1 != 0 && n1 < n0) {
32             n1 = number1;
33         }
34
35     critical_section:
36
37         number0 = 0;
38         mfence;
39     }
40 }
41
42 thread {
43     while (true) {
44         entering1 = true;
45         mfence;
46
47         max = number0;
48         number1 = 1 + max;
49         mfence;
50
51         entering1 = false;
52         mfence;
53
```

```
54          e = entering0;
55          while (e) {
56              e = entering0;
57          }
58
59          n0 = number0;
60          n1 = number1;
61
62          while (n0 != 0 && n0 <= n1) {
63              n0 = number0;
64          }
65
66      critical_section:
67
68          number1 = 0;
69          mfence;
70      }
71  }
```

Programme A.2 – Bakery

```
1   /* Specification:
2      (0:bp,1:bp) 1:i >= 5 and 1:i <= 5 and 0:j >= 5 and 0:j <= 5
3   */
4
5   int i, j;
6
7   // Initial condition
8   assert(true);
9
10  thread {
11      i = 0;
12      t = i;
13      while (t <= 4) {
14          t = t + 1;
15          i = t;
16          mfence;
17      }
18  bp:
19  }
20
21  thread {
22      j = 0;
23      u = j;
24      while (u <= 4) {
25          u = u + 1;
26          j = u;
27          mfence;
28      }
29  bp:
30  }
```

Programme A.3 – Concloop

```
1   /* Specification :
2      (0:critical, 1:critical) false
3   */
4
5   int turn, x;
6   bool flag0, flag1;
7
8   // Initial condition
9   assert(turn = 0 && not flag0 && not flag1 && x = 0);
10
11  thread {
12      while (true) {
13          flag0 = true;
14          mfence;
15
16          f1 = flag1;
17          while (f1) {
18              t1 = turn;
19              if (t1 > 0) {
20                  flag0 = false;
21                  // mfence
22                  t1 = turn;
23                  while (t1 > 0) {
24                      t1 = turn;
25                  }
26                  flag0 = true;
27                  mfence;
28              }
29              f1=flag1;
30          }
31
32      critical:
33          turn = 1;
34          // mfence
35          flag0 = false;
36          // mfence
37      }
38  }
39
40  thread {
41      while (true) {
42          flag1 = true;
43          mfence;
44
45          f2 = flag0;
46          while (f2) {
47              t2 = turn;
48              if (t2 < 1) {
49                  flag1 = false;
50                  // mfence
51                  t2 = turn;
52                  while(t2 < 1) {
53                      t2 = turn;
```

```
54                  }
55                  flag1 = true;
56                  mfence;
57              }
58              f2 = flag0;
59          }
60
61      critical:
62          turn = 0;
63          // mfence
64          flag1 = false;
65          // mfence
66      }
67  }
```

Programme A.4 – Dekker

```
1   /* Specification:
2      (0:bp1, 1:bp2) false
3   */
4
5   bool b0, b1, turn0, turn1;
6   int x;
7
8   // Initial condition
9   assert (not b0 and not b1 and x = 0);
10
11  thread {
12      while (true) {
13          b0 = true;
14          mfence;
15          local = turn1;
16          turn0 = local;
17          mfence;
18          local_b1 = b1;
19          local_turn1 = turn1;
20          while (local_b1 && (!local  local_turn1) && (local  !local_turn1)) {
21              local_b1 = b1;
22              local_turn1 = turn1;
23          }
24          x = 3;
25          mfence;
26      bp1:
27          b0 = false;
28          // mfence
29      }
30  }
31
32  thread {
33      while (true) {
34          b1 = true;
35          mfence;
36          local = not turn0;
```

```
37          turn1 = local;
38          mfence;
39          local_b0 = b0;
40          local_turn0 = turn0;
41          while not (! local_b0  local && local_turn0  !local && !local_turn0) {
42                  local_b0 = b0;
43                  local_turn0 = turn0;
44              }
45          x = 4;
46          mfence;
47      bp2:
48          b1 = false;
49          // mfence
50      }
51  }
```

Programme A.5 – Kessel

```
1   /* Specification:
2      @ (1:fail) false
3   */
4
5   int p0, p1, t0, t1;
6
7   // Initial condition
8   assert (t0 = 0 && t1 = 0);
9
10  thread /* Reader */ {
11      time0 = 10;
12      p0 = 0;
13      // mfence;
14      temp = p0;
15      while (temp < 10) {
16          temp = t0;
17          temp = temp + time0;
18          t0 = temp;
19          // mfence;
20          local_t0 = t0;
21          local_t1 = t1;
22          while (not (local_t0 <= local_t1)) {
23              local_t0 = t0;
24              local_t1 = t1;
25          }
26          temp = p0;
27          temp = temp + 1;
28          p0 = temp;
29          // mfence;
30      }
31  }
32
33
34  thread /* Writer */ {
35      time1 = 20;
```

```
36      p1 = 0;
37      // mfence;
38      temp = p1;
39      while (temp < 10) {
40          temp = t1;
41          temp = temp + time1;
42          t1 = temp;
43          // mfence;
44          local_t0 = t0;
45          local_t1 = t1;
46          while (not (local_t1 <= local_t0)) {
47              local_t0 = t0;
48              local_t1 = t1;
49          }
50          temp = p1;
51          temp = temp + 1;
52          p1 = temp;
53          // mfence;
54      }
55  fail: // Specification is that this label is not reachable;
56  }
```

Programme A.6 – Loop2 TLM

```
1   /* Specification:
2      (0:critical, 1:critical) false
3   */
4
5   bool flag0, flag1, turn;
6
7   // Initial condition
8   assert (true);
9
10  thread /* T0 */ {
11      while (true) {
12          flag0 = true;
13          mfence;
14          turn = true;
15          mfence;
16
17          lturn0 = turn;
18          lflag1 = flag1;
19          while (lflag1 and lturn0) {
20              lflag1 = flag1;
21              lturn0 = turn;
22          }
23      critical:
24          flag0 = false;
25          // mfence;
26      }
27  }
28
29  thread /* T1 */ {
```

```
30      while (true) {
31          flag1 = true;
32          mfence;
33          turn = false;
34          mfence;
35
36          lturn1 = turn;
37          lflag0 = flag0;
38          while (lflag0 and not lturn1) {
39              lturn1 = turn;
40              lflag0 = flag0;
41          }
42      critical:
43          flag1 = false;
44          // mfence;
45      }
46  }
```

Programme A.7 – Peterson

```
1   /* Specification:
2      (0:bp, 1:bp) 1:tail < 0:h1
3   */
4
5   int head;
6
7   // Initial condition
8   assert (true);
9
10  thread /* Enqueue */ {
11      while (true) {
12          h1 = head;
13      bp:
14          h1 = h1 + 1;
15          head = h1;
16          // mfence
17      }
18  }
19
20  thread /* Dequeue */ {
21      tail = head;
22      while (true) {
23          h2 = head;
24          while (tail >= h2) {
25              h2 = head;
26          }
27      bp:
28          tail = tail + 1;
29      }
30  }
```

Programme A.8 – Queue

Bibliographie

[AAJL16] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In *International Conference on Computer Aided Verification*, pages 134–156. Springer, 2016. [Cité en page 53.]

[AAN15] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. The best of both worlds : Trading efficiency and optimality in fence insertion for TSO. In *ESOP*, pages 308–332. Springer, 2015. [Cité en page 53.]

[ABBM10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *ACM Sigplan Notices*, volume 45, pages 7–18. ACM, 2010. [Cité en page 53.]

[AG96] Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models : A tutorial. *IEEE Computer*, 29(12) :66–76, 1996. [Cité en page 7.]

[AKL+11] Jade Alglave, Daniel Kroening, John Lugton, Vincent Nimal, and Michael Tautschnig. Soundness of data flow analyses for weak memory models. In *Programming Languages and Systems*, pages 272–288. Springer, 2011. [Cité en pages 53 et 91.]

[AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *Programming Languages and Systems*, pages 512–532. Springer, 2013. [Cité en page 53.]

[BCC+02] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The essence of computation*, pages 85–108. Springer, 2002. [Cité en page 69.]

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against TSO. In *Programming Languages and Systems*, pages 533–553. Springer, 2013. [Cité en page 53.]

[BGOS18] Sam Blackshear, Nikos Gorogiannis, Peter W O'Hearn, and Ilya Sergey. Racerd : Compositional static race detection. *Proceedings of the ACM on Programming Languages*, 1(1), 2018. [Cité en page 91.]

[BHRZ03] Roberto Bagnara, Patricia M Hill, Elisa Ricci, and Enea Zaffanella. Precise widening operators for convex polyhedra. In *International Static Analysis Symposium*, pages 337–354. Springer, 2003. [Cité en page 22.]

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and their Applications*, pages 128–141. Springer, 1993. [Cité en pages 41, 55, 79 et 100.]

[Bry86] Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8) :677–691, 1986. [Cité en page 25.]

[CC77a] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977. [Cité en pages 5, 19, 24, 38, 40 et 93.]

[CC77b] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conference on Formal Description of Programming Concepts, St. Andrews, NB, Canada*. North-Holland Publishing Company, 1977. [Cité en page 111.]

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 269–282. ACM, 1979. [Cité en pages 65, 125, 126 et 128.]

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *The Journal of Logic Programming*, 13(2-3) :103–179, 1992. [Cité en page 153.]

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005. [Cité en pages 152 et 155.]

[CCF+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the Astrée static analyzer. In *Annual Asian Computing Science Conference*, pages 272–300. Springer, 2006. [Cité en page 146.]

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *ACM SIGPLAN Notices*, volume 46, pages 105–118. ACM, 2011. [Cité en pages 149, 150 et 152.]

[CFS07] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles. Designing a generic graph library using ml functors. *Trends in functional programming*, 8 :124–140, 2007. [Cité en page 79.]

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 84–96. ACM, 1978. [Cité en page 22.]

[Che68] NV Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6) :282–293, 1968. [Cité en page 22.]

[Cou03] Patrick Cousot. Verification by abstract interpretation. In *Verification : Theory and Practice*, pages 243–268. Springer, 2003. [Cité en pages 149 et 150.]

[CWZ90] David R Chase, Mark Wegman, and F Kenneth Zadeck. *Analysis of pointers and structures*, volume 25. ACM, 1990. [Cité en page 69.]

[DDC12] Marco Danelutto and Roberto Di Cosmo. A "minimal disruption" skeleton experiment : Seamless map & reduce embedding in OCaml. In *Procedia Computer Science*, volume 9, pages 1837–1846. Elsevier, 2012. [Cité en page 154.]

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3 : An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. [Cité en page 80.]

[DMVY14] Andrei Dan, Yuri Meshman, Martin Vechev, and Eran Yahav. Effective abstractions for verification under relaxed memory models. In *Verification, Model Checking, and Abstract Interpretation*, pages 449–466. Springer, 2014. [Cité en pages 53, 79, 80 et 81.]

[GBCS07] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *ACM SIGPLAN Notices*, volume 42, pages 266–277. ACM, 2007. [Cité en page 91.]

[GDD$^+$04] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 512–529. Springer, 2004. [Cité en pages 69, 71, 72, 76, 77 et 78.]

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4) :165–190, 1989. [Cité en page 126.]

[GRS05] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. *ACM SIGPLAN Notices*, 40(1) :338–350, 2005. [Cité en page 69.]

[HMVW17] Lukáš Holík, Roland Meyer, Tomáš Vojnar, and Sebastian Wolff. Effect summaries for thread-modular analysis. In *International Static Analysis Symposium*, pages 169–191. Springer, 2017. [Cité en page 91.]

[HP08] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Notices*, volume 43, pages 339–348. ACM, 2008. [Cité en page 149.]

[ISO11] ISO. *ISO/IEC 14882 :2011 Information technology — Programming languages — C++*. 2011. [Cité en page 49.]

[Jea03] Bertrand Jeannet. Dynamic partitioning in linear relation analysis : Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1) :5–37, 2003. [Cité en page 27.]

[Jea09] Bertrand Jeannet. The BDDApron logico-numerical abstract domains library. http://pop-art.inrialpes.fr/~bjeannet/bjeannet-forge/bddapron/, 2009. [Cité en page 79.]

[Jea13] Bertrand Jeannet. Relational interprocedural verification of concurrent programs. *Software & Systems Modeling*, 12(2) :285–306, 2013. [Cité en page 80.]

[JLB$^+$15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified c static analyzer. *ACM SIGPLAN Notices*, 50(1) :247–259, 2015. [Cité en page 4.]

[JM09] Bertrand Jeannet and Antoine Miné. Apron : A library of numerical abstract domains for static analysis. In *Computer Aided Verification*, pages 661–667. Springer, 2009. [Cité en page 79.]

[Jou16] Jacques-Henri Jourdan. Sparsity preserving algorithms for octagons. In *Numerical and Symbolic Abstract Domains Workshop (NSAD)*, volume 331, pages 57–70. Elsevier, September 2016. [Cité en page 23.]

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4 : Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009. [Cité en page 4.]

[KHL+17] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *ACM SIGPLAN Notices*, volume 52, pages 175–189. ACM, 2017. [Cité en page 155.]

[KVY11] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *ACM SIGPLAN Notices*, volume 46, pages 187–198. ACM, 2011. [Cité en page 53.]

[KVY12] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *ACM SIGACT News*, 43(2) :108–123, 2012. [Cité en page 53.]

[KW16] Markus Kusano and Chao Wang. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 799–809. ACM, 2016. [Cité en page 91.]

[KW17] Markus Kusano and Chao Wang. Thread-modular static analysis for relaxed memory models. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 337–348. ACM, 2017. [Cité en pages 91 et 154.]

[Lam74] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8) :453–455, 1974. [Cité en page 83.]

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9) :690–691, 1979. [Cité en pages 6 et 43.]

[Lee59] Chang-Yeong Lee. Representation of switching circuits by binary-decision programs. *Bell system Technical journal*, 38(4) :985–999, 1959. [Cité en page 25.]

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009. [Cité en page 4.]

[LLF+96] Jacques-Louis Lions, Lennart Lübeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, and Colin O'Halloran. Ariane 5 flight 501 failure report by the inquiry board, 1996. [Cité en page 1.]

[LT93] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7) :18–41, 1993. [Cité en page 1.]

[LV92] Hervé Le Verge. *A note on Chernikova's algorithm.* PhD thesis, INRIA, 1992. [Cité en page 22.]

[Mar17] Alexandre Maréchal. *New Algorithmics for Polyhedral Calculus via Parametric Linear Programming. (Nouvelle Algorithmique pour le Calcul Polyédral via Programmation Linéaire Paramétrique).* PhD thesis, Grenoble Alpes University, France, 2017. [Cité en page 22.]

[MDVY14] Yuri Meshman, Andrei Dan, Martin Vechev, and Eran Yahav. Synthesis of memory fences via refinement propagation. In *Static Analysis*, pages 237–252. Springer, 2014. [Cité en pages 53 et 80.]

[MFK+16] Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. Polyhedral approximation of multivariate polynomials using handelman's theorem. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 166–184. Springer, 2016. [Cité en page 22.]

[MG16] David Monniaux and Laure Gonnord. Cell morphing : from array programs to array-free horn clauses. In *International Static Analysis Symposium*, pages 361–382. Springer, 2016. [Cité en page 149.]

[Min01] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *Programs as Data Objects*, pages 155–172. Springer, 2001. [Cité en page 140.]

[Min06] Antoine Miné. The octagon abstract domain. *Higher-order and symbolic computation*, 19(1) :31–100, 2006. [Cité en page 23.]

[Min11] Antoine Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *ESOP*, volume 11, pages 398–418. Springer, 2011. [Cité en pages 87, 89, 91, 108 et 154.]

[Min12] Antoine Miné. Static analysis by abstract interpretation of sequential and multithread programs. In *10th School of Modelling and Verifying Parallel Processes*, pages 35–48, 2012. [Cité en page 99.]

[Min14] Antoine Miné. Relational thread-modular static value analysis by abstract interpretation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 39–58. Springer, 2014. [Cité en pages 87, 89, 91, 100, 109 et 112.]

[MM17] Raphaël Monat and Antoine Miné. Precise thread-modular abstract interpretation of concurrent programs using relational interference abstractions. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 386–404. Springer, 2017. [Cité en pages 91, 100, 108, 111, 117 et 118.]

[Mon05] David Monniaux. The parallel implementation of the Astrée static analyzer. In *Asian Symposium on Programming Languages and Systems*, pages 86–96. Springer, 2005. [Cité en page 154.]

[Mon10] David Monniaux. Quantifier elimination by lazy model enumeration. In *International Conference on Computer Aided Verification*, pages 585–599. Springer, 2010. [Cité en page 24.]

[MPA05] Jeremy Manson, William Pugh, and Sarita V Adve. *The Java memory model*, volume 40. ACM, 2005. [Cité en page 49.]

[MPS+17] Suvam Mukherjee, Oded Padon, Sharon Shoham, Deepak D'Souza, and Noam Rinetzky. Thread-local semantics and its efficient sequential abstractions for race-free programs. In *International Static Analysis Symposium*, pages 253–276. Springer, 2017. [Cité en page 91.]

[NQ10] Duong Nguyen Que. *Robust and generic abstract domain for static program analyses : the polyhedral case*. PhD thesis, École Nationale Supérieure des Mines de Paris, France, 2010. [Cité en page 24.]

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3) :115–116, 1981. [Cité en pages 84 et 130.]

[PR05] François Pottier and Didier Rémy. The essence of ML type inference. In *Advanced Topics in Types and Programming Languages*, pages 389–489. The MIT Press, 2005. [Cité en page 4.]

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, 1953. [Cité en page 2.]

[Rid10] Tom Ridge. A rely-guarantee proof system for x86-TSO. In *International Conference on Verified Software : Theories, Tools, and Experiments*, pages 55–70. Springer, 2010. [Cité en page 91.]

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5) :26, 2007. [Cité en page 154.]

[RSW04] Thomas W Reps, Mooly Sagiv, and Reinhard Wilhelm. Static program analysis via 3-valued logic. In *International Conference on Computer Aided Verification*, pages 15–30. Springer, 2004. [Cité en page 69.]

[ŠA08] Jaroslav Ševčík and David Aspinall. On validity of program transformations in the java memory model. In *ECOOP 2008–Object-Oriented Programming*, pages 27–51. Springer, 2008. [Cité en page 155.]

[Sch98] Alexander Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998. [Cité en page 22.]

[SJ11] Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *International Static Analysis Symposium*, pages 233–248. Springer, 2011. [Cité en page 27.]

[SS12] Holger Siegel and Axel Simon. Summarized dimensions revisited. *Electronic Notes in Theoretical Computer Science*, 288 :75–86, 2012. [Cité en pages 76, 78 et 79.]

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. *ACM SIGPLAN Notices*, 46(6) :175–186, 2011. [Cité en page 49.]

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O Myreen. x86-TSO : a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7) :89–97, 2010. [Cité en page 44.]

[Tur37] Alan M Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4) :153–163, 1937. [Cité en page 2.]

RÉSUMÉ

Parmi les méthodes de certification de logiciels critiques, l'analyse statique vise à établir l'absence d'erreurs dans toutes les exécutions possibles d'un programme donné. L'interprétation abstraite fournit un cadre théorique général permettant de concevoir de telles analyses sûres par construction : elles n'oublient aucun comportement de la cible.

Nous nous intéressons dans cette thèse à la vérification de programmes concurrents s'exécutant dans des modèles mémoire dits faiblement cohérents. En plus des exécutions séquentiellement cohérentes générées par les entrelacements des processus, ces modèles autorisent des comportements contre-intuitifs rendant le raisonnement d'autant plus difficile.

Nous proposons des domaines abstraits dédiés pour analyser les programmes s'exécutant dans de tels domaines. Résultats expérimentaux à l'appui, nous montrons comment ces domaines permettent, à l'aide de méthodes de calcul adaptées, de vérifier avec précision la correction d'algorithmes classiques de programmation concurrente.

Pour permettre un meilleur passage à l'échelle, nous étendons ces travaux à la conception d'une analyse modulaire. Nous montrons par l'expérience comment, en tirant profit d'abstractions spécifiques et d'une stratégie d'itération optimisée, cette méthode permet d'analyser efficacement des programmes comportant un plus grand nombre de processus.

Nous définissons finalement des domaines permettant d'inférer avec précision des relations spécifiques au modèle mémoire afin de pouvoir certifier des programmes aux invariants complexes.

MOTS CLÉS

Interprétation abstraite, Analyse statique modulaire, Mémoire faiblement cohérente, Programmes concurrents

ABSTRACT

Static analysis aims to certify critical software by establishing the absence of errors amongst every possible execution of a given program. Abstract interpretation provides a general theoretical framework to build such analysis that are sound-by-design : they take every possible behaviour of the target into account.

In this thesis, we address the verification of concurrent programs that run in weakly consistent memory models. In addition to sequentially consistent executions that match the interleavings of the threads, these models allow counter-intuitive behaviours that further complicate human reasoning.

We propose dedicated abstract domains to analyse the programs that run under such models. Supported by experimental results, we show how these domains, by means of adapted iteration methods, allow precisely verifying the correction of typical concurrent programming algorithms.

To improve the scaling, we extend this work to the design of a thread-modular analysis. We show by experimentations how, by leveraging specific abstractions and an optimised iteration strategy, this method allows efficiently analysing programs that involve a greater number of threads.

We eventually define abstract domains allowing to precisely infer relations that are specific to the memory model, in order to certify programs entailing complex invariants.

KEYWORDS

Abstract interpretation, Thread-modular static analysis, Weakly consistent memory, Concurrent program