Automatisation de la Certification Formelle de Systèmes Critiques par Instrumentation d'Interpréteurs Abstraits

Manuel Garnacho

▶ **To cite this version:**

Manuel Garnacho. Automatisation de la Certification Formelle de Systèmes Critiques par Instrumentation d'Interpréteurs Abstraits. Logique en informatique [cs.LO]. Université de Grenoble, 2010. Français. NNT : . tel-00720595

HAL Id: tel-00720595
https://theses.hal.science/tel-00720595

Submitted on 25 Jul 2012

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.

UNIVERSITÉ DE GRENOBLE

N° attribué par la bibliothèque

THÈSE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : Informatique

préparée au Laboratoire VERIMAG

dans le cadre de l'École Doctorale Mathématiques,
Sciences et Technologies de l'Information, Informatique

présentée et soutenue publiquement

par

Manuel GARNACHO

le 27 août 2010

Automatisation de la Certification Formelle de Systèmes Critiques par Instrumentation d'Interpréteurs Abstraits

---

Directeurs de thèse : Michaël Périn et Yassine Lakhnech

devant le jury composé de :

Laurence PIERRE	(Pr. Université Joseph Fourier)	Présidente
Olga KOUCHNARENKO	(Pr. Université de Franche-Comté)	Rapporteur
Thomas JENSEN	(Directeur de Recherche IRISA)	Rapporteur
Solange COUPET-GRIMAL	(MCF. Université de Provence)	Examinateur
Gilles BARTHE	(Pr. IMDEA Software Faculty)	Examinateur
Michaël PÉRIN	(MCF. Université Joseph Fourier)	Examinateur

Résumé : Les travaux menés dans cette thèse portent sur la certification de programmes impératifs utilisés dans des applications critiques. Les certificats établissent la validité des propriétés sémantiques des programmes. Ils sont produits sous forme de preuves déductives vérifiables par machine. Le défi relevé dans cette thèse est d'automatiser la construction des preuves de correction de programmes. Nous avons suivie l'approche de Floyd-Hoare pour prouver que les propriétés sémantiques sont des invariants du programme et nous avons utilisé le système Coq pour vérifier la validité des preuves.

Dans nos travaux, on considére le cas où les propriétés sémantiques sont calculées par des analyseurs statiques de programmes qui s'appuient sur la théorie de l'interprétation abstraite. Nous proposons une méthode d'instrumentation des analyseurs statiques de programmes afin de leur faire générer automatiquement un certificat pour chaque propriété sémantique calculée. L'instrumentation consiste à associer à certaines fonctions d'un analyseur statique des patrons de preuve qui à l'exécution généreront les certificats. De tels analyseurs instrumentés ne sont pas pour autant certifiés puisque leur correction pour toutes leurs entrées possibles n'est pas garantie, néanmoins ils sont capables de justifier par une preuve formelle la correction du résultat de chacun de leurs calculs. Bien sûr, si pour une entrée possible l'exécution de l'analyseur est boguée le résultat sera erroné et la justification produite ne sera pas une preuve valide ; elle sera rejettée par le vérificateur de preuves. Cette approche permet de certifier avec un très haut-niveau de confiance (celui du vérificateur de preuves) les résultats d'outils existants, même à l'état de prototypes. Pour que l'instrumentation soit applicable en pratique, nous avons cherché à limiter le nombre de fonctions à instrumenter. Nous avons dégagé un ensemble restreint de fonctions, commun à tout analyseur, à instrumenter pour qu'un analyseur existant devienne un outil de certification automatique.

Nous avons appliqué cette technique d'instrumentation à un analyseur de programmes manipulant des tableaux et s'appuyant sur des domaines abstraits non triviaux. Cet analyseur calcule des propriétés sur le contenu des tableaux manipulés par les programmes et grâce à notre instrumentation il génère désormais des certificats vérifiables par Coq attestant de la validité des propriétés découvertes par l'analyse statique. Nous montrons enfin comment utiliser cet analyseur instrumenté pour certifier un protocole de communication pour systèmes multi-tâches destiné à l'avionique. La garantie de la correction des programmes est cruciale dans le domaine des systèmes embarqués. Face aux coûts et la durée d'une procédure de certification formelle, le développement d'outils automatiques de certification représente un enjeu économique majeur. La transformation, par instrumentation, d'outils d'analyses existants en outils de certification est une réponse possible qui évite la certification des outils d'analyse.

Mots importants : Certification, preuve déductive, Coq, interprétation abstraite, sémantique des programmes, instrumentation, arithmétique linéaire, système réactif, application embarquée.

Remerciements

Je tiens tout d'abord à remercier l'ensemble des membres du jury pour avoir accepté de juger ces travaux et me permettre de tourner la page sur cette première expérience de recherche. Je remercie particulièrement Olga Kouchnarenko, Thomas Jensen et Solange Coupet-Grimal pour avoir relu minutieusement ce manuscrit et m'apporter leurs remarques qui m'ont permis d'en améliorer considérablement la qualité. Je remercie également Laurence Pierre pour avoir accepté de présider cette soutenance ainsi que Gilles Barthe pour s'être porté examinateur.

Je remercie mon directeur de thèse, Michaël Périn, pour m'avoir formé pendant trois années de Magistère avant de m'encadrer dans ces travaux de doctorat. Sans sa disponibilité, son écoute permanente et la liberté d'action qu'il m'a laissée, ces travaux n'auraient sans doute jamais vu le jour.

Je remercie Yassine Lakhnech pour m'avoir intégré à l'équipe DCS qui offre un cadre de travail confortable, agréable et sérieux. Une pensée va également à tous les membres du laboratoire Verimag et notamment à Jean-Claude Fernandez et Paul Caspi qui m'ont sans doute, dans une décénie antérieure, donné le goût de la science.

La bonne humeur qui a régnée au CTL ces quatre dernières années m'a permis de venir tous les jours travailler avec le sourire et je tiens, pour cela, à remercier : Mathias, Jacques, Simon, Loic, Hugo, Guillaume, FX, Ylies, Yassin, Marc, Sophie, Mohamad, Hung, Ananda, Polyvios, Marius, Imen, Vasso, Manos, Paris, Tesnim, Radu, Pierre, Cristian, Phillipe, Pascal, et tous ceux que j'ai pu oublier. Merci à vous, que de belles rencontres.

Je crois au déterminisme et je pense qu'un être devient ce qu'il est en grande partie grâce à son environnement initial et pour cela, je tiens à remercier mes parents : Fatima et Antoine. Ma soeur Maïa qui m'a tant influencé. Mes amis de toujours pour leur fidélité : Nacim, Kacem, Abdallah, Mehdi, Sami. Mon cousin Pierre à qui, malgré tout, je dois énormément. Isiah qui égaie mes journées depuis douze ans à chaque fois que l'on se voit. Et tous les autres membres de ma famille, enfants et adultes, que j'ai vu ou qui m'ont vu grandir et évoluer.

Il paraît qu'il y a dans certaines rencontres des phénomènes prodigieux sur la trajectoire qu'une vie peut prendre. J'en ai fait l'expérience et je tiens notamment à remercier Norbert et Jérémie qui me gratifient depuis dix ans de leur amitié, leur confiance et leur patience.

Une pensée va également à tou(te)s mes ami(e)s. Tou(te)s vous citer serait trop long mais sachez qu'une pensée vous est personnellement adressée, à chacun(e).

Je remercie mon épouse Myriam qui m'a permis d'être ce que j'aspirais à devenir et sans qui (grâce à son soutien et sa présence de tous les instants) ce travail n'aurait pas été le même.

Enfin, en reprenant les mots de Sylvie Tissot et Pierre Tevanian, je dédie ce travail à toutes celles et ceux qui s'engagent en pensée, en paroles et en actes, et s'efforcent jusque dans leur vie professionnelle et affective de faire vivre un minimum les mots émancipation, égalité et amitié.

Table des matières

1 Introduction **1**
1.1 Naissance de la logique mathématique . . . 1
1.2 L'informatique au service de la démonstration . . . 4
1.3 Raisonner mathématiquement sur les programmes . . . 5
1.4 Quand les programmes vérifient les programmes . . . 6
1.5 Les preuves formelles pour convaincre . . . 7
1.6 Motivations et Contributions . . . 8
1.7 Notions essentielles . . . 9
1.7.1 Langages de programmation . . . 9
1.7.1.1 Syntaxe du langage de programmation considéré . . . 10
1.7.1.2 Sémantique du langage de programmation considéré . . . 11
1.7.2 Programmes et systèmes de transitions . . . 11
1.7.3 Logique du premier ordre . . . 13
1.7.4 Systèmes formels . . . 15
1.7.4.1 La déduction naturelle NJ . . . 15
1.7.4.2 Théorie de l'Arithmétique de Presburger . . . 17
1.7.5 Sémantique axiomatique . . . 18
1.7.5.1 Logique de Hoare . . . 19
1.7.5.2 Semantique de transformation des prédicats . . . 21
1.8 Plan du manuscrit de thèse . . . 22

2 Autour de la certification formelle de programmes **25**
2.1 Calcul des Constructions Inductives . . . 25
2.1.1 Présentation du système Coq . . . 26
2.1.1.1 Modélisation de la logique propositionnelle . . . 26
2.1.1.2 Modélisation de la sémantique axiomatique . . . 29
2.1.1.3 Preuves de corrections. . . . 32
2.1.1.4 Calcul de plus faible pré-condition . . . 34
2.1.1.5 Conclusion . . . 35
2.1.2 Raisonner sur les programmes impératifs dans Coq . . . 35
2.1.2.1 Modélisation des tableaux dans le système Coq . . . 35
2.1.2.2 Calcul de plus pré-condition correct et complet pour les affectations de tableaux . . . 37
2.1.2.3 Données mutables et logique de séparation . . . 42
2.1.3 Conclusion . . . 43
2.2 Approche pour la certification de programmes impératifs . . . 43

2.2.1 Traduction des programmes vers le langage Why 44
2.2.2 Obligations de preuve et correction des programmes 44
2.2.3 Conclusion sur l'approche du système Why 45
2.3 Certification de méta-programmes 45
2.3.1 Certification de compilateurs 46
2.3.2 Certification d'interpreteurs abstraits 47
2.3.3 Utilisation du système Coq 48
2.3.4 Conclusion sur la certification des méta-programmes 49
2.4 Certification des résultats des méta-programmes 50
2.4.1 Principe des méta-programmes certifiant 50
2.4.2 Méta-programmes certifiants pour le Proof-Carrying Code 51
2.4.3 Génération de preuves guidée par analyse statique 52
2.4.4 Conclusion sur les méta-programmes certifiants 53
2.5 Discussion et direction choisie 54

3 Certification automatique de programmes par instrumentation de méta-programmes 57
3.1 Principe de l'instrumentation de méta-programmes 59
3.1.1 Passage de la validation à la certification grâce à l'instrumentation 59
3.1.2 Instrumenter pour justifier 60
3.1.2.1 Correction et justification 62
3.1.2.2 Justification de fonctions 63
3.1.2.3 Justifications résultats d'un méta-programme 64
3.1.3 Construction et vérification des preuves dans le système Coq . . . 66
3.1.4 Conclusion 69
3.2 Instrumentation d'analyses statiques 69
3.2.1 Analyse statique par interprétation abstraite 71
3.2.1.1 Base de l'interprétation abstraite 71
3.2.1.2 Abstraction correcte de sémantiques 73
3.2.1.3 Illustration : Le domaine des intervalles 74
3.2.1.4 Spécification de sémantiques par calcul de points fixes . . 75
3.2.1.5 Partitionnement de sémantiques pour les systèmes de transitions 76
3.2.1.6 Calcul de (post-)points fixes abstraits 77
3.2.2 Concrétisation vers la logique mathématique 80
3.2.2.1 Prouver les propriétés sémantiques dans le domaine concret 81
3.2.2.2 Domaines abstraits considérés 82
3.2.3 Justification d'une analyse statique 82
3.2.3.1 Stratégie de preuve pour justifier une analyse statique . . 83
3.2.3.2 Justification de la fonction sémantique 85
3.2.4 Complétude de l'instrumentation avec patrons de preuve 86
3.2.4.1 Justifications de l'analyse des affectations 87
3.2.4.2 Justifications de l'analyse des gardes 93
3.3 Conclusion 95

4 Étude de cas : Instrumentation de l'analyseur statique Enkidu 97
4.1 Domaines abstraits de l'analyseur Enkidu . . . 99
4.1.1 Le domaine abstrait des zones . . . 99
4.1.1.1 Système de contraintes de zone . . . 100
4.1.1.2 Implantation des contraintes de zone par des matrices . . . 101
4.1.1.3 Normalisation des DBMs . . . 102
4.1.1.4 Instrumentation de la normalisation des DBMs . . . 103
4.1.1.5 Opérateurs du domaine des zones . . . 104
4.1.1.6 Instrumentation des opérateurs du domaine des zones . . . 104
4.1.1.7 Conclusion sur le domaine abstrait des zones . . . 107
4.1.2 Domaine abstrait du contenu des tableaux $\mathscr{A}$ . . . 107
4.1.2.1 Instances du domaine des zones utilisées par $\mathscr{A}(P)$ . . . 110
4.1.2.2 Opérateurs du domaine abstrait des propriétés de tableaux 113
4.1.2.3 Instrumentation du test d'inclusion dans $\mathscr{A}(P)$ . . . 114
4.2 Modification des indices de tableaux . . . 115
4.2.1 Gestion des décalages d'indices dans l'analyse et sa justification . . . 115
4.2.1.1 Problématique pour générer la preuve de correction . . . 116
4.2.1.2 Solutions pour justifier l'analyse de la modification des indices . . . 117
4.2.2 Justification de l'analyse de la modification des indices par une stratégie *ad hoc* . . . 118
4.2.2.1 Algorithme du calcul de meilleure information . . . 118
4.2.2.2 Principe de l'instrumentation du calcul de meilleure information . . . 119
4.2.2.3 Details de l'instrumentation du calcul de meilleure information . . . 121
4.2.3 Justification de l'analyse de l'affectation des indices par modification du calcul de plus faible pré-condition . . . 124
4.2.3.1 Modification du calcul de plus faible pré-condition . . . 126
4.2.3.2 Test d'inclusion générique de $\mathscr{A}$ . . . 127
4.2.4 Conclusion sur la justification de l'analyse des affectations d'indices de tableaux . . . 128
4.3 Modification des tableaux . . . 130
4.3.1 Interprétation abstraite de l'affectation d'un tableau . . . 130
4.3.1.1 Traduction de l'affectation vers le domaine abstrait des propriétés de tableaux. . . . 131
4.3.1.2 Fonction de transfert de l'affectation de tableau . . . 131
4.3.2 Justification de l'analyse de l'affectation de tableau . . . 133
4.3.2.1 Principe de la justification . . . 133
4.3.2.2 Détails de la justification . . . 135
4.3.3 Conclusion sur la justification de l'analyse des affectations de tableaux . . . 140
4.4 Transitions gardées . . . 141
4.4.1 Analyse des transitions gardées . . . 142
4.4.1.1 Traduction de la garde vers le domaine asbtrait des propriétés de tableaux . . . 142
4.4.1.2 Fonction de transfert des transitions gardées . . . 143

4.4.2 Justification de l'analyse des gardes . . . 144
4.4.2.1 Justification de la traduction des gardes . . . 144
4.4.2.2 Justification de la fonction de transfert des gardes . . . 145
4.4.3 Conclusion sur la justification de l'analyse des gardes . . . 146
4.5 Développement de l'instrumentation d'Enkidu . . . 147
4.5.1 Implantation . . . 147
4.5.2 Expérimentation . . . 148
4.6 Conclusion . . . 149

5 Vers la certification d'un protocole de communication pour systèmes multi-tâches 151
5.1 Contexte et motivations . . . 151
5.2 Modélisation du système réactif multi-tâches . . . 153
5.2.1 Modèles du système multi-tâches avec évènements . . . 153
5.2.1.1 Exécution d'une tâche . . . 154
5.2.1.2 Traces d'exécution . . . 154
5.2.1.3 Modèle idéal . . . 155
5.2.1.4 Modèle réaliste avec contraintes d'implantation . . . 156
5.2.2 Communications inter-tâches avec priorités statiques . . . 157
5.2.2.1 Priorités et interruptions . . . 157
5.2.2.2 Communications entre les tâches . . . 159
5.2.2.3 Restriction des délais de communication . . . 160
5.3 Préservation de la sémantique grâce au DBP protocole . . . 161
5.3.1 Problème avec une implantation "simple" . . . 161
5.3.2 Principe du DBP protocole . . . 163
5.3.2.1 Communication dans le sens *high to low* . . . 164
5.3.2.2 Communication dans le sens *low to high* . . . 165
5.4 Preuve de correction du DBP protocole . . . 169
5.4.1 Simplification du système pour raisonner sur deux tâches . . . 170
5.4.1.1 Code du protocole simplifié . . . 171
5.4.1.2 Propriété de correction dans le cas à deux tâches . . . 173
5.4.2 Calcul de la sémantique du protocole à deux tâches . . . 173
5.4.2.1 Calcul des séquences d'exécution possibles . . . 174
5.4.2.2 Interprétation abstraite du code protocole . . . 174
5.4.3 Certification de la correction du protocole . . . 176
5.5 Conclusion . . . 177

6 Conclusions et Perspectives 179
6.1 Bilan sur les contributions de nos travaux . . . 179
6.2 Perspectives . . . 182

Bibliographie 185

J'ai longtemps pensé et affirmé que la notion de vérité était relative et qu'elle pouvait varier d'un individu à l'autre : tout n'est alors qu'interprétation et il existe potentiellement autant de vérités que d'individus. Les êtres humains, par leur faculté à penser, déterminent le vrai en fonction de certains précèptes acquis malgré eux.

Je doute aujourd'hui de cette vision de la vérité. Le fait de pratiquer pendant plus de trois ans une activité intellectuelle intense sur des questions relativement abstraites, permet de démystifier un nombre conséquent d'idées reçues, ce qui induit une ouverture d'esprit certaine. Il me paraît tout à fait clair aujourd'hui que le monde et les choses qui le composent contiennent des vérités objectives, indépendantes de qui les observe. Il existe des faits réels, certains m'échappent pour l'instant, d'autres ne me seront jamais accessibles. Et si parfois mon interprétation m'ammène à contredire l'un de ces faits, il n'en demeure pas moins un fait, une vérité, mais qui seulement n'est pas à la portée de mon imagination. Essayer de découvrir et comprendre ces faits me semble être une noble entreprise.

Chapitre 1

Introduction

> « *Ce que comprenaient bien les penseurs des Lumières, mais qui a été en partie oublié depuis lors, c'est que l'approche scientifique (en y incluant la connaissance ordinaire) nous donne les seules connaissances objectives auxquelles l'être humain a réellement accès. Si l'approche scientifique nous donne une vision partielle de la réalité, c'est parce que nous n'avons pas accès, de par notre nature finie, à la réalité ultime des choses. Mais il y a une grande différence entre dire que la science nous donne une description complète de la réalité et dire qu'elle en donne la seule connaissance accessible à l'être humain.* »
>
> Jean Bricmont

Avant toutes choses, nous tenons à préciser que par *certification formelle* nous entendons *preuves déductives en logique mathématique de la correction* de programmes informatiques. Ces programmes informatiques sont écrits dans des langages dits de *hauts niveaux*, dans un style *impératif*, et servent à implanter les systèmes critiques que nous considérerons. Ensuite, par *automatique* nous entendons que ces preuves doivent être aussi bien *construites* que *vérifiées par machine*, au moyen de l'exécution d'autres programmes informatiques par exemple. Enfin, la thèse ici présentée est de nature scientifique et tente d'apporter des réponses à des problèmes de cette même nature. Néanmoins, une partie de sa motivation provient d'un enjeux sociétal : les programmes informatiques sont, à l'heure d'aujourd'hui, présents dans presque tous les objets qui entourent et facilitent notre quotidien, garantir la fiabilité de leurs comportements est essentiel, et particulièrement lorsque des vies en dépendent.

Ces précisions étant faites, nous revenons brièvement sur la naissance de la logique mathématique puis sur celle de l'informatique et l'évolution de leurs utilisations, ainsi qu'une partie des problèmes que ces disciplines ont posées et rencontrées à travers le vingtième sciècle.

1.1 Naissance de la logique mathématique

Voici un siècle que David Hilbert établit sa liste de problèmes veillant à encadrer et orienter les recherches dans le domaine des mathématiques [Bro76, Det86]. Trois

décénies plutôt, Friedrich Ludwig Gottlob Frege proposait le calcul propositionnel puis le calcul des prédicats du premier ordre [Fre84, Fre60] (aussi appelé logique du premier ordre), formalisant ainsi pour la première fois la logique dans une syntaxe qui lui était propre. Véritable fondateur de la logique moderne, il apporta d'importants élements de réponse aux deux questions majeures pour de nombreux philosophes et mathématiciens de l'époque : « *Qu'est ce qu'un raisonnement logique ? Comment caractériser mathématiquement la notion de vérité ?* » En effet, l'accroissement de la complexité des théories mathématiques développées leur imposa de s'interroger sur la notion de *démonstration valide*. Ainsi naquirent plusieurs langages exprimant concepts, relations et propositions logiques, accompagnés de régles permettant de déduire la validité des uns à partir de celle des autres. De cela émana le *logicisme*, théorie considérant que l'ensemble des mathématiques est réductible à la logique formelle.

En 1902, Bertrand Russell adressa une lettre à Gottlob Frege qui contenait une démonstration de l'incohérence du système de déduction de ce dernier, qu'il publia un an plus tard dans son ouvrage *The Principles of Mathematics* [Rus03]. Cependant, si Russell réfuta la logique de Fredge en y trouvant un paradoxe, il demeura fortement inspiré et influencé par ses travaux, et dirigea les siens dans le même sens. Quelques années plus tard, en collaboration avec Alfred North Whitehead, il inventa la *théorie des types* [Rus08], résolvant les paradoxes des sytèmes de déduction susmentionnés. Ces travaux amenant les premiers systèmes logiques corrects furent publiés dans leur célèbre ouvrage, *Principia Mathematica* [RW13]. Les mathématiques modernes étaient nées et les théories se fondaient dès lors sur une hiérarchie de propositions valides (des théorèmes), dont les premières sont nommées les *axiomes*. Les axiomes d'une théorie mathématique fondent la base de déduction de tous les théorèmes de cette théorie : un axiome est une vérité première qui ne doit pas être démontrable à partir des autres axiomes de la théorie (dans le cas contraire il serait relégué au rang de théorème).

David Hilbert ne fut pas en reste et apporta sa contribution en développant son système logique éponyme, puis présenta en 1920 un programme de recherche dans la continuité des vingt-trois problèmes énoncés vingt ans auparavant. Ce programme ayant pour but d'assurer les fondements des mathématiques et de prouver leur cohérence correspond à la vision *formaliste* des mathématiques : « *Les mathématiques sont un pur jeu de symboles, l'essentiel est la cohérence formelle des règles de ce jeu* » . Le deuxième problème du programme de Hilbert concernait la cohérence de l'arithmétique élémentaire, et posait la question suivante : « *peut-on prouver dans l'arithmétique elle-même la cohérence de l'arithmétique ?* ». Hilbert cherchait donc à savoir si l'arithmétique admettait un autre *point aveugle* [Gir06] que ses axiomes : « *en supposant que l'arithmétique soit une théorie cohérente, peut-elle le voir d'elle-même ?* » .

Parallèlement Luitzen Egbertus Jan Brouwer fonda *l'intuitionnisme* [Tro69, Bro81] et y voyait une réponse alternative à la crise des fondements des mathématiques. La logique intuitionniste considère que les mathématiques sont intuitives et ne peuvent être purement hypothético-déductives, s'opposant ainsi au logicisme de Russell et Frege, au formalisme de Hilbert (avec qui s'ensuivirent de violents affrontements). Rejetant une *définition simpliste* de la vérité tendant à se limiter à la dichotomie *vrai/faux*, les intuitionnistes considèrent *l'inconnue* comme valeur tiers aux propositions logiques prenant sens dans des domaines infinis. Brouwer accorda la plus grande importance aux moyens d'atteindre les choses et estima que leur vérité en dépendait. Les mathématiques constructives, consistant à n'accepter que les démonstrations qui permettent de construire les

objets dont elles prouvent l'existence, etaient nées.

Deux branches s'opposèrent [Pra77b], jouant à jeu non égal : Hilbert imposa *le formalisme* dans la communauté comme *paradigme de référence* tandis que Brouwer sombra dans l'oubli, évincé des institutions prestigieuses de l'époque. En 1930 Kurt Gödel énonca et démontra un *théorème de complétude* du calcul des prédicats du premier ordre [Göd80], enfonçant un peu plus Brouwer dans le fossé qu'Hilbert lui avait creusé. Ce théorème établit que toute proposition vraie du premier ordre est démontrable *i.e.* il existe un systéme de déduction tel que toutes les propositions satisfaites dans tous les modèles sont démontrables dans ce système.

Cependant, afin de ne décevoir personne, ce même Kurt Gödel énonça un an plus tard deux *théorèmes d'incomplétude* de la logique mathématique [Göd80]. Le premier affirmait que dans une théorie respectant certaines conditions (formaliser l'arithmétique élémentaire notamment), il existe des propositions sur lesquelles on ne pourra jamais rien dire, *i.e.* ces propositions ne sont ni démontrables ni réfutables. Le second théorème, sous les mêmes hypothèses que le premier, affirme qu'il existe un énoncé exprimant la cohérence de la théorie considérée sans que celui-ci puisse être démontrée dans la théorie elle même ; Pour prouver la cohérence d'une théorie, il faut en sortir ; il s'ensuit une course sans fin pour montrer la cohérence d'une théorie en faisant pas à pas appel à une théorie plus expressive... Cela résolut le *deuxième des problèmes de Hilbert* mais malheureusement pas dans le sens souhaité. Pour ne retenir que deux choses, Gödel avait montré qu'il existe des choses vraies qu'on ne puisse pas démontrer – un peu comme un avocat convaincu de la culpabilité d'un citoyen mais dans l'incapacité d'en convaincre les jurés par manque d'arguments matériels – et parmi ces choses non démontrables se trouvait la cohérence des axiomes d'une certaine théorie : la cohérence s'avère donc être un point aveugle de toute théorie exprimant au moins l'arithmétique élémentaire (ou de Peano [Pea89] plus précisément) et cela aurait dû mettre définitivement fin à l'approche formaliste.

Enfin, ce fut sans compter sur Gerhard Gentzen qui en 1936 apporta une démonstration de la cohérence de l'arithmétique en dépit des théorèmes de Gödel (au moyen d'une astuce) [Gen69]. Le côté intéressant de la chose est que pour produire cette *pseudo preuve*, il eut besoin d'outils. Ces outils sont des systèmes de déduction pour le calcul des prédicats ayant des propriétés structurelles bien différentes de ceux à la Hilbert [Gen69, Pra65]. En effet ces derniers ont beaucoup d'axiomes, peu de règles de déduction et il en résulte une quasi impossibilité de faire de la déduction automatique. Les systèmes de Gentzen, nommés la *déduction naturelle* et le *calcul des séquents*, sont tout le contraire (très peu d'axiomes, un couple de règles pour chaque connecteur logique) et aujourd'hui reconnus comme faisant partie des créations mathématiques les plus conséquentes du siècle dernier. En s'appuyant sur le calcul des séquents, Gentzen énonça et démontra un *théorème d'élimination des coupures*, affirmant que toute démonstration admet une forme normale sans détours *i.e.* aucun concept qui ne soit pas contenu dans son résultat final n'est employé pour obtenir ce résultat.

Dans le même temps, dans la lignée des travaux de Gödel, Alonzo Church et Alan Mathison Turing qui focalisaient leurs recherches sur la calculabilité et la conception d'un appareil mécanique de calcul, démontrèrent séparément que les logiques frappées de plein fouet par les théorèmes d'incomplétudes étaient algorithmiquement indécidables [Tur36], *i.e* il n'existe pas de processus systématique calculatoire permettant de décider si cette proposition est un théorème (elle admet une demonstration) ou non.

Les années cinquante, sous l'impulsion des modèles de calcul de Church et Turing (respectivement le λ-calcul et les *machines de Turing*) ont vu l'avènement de la programmation et de l'informatique comme *science du calculable* et concernant leurs acteurs on parle dorénavant d'informaticiens. Alan Turing, outre ses résultats fondateurs de la science de l'informatique, démontra de plus que le simple calcul des prédicats etait quant à lui semi-décidable. L'idée d'utiliser la programmation afin de faire de la *démonstration automatique de théorèmes* qui était dans l'air du temps, aurait une nouvelle fois dû être blessée d'une langueur définitive. Mais l'obstination est sans doute l'une des caractéristiques les plus communes des scientifiques : sans tenir compte des résultats démoralisants de Turing, des informaticiens développèrent malgré tout des méthodes efficaces *en pratique* pour calculer des démonstrations de théorèmes dans la logique du premier ordre.

1.2 L'informatique au service de la démonstration

La plus connue des méthodes de démonstration automatique est sans doute *la résolution* [Rob65]. Elle fut proposée par John Alan Robinson en 1965 et son succès fut tel que la *programmation logique* en découla, puis se concrétisa via le langage Prolog [Col89]. Par une réduction du problème sous forme clausale, elle démontre que la négation de la proposition intiale conduit à l'absurde, puis par *modus tollens* la proposition initiale est ainsi validée. Ces techniques permettent d'automatiser le raisonnement et de démontrer des théorèmes complexes, ou de tailles importantes et cela présente un interêt majeur. Mais, ratrappées par les résultats d'indécidabilité de Turing, ces méthodes s'essoufflèrent, souffrant de la pauvreté du langage des propositions. En effet, elles rencontrent de grandes difficultés face à la transitivité de l'égalité notamment. Pour palier ces problèmes, des règles de déduction spécifique furent proposées telle la *paramodulation* ou d'autres *théories équationelles* [RV01]. Néanmoins, l'autre tare de ces méthodes est qu'elles dénaturent totalement le problème pour le résoudre, et il devient compliqué de *(se) convaincre* que la réponse obtenue est bien celle initialement désirée. À mon sens, le sens et le but premier des mathématiques (et de la science en générale), outre de faire lumière sur les faits qui constituent la réalité objective, est de fournir des méthodes produisant des *arguments convaincants* pour approcher et sentir des vérités relatives (aux croyances de chacun). Les méthodes formelles ont pour utilité d'*aider à convaincre* de la véracité de certains faits, mais elles ne sont pas *intrinséquement convaincantes*. Ainsi, si ces méthodes de démonstration automatique sont efficaces mais peu convaincantes, quelle valeur ont-elles ? Ce questionnement a aiguillé continuellement l'avancement de la thèse ici présentée.

Au delà de la démonstration de théorèmes, l'informatique amorça la troisième révolution industrielle et la programmation devint un outil fondamental dans presque tous les secteurs économiques. En trente ans, l'informatique s'est implantée partout (electroménager, avionique, automobile, aéro-spatial, ...), et les sociétés occidentales sont devenues dépendantes du bon fonctionnement des programmes. Cependant nous constatons chaque jour des dysfonctionnements informatiques dans notre quotidien (guichet automatique en panne, serveur internet indisponible, ordinateur personnel qui plante, etc ...). Tant que cela touche à nos activités les plus banales les conséquences sont moindres et les dysfonctionnements demeurent sans gravité et nous pouvons ainsi facilement en faire abs-

traction. Toutefois, il est possible de distinguer parmi les systèmes informatiques, ceux qui dans leurs dysfonctionnements entraînent des pertes financières considèrables (fusée qui explose en plein ciel) ou pire, des morts humaines (avion qui s'écrase). Ces sytémes sont dits *critiques* et garantir leur bon fonctionnement est la motivation principale de ces travaux de thèse.

1.3 Raisonner mathématiquement sur les programmes

L'informatique, qui fut inventée par des mathématiciens[1], avait pour but premier de servir ses créateurs dans leurs recherches fondamentales (le vrai et le faux), mais victime de son succés, vit la quantité de programmes développés et la diversité de ses domaines d'application exploser. Ainsi, la programmation qui à la base est une science exacte et consiste à la manipulation formelle de symboles (et en ce sens n'est pas si éloignée des mathématiques) muta vers une activité expérimentale – voire artisanale – dont la fiabilité des objets générés devint plus que médiocre. Il en découle qu'une grande partie du temps necessaire au développement d'un programme est consacrée à la correction de ses erreurs. Ces corrections sont malheureusement souvent effectuées par un jeu *d'essai/erreur* et reposent plus sur des tests que de véritables raisonnements. Cela est évidement insuffisant lorsque l'on sait que le domaine des variables d'un programme est la plus part du temps infini. « *Tester un programme peut démontrer la présence de bugs, jamais leur absence* » , pour reprendre l'aphorisme d'Edsger Wybe Dijkstra.

En réponse à cela, Robert Floyd [Flo67], Charles Antony Richard Hoare [Hoa69], Edsger Wybe Dijkstra [Dij75], et d'autres, au cours des années soixante-dix ont proposé un cadre formel permettant la construction de programmes corrects (*i.e* qui respectent les comportements que l'on attend d'eux), et ont fourni un ensemble de méthodologies permettant de s'en convaincre. Ces travaux s'appuient sur ce que l'on appelle, *la sémantique* des programmes, et *la sémantique* des langages de programmation. La sémantique d'un langage de programmation est une définition mathématique de l'interpretation que l'on doit se faire de chacune des instructions qu'offre le langage [Ten91]. La sémantique d'un programme est quant à elle une formalisation mathématique des comportements de ce programme. Si la sémantique d'un langage de programmation permet d'exécuter un programme écrit dans ce langage, ou de traduire ce programme dans un autre langage (lorsqu'on compile le programme par exemple), elle ne permet pas en revanche de calculer, dans le cas général, la sémantique de tout programme écrit dans ce langage. Néanmoins, afin d'établir qu'un programme respecte les comportements que l'on attend de lui, les methodologies développées à cette époque utilisent la sémantique du langage de programmation employé afin de construire (manuellement) des arguments rigoureux, *les preuves de programmes*.

Dans une autre direction, Patrick et Radhia Cousot définissaient en 1977 la théorie de *l'interprétation abstraite* [CC77] dont l'utilité principale réside dans *l'analyse statique et automatique* de programmes [CC76]. Analyser un programme en calculant sa sémantique sans avoir à les exécuter – afin de vérifier qu'elle est conforme à celle que l'on attend ou juste pour la découvrir – est une tâche qui n'est pas, de manière générale, automatisable (notamment du fait que la sémantique doit capturer l'existence potentielle de séquences d'exécution infinies). Cette théorie formalise l'idée que la sémantique des programmes

[1] Des logiciens plus précisément.

a une précision relative au degrés d'abstraction avec lequel on les observe. Ainsi, l'interprétation abstraite permet d'approximer les comportements des programmes afin de rendre leur sémantique calculable. Une analyse statique utilisation l'interprétation abstraite a de ce point de vue l'avantage d'être totalement automatique et permet donc de traiter des programmes de grandes tailles. Cependant, le *théorème de Rice* [Ric53] assure qu'aucun programme n'est capable de décider si une propriété *non triviale* est satisfaite (ou non) par tous les programmes d'un langages de programmation atteignant une certaine expressivité (celle des machines de Turing en l'occurrence). En d'autres termes, ce résultat fondamental de calculabilité exprime que développer un outil automatique capable de garantir la correction de n'importe quel programme est impossible, ou ne calculera rien de bien intéressant.

Pour accroître l'intérêt des sémantiques calculées, des approches dites *intéractives* (dans le sens où l'outil sollicite des interventions humaines durant son calcul) ont été proposées, telle la *preuve assistée par ordinateur*, et ont permis de combler ce faussé d'indécidabilité. L'inconvénient de ces approches est que par leur coté non automatique, il est beaucoup moins réaliste de traiter des programmes aussi grands et complexes que l'analyse statique sait le faire.

Face à l'augmentation de la taille et la complexité des systèmes informatiques, accompagnée d'une baisse proportionelle de leur fiabilité, les recherches se sont axées autour de la *vérification*, la plus automatisée possible, de ces systèmes, quitte à ne vérifier que partiellement leur correction. De nombreuses approches, mélangeant les théories énoncées plus haut et d'autres, furent développées dans cette perspective de vérifier formellement les programmes après leur conception. Ces approches, dont les théories sur lesquelles elles reposent sont validées, offrent la possibilité de (se) persuader mathématiquement de la correction et la fiabilité d'un programme, au moyen d'algorithmes complexes. Dès années quatre-vingt à nos jours, poussées par des techniques telles le *model-cheking* [SQ82] notamment, elles connaissent un grand succés dans la communauté des méthodes formelles au détriment des approches orientées *preuves de programmes* [Pau87], qui elles manquent d'efficacité. La preuve de programme est pourtant la jonction parfaite entre l'informatique et les mathématiques : en 1969 William Alvin Howard remarqua que les démonstrations en déduction naturelle intuitionniste pouvaient être vues comme des objets du λ-calcul de Church [Bar81]. Cette correspondance *preuve/programme* fut déjà observée onze années auparavant par Haskell Brooks Curry entre les systèmes à la Hilbert et un modèle de calcul inventé par lui même (*la logique combinatoire*) [Cur63]. Cette correspondance, fut nommée l'*isomorphisme de Curry-Howard* [GLT89] et joua un rôle fondamentale dans l'évolution de la logique mathématique et de la théorie de la démonstration.

1.4 Quand les programmes vérifient les programmes

Les domaines d'utilisation de la programmation étant diverses et variés, les méthodologies de *vérification formelles de programmes* furent proposées de manière très spécifique aux problèmes et aux applications visés : systèmes distribués, systèmes réactifs, protocoles cryptographiques, sécurité des systèmes informatiques, programmes synchrones embarqués, etc... Afin de pouvoir exploiter ces méthodologies sur des applications concrètes, elles furent à leur tour implantées et un grand nombre d'*outils de vérification* fut pro-

grammé : astrée [CCF+05], spin [Hol05], blast [HJMS03, BHJM07], cadp [FGK+96], hermes [BLP03], lesar [Ray08], *etc.* Ces outils de vérification appartiennent à la classe des *méta-programmes* (des programmes qui prennent d'autres programmes en entrées).

Outre le fait que les théories en amont soient valides, le passage à l'implantation est intrinsèquement une source d'erreur. Lorsqu'un outil de vérification se contente de répondre « *OK* » , quel crédit peut-on lui accorder bien que l'on soit convaincu par la théorie sur laquelle il repose ? Cette question fut, et demeure souvent ignorée dans la communauté de la vérification formelle de programmes, négligeant les difficultés qu'entraine le passage à l'implantation. La confrontation d'idées conceptuelles à l'effectivité des possibilités qu'offre le monde réel amène bien souvent de grandes désillusions. Comment attribuer une quelconque légitimité dans un verdict, censé garantir la fiabilité d'un programme d'une centaine de lignes de codes, provenant d'un programme complexe de plusieurs milliers de lignes de code ?

Développer des *super-vérificateurs* (des méta-programmes qui traitent des méta-programmes ...) permettant de vérifier et garantir la fiabilité des vérificateurs de programmes n'est pas sans rappeler les soucis posés par le second théorème d'incomplétude de Godël, et la course sans fin qui en découle. Qui garantira la fiabilité des super-vérificateurs ?

Il faut donc créer une rupture dans la démarche existante afin de ne pas courir infiniment après la fiabilité des vérificateurs. C'est là que les approches orientées preuves formelles de programme retrouvent un certain intérêt, à travers la *certification formelle de programme*. Certifier – fournir une preuve logique de la correction – des outils de vérification est une réponse possible [Mon98, Pic05, CJPR05, Ler06, Ler09, CP10], cette thèse a pour but d'en proposer une autre, moins coûteuse, plus facile à mettre en pratique.

1.5 Les preuves formelles pour convaincre

La construction de preuves entièrement formelles est une activité rébarbative et difficile. Si les preuves formelles présentent une solution pertinente au problème de fiabilité des programmes informatiques, la question qu'il convient de se poser désormais est « *quels programmes certifier ?* » Faut-il prouver directement la correction des programmes visés ou la correction des vérificateurs de programmes afin de garantir leurs verdicts ?

Prouver la correction de chaque programme visé implique qu'il faut construire autant de preuves que de programmes, et nous savons que construire de telles preuves est un exercice compliqué. De l'autre côté, prouver la correction des vérificateurs de programmes factorise le nombre de preuves à construire – la preuve de correction d'un vérificateur permet de certifier toute une classe de programmes – mais ces vérificateurs sont d'une grande complexité algorithmique et de taille importante et prouver leur correction est d'autant plus difficile.

Dans un cas comme dans l'autre des questions doivent être abordées et éclaircies :

- Comment construire de telles preuves ? car prouver la correction d'un programme de taille importante implique que la preuve sera sûrement de taille importante.
- Comment vérifier la correction de telles preuves ? car vérifier manuellement une preuve de plusieurs milliers de lignes afin de s'assurer de sa correction paraît peu raisonnable.

– En supposant que la preuve puisse être vérifiée automatiquement, quel crédit accorder aux verdicts de ce vérificateur de preuves ? Car ce vérificateur, s'il se veut automatique sera aussi modélisé par un programme.

Construire des preuves de programmes peut se faire selon deux grandes approches : celle dite automatique (dans le même esprit que les démonstrateurs automatiques de théorèmes mathématiques) et celle dite *intéractive*.

Les outils de preuve interactifs sont accompagnés de vérificateurs automatiques, propre à la logique qu'ils implantent, et répondent par « *oui ou non* » si les preuves construites à l'aide de l'outils sont respectivement valides ou pas. Cela ne semble pas éloigné des outils de vérification dont on doute de la fiabilité de leurs jugements, à la différence que ces vérificateurs de preuves sont de taille modeste. En adoptant une *philosophie sceptique* de l'outillage informatique, il est tout de même possible de définir une *hiérarchie de confiance* entre les outils de vérification ou de certification, en fonction de la taille de leur code dans lequel nous n'avons pas d'autre choix que d'avoir une foi aveugle. Ce minimum de code ni vérifié ni certifié est nommé (dans la communauté de la certification formelle de programme) le TCB (*trusted code base*) et plus il est réduit et simple, plus l'outil concerné est considéré comme fiable. La fiabilité des outils de vérification et de certification n'est donc pas totale, mais relative à un critère (dit de *de Bruijn*) subjectif dont la pertinence ne sera pas discutée dans la suite. Chacun des systèmes de preuves intéractifs HOL [HOL], Isabelle [ISA] et Coq [BC04] a un « *noyau* » de taille raisonable conscaré à la verification des preuves et en ce sens satisfait ce critère. Ces systèmes permettent ainsi de valider des preuves de grandes tailles et manipulant des concepts non triviaux avec un niveau de confiance assez haut.

1.6 Motivations et Contributions

Cette thèse a pour but de proposer une méthodologie de certification de programmes informatiques par la logique mathématique, *automatique et convaincante*, capable d'aborder des problèmes de complexité industrielle. Un certain nombre de travaux ont déjà pris cette direction avec plus ou moins d'originalité, d'efficacité et de succés. S'il est prudent de penser que les outils de vérification et d'analyse de programmes développés depuis vingt ans ne sont pas aussi dignes de confiance que leurs concepteurs le voudraient, nous nous sommes efforcés de ne pas tourner le dos à deux décénies de recherche et de développement, en préservant les aspects positifs qu'elles ont su apporter. L'idée directrice de ces travaux est de relier la fiabilité des systèmes de déduction formelle à la puissance d'expréssivité des approches intéractives et à la facilité d'utilisation des approches automatiques. Le tout, conduit par un souci permanent d'effectivité. Pour cela, nous proposons une méthodologie consistant à instrumenter des méta-programmes existants, afin de leur faire générer des certificats de leurs propres résultats. Une difficulté consiste à formaliser ces certificats de sorte à ce qu'ils puissent être vérifiés automatiquement par machine.

Dans l'étude que nous avons menée, les certificats sont des preuves logiques formalisées dans le système Coq. Ces certificats sont générés grâce à l'instrumentation d'un analyseur statique dévellopé au laboratoire Verimag, dont la fonction initiale était de *découvrir* des propriétés invariantes sur le comportement des programmes, et non de les certifier. La classe des programmes considérée par cet analyseur est celle des *programmes*

impératifs et itératifs manipulant des tableaux. Un tableau est une structure de données récursive représentant une collection d'éléments, dont chacun est identifié par un (ou plusieurs) indice(s) (selon la dimension du tableau). Cette structure est à la base de beaucoup d'autres et permet de modéliser des objets complexes et ainsi de traiter des problèmes algorithmiques conséquents. Considérer les tableaux éxige d'être en mesure de gérer des *problèmes d'alias* – plusieurs représentations syntaxiques d'un même objet.

L'une des contributions principales de cette thèse a été d'établir, via une méthodologie conceptuelle basée sur de patrons de preuve (voir définition 3.1.1), renforcée d'une étude de cas concrète, qu'il est possible d'accroître considérablement le niveau de confiance des méta-programmes (une certaine classe, du moins) existants, à un coût modéré. Ce niveau de confiance se mesure par le degrés de fiabilité que l'on peut avoir dans les logiciels mis à contribution pour certifier la correction des résultats retournés par ces méta-programmes. Un logiciel tel que Coq, ayant un noyau de trés petite taille, est extrêmement fiable. Et s'appuyer sur un tel logiciel pour valider les résultats des méta-programmes permet d'atteindre un niveau de confiance des plus élevés.

La seconde question abordée fut « *comment utiliser et combiner les techniques de certification existantes à la nôtre pour traiter des problèmes sortant de l'univers académique ?* » Des éléments de réponse sont apportés en s'appuyant sur la certification d'un protocole d'échange de données multi-tâches, implanté sur un système d'exploitation réactif et utilisé dans l'avionique. La solution consiste à transformer le problème de sorte à ce qu'il puisse être traité par l'outil de certification que nous avons développé. Les techniques de certification existantes (comme la preuve intéractive) nous sont utiles pour justifier formellement cette transformation du problème.

La suite de cette première partie introduit les notions essentielles et nécessaires à la lecture de cette thèse.

1.7 Notions essentielles

1.7.1 Langages de programmation

Les *langages de programmation* se distinguent des langages naturels par le fait qu'ils sont, en général, définis formellement. Cette dimension formelle permet de raisonner mathématiquement sur leur *forme* et leur *sens*. La forme (ou représentation) d'un langage de programmation est appelée *la syntaxe*, et le sens est appelé *la sémantique*. Le type de langage de programmation auquel nous nous intéresserons est *impératif*.

Paradigme de la programmation impérative. Un langage impératif est défini par une syntaxe admettant une instruction d'affectation, permettant de modifier l'état mémoire des variables. Les affectations sont composées séquentiellement et certaines parties peuvent être gardées par des expressions booléenes *i.e* ces parties ne seront exécutées que si les expressions booléenes sont évaluées comme étant vraies. La dimension itérative permet d'exécuter plusieurs fois une séquence d'instructions tant qu'une certaine expression booléene est évaluée à vrai.

Les quatre instructions de base d'un langage impératif itératif sont :

– *l'affectation*, permettant de modifier la valeur d'une variable, en lui attribuant la valeur d'une expression (de type de compatible).

– *la séquence*, permettant de composer et d'ordonner les autres instructions du langage.
– *le branchement conditionnel* permettant d'exécuter des séquences d'instructions différentes selon l'évaluation d'une certaine expression.
– *l'itération*, permettant d'exécuter plusieurs fois (parfois même à l'infini) une même séquence d'instructions tant qu'une certaine expression est évaluée positivement.

1.7.1.1 Syntaxe du langage de programmation considéré

La syntaxe des programmes considérés dans cette thèse est définie en figure 1.1. Ce langage de programmation est *Turing complet* [Tur36] (cela signifie que les fonctions calculable avec ce langage sont les mêmes que celles d'une machine de Turing), et ne manipule que des variables entières et des *tableaux d'entiers*. Pour distinguer les différents types de variables, nous definissons les ensembles suivants :

– Constantes entières : $\mathbb{Z}$
– Variables entières : $\mathcal{X}$, dans lequel nous distinguons les *indices* des *contenus des tableaux* et des *variables scalaires*. Ainsi $\mathcal{X}$ est l'union de deux sous ensembles : $\mathcal{X} \stackrel{def}{=} Indices \cup Contenus$.
– Variables de type tableau : *Arrays*

$k \in \mathbb{Z}$, $i \in Indices$, $x \in Contenus$, $A \in Arrays$
$\diamond \in \{+, -\}$, $\bowtie \in \{=, \neq, \leq, <\}$

$$
\begin{array}{rcl}
\langle programme \rangle & ::= & \langle instruction \rangle \\
& | & \langle instruction \rangle \,;\, \langle programme \rangle \\
\langle instruction \rangle & ::= & \langle affectation \rangle \\
& | & \texttt{while}\ \langle condition \rangle\ \texttt{do}\ \langle programme \rangle \\
& | & \texttt{if}\ \langle condition \rangle\ \texttt{then}\ \langle programme \rangle \\
& & \qquad\qquad \texttt{else}\ \langle programme \rangle \\
\langle affectation \rangle & ::= & i := \langle expression \rangle \\
& | & x := \langle expression \rangle \\
& | & A[\langle expression \rangle] := \langle expression \rangle \\
\langle condition \rangle & ::= & \texttt{not}\ \langle condition \rangle \\
& | & \langle condition \rangle\ \texttt{and}\ \langle condition \rangle \\
& | & \langle condition \rangle\ \texttt{or}\ \langle condition \rangle \\
& | & \langle expression \rangle \bowtie \langle expression \rangle \\
\langle expression \rangle & ::= & k \\
& | & i \\
& | & x \\
& | & A[\langle expression \rangle] \\
& | & \langle expression \rangle \diamond \langle expression \rangle
\end{array}
$$

Fig. 1.1 – Syntaxe des programmes considérés

1.7.1.2 Sémantique du langage de programmation considéré

La sémantique d'un langage de programmation est un ensemble de règles formelles, permettant de caractériser l'effet de chaque instruction du langage selon un certain critère. Ce critère est variable selon l'utilisation que nous voulons faire de cette sémantique : d'exécuter les instructions du langage, à les traduire vers un autre langage, en passant par prouver des propriétés mathématiques sur ses instructions. De nombreux formalismes ont été proposés pour définir la sémantique d'un langage : *opérationelle (naturelle ou structurelle), dénotationelle, axiomatique,* etc. Tous ces formalismes décrivent des aspects différents d'un langage de programmation, et ne doivent en aucun cas être comparés les uns aux autres car ils forment un ensemble complémentaire.

Ce qu'il est important de distinguer, c'est la sémantique d'un langage, de la sémantique des objets que l'on peut construire grâce à ce langage, *les programmes.*

1.7.2 Programmes et systèmes de transitions

Un *programme* est une représentation d'un algorithme dans un langage dont la syntaxe et la sémantique sont formellement définies. Un programme est défini par une liste d'instructions de ce langage ayant pour but de calculer un résultat désiré à partir d'un ensemble de données d'entrées. La finalité d'un programme est qu'une machine exécute l'algorithme qu'il formalise.

Exemple 1.7.1. *Nous considérons un algorithme très simple, qui à partir d'une variable* `x` *initialisée à* 1*, l'évalue et l'incrémente tant que sa valeur n'a pas dépassée* 100*. Cela se formalise par le programme* `p` *suivant :*

```
void exemple(){
    int x = 1;
    while (x <= 100) do
       {
       x = x + 1;
       }
}
```

Sémantique des programmes. La sémantique d'un programme est une expression (formelle ou intuitive) permettant de caractériser les comportements de ce programme. Il existe une multitude de formalismes permettant de définir la sémantique d'un programme et celles-ci sont liées aux sémantiques du langage employé. La sémantique d'un programme peut être vue comme une propriété invariante des comportements de ce programme pour toutes ses exécutions. De manière générale, il n'est pas possible de *calculer automatiquement* la sémantique d'un programme, quelque soit le formalisme choisi [Ric53]. Il n'est également pas possible de *prouver automatiquement* qu'une expression quelconque est ou n'est pas une propriété sémantique d'un programme donné.

Exemple 1.7.2. *Une sémantique informelle (et triviale) pour l'exemple 1.7.1 pourrait être :* à l'entrée de la boucle, la valeur de `x` est 1, et à sa sortie elle est toujours de 101.

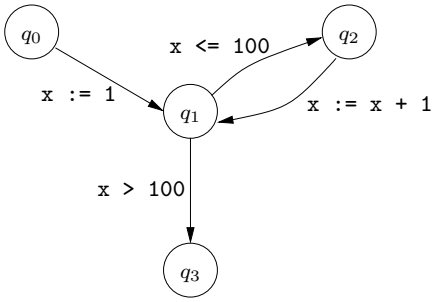
Nous verrons dans la suite de cette section comment formaliser cette sémantique informelle, puis la démontrer en logique mathématique.

Une technique utile pour réaliser cela consiste à représenter les programmes par des *systèmes de transitions.*

Les systèmes de transitions sont une autre représentation des algorithmes, sous forme graphique, équivalentes aux programmes (en terme d'expressivité). Nous en donnons la définition.

Définition 1.7.1 (Système de transitions)**.** *Un système de transitions est défini par un triplet (Q, T, Q_{init}) où Q est un ensemble de points de contrôle, $T \subset Q \times Q$ la relation de transition entre les points de contrôle et leurs successeurs, et $Q_{init} \subset Q$ l'ensemble des points de contrôle initiaux. Les transitions sont étiquetées par des instructions algorithmiques de type garde et affectation. Une garde est définie par une expression booléene.*

Exemple 1.7.3. *Le programme donné dans l'exemple 1.7.1 peut se représenter par le système de transitions suivant :*



*Le point de contrôle q_0 est l'unique point de contrôle initial de ce système ($Q_{init} \stackrel{def}{=} \{q_0\}$). Le point de contrôle q_2 (*resp. *q_3) est accessible à partir de q_1 si l'évaluation de la condition de boucle $x \leq 100$ est positive (*resp. *négative). Le point de contrôle q_1 est quant à lui directement accessible à partir de q_0 par la transition* `x := 1`.

Un système de transitions est un objet non voué à être exécuté par une machine mais permettant de raisonner sur les algorithmes de façon plus intuitive que les programmes correspondants. Cette représentation des algorithmes permet de s'abstraire de toute autre instruction que la garde et l'affectation. L'itération, la séquence, et le branchement sont exprimés implicitement par la structure graphique du système, comme il est possible de l'observer sur l'exemple 1.7.3. Cette méthode, décomposant le programme en ses instructions les plus simples, facilite les raisonnements sur les algorithmes car elle permet d'associer à chaque point de contrôle, une assertion invariante, essentielle pour prouver la correction du programme.

Définition 1.7.2 (Invariant)**.** *Un invariant à un point de contrôle q d'un programme* `p`*, est une propriété sur les variables de* `p`*, qui est vraie lorsque le point de contrôle est*

atteint. Un invariant est dit inductif *à un point de contrôle q, s'il est vrai lorsque q est atteint pour la première fois et demeure vrai après chaque cycle conduisant de nouveau à q.*

Exemple 1.7.4. *Il est possible d'affiner la sémantique définie dans l'exemple 1.7.2 en associant à chaque point de contrôle un invariant :*

- En q_0, la valeur de `x` est inconnue
- En q_1, la valeur de `x` est soit comprise entre 1 et 100 (inclus), soit égale à 101.
- En q_2, la valeur de `x` est comprise entre 1 et 100
- En q_3, la valeur de `x` est égale à 101

Les invariants en q_1 et q_2 sont inductifs. Malheureusement, ces invariants ne sont pas formels et il est difficile de les exploiter dans une approche scientifique permettant de prouver automatiquement la correction des programmes.

Le formalisme que nous utiliserons pour définir ces propriétés sémantiques (les invariants) sur les programmes est la logique du premier ordre que nous définissons maintenant.

1.7.3 Logique du premier ordre

Nous définissons dans la section qui suit le langage des formules du premier ordre que nous nommons FOL, et une partie des méthodes formelles de raisonnement qui l'accompagne.

Définition 1.7.3 (Signature)**.** *Une signature Σ est un ensemble de symboles de fonction d'arité $n \geq 0$ (pour exemple, $=$, $+$, $\leq$, etc...). Les symboles d'arité 0 sont nommés* constantes*.*

Définition 1.7.4 (Terme)**.** *Soit $\mathcal{X}$ un ensemble de variables. Un* terme *est un* arbre *(un graphe acyclique orienté possédant une unique racine) fini t dont la racine peut être :*

- *soit une variable $x \in \mathcal{X}$, et l'arbre se limite à x.*
- *soit un symbole de fonction f d'arité n dans Σ, et l'arbre doit avoir n branches dont les racines sont les termes $t_1, \ldots, t_n$.*

  Nous dénommons par $\mathcal{T}(\Sigma, \mathcal{X})$ l'ensemble des termes du premier ordre *sur Σ et $\mathcal{X}$.*

Définition 1.7.5 (Substitution)**.** *Une substitution σ est une fonction des variables vers l'ensemble des termes ($\sigma : \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$). On note $Dom(\sigma) \subseteq \mathcal{X}$ le domaine de σ tel que pour toute variable $x \notin Dom(\sigma), \sigma(x) = x$. On définit l'application de la substitution σ au terme t ($t\sigma$) par :*

- $x\sigma = \sigma(x)$
- $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$

Dans le cas particulier où la substitution σ ne porte que sur une variable x, nous noterons l'application de σ sur un terme t, $t[x/e]$ si $\sigma(x) = e$.

Définition 1.7.6 (Prédicat, Atome)**.** *Soit $P \subseteq \Sigma$ un sous ensemble de symboles de*

fonctions de la signature Σ.

Un prédicat $p(s_1, \ldots, s_n)$ *d'arité* n *est un élement de* P, *où* $s_1, \ldots, s_n$ *sont des symboles de* Σ. *Les prédicats d'arité* 0 *sont nommés* symboles propositionnels.

Un atome a *est un terme de* $\mathcal{T}(P, \mathcal{X})$.

Définition 1.7.7 (Formule)**.** *Une* formule $F \in$ FOL *est un objet défini par la syntaxe suivante :*

$$\begin{array}{rcl} \textit{Formule} & ::= & a \\ & | & \bot \\ & | & \top \\ & | & \neg \textit{Formule} \\ & | & \textit{Formule} \Rightarrow \textit{Formule} \\ & | & \textit{Formule} \wedge \textit{Formule} \\ & | & \textit{Formule} \vee \textit{Formule} \\ & | & \forall x, \textit{Formule} \\ & | & \exists x, \textit{Formule} \end{array}$$

où a *est un atome,* $\bot$ *est le symbole propositionnes représentant* l'absurde, $\top$ *est le symbole propositionnel représentant* le vrai, *les connecteurs* $\wedge$ *et* $\vee$ *correspondent respectivement à la* conjonction *et à la* disjonction *logique. Le symbole* $\Rightarrow$ *formalise* l'implication logique. $\forall$ *et* $\exists$ *sont les quantificateurs universels et existentiels : la formule* $\forall x, F(x)$ *exprime que la formule* $F(t)$ *est valide quelque soit le terme* t, *alors que* $\exists x, F(x)$ *exprime l'existence d'un terme* t *tel que* $F(t)$ *soit valide.*

Variables libres. Les variables libres d'une formule logique du premier ordre sont celles qui ne sont pas introduites par un quantificateur apparaissant dans cette formule.

Définition 1.7.8 (Variables libres ou liées)**.** *L'ensemble des variables libres d'une formule est défini récursivement par la fonction* VL *suivante :*

$$\begin{array}{rcl} VL(a) & = & \{v \mid \exists p \in Pos(a), a|_p = v\} \\ VL(\bot) & = & \emptyset \\ VL(\top) & = & \emptyset \\ VL(\neg F) & = & VL(F) \\ VL(F_1 \bowtie F_2) & = & VL(F_1) \cup VL(F_2) \text{ avec } \bowtie \in \{\Rightarrow, \wedge, \vee\} \\ VL(Qx, F) & = & VL(F) \setminus \{x\} \text{ avec } Q \in \{\forall, \exists\} \end{array}$$

Les variables de F *qui n'appartiennent pas* $VL(F)$ *sont dites liées dans* F.

Modèle, satisfiabilité, validité. Un modèle est une fonction d'interprétation des formules qui associe à chaque variable du langage considéré une valeur, permettant d'évaluer positivement une formule donnée F.

Si une formule admet au moins un modéle on dit alors qu'elle est *satisfiable*. Soit $\mathcal{M}$ un modèle de la formule F, nous exprimons la *satisfiabilité* de F dans $\mathcal{M}$ (*i.e.* F est interprétée par *vraie* dans $\mathcal{M}$) par $\models_{\mathcal{M}} F$. Si F est satisfiable pour toutes les fonctions d'interprétation, alors F est dite *valide* et nous notons cela $\models F$.

1.7.4 Systèmes formels

Définition 1.7.9 (Règle d'inférence)**.** *Une* règle d'inférence *r est une structure composée d'un nombre fini (eventuellement nul) de formules $P_1, \ldots, P_n$ nommées* prémisses*, et d'une formule C nommée* conclusion *de la règle. La représentation d'une telle règle d'inférence sera donnée sous la forme suivante :*

$$\frac{P_1 \quad \ldots \quad P_n}{C}\, r$$

> *« Si j'ai déduit les prémisses $P_1, \ldots, P_n$ alors, grâce à la règle r je peux déduire la conclusion C. »*

Une règle d'inférence ne comptant pas de prémisses est un axiome*.*

Définition 1.7.10 (Système d'inférence, Dérivation, Preuve)**.** *Un* système d'inférence *$\mathcal{R}$ est un ensemble de k règles d'inférence $r_1, \ldots, r_k$.*

Une dérivation *d'un tel système $\mathcal{R}$ est un arbre fini dont les noeuds sont étiquetés par des couples (C, r), où C est une formule et r une règle d'inférence. La racine mise à part, chaque conclusion des instances des règles employées est une prémisse de la règle du noeud père :*

$$\dfrac{\dfrac{\dfrac{\vdots \quad \ldots \quad \vdots}{P_{1,1} \qquad\quad P_{1,i}}\, r_1 \qquad \ldots \qquad \dfrac{\vdots \quad \ldots \quad \vdots}{P_{n,1} \qquad\quad P_{n,j}}\, r_n}{P_1 \qquad\quad\; \ldots \qquad\quad P_n}\, r}{C}$$

> *« Les prémisses $P_1, \ldots, P_n$ sont les conclusions des instances respectives des règles $r_1, \ldots, r_n$. Les prémisses $P_{1,1}, \ldots, P_{1,i}, \ldots, P_{n,1}, \ldots, P_{n,j}$ sont les conclusions des instances de règles utilisées au niveau suivant. »*

Une preuve *est une dérivation dont les feuilles sont étiquetées par des axiomes.*

Dérivabilité. La *dérivabilité* d'une formule s'exprime relativement à un système d'inférence. Un formule F est dérivable dans un système $\mathcal{R}$ s'il existe un arbre de preuve ∇_F construit avec les règles d'inférence de $\mathcal{R}$ dont la conclusion est F. Nous exprimerons que *la formule F est dérivable dans $\mathcal{R}$* par $\vdash_{\mathcal{R}} F$.

1.7.4.1 La déduction naturelle NJ

La déduction naturelle est un système d'inférence, proposé par G.Gentzen [Gen35], puis rendu populaire par Prawitz [Pra65] au milieu du sciècle dernier, permettant de représenter les preuves des théorèmes de la logique du premier ordre. Ce système (contrairement au système à la Hilbert) se caractérise par la dualité symétrique de ses règles portant sur les connecteurs logiques $(\wedge, \vee, \Rightarrow, \forall, \exists)$. À chaque connecteur sont associées une règle d'*introduction* et une règle d'*élimination* (voir figure 1.2).

$$\frac{\vdash_{\text{NJ}} A \qquad \vdash_{\text{NJ}} B}{\vdash_{\text{NJ}} A \wedge B}\ \wedge_i$$

$$\frac{\vdash_{\text{NJ}} A \wedge B}{\vdash_{\text{NJ}} A}\ \wedge_{e_1} \qquad \frac{\vdash_{\text{NJ}} A \wedge B}{\vdash_{\text{NJ}} B}\ \wedge_{e_2}$$

$$\frac{\vdash_{\text{NJ}} A}{\vdash_{\text{NJ}} A \vee B}\ \vee_{i_1} \qquad \frac{\vdash_{\text{NJ}} B}{\vdash_{\text{NJ}} A \vee B}\ \vee_{i_2}$$

$$\frac{\vdash_{\text{NJ}} A \vee B \qquad \begin{array}{c}\overbrace{\vdash_{\text{NJ}} A}^{H_1}\\ \vdots\\ \vdash_{\text{NJ}} C\end{array} \qquad \begin{array}{c}\overbrace{\vdash_{\text{NJ}} B}^{H_2}\\ \vdots\\ \vdash_{\text{NJ}} C\end{array}}{\vdash_{\text{NJ}} C}\ \vee_e(H1, H2)$$

$$\frac{\begin{array}{c}\overbrace{\vdash_{\text{NJ}} A}^{H}\\ \vdots\\ \vdash_{\text{NJ}} B\end{array}}{\vdash_{\text{NJ}} A \Rightarrow B}\ \Rightarrow_i (H) \qquad \frac{\vdash_{\text{NJ}} A \qquad \vdash_{\text{NJ}} A \Rightarrow B}{\vdash_{\text{NJ}} B}\ \Rightarrow_e$$

$$\frac{\vdash_{\text{NJ}} F(a)}{\vdash_{\text{NJ}} \forall x F(x)}\ \forall_i \qquad \frac{\vdash_{\text{NJ}} \forall x F(x)}{\vdash_{\text{NJ}} F(a)}\ \forall_e[x/a]$$

$$\frac{\vdash_{\text{NJ}} F(a)}{\vdash_{\text{NJ}} \exists x F(x)}\ \exists_i \qquad \frac{\vdash_{\text{NJ}} \exists x F(x) \qquad \begin{array}{c}\overbrace{\vdash_{\text{NJ}} F(a)}^{H}\\ \vdots\\ \vdash_{\text{NJ}} C\end{array}}{\vdash_{\text{NJ}} C}\ \exists_e(H)$$

$$\frac{\vdash_{\text{NJ}} F \Rightarrow \bot}{\vdash_{\text{NJ}} \neg F}\ \neg_{def} \qquad \frac{\vdash_{\text{NJ}} \neg F}{\vdash_{\text{NJ}} F \Rightarrow \bot}\ \neg_{def} \qquad \frac{\vdash_{\text{NJ}} \bot}{\vdash_{\text{NJ}} F}\ \bot_e$$

Fig. 1.2 – Règles d'inférence de la Déduction naturelle

Conditions d'application des règles. Les règles $\forall_i$ et $\exists_e$ s'appliquent à condition que la variable a soit fraîche *i.e.* qu'elle ne soit ni libre dans F, ni libre dans les hypothèses disponibles. Cela assure dans le cas de $\forall_i$ que F est bien prouvée pour tout x (soit un a quelconque) et dans l'autre cas, que a soit un témoin arbitrairement choisi (soit un a quelconque) de la validité de $\exists x F(x)$. Les règles d'introduction ne s'appliquent que si le connecteur principal de la conclusion coïncide. Dans les règles d'élimination, on distingue la *prémisse principale* (une et une seule) des *prémisses mineures* (il peut ne pas y en avoir). La prémisse principale porte forcément sur le connecteur éliminé.

Exemple 1.7.5. *Soient P et Q deux prédicats d'arité 1 appartenant à la signature Σ. Nous cherchons à prouver que si il existe un variable y de $\mathcal{X}$ telle que pour tout x, la formule $P(x) \wedge Q(y)$ est vrai, alors pour tout z il existe un w tel que $P(z) \wedge Q(w)$. Cela se formalise par la formule $(\exists y\ \forall x\ P(x) \wedge Q(y)) \Rightarrow (\forall z\ \exists w\ P(z) \wedge Q(y))$. La preuve de cette formule est donnée ci dessous :*

$$\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\exists y\ \forall x\ P(x) \wedge Q(y)}^{H_1} \qquad \dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\forall x\ P(x) \wedge Q(b)}^{H_2}}{\vdash_{\text{NJ}} P(a) \wedge Q(b)}\ \forall_e[x/a]}{\vdash_{\text{NJ}} \exists w\ P(a) \wedge Q(w)}\ \exists_i}{\dfrac{\vdash_{\text{NJ}} \exists w\ P(a) \wedge Q(w)}{\vdash_{\text{NJ}} \forall z\ \exists w\ P(z) \wedge Q(w)}\ \forall_i}\ \exists_e(H_2)}{\vdash_{\text{NJ}} (\exists y\ \forall x\ P(x) \wedge Q(y)) \Rightarrow (\forall z\ \exists w\ P(z) \wedge Q(w))}\ \Rightarrow_i (H_1)$$

Dans la preuve ci-dessus, l'application de la règle $\exists_e(H_2)$ est correcte car l'hypothèse $H_2 \stackrel{def}{=} \forall x\ P(x) \wedge Q(b)$, issue de la formule $\exists y\ \forall x\ P(x) \wedge Q(y)$, contient la variable fraîche b. Puis, l'application de la règle $\forall_i$ est également correcte car la variable a est fraîche.

Pour sa clarté, sa facilité de lecture et son côté intuitif, nous choisissons la *déduction naturelle* pour représenter les preuves manuscrites de cette thèse.

Nous finissons cette présentation de la logique du premier ordre, en définissant les théories du premier ordre.

Définition 1.7.11 (Théorie du premier ordre)**.** *Une théorie du premier ordre T_Σ est l'ensemble des formules du premier ordre valides qui peuvent être construites sur la signature Σ. Une théorie comprend un certain nombre d'axiomes, relatifs à Σ, non prouvable dans la théorie elle-même, qui permettent de prouver grâce à un système d'inférence associé à la logique du premier ordre (la déduction naturelle par exemple), une partie des théorèmes de T_Σ. Si certains théorèmes de la théorie T_Σ ne sont pas prouvables à partir des axiomes de T_Σ, alors la théorie est dite incomplète.*

1.7.4.2 Théorie de l'Arithmétique de Presburger

L'arithmétique de Presburger est une théorie du premier ordre construite initialement sur la signature $\{+, =, \mathbb{N}\}$. Cette théorie est bien connue et souvent utilisée car elle est

cohérente (*i.e.* on ne peut pas prouver dans cette théorie une formule et sa négation), complète (*i.e.* pour chaque formule de cette théorie, il est soit possible de la prouver, soit il est possible de prouver sa négation), décidable (*i.e.* il existe un algorithme qui décide pour une formule donnée de cette théorie, si elle est vraie ou fausse). Nous en donnons ici une définition quelque peu différente, en ajoutant des symboles à la signature, et des axiomes par conséquent, mais préservant les propriétés de la définition initiale (cohérente, complète et décidable). Cette redéfinition nous est utile pour énoncer et prouver des *formules d'arithmétique linéaire* plus facilement. Cela sera précisé par la suite (section 3.2.2). La signature est donc élargie des entiers naturels aux relatifs, des égalités aux inégalités[2], et nous rajoutons la soustraction aux opérateurs, comme inverse de l'addition : $\Sigma \stackrel{def}{=} \{+, -, =, \neq, \leq, \mathbb{Z}\}$. Les axiomes et les règles de cette arithmétique sont donnés en figure 1.3.

$$\frac{}{\vdash_{\text{NJ}} x = x}\ =_{refl} \qquad \frac{\vdash_{\text{NJ}} x = y \qquad \vdash_{\text{NJ}} F}{\vdash_{\text{NJ}} F[x/y]}\ =_{e} \qquad \frac{\vdash_{\text{NJ}} \neg(x = y)}{\vdash_{\text{NJ}} x \neq y}\ \neq_{def}$$

$$\frac{\vdash_{\text{NJ}} x \leq y}{\vdash_{\text{NJ}} x + z \leq y + z}\ +_c(z) \qquad \frac{\vdash_{\text{NJ}} x \leq y \qquad \vdash_{\text{NJ}} y \leq z}{\vdash_{\text{NJ}} x \leq z}\ \leq_{trans}$$

$$\frac{}{\vdash_{\text{NJ}} x \leq x + 1}\ +1 \qquad \frac{}{\vdash_{\text{NJ}} x = x + 0}\ +0$$

$$\frac{}{\vdash_{\text{NJ}} x \leq c - 1 \vee x = c \vee c + 1 \leq x}\ \mathbb{Z}_{def}$$

avec $x, y, z \in \mathcal{X}$ *et* $F \in \text{FOL}$

Fig. 1.3 – Axiomes et règles d'inférence de l'arithmétique de Presburger

1.7.5 Sémantique axiomatique

Nous avons vu en section 1.7.2 qu'il existe plusieurs formalismes pour définir la sémantique d'un programme. La sémantique axiomatique permet de formaliser le comportement des programmes au moyen de formules de la logique du premier ordre, et permet ainsi de relier les programmes à la logique. La logique de Hoare est la sémantique axiomatique la plus connue. Elle est accompagnée d'un système d'inférence (section 1.7.5.1) permettant de raisonner sur les propriétés des programmes. Une autre façon de raisonner sur les programmes à partir du formalisme de logique de Hoare, est de transformer les termes de cette logique en formule de la logique du premier ordre, puis d'utiliser la déduction naturelle pour prouver ces invariants de programmes (1.7.5.2).

[2]Une inégalié peut être exprimé par une disjonction d'égalités.

1.7.5.1 Logique de Hoare

La logique de Hoare est un formalisme permettant de spécifier, puis de raisonner sur le comportements des programmes. Ce formalisme logique consiste à annoter les instructions d'un programmes avec des *assertions* (des formules portant sur les variables d'entrées et de sorties du programme). Cette méthode qui est la première du genre demeure également la plus élégante pour prouver qu'une propriété (formalisée en logique du premier ordre) est une sémantique correcte d'un programme. L'objet central de cette logique est le *triplet de Hoare*.

Définition 1.7.12 (*Triplet de Hoare*)**.** *Un triplet de Hoare est un objet composé de deux assertions P et Q nommées respectivement* pré et post-condition*, et d'un programme p :*

$$\{P\}\mathsf{p}\{Q\}$$

> « *Si la pré-condition P est vérifiée et que le programme p est exécuté et termine, alors la post-condition Q est également vérifiée* ».

Les triplets de Hoare permettent de spécifier la sémantique des programmes de manière très naturelle, en représentant leurs états à chaque instruction par des formules de la logique du premier ordre.

Définition 1.7.13 (Correction des programmes)**.** *La correction d'un programme p se spécifie par un triplet de Hoare $\{P\}\mathsf{p}\{Q\}$ où P (la pré-condition) est une condition d'appel de p sur ses variables d'entrées, et Q est une condition sur les variables d'entrées et de sorties que l'exécution de p doit satisfaire.*

La correction d'un programme est établie lorsqu'une preuve du triplet $\{P\}\mathsf{p}\{Q\}$ est fournie.

La logique de Hoare fournit un ensemble de règles d'inférence structurelles permettant de prouver les triplets de Hoare. Les langages visés par ce système d'inférence, sont généralement *impératifs*, dont le principe est de modifier l'état des programmes par effet de bord, au moyen d'une instruction d'*affectation* de variable. Nous définissons en figure 1.4 les règles d'inférences permettant de prouver des triplet de Hoare portant sur les *programmes impératifs*. Ce système d'inférence sera nommé H.

Exemple 1.7.6. *Dans le cas de l'exemple 1.7.1, cela conduit à construire un arbre de dérivation guidé par les règles d'inférence spécifiques aux instructions du langage de programmation utilisées par le programme.*

Nous posons $I \stackrel{def}{=} 1 \leq x \leq 101$, comme étant l'invariant de la boucle du programme.

$$\dfrac{\vdash_{\text{NJ}} I \wedge x \leq 100 \Rightarrow 1 \leq x \leq 100 \qquad \dfrac{}{\vdash_{\text{H}} \{1 \leq x \leq 100\}\texttt{x := x+1}\{2 \leq x \leq 101\}}\,asg \qquad \vdash_{\text{NJ}} 2 \leq x \leq 101 \Rightarrow I}{\vdash_{\text{H}} \{I \wedge x \leq 100\}\texttt{x := x+1}\{I\}}\,strong$$

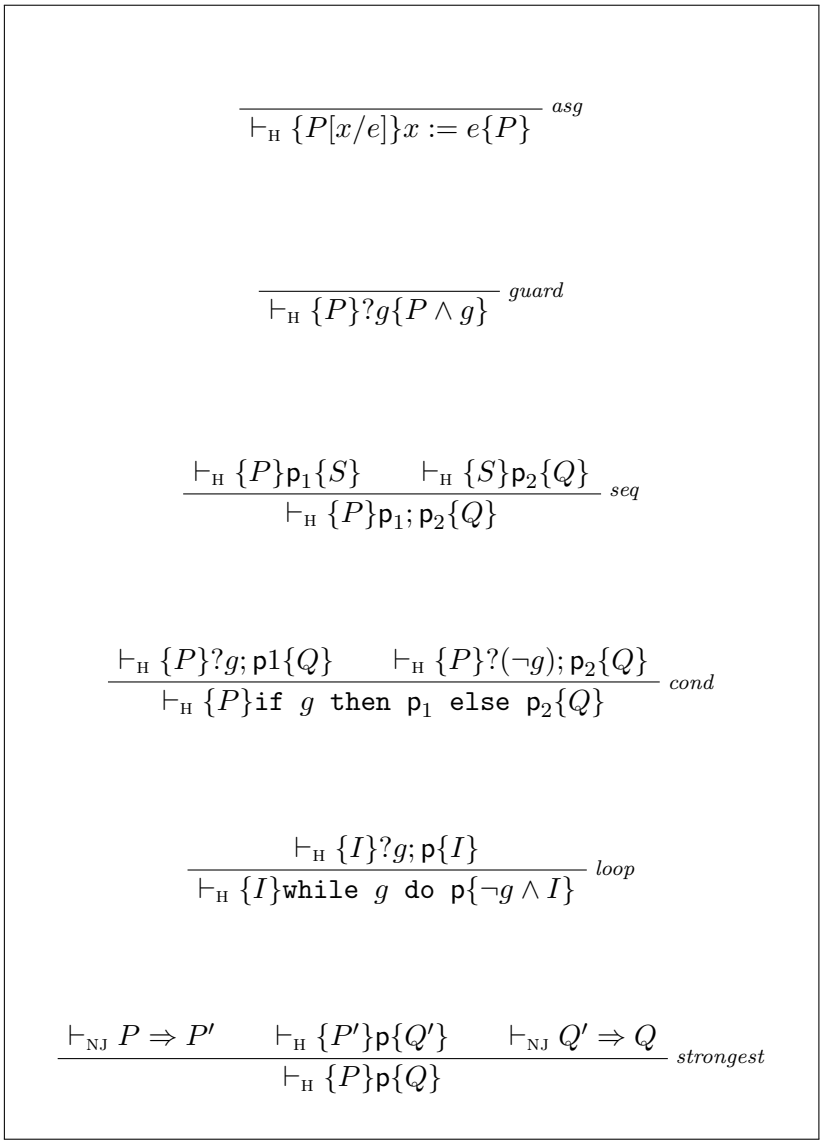


Fig. 1.4 – Règles d'inférence de la logique de Hoare

$$\dfrac{\dfrac{}{\vdash_{\text{H}} \{I\}\,\texttt{?(x<=100)}\{I \wedge x \leq 100\}}\ guard \qquad \begin{array}{c}\vdots\\ \vdash_{\text{H}} \{I \wedge x \leq 100\}\texttt{x := x+1}\{I\}\end{array}}{\vdash_{\text{H}} \{I\}\,\texttt{?(x<=100) ; x := x+1}\{I\}}\ seq$$

$$\dfrac{\vdash_{\text{NJ}} x = 1 \Rightarrow I \qquad \dfrac{\begin{array}{c}\vdots\\ \vdash_{\text{H}} \{I\}\,\texttt{?(x<=100) ; x := x+1}\{I\}\end{array}}{\vdash_{\text{H}} \{I\}\texttt{p'}\{I \wedge x > 100\}}\ loop \qquad \vdash_{\text{NJ}} I \wedge x > 100 \Rightarrow x = 101}{\vdash_{\text{H}} \{x = 1\}\underbrace{\texttt{while x <= 100 do x := x+1}}_{\texttt{p'}}\{x = 101\}}\ strongest$$

$$\dfrac{\begin{array}{c}\vdots\\ \vdash_{\text{H}} \{\top\}\texttt{x := 1}\{x = 1\}\end{array} \qquad \begin{array}{c}\vdots\\ \vdash_{\text{H}} \{x = 1\}\texttt{tant que x <= 100 faire x := x+1}\{x = 101\}\end{array}}{\vdash_{\text{H}} \{\top\}\texttt{x := 1 ; while x <= 100 do x := x+1}\{x = 101\}}\ loop$$

Cette preuve, pour être validée, demande à vérifier quatre implications logique entre des formules purement arithmétiques. Ces sous preuves, pour pouvoir être construites, nécessitent que le système d'inférences $\vdash_{\text{NJ}}$ *soit augmenté d'un certain nombre d'axiomes et de règles permettant de raisonner sur les predicats définis par sa signature (*$\{\leq, =\}$ *dans cet exemple). Nous ne donnerons pas ces règles ici, et nous expliquons seulement informellement pourquoi ces implications sont correctes :*

1. $x = 1 \Rightarrow I$ *:*
2. $I \wedge x > 100 \Rightarrow x = 101$ *:*
3. $2 \leq x \leq 101 \Rightarrow I$ *:*
4. $I \wedge x \leq 100 \Rightarrow 1 \leq x \leq 100$ *:*

La première implication du fait que si x *vaut* 1 *alors il est imprécis mais correcte d'affirmer que* x *appartient à l'intervalle* $[1, 101]$*. La seconde se prouve en exploitant le fait que* x *est strictement supérieure à* 100 *et inférieure ou ègale à* 101*. Une seule valeur est possible,* 101*. La troisième implication se prouve comme la première, par une perte de précision. Si* x *appartient à l'intervalle* $[2, 101]$*, il est correcte d'en déduire que* x *appartient à l'intervalle* $[1, 101]$*. Enfin, la dernière implication se prouve par restriction de l'intervalle désigné par* I *en tenant compte du fait que* x *est inférieure ou ègale à* 100*. En conséquence, il est correcte de déduire que* x *appartient à l'intervalle* $[1, 100]$*.*

Une autre approche, permettant de raisonner sur les triplets de Hoare, mais d'un point de vue fonctionnel (plus facile à modéliser), est la *sémantique de transformation de predicats*.

1.7.5.2 Semantique de transformation des prédicats

Cette sémantique, proposée par Edsger Wybe Dijkstra [Dij75, Dij76], est équivalente à la logique de Hoare (en terme d'expressivité), mais ne comporte pas d'axiome et n'a qu'une seule règle d'inférence. En revanche elle emploie une *fonction* de transformation des formules calculant *la plus faible pré-condition* relativement à un programme et une

formule donnés. La notion de « *faible* » est à prendre au sens de l'implication logique. Nous dirons qu'une formule B est plus faible qu'une formule A, si et seulement si $A \Rightarrow B$. Réciproquement, A est dite plus « *forte* » que B.

Étant donnés un programme p et une formule Q, la *plus faible pré-condition* qui leur est associée est une formule $wp(\mathsf{p}, Q)$ telle que :

- *correction* - le triplet $\{wp(\mathsf{p}, Q)\}\mathsf{p}\{Q\}$ est valide
- *précision* - pour toute formule $P \in$ FOL telle que $\{P\}\mathsf{p}\{Q\}$ est un triplet de Hoare valide, on a $P \Rightarrow wp(\mathsf{p}, Q)$.

Définition 1.7.14 (Plus faible pré-condition)**.** *Nous donnons la définition de cette fonction pour les instructions définies en section 1.7.1*

- $wp(x := e, Q) \stackrel{def}{=} Q[x/e]$
- $wp(?g, Q) \stackrel{def}{=} g \Rightarrow Q$
- $wp(\mathsf{p}_1; \mathsf{p}_2, Q) \stackrel{def}{=} wp(\mathsf{p}_1, wp(\mathsf{p}_2, Q))$
- $wp(si\ g\ alors\ \mathsf{p}_1\ sinon\ \mathsf{p}_2, Q) \stackrel{def}{=} (g \Rightarrow wp(\mathsf{p}_1, Q)) \bigwedge (\neg g \Rightarrow wp(\mathsf{p}_2, Q))$

Grâce à ce calcul de plus faible pré-condition, il en découle qu'une unique règle d'inférence est necessaire pour exprimer la sémantique des programmes :

$$\frac{\vdash_{\text{NJ}} P \Rightarrow wp(\mathsf{p}, Q)}{\vdash_{\text{H}} \{P\}\mathsf{p}\{Q\}}\ sem$$

Ces méthodes ont initiallement été développées pour fournir aux programmeurs des techniques de raisonnement leur permettant de construire des programmes *correctes* par construction. Cependant, elles furent relativement peu utiliser dans ce cadre là car elles présentent deux principaux problèmes :

Le premier, est que dans le cas des boucles, l'invariant I n'est pas simple à trouver intuitivement et n'est pas, en général, calculable. De nombreux travaux ont été réalisés sur le calcul des invariants de boucle [CH78, Sif82, BBM95, BM08] mais une méthode globale n'existe pas encore à ce jour. Le second, est que les preuves pour des programmes de taille quelque peu plus conséquente deviennent vite difficiles à construire à la main. Ces deux difficultés font que ces méthodes ne sont pas effectives en pratique.

Une technique pour remédier au second problème, est de représenter les programmes par des systèmes de transitions. Cela permet ainsi de décomposer le problème de la correction d'un programme donné, en autant de sous problèmes que le système homologue contient de transitions. Quant au problème posé par l'automatisation de ces méthodes de preuves de programme, il sera abordé tout au long de la suite de ce manuscrit.

1.8 Plan du manuscrit de thèse

La suite de ce manuscrit s'organise de la façon suivante : Le chapitre 2 présente le logiciel Coq, qui nous servira de vérificateur de preuves, puis un certain nombre de travaux réalisés autour de la certification de programmes s'appuyant sur ce logiciel. Le chapitre 3 présente une technique d'instrumentation de méta-programmes qui s'appuie sur des patrons de preuve permettant à ces méta-programmes de produire des justifications formelles de leurs résultats sous forme de preuves déductives vérifiables par Coq. Dans

le chapitre 4 nous relatons une étude de cas d'instrumentation menée sur un analyseur statique non trivial. Cet analyseur calcule des propriétés sur les contenu des tableaux manipulés par les programmes et notre instrumentation lui permet désormais d'apporter automatiquement des justifications formelles de la correction des ces propriétés. Le chapitre 5 présente la certification d'un protocole de communication pour système multi-tâches et réactifs utilisé dans l'avionique. L'interêt de ce chapitre est d'étudier comment les outils de certification automatiques peuvent être employés pour faciliter la construction de preuves de correction de systèmes critiques et complexes. Enfin, nous conclurons ce manuscrit dans le chapitre 6 en résumant plus en détail les contributions apportées et les perspectives qui nous paraissent intéressantes que nos travaux ont ouvertes.

Chapitre 2

Autour de la certification formelle de programmes

> « *Douter de tout ou tout croire sont deux solutions également commodes, qui l'une et l'autre nous dispensent de réfléchir.* »
>
> Henri Poincaré

Nous présentons dans ce chapitre différents travaux réalisés autour de la certification de programmes. Nous commençons par une partie de l'outillage qui les accompagne et que nous adoptons à notre tour, le logiciel Coq.

Il est la base logicielle que nous utilisons pour garantir la correction des programmes et dont nous ne doutons pas de sa correction. Le vérificateur de preuves du logiciel Coq est le TCB (*cf.* section 1.5) de nos travaux, *i.e.* nous ne doutons pas des verdicts rendus par le logiciel Coq concernant la validité des preuves que nous lui soumettons.

2.1 Calcul des Constructions Inductives

Le développement récent de logiciels d'aide à la démonstration mathématique (tels que PVS, HOL, Twelve ou Coq) pose la question du langage qu'il convient d'utiliser pour formaliser les mathématiques *sur machine*. Si la théorie des ensembles reste encore la référence pour la plupart des mathématiciens, force est de reconnaître qu'elle est peu adaptée à un traitement mécanisé des démonstrations. Les travaux autour de la *correspondance de Curry-Howard* ont fait émerger une nouvelle famille de formalismes : la théorie des types et les systèmes de types, qui intègrent au sein d'un même langage les propositions mathématiques, les preuves, les types de données et les programmes.

Le *calcul des constructions inductives* (CIC) [CH88, PM89a, BC04] est un langage formel basé sur la théorie des types, permettant de représenter des termes d'ordre supérieur. C'est un λ-calcul typé [Bar81] admettant des types dépendants et inductifs. Il interprète chacun de ses termes comme étant une preuve de son type (voir figure 2.1). La relation de typage $\Gamma \vdash t : T$ signifiant que dans l'environnement de typage Γ, t a le type T, s'interprète parallèlement par « *sous les hypothèses* Γ*, on peut déduire que* t *est une preuve du théorème* T » .

L'outil Coq [BC04, Chl08] a été implanté sur les fondements de ce calcul. Coq est un interpréteur et un vérificateur de type pour ce langage et l'interprétation des termes de ce langage comme des preuves logiques en fait un assistant à la preuve intéractif. De nombreux travaux autour de la certification de programmes qu'il nous parait essentiel de présenter ont été réalisés par l'entremise du système Coq. Les chapitres qui vont suivre, contenant les contributions de cette thèse, s'appuient également en partie sur le système Coq. Pour ces raisons, nous présentons une brève introduction à Coq, accessible à tout informaticien ayant un minimum de bases en programmation fonctionnelle et/ou en théorie des types.

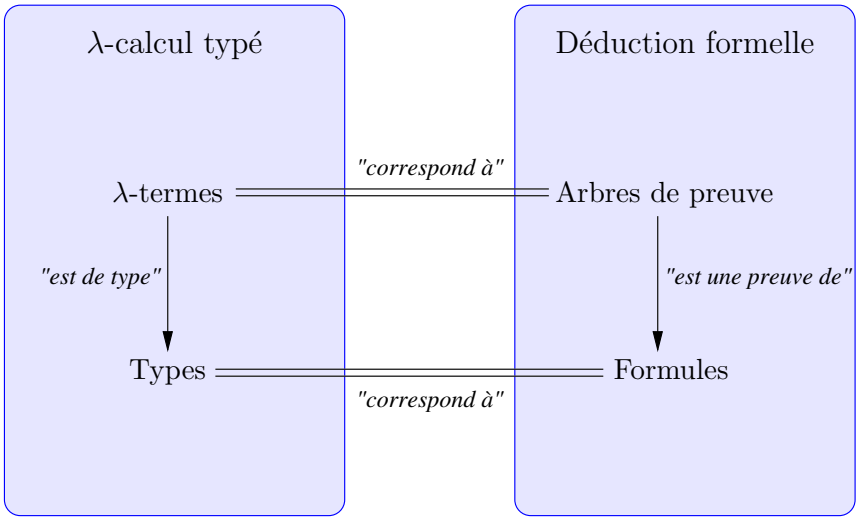


Fig. 2.1 – Correspondance de Curry-Howard

2.1.1 Présentation du système Coq

Plusieurs présentations et manuels d'utilisation ont été rédigés, notamment l'originale approche de David Pichardie axée sur la *programmation* [Pic05], dont nous encourageons la lecture pour sa clarté et la version plus complète et tout aussi claire de Yves Bertot et Pierre Castéran [BC04]. Pour notre part, nous nous prêtons également à l'exercice mais en s'appuyant sur une modélisation de la logique de Hoare sur un langage impératif simple, qui nous sera utile pour la suite.

2.1.1.1 Modélisation de la logique propositionnelle

Nous présentons un langage simple permettant de représenter des formules de la logique propositionnelle. Le système Coq est muni d'une *sorte* `Prop` permettant déjà cela. Cependant, l'intérêt de définir ce langage est double : (1) la structure du langage recouvre relativement bien les différentes façons de définir un nouveau type Coq et donne une bonne illustration des possibilités de modélisation offertes par ce système ; (2) il n'est pas possible d'effectuer du *pattern matching* sur les éléments de `Prop` et par conséquent impossible de programmer une fonction de substitution purement syntaxique (utile pour définir la logique de Hoare dans le cas des affectations) sur le type `Prop`.

Les quantificateurs du premier ordre seront ignorés dans cette présentation du système Coq dans un pur souci de clarté. Nous faisons le choix de nous en passer dans cette introduction car nous visons de modéliser la sémantique axiomatique et passer au premier ordre n'apportera rien d'intéressant à observer et n'ajoutera que des difficultés techniques.

Variables scalaires entières. Nous représentons les variables scalaires du langage que nous allons définir par un couple dont le premier élement est un entier naturel (de type `nat` en Coq) dénotant son numéro d'identifiant, et le second sa valeur par un entier relatif (de type `Z` en Coq). Les numéros d'identifiant agissent comme la représentation syntaxique des variables et permettent de distinguer les variables les unes des autres. Cela est notamment utile pour définir la fonction de substitution.

Ainsi la variable $x \in V$ sera representée, dans notre modélisation Coq, par un couple `(id_x, x)`, où `x` est la valeur de x :

```
Variable x : Z.                    Variable y : Z.
Let var_x : nat*Z := (1, x)        Let var_y : nat*Z := (2, y)
```

Ici la variable `var_x` de type `nat*Z` est identifiée par l'entier 1 et a pour valeur l'entier relatif `x`.

La commande `Check` du système Coq permet de vérifier qu'un terme est bien formé et nous renvoie le type de celui-ci. Ainsi :

```
Check 1.                           Check 2.
 1 : nat                            2 : nat
Check x.                           Check x.
 x : Z                              y : Z
Check var_x.                       Check var_x.
 var_x : nat*Z                      var_y : nat*Z
```

Dans le système Coq, il est possible de nommer les types composés. Ainsi, nous définissons le type `prog_var` qui permet au moyen du constructeur `p_var` de construire des éléments de type `nat*Z`, à partir d'un entier naturel (l'identifiant de la variable) et d'un entier naturel (la valeur de la variable) :

```
Inductive prog_var : Type :=
  | p_var : nat -> Z -> prog_var.
```

Nous pouvons de cette manière redéfinir la variable `var_x` en utilisant ce type plutôt que `nat * Z` :

```
Let var_x : prog_var := (p_var 1 x)
```

Formules logiques. Nous définissons ensuite le type des formules `Formule` construit sur les types des prédicats `Pred` et des expressions arithmétiques `Expr`, constituant le langage de la logique propositionnelle :

```
Inductive Expr : Type :=
  | E_var : prog_var -> Expr
  | const : Z -> Expr
  | add : Expr -> Expr -> Expr
  | sub : Expr -> Expr -> Expr.

Inductive Pred : Type :=
  | equal : Expr -> Expr -> Pred
  | less : Expr -> Expr -> Pred.
```

```
Inductive Formule : Type :=
  | atome : Pred -> Formule
  | bot : Formule
  | conj : Formule -> Formule -> Formule
  | disj : Formule -> Formule -> Formule
  | impl : Formule -> Formule -> Formule
  | not : Formule -> Formule.
```

Contrairement à la première définition de la variable x, les propositions logiques sont définies par des types déclarés par le mot clé `Inductive` (tout comme le type `prog_var`). Si l'on prend le temps de regarder le type des expressions `Expr`, on remarque l'autre mot clé de cette définition, `Type`, indiquant que `Expr` est un *type* de *sorte* `Type`. Tous les objets du système Coq sont typés par un objet de *dimension* supérieure. Les élements de ce type se construisent au moyen de quatre constructeurs ($\{$`E_var`, `const`, `add`, `sub`$\}$). Pour exemple la variable x peut être utilisée pour construire `e_x`, qui modélise la variable x par une expression de type `Expr` et non plus comme une variable de programme de type `prog_var` :

```
Let e_x := E_var var_x.
```

Ainsi la commande `Check` retourne

```
Check e_x.
  e_x : Expr
```

`e_x` est la variable x exprimée dans le type des expressions. Cela nous permet de construire des expressions arithmétiques contenant x, comme `e1` $\stackrel{def}{=} x + 0$ au moyen du constructeur `add` :

```
Let e1 := add e_x (const 0).
Check e1.
  e1 : Expr
```

Le type des prédicats que nous avons défini permet d'exprimer des expressions booléenes modélisant l'égalité ($=$) et l'inégalité stricte ($<$) au moyen des constructeurs `equal` et `less` respectivement. Il est donc possible de définir le prédicat `p1` $\stackrel{def}{=} x = x + 0$ dans le type `Pred` :

```
Let p1 := equal e_x e1.
Check p1.
  p1 : Pred
```

Pour finir si nous souhaitons définir ce prédicat comme une formule, il est possible de le faire grâce au constructeur `atome` du type `Formule` :

```
Let f1 := atome p1.
Check f1.
 f1 : Formule
```

Si `f1` et `p1` exprime la même *idée* ($x = x + 0$ en l'occurrence), ces deux objets n'ont pourtant pas le même type.

2.1.1.2 Modélisation de la sémantique axiomatique

Pour modéliser la sémantique axiomatique, il nous faut au préalable définir un programme effectuant la substitution utile pour définir l'axiome de l'affectation. C'est chose possible en Coq car le système est muni d'un langage de programmation proche de la syntaxe du langage Ocaml.

Programmation en Coq. Nous donnons immédiatement la fonction de substitution `subst_e`, substituant dans une expression `E` la variable `v` par l'expression `e` ($E[v/e]$ formellement), programmée en Coq. La substitution est précedée d'un parcours de `E` à travers sa structure et s'effectue lorsqu'une variable, reconnue par le constructeur `var`, est identifiée comme étant `v`. Cette identification se fait par la comparaison de leur identifiant respectif, au moyen de la fonction `eq_nat_dec` qui décide si deux entiers sont égaux :

```
Fixpoint subst_e (E : Expr) (v : prog_var) (e : Expr) : Expr :=
  match E with
  | E_var (p_var id1 x1) => let (id2, x2) := v in
              if (eq_nat_dec id1 id2) then e else E_var E
  | const c => const c
  | add e1 e2 => add (subst_e e1 v e) (subst_e e2 v e)
  | sub e1 e2 => sub (subst_e e1 v e) (subst_e e2 v e)
  end.
```

On peut remarquer la ressemblance de syntaxe avec le langage Ocaml où le mot clé `Fixpoint` joue le rôle du `let rec`.

La substitution sur les expressions est bien sûr utilisée pour effectuer des substitutions sur les formules. Cela est implanté par la fonction récursive `subst` qui parcourt la structure des formules pour appeler `subst_e` lorsqu'elle *rencontre* des expressions (dans les atomes exclusivement) :

```
Fixpoint subst (F : Formule) (v : prog_var) (e : Expr) : Formule :=
  match F with
  | atome p => match p with
               | equal x y => atome (equal (subst_e x v e) (subst_e y v e))
               | less x y => atome (less (subst_e x v e) (subst_e y v e))
               end
  | bot => bot
  | conj F1 F2 => conj (subst F1 v e) (subst F2 v e)
  | disj F1 F2 => disj (subst F1 v e) (subst F2 v e)
  | impl F1 F2 => impl (subst F1 v e) (subst F2 v e)
  | not F' => not (subst F' v e)
```

```
end.
```

Types paramétrés et dépendants. Nous pouvons désormais, grâce à la définition de cette fonction de substitution, proposer le type inductif et *dépendant* des *triplets de Hoare*, `Triplet`, de type `Formule -> Transition -> Formule -> Prop`. Avant d'en donner la définition nous donnons quelques précisions concernant son type.

La première chose que nous pouvons remarquer, c'est qu'un triplet de la forme {`Formule`}`Transition`{`Formule`} est un objet de type `Prop`, qui est valide ou non. Si nous avons défini un langage pour modéliser les formules afin de pouvoir effectuer des substitutions dessus (ce qui n'est pas possible sur les objets de `Prop`, nous le rappelons), il demeure que le type `Prop` est muni d'un *environnement de preuves* et d'un *langage de tactiques* permettant de raisonner et de prouver les objets de type `Prop`. Le type `Triplet`, outre le fait qu'il permet de modéliser la sémantique des programmes, fait donc le lien entre les programmes et la logique. De plus, la logique est modélisée dans un langage (le CIC) propice pour y démontrer ses théorèmes dans le système Coq et ainsi prouver la validité des triplets de Hoare.

La seconde remarque concerne le fait que le type `Triplet` prend en argument deux objets de type `Formule` et un objet de type `Transition` (que nous allons définir immédiatement). Le fait est que le type `Triplet` *est paramétré* ; en conséquence il ne constitue pas un type en lui-même (dans le sens où il n'existe aucun élement `t` de type `Triplet`) mais une *famille de types*. L'objet `Triplet F1 T F2` est quant à lui un type, à partir du moment où `F1` et `F2` sont de type `Formule` et `T` est de type `Transition`. Mais attention, tous les types génerés par `Triplet` ne contiennent pas forcément des élements. Seulement ceux pouvant être construits à partir des constructeurs de `Triplet` en sont pourvus. Afin de pouvoir clarifier cela aux travers d'exemples, nous définissons le type `Transition` modélisant les instructions des programmes impératifs représentés par des systèmes de transitions. Ainsi le type `Transition` admet seulement deux constructeurs, l'un pour les gardes, l'autre pour les affectations :

```
Inductive Transition : Type :=
  | guard : Formule -> Transition
  | asg : prog_var -> Expr -> Transition.
```

Nous pouvons donner maintenant une première définition de `Triplet`, admettant trois constructeurs, l'un reconnaissant les triplets de Hoare valides pour les transitions gardées ($\{P\}?g\{P \wedge g\}$), le second pour les affectations ($\{P[v/e]\}v := e\{P\}$) et le dernier permettant de montrer la validité d'autres triplets de Hoare ($\{P\}t\{Q\}$) non reconnus par les deux autres constructeurs, au moyen d'une implication logique ($P \Rightarrow P'$ et $\{P'\}t\{Q\}$ est valide). Pour le troisième constructeur, il est donc nécessaire de déduire l'implication entre P et P' par un système d'inférences tel que la *déduction naturelle*. Pour déduire cela, Coq fournit un environnement de preuve pour les objets de type `Prop`. Si nous souhaitons en bénificier, nous devons donc traduire cette implication dans le type `Prop`. Ceci est possible en utilisant les fonctions récursives de `Expr` vers `Z` et `Formule` vers `Prop` suivantes :

```
Fixpoint Trad_expr (E : Expr) : Z :=
```

```
    match E with
    | E_var (p_var id x) => x
    | const c => c
    | add e1 e2 => (Trad_expr e1) + (Trad_expr e2)
    | sub e1 e2 => (Trad_expr e1) - (Trad_expr e2)
    end.
```

```
Fixpoint Traduction (F : Formule) : Prop :=
   match F with
     | atome p => match p with
                    | equal e1 e2 => (Trad_expr e1) = (Trad_expr e2)
                    | less e1 e2 => (Trad_expr e1) < (Trad_expr e2)
                  end
     | bot => False
     | conj F1 F2 => (Traduction F1) /\ (Traduction F2)
     | disj F1 F2 => (Traduction F1) \/ (Traduction F2)
     | impl F1 F2 => (Traduction F1) -> (Traduction F2)
     | not F' => ~(Traduction F')
  end.
```

La définition de `Triplet` se décline donc comme ceci :

```
Inductive Triplet : Formule -> Transition -> Formule -> Prop :=
  | t_guard : forall (P g: Formule), Triplet P (guard g) (conj P g)
  | t_asg : forall (P : Formule) (v : prog_var) (e : Expr),
            Triplet (subst P v e) (asg v e) P
  | t_strong : forall (P P' Q: Formule) (t : Transition),
               (((Traduction P) -> (Traduction P')) /\ (Triplet P' t Q))
               -> (Triplet P t Q)
```

Le type `Triplet`, en plus d'être paramétré est *dépendant*. En effet, chaque constructeur dépend des valeurs de certains termes de type `Formule`, `prog_var`, `Expr`, *etc.* Le type d'un terme construit avec le constructeur `t_guard` dépend de deux termes de type `Formule P` et `g` et en ce sens ne construit pas des triplets (des termes de type `Triplet F1 g F2`) pour tout couple de formules `F1` et `F2` et pour toute garde `g`. Cette définition des triplets de Hoare permet de construire uniquement des triplets valides. Elle capture donc la sémantique de Hoare.

Regardons quelques exemples afin de saisir la notion de type paramétré et les familles de types qui en découlent. Soit `f2` $\stackrel{def}{=} y < 1$ et `f3` $\stackrel{def}{=} x = 0$, modélisés en Coq par :

```
Let f2 := atome (less (E_var var_y) (const 1)).
Check f2.
 f2 : Formule
```

```
Let f3 := atome (equal (E_var var_x) (const 0)).
Check f3.
 f3 : Formule
```

D'après notre modélisation de la sémantique axiomatique, le triplet de Hoare $\{x = 0\}?(y < 1)\{x = 0 \wedge y < 1\}$ modélisé par le type `Triplet f3 (guard f2) (conj f3 f2)` contient l'élément `t_guard f3 f2` :

```
Check t_guard f3 f2.
 t_guard f3 f2 : Triplet f3 (guard f2) (conj f3 f2)
```

L'objet `t_guard f3 f2` agit donc comme *témoin* du type `Triplet f3 (guard f2) (conj f3 f2)` et d'après la correspondance de Curry-Howard, `t_guard f3 f2` peut être vu comme une preuve du théorème `Triplet f3 (guard f2) (conj f3 f2)`.

En revanche le type `Triplet f3 (guard f2) (not f2)`, ne contient aucun élément car ni `t_guard`, ni `t_asg` ni `t_strong` n'est en mesure de construire un objet de ce type. Ceci est rassurant car le triplet de Hoare, $\{x = 0\}?(y < 1)\{\neg(y < 1)\}$ modélisé par cette objet Coq n'est évidemment pas valide.

À contrario, le triplet $\{x = 0\}y := x\{y < 1\}$ modélisé par le type `Triplet f3 (asg var_y (E_var var_x)) f2` ne contient aucun élément constructible par l'un des deux premiers constructeurs du type `Triplet`, mais est pourtant valide. Un élément de ce type peut être construit grâce au constructeur `t_strong` qui exige de démontrer une implication logique via *une formule tiers*, $x = 0 \Rightarrow x < 1$. Un tel élément peut être `t_strong f3 (atome (less (E_var var_x) (const 1))) f2 (asg var_y (E_var var_x))`. L'objet de type `Prop`, `f3 -> (atome (less (E_var var_x) (const 1)))` doit donc être démontré pour valider le triplet $\{x = 0\}y := x\{y < 1\}$.

2.1.1.3 Preuves de corrections.

Si nous souhaitons prouver la correction de l'instruction $y := x$ ayant comme pré-condition $x = 0$ et comme post-condition $y < 1$, il nous faut prouver que :

1. $\{x < 1\}y := x\{y < 1\}$ est un triplet valide ;
2. La formule propositionnelle $x = 0 \Rightarrow x < 1$ est valide.

Déduction dans le système Coq. Le système Coq permet de prouver de tels objets (formalisés dans le type `Prop`) au moyen d'un environnement de preuve délimité par les mots clés `Proof` et `Qed`. Les commandes placées entre ces deux mots clés sont des tactiques de raisonnement indiquant à Coq comment procéder pour déduire et prouver la validité logique de l'objet souhaité. Une tactique formalise un (ou plusieurs) pas de raisonnement formelle de système d'inférence comme la déduction naturelle, ou de réécriture. L'objet à prouver peut être déclaré par les mots clés `Theorem` ou `Lemma` permettant de nommer les démonstration afin de les utiliser dans d'autres démonstration, ou simplement `Goal` pour les démonstrations *anonymes*. Ainsi, si nous souhaitons prouver la validité du triplet $\{x < 1\}y := x\{y < 1\}$ nous pouvons donner la démonstration anonyme suivante :

```
Goal Triplet f3 (asg var_y (E_var var_x)) f2
```

```
Proof.
  apply (t_strong _ (subst f2 var_y (E_var var_x)) _ _)
  split.
  simpl.
  omega.
  apply (t_asg f2).
Qed.
```

Détaillons un peu les étapes de cette preuve. La première tactique employée est `apply` avec en argument le constructeur `t_strong`. `apply` permet d'appliquer une hypothèse ou un constructeur (de type compatible) avec le but à prouver. Ici, `apply` est utilisée avec le constructeur de `Triplet`, `t_strong`. Ce constructeur prend en arguments trois objets de type `Formule` et une `Transition`, soit quatre arguments. Trois d'entre eux peuvent être (et sont) inférés par le système Coq à partir du but à prouver et sont donc implicitement représentés par "_". L'objet ne pouvant être inféré est la `Formule P'` (cf la définition de `t_strong`) car celui-ci n'apparaît ni dans le résultat du constructeur, ni dans le but à prouver. Dans notre exemple, la `Formule` qui instancie `P'` est `subst f2 var_y (E_var var_x)` ($(y < 1)[y/x]$) qui est équivalent à `atome (less (E_var var_x) (const 1))` ($x < 1$) aprés réduction de la fonction `subst`. Si nous adoptons cette représentation ($(y < 1)[y/x]$ en l'occurrence) de cette `Formule` c'est pour que le constructeur `t_asg` puisse reconnaître le triplet de Hoare que `t_strong` mène à prouver, car sa définition utilise la fonction `subst`. Le résutat de la tactique `apply (t_strong _ (subst f2 var_y (E_var var_x)) _ _)` conduit donc à prouver une `Prop` conjonctive,

```
1 subgoal
====================================(1/1)
(Traduction f3 -> Traduction (subst f2 var_y (E_var var_x))) /\
Triplet (subst f2 var_y (E_var var_x)) (asg var_y (E_var var_x)) f2
```

que nous séparons en deux sous-buts par la tactique `split`, qui modélise la règle d'inférence $\wedge_i$ de la déduction naturelle (*cf* figure 1.2) :

```
2 subgoals
====================================(1/2)
Traduction f3 -> Traduction (subst f2 var_y (E_var var_x))

====================================(2/2)
Triplet (subst f2 var_y (E_var var_x)) (asg var_y (E_var var_x)) f2
```

Pour prouver le premier sous-but, nous utilisons la tactique `simpl` qui réduit l'expression qui constitue le sous-but courant. Dans notre exemple, `simpl` réduit les fonctions `Traduction` et `subst`, ce qui simplifie le sous-but à

```
====================================(1/2)
x = 0 -> x < 1
```

qui est l'implication traduite dans le type `Prop`. Afin de prouver cette implication nous utilisons la tactique `omega` qui décide l'arithmétique de Presburger.

Cela nous amène à prouver le second sous-but :

```
1 subgoals
==================================(1/1)
Triplet (subst f2 var_y (E_var var_x)) (asg var_y (E_var var_x)) f2
```

Si nous suivons le script de la preuve, la tactique suivante est de nouveau la tactique `apply`, mais cette fois-ci utilisée avec le constructeur `t_asg`. Tous les arguments de `t_asg` peuvent être inférés par le système car ils apparaissent explicitement dans le sous-but. La validité du triplet est donc démontrée par définition du constructeur `t_asg`. Lorsque la preuve est terminée l'assistant à la preuve retourne `Proof completed` qui annonce que Coq valide la démonstration.

2.1.1.4 Calcul de plus faible pré-condition

Nous finissons cette présentation du système Coq, comme nous avons fini le chapitre précedent, en définissant le calcul de plus faible pré-condition. Comme nous l'avons expliqué dans le chapitre d'introduction, il existe un autre moyen de définir la sémantique des programmes par un système d'inférences, au moyen d'un tranformateur de predicats. Ainsi, il est possible de redéfinir le type `Triplet` en utilisant une modélisation Coq du calcul de plus faible pré-condition (nous nous limitons aux instructions des systèmes de transitions) :

```
Fixpoint wp (P : Formule) (t : transition) : Prop :=
  match t with
  | guard g => (Traduction g) -> (Traduction P)
  | asg v e => Traduction (subst P v e)
  end.
```

Ainsi le type `Triplet` peut modéliser la règle d'inférence,

$$\frac{\vdash_{\text{NJ}} P \Rightarrow wp(Q,t)}{\vdash_{\text{H}} \{P\}t\{Q\}}$$

au moyen du constructeur `t_wp` reconnaissant tout les triplets de Hoare $\{P\}t\{Q\}$ tels que $P \Rightarrow wp(Q,t)$ soit démontrable dans la déduction naturelle :

```
Inductive Triplet : Formule -> Transition -> Formule -> Prop :=
  | t_wp : forall (P Q : Formule) (t : transition),
          ((Traduction P) -> (wp Q t)) -> (triplet P t Q).
```

Ces deux définitions du type `Triplet` sont équivalentes, dans le sens où elles admettent les mêmes éléments *i.e.* les triplets de Hoare démontrables sont les mêmes dans une définition que dans l'autre.

L'avantage d'utiliser le calcul de plus faible pré-condition (plutôt que le système d'inférence de la logique de Hoare) dans une modélisation Coq, vient du fait que si l'on souhaite étendre la logique de Hoare, de manière à pouvoir traiter des affectation de tableaux par exemple (voir section 2.1.2.2), il est plus pratique de modifier la fonction `wp` que de modifier le type `Triplet`.

2.1.1.5 Conclusion

Le Coq permet donc modéliser programmes et spécifications de programmes, puis de prouver interactivement avec l'utilisateur que les programmes respectent leur spécification. Dans la modélisation que nous avons donnée, la spécification des programmes est modélisée par des triplets de Hoare, ce qui permet de construire ces preuves en suivant la méthode de Floyd-Hoare [Flo67, Hoa69].

Un autre avantage que présente l'utilisation du système Coq pour certifier la correction des programmes, est que son vérificateur de preuves satisfait le critère de *de Bruijn*, *i.e.* le noyau du vérificateur de preuves est de petite taille. Utiliser Coq comme vérificateur de preuves, permet d'atteindre un niveau de confiance dès plus élevé car la validité de la certification nécéssitera de faire aveuglement confiance uniquement à ce noyau de petite taille, maintes fois vérifié.

Pour étendre cette modélisation de la logique formelle à l'ordre supérieur nous avons repris les travaux de thèse de Pierre Corbineau, concernant les preuves reflexives [Cor05]. Cela permet de modéliser des formules dont les variables sont quantifiées, dans un type `Formule` sur lequel il est possible d'effectuer des substitutions.

2.1.2 Raisonner sur les programmes impératifs dans Coq

Les programmes impératifs décrivent des séquences d'instructions modifiant l'état de la mémoire de la machine qui l'exécute. Ainsi, les programmes impératifs manipulent, outre les variables scalaires auxquelles est alloué un espace mémoire minimal, des tableaux auxquels est allouée une sous partie de la mémoire ou des structures de données mutables (les pointeurs) permettant de contrôler avec la plus grande précison les adressages mémoires effectués par les programmes. Ces structures de données compliquent les raisonnements sur les programmes, dans le sens où la logique de Hoare, telle que nous l'avons présentée, n'est ni correcte, ni complète en présence de ces structures de données. Nous présentons ici des solutions proposées pour raisonner formellement, dans le système Coq, sur de tels programmes.

2.1.2.1 Modélisation des tableaux dans le système Coq

Une façon de modéliser les tableaux dans le système Coq, est d'utiliser des fonctions des entiers vers les entiers.

```
Definition array := Z -> Z.
```

Nous avons vu dans la section 2.1.1.1 précédente que pour distinguer deux variables entre elles, celles-ci doivent être modélisées par un doublé comprenant un numéro d'identifiant et la valeur de cette variable. Une fois que la sémantique de transformation de prédicats nous a ramenés vers la logique du premier ordre, seule la représentation symbolique des variables nous importera. Ainsi (nous avons également vu que) la fonction de traduction de `Formule` vers `Prop` traduit la variable `var_x` $\stackrel{def}{=}$ `(p_var id x)` en `x`.

Il doit en être de même pour les tableaux. Pour pouvoir décider si deux représentations de tableau désignent le même tableau, nous associons à chaque terme de type `array` un numéro d'identifiant. S'ajoute également à cette définition un second entier précisant la longueur du tableau. De la sorte, nous définissons le type inductif `prog_array` ayant

un unique constructeur `p_array`, modélisant *concrètement* les tableaux dans le système Coq ;

```
Inductive prog_array : Type :=
    | p_array : (array * nat * nat) -> prog_array.
```

Le constructeur `p_array` prend un triplet composé d'un terme de type `array` (une fonction des entiers vers les entiers) et deux entiers : un pour identifier le tableau, le second pour définir sa longueur.

Nous intégrons ensuite les tableaux au type des expressions, `Expr`, donné en section 2.1.1.1. Conformément à la définition de la syntaxe des expressions (figure 1.1) du chapitre introductif, nous étendons la définition Coq des expressions pour prendre en compte les variables que représentent les cases d'un tableau. Dans cette définition (ci-dessous), le constructeur d'expressions d'une case d'un tableau est `E_array` et prend en argument un objet de type `prog_array` et une expression de type `Expr` qui sert d'indice à ce tableau. Du fait que l'indice des tableaux soit de type `Expr`, celui-ci peut être notamment une (autre ou la même) case d'un (autre ou du même) tableau. En revanche, si cette définition permet de construire des tableaux de tableaux, elle ne permet pas de construire des tableaux multi-dimension. Nous avons fait le choix de nous limiter à des tableaux à une dimension dans cette présentation car augmenter la dimension des tableaux ne change rien à la problèmatique que nous allons traiter (les alias de tableaux).

```
Inductive Expr : Type :=
    | E_var : prog_var -> Expr
    | E_array : prog_array -> Expr -> Expr
    | const : Z -> Expr
    | add : Expr -> Expr -> Expr
    | sub : Expr -> Expr -> Expr.
```

Par exemple il est possible de définir le tableau $A[\,]$ des trois façons (imbriquées) suivantes :

1. le tableau $A[\,]$ sous sa forme fonctionnelle : `Let A : array` ;
2. le tableau $A[\,]$ sous le type `prog_array` avec un numéro d'identifiant (4 ici) et une longueur ($A[\,]$ dispose de 20 adresses mémoire) : `Let var_A : prog_array := (p_array A 4 20)` ;
3. le tableau $A[\,]$ sous forme d'une expression, utilisable dans la logique propositionnelle sur les termes que nous avons définis : `Let expr_A : Expr -> Expr := (E_array var_A)`

Cette dernière représentation de $A[\,]$, `expr_A` n'est pas de type `Expr`, mais de type `Expr -> Expr` car elle attend un terme de type `Expr` pour désigner l'une de ses cases. Nous sommes donc passés d'une définition fonctionnelle des entiers vers les entiers, des expressions vers les expressions.

2.1.2.2 Calcul de plus pré-condition correct et complet pour les affectations de tableaux

Avant de présenter une nouvelle définition du calcul de plus faible pré-condition correcte et complète pour les affectations de tableaux, nous illustrons en figure 2.2 et 2.3 les problèmes rencontrés en utilisant le *wp* classique.

La logique de Hoare et la sémantique de transformation de prédicats, dans leur définition initiale [Hoa69, Dij75], consistent dans le cas de l'affectation d'une *variable scalaire* à substituer la variable modifiée par l'expression affectée dans la post-condition. Le langage de programmation que nous souhaitons traiter dans ces travaux permet d'affecter, outre les variables scalaires, des tableaux (dont les indices peuvent être également des tableaux). Dans ce cadre, le calcul de plus faible pré-condition peut conduire à valider des triplets de Hoare non valides (*cf.* figure 2.2) : il ne prend pas en compte les *alias potentiels* entre les différentes variables d'indices d'un même tableau. Ce problème s'observe lorsque deux variables (`i` et `j`) ont même valeur et que la case du tableau indicée par ces variables est modifiée – après l'affectation `A[j] := 0`, que vaut $A[i]$? – (figure 2.2). Ce problème a été résolu, de façon algorithmique, par Richard Bornat [Bor00] à partir de travaux initiaux de J.M.Morris [Mor82], James.A Painter et John McCarthy [MP67] et Hoare lui-même [HW73]. Jean-Christophe Filliâtre a implanté dans l'outil WHY une variante de cette solution [BC04].

La solution consiste à *expliciter tous les alias potentiels* par une instruction conditionnelle, `if (i = j) then A[i] = 0` dans l'exemple précédent. Dans notre modélisation Coq, les tableaux sont représentés par des fonctions de `nat -> nat`. Par exemple, un tableau $A[\,]$ initialisé à 0 est représenté dans le CIC par $\lambda i.0$, soit $\forall i, A[i] = 0$ en logique du premier ordre. Nous donnons ci-dessous une généralisation de ce traitement des alias de tableaux par un calcul de plus faible pré-condition, qui consiste à substituer dans une formule Φ en post-condition, la fonction qui représente le tableau $A[\,]$ par une fonction de même type, explicitant les alias potentiels :

$$wp(A[e] := e', \Phi) = \Phi[A/(\lambda i.\ \texttt{if}\ i = e\ \texttt{then}\ e'\ \texttt{else}\ A[i])]$$

Nous avons implanté dans Coq ce calcul préservant l'équivalence logique de toutes formules, tout en *explicitant* dans la condition du `if` les *alias potentiels* relatifs à la modification d'une case d'un tableau, *i.e.* les *alias* potentiels entre la case du tableau modifié et les cases de ce même tableau qui apparaissent dans la post-condition. Ces alias potentiels sont précisés en établissant un ensemble d'égalités entre la variable d'indice du tableau modifié par le programme (`j` dans notre exemple) et les variables (il peut il y en avoir plusieurs car le tableau peut avoir plusieurs occurrences dans la post-condition) qui indicent ce tableau dans la post-condition (i dans notre exemple) :

```
Let wpa  (F : Formule) (A: prog_array) (e e' :  Expr) : Formule :=
  subst_array F A
     (fun i => if (Z_eq_dec (Trad_expr i) (Trad_expr e)) then e'
               else (E_array A i)).
```

La règle de l'affectation de la sémantique de transformation de prédicats, dans sa version standard, est incorrecte en présence d'alias de tableaux ; elle permet de démontrer le triplet non valide suivant par exemple : $\{\top\}$`A[i] := 0 ; j :=i ; A[j] := 1`$\{A[i] = 0\}$

$$\dfrac{\dfrac{\dfrac{}{\vdash_{\text{NJ}} 0 = 0}\,{=_{refl}}}{\vdash_{\text{NJ}} \top \Rightarrow (A[i] = 0)[A[i]/0]}\,{\Rightarrow_i [H_3]}}{\vdash_{\text{H}} \{\top\}\texttt{A[i] := 0}\{A[i] = 0\}}\,{sem} \qquad \dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{A[i] = 0}^{H_2}}{\vdash_{\text{NJ}} A[i] = 0 \Rightarrow (A[i] = 0)[j/i]}\,{\Rightarrow_i [H_2]}}{\vdash_{\text{H}} \{A[i] = 0\}\texttt{j := i}\{A[i] = 0\}}\,{sem}}{\vdash_{\text{H}} \{\top\}\texttt{A[i] := 0 ; j := i}\{A[i] = 0\}}$$

$$\dfrac{\begin{array}{c}\vdots\\ \vdash_{\text{H}} \{\top\}\texttt{A[i] := 0 ; j := i}\{A[i] = 0\}\end{array} \qquad \dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{A[i] = 0}^{H_1}}{\vdash_{\text{NJ}} A[i] = 0 \Rightarrow A[i] = 0[A[j]/1]}\,{\Rightarrow_i [H_1]}}{\vdash_{\text{H}} \{A[i] = 0\}\texttt{A[j] := 1}\{A[i] = 0\}}\,{sem}}{\vdash_{\text{H}} \{\top\}\texttt{A[i] := 0 ; j := i ; A[j] := 1}\{A[i] = 0\}}\,{seq}$$

La preuve ci-dessus est une application correcte des règles de déduction de la logique de Hoare ; pourtant la conclusion est fausse car après l'affectation `j := i`*, i et j sont égales et un alias est crée entre $A[j]$ et $A[i]$. En conséquence, l'affectation* `A[j] := 1` *modifie la case du tableau $A[\,]$ indicée par j, mais également celle indicée par i (car $i = j$), $A[i]$ en l'occurence. De fait, $A[i]$ ne vaut plus 0. Le problème vient de la règle sem, car la substitution effectuée par wp ne capte pas les alias entre les cases d'un même tableau.*

Fig. 2.2 – Illustration du problème de correction posé par l'affectation des tableaux

La règle de l'affectation de la sémantique axiomatique standard est incomplète en présence d'alias ; elle ne permet pas de démontrer le triplet suivant :

$$\dfrac{\dfrac{\dfrac{}{\vdash_{\mathrm{NJ}} j = j}\,{=_{refl}}}{\vdash_{\mathrm{NJ}} \top \Rightarrow (i=j)[i/j]}\,{\Rightarrow_i [H_2]}}{\vdash_{\mathrm{H}} \{\top\}\texttt{i := j}\{i=j\}}\,{sem} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\textit{non démontrable}}\quad\dfrac{\vdots}{\vdash_{\mathrm{NJ}} A[i]=1}}{\vdash_{\mathrm{NJ}} i=j \Rightarrow (A[i]=1)[A[j]/1]}\,{\Rightarrow_i [H_1]}}{\vdash_{\mathrm{H}} \{i=j\}\texttt{A[j] := 1}\{A[i]=1\}}\,{sem}}{\vdash_{\mathrm{H}} \{\top\}\texttt{i := j ; A[j] := 1}\{A[i]=1\}}\,{seq}$$

L'affectation `i := j` *a pour post-condition l'égalité* $i = j$*. Cette égalité est pré-condition de l'affectation suivante,* `A[j] := 1` *en l'occurrence. La substitution de* $A[j]$ *par* i*, induite par cette seconde affectation, n'a pas d'effet sur la proposition* $A[i] = 1$ *et nous conduit à prouver que* $i = j \Rightarrow (A[i] = 1)$*. Cela n'est pas possible dans la mesure où nous ne disposons pas d'hypothèse sur* $A[i]$*, ni sur* $A[j]$*.*

Fig. 2.3 – Illustration du problème de complétude posé par l'affectation des tableaux

La fonction `wpa` substitue donc dans la formule de `F` toutes les occurrences du tableau `A` par la fonction $\lambda i.$ `if` $i = e$ `then` e' `else` $A[i]$ de type `Expr -> Expr` dans son implantation ci-dessus. Cette substitution s'effectue grâce à la fonction `sub_array` suivante, qui remplace toutes les occurrence de A sous forme d'expression (des termes de type `Expr -> Expr` comme nous l'avons vu) par une fonction `f : Expr -> Expr` :

```
Fixpoint subst_array (F: Formule) (A: prog_array) (f: Expr->Expr) : Formule :=
         match F with
           | bot => bot
           | atome a => atome (subst_array_in_pred a A f)
           | conj F1 F2 => conj (subst_array F1 A f) (subst_array F2 A f)
           | disj F1 F2 => disj (subst_array F1 A f) (subst_array F2 A f)
           | impl F1 F2 => impl (subst_array F1 A f) (subst_array F2 A f)
           | not F' => not (subst_array F' A f)
         end.
```

avec,

```
Fixpoint subst_array_in_expr (E: Expr) (A: prog_array) (f: Expr->Expr) : Expr :=
         match E with
           | E_var (p_var id x) => E
           | E_array (p_array a id l) e =>
                  match A with
                    | p_array a' id' l' => if (eq_nat_dec id id') then
                                             f (subst_array_in_expr e A f)
```

```
                                        else subst_array_in_expr e A f
            end
  | const c => const c
  | add e1 e2 => add (subst_array_in_expr e1 A f)
                     (subst_array_in_expr e2 A f)
  | sub e1 e2 => sub (subst_array_in_expr e1 A f)
                     (subst_array_in_expr e2 A f)
 end.
```

La fonction `sub_array_in_expr`, dans le cas où l'expression `E` est un tableau, est récursive sur l'indice `e` pour gérer les tableaux de tableaux, si jamais `e` porte sur le même tableau ($A[e]$ avec $e \stackrel{def}{=} A[i]$ par exemple).

Nous donnons pour finir la fonction `sub_array_in_pred` qui fait le lien entre les deux fonctions précédentes :

```
Fixpoint subst_array_in_pred (p: Pred) (A: prog_array) (f: Expr->Expr) : Pred :=
         match p with
           | equal x y => equal (subst_array_in_expr x A f)
                                (subst_array_in_expr y A f)
           | less x y => less (subst_array_in_expr x A f)
                              (subst_array_in_expr y A f)
         end.
```

Ainsi, nous pouvons intégrer à notre fonction `wp`, qui implante le calcul de plus faible pré-condition, ce traitement logique de l'effet des affectations de tableaux. Nous redéfinissons `wp` comme suit :

```
Fixpoint wp  (F : Formule) (t : transition) : Formule :=
      match t with
        | guard g => impl g F
        | asg v e => subst F (var v) e
        | asg_array (a,e) e' => wpa F a e e'
      end.
```

où `asg_array (a,e) e'` modélise une transition de la forme $A[e] := e'$.

Exemple 2.1.1. *Nous montrons dans cet exemple comment le triplet non démontrable avec le wp classique de la figure 2.3 peut être prouvé dans le système* Coq *avec la fonction* `wp` *que nous venons de redéfinir. Nous nous concentrons sur la deuxième affectation en supposant que la première (*`i := j`*) nous a permis de déduire la formule* $i = j$ *:*

```
Let A : array.

(* le tableau A a l'identificateur 4, et contient 20 adresses memoire *)
Let var_A := (p_array A 4 20) .

Let var_i := (p_var i 2 nat).
Let var_j := (p_var j 3 nat).

(* i = j *)
Let F1 := atome (equal (E_var var_i) (E_var var_j)).

(* A[i] = 1 *)
Let F2 := atome (equal (E_array var_A (E_var var_i)) (const 1)).

(*{i = j}A[j] := 1{A[i] = 1} *)
Goal triplet F1 (asg_a (var_A, (E_var var_j)) (const 1)) F2.
   Proof.
      apply t_gen.
      simpl.
      intro.
      case Z_eq_dec; simpl.
      intro. auto.

      simpl in H.
      intro. intuition.
   Qed.
```

Nous detaillons quelques étapes de cette preuve. Après l'application du constructeur `t_gen` *sur le but initial, nous devons prouver le sous-but suivant :*

```
1 subgoals
==================================(1/1)
Traduction F1 ->
Traduction (wp F2 (asg_array (var_A, var var_j) (const 1)))
```

La tactique `simpl ;intro` *que nous appliquons ensuite, exécute la fonction* `Traduction` *et transforme la partie gauche de l'implication de ce sous-but en un terme purement logique de type* `Prop` *qui est immediatement introduit en hypothèse. Puis cette tactique reduit la fonction* `wp` *à son code appliqué aux termes* `F2`, `var_A`, `(var var_j)` *et* `(const 1)`. *Cela nous conduit donc à appliquer la fonction* `wpa` *que nous venons de définir.* `wpa` *fait appel à la fonction* `sub_array` *qui trouve dans* `F2` *une occurrence du tableau* `var_A` *et la remplace par le test permettant de traiter les alias potentiels :*

```
1 subgoals
H : i = j
==================================(1/1)
```

```
Trad_expr (if Z_eq_dec i j then const 1 else E_array var_A (var var_i))
 = 1
```

Ce nouveau sous-but est traité par la tactique `case Z_eq_dec ;simpl` *qui effectue une analyse par cas selon le test* `if Z_eq_dec i j`*, réduit les termes obtenues dans la foulée, et retourne deux sous-buts :*

```
2 subgoals
H : i = j
====================================(1/2)
i = j -> 1 = 1

====================================(2/2)
i <> j -> A i = 1
```

Le premier sous-but est prouvé trivialement par la tactique `trivial` *et le second est prouvé par l'absurde grâce à la tactique* `intuition` *qui exploite une contradiction entre* `i = j` *et* `i <> j`*.*

2.1.2.3 Données mutables et logique de séparation

Les programmes que nous souhaitons certifier ne manipulent pas de données mutables de type pointeur d'adresse. Malgré cela, afin de présenter le plus largement possible les travaux réalisés autour de la certification de programmes, nous dirons quelques mots concernant le traitement des pointeurs. Les travaux de Adam Chlipala, Greg Morrisett *et al* [CMM+09] ont pour objectif de certifier des programmes impératifs avec pointeurs au moyen de l'assistant à la preuve Coq.

Ces travaux sont basés sur le système Ynot [NMB06, NMB08, NMS+08], qui permet de prouver la correction des programmes manuellement en offrant une large librairie de tactiques Coq, facilitant la construction des preuves, en présence de pointeurs d'adresse notamment. Cette librairie implante, au delà de la logique de Hoare standard, une *logique de séparation* [Rey02, CIO00, OY02, O'H04] accompagnée de tactiques permettant de démontrer des propriétés sémantiques de programmes sur des parties disjointes de la mémoire. La logique de séparation est une extension de la logique de Hoare. Elle introduit un nouvel opérateur logique $\phi * \varphi$, nommé *conjonction spatiale*, qui formalise le fait que ϕ et φ sont des prédicats valides sur des parties disjointes des adresses mémoires allouées par le programme. Cet opérateur est accompagné de *la frame rule* qui formalise dans le système d'inférence cette séparation de la mémoire :

$$\frac{\vdash_{\text{H}} \{\phi\}\mathsf{p}\{\psi\}}{\vdash_{\text{H}} \{\phi * \varphi\}\mathsf{p}\{\psi * \varphi\}}\ \textit{frame}$$

Grâce à la logique de séparation, il est possible de raisonner localement sur l'effet d'une affectation sur la mémoire et ainsi de garantir que le reste de la mémoire n'est pas modifié par cette affectation.

Ces travaux rejoignent ceux de Xavier Leroy *et al* [Ler06, AB07], qui ont dû utiliser cette logique pour certifier dans Coq un compilateur de programmes C vers un langage assembleur. Nous y reviendrons en section 2.3.1.

2.1.3 Conclusion

Tout ceci nous fournit donc un cadre pour certifier les programmes impératifs dans le système Coq, mais pas de réelle approche pour construire des certificats efficacement. Nous sommes en mesure de raisonner logiquement sur des programmes manipulant des tableaux et de faire valider ces raisonnements par le système Coq, dans lequel nous avons confiance. Néanmoins, construire les preuves intéractivement avec l'assistant à la preuve de Coq est fastidieux. La suite de ce chapitre recense différentes méthodes proposées au cours de la dernière décénie permettant de faciliter la construction de ces certificats.

2.2 Approche pour la certification de programmes impératifs

Une approche incontournable concernant la certification de programmes impératifs – utilisant le système Coq – est celle proposée par Jean-Christophe Filliâtre [Fil99], implantée par la suite dans la plateforme why [Fil03, FM04, FM07].

why est à notre connaissance le premier outil dédié à la certification de programmes. L'approche utilisée par ce méta-programme consiste à générer des *obligations de preuves* à partir de programmes impératifs annotés. L'intérêt de ces obligations de preuves est de simplifier le problème que pose la certification de programmes en ramenant leur spécification à des formules logiques du premier ordre. Cet outil permet de passer d'un problème concernant la sémantique des programmes, basé sur leur syntaxe, à une problématique purement logique. Dit comme cela, why ne semble être qu'une implantation de la sémantique de transformation de prédicats. En réalité, why est plus que cela et comprend un langage intermédiaire vers lequel différents langages de programmation peuvent être traduits. Ce langage intermédiaire se nomme également **Why**, et permet de spécifier la sémantique des programmes écrits initialement dans des langages différents (c, java, *etc*), dans une représentation fonctionnelle. Un programme **Why** est un ensemble de fonctions annotées de pré et post-conditions [FM07]. Ce langage ne gère pas d'alias entre ces variables et n'autorise donc pas les pointeurs d'adresse. En revanche les programmes sources traduits en **Why** pour être certifiés peuvent manipuler des tableaux : le calcul de plus faible pré-condition utilisé pour générer les obligations de preuves intégre la solution pour gérer les alias de tableaux présenté en section 2.1.2.2.

En amont la plateforme why utilise des outils d'analyse statique et de traduction, permettant de spécifier la sémantique des programmes à certifier, puis de les traduire vers le langage intermédiaire **Why** facilitant le calcul des obligations de preuves. Cela rend cette approche portable et lui permet de certifier des programmes écrits dans autant de langages sources que de traducteurs existant vers ce langage intermédiaire.

Pour établir qu'un programme est correct, il suffit de prouver les obligations de preuves que le coeur de why génère. why propose un large éventail de formats de sortie de ces obligations de preuves, afin que celles-ci puissent être prouvées interactivement (en utilisant Coq, pvs, Isabelle,...) ou automatiquement (en utilisant des prouveurs automatiques tels que Simplify, Alt-Ergo, ou d'autres SMT-*solvers*).

Nous donnons quelques détails concernant la réalisation de cette méthode de certification, pour présenter les différentes étapes à suivre pour certifier un programme efficacement dans le système Coq.

2.2.1 Traduction des programmes vers le langage Why

Le langage intermédiaire **Why** de cette approche est un *langage While* relativement simple, n'autorisant pas les alias entre variables et a une structure fonctionnelle. La traduction de programmes définis dans des langages impératifs consiste donc à traduire les programmes par un ensemble de fonctions. Ceci est une idée bien connue et est à la base de ce que l'on nomme la *sémantiques dénotationelle* : à chaque programme est associée une fonction mathématique décrivant son effet.

Cette traduction ne consiste pas seulement à opérer une translation d'un langage vers un autre mais à integrer au code du programme source sa spécification (pré et post-condition, invariant de boucle, *etc*), induite des annotations, dans sa traduction fonctionnelle. Le langage **Why** permet donc d'exprimer dans un même terme, un programme et sa spécification en logique formelle.

Cette analyse nécessite et dépend d'une annotation des programmes décrivant sous forme textuelle (des commentaires avec une syntaxe bien définie dans le code source) leurs effets attendus. Pour la certification de programme C, WHY utilise l'outil Frama-C [CSB+09, CCM09], effectuant cette analyse et cette traduction.

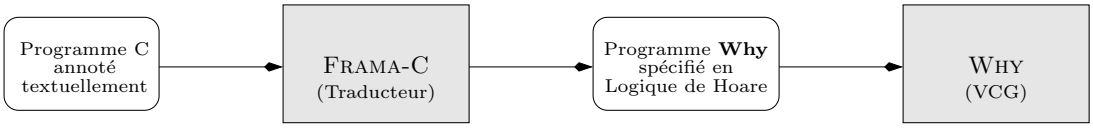


Fig. 2.4 – Traduction de programmes C vers des programmes **Why**

Pour ne pas se limiter à la certification de programmes C, cette traduction fonctionnelle à été défini de façon trés générique en recourant à la notion de monade [Mog89]. Les monades permettent d'introduire des opérateurs génériques pour manipuler des instructions impératives dans un cadre fonctionnel. Ainsi, il est également possible de certifier des programmes JAVA grâce à WHY via l'outil Krakatoa [FM07], analogue à Frama-C.

2.2.2 Obligations de preuve et correction des programmes

Nous considérons que les obligations de preuve sont formalisées dans le CIC afin que celles-ci puissent être prouvées interactivement dans le systéme Coq.

Chaque effet du programme, modélisé par un triplet de Hoare, est par conséquent traduit par un objet du CIC. Prouver chacun de ces objets permet de garantir que le programme se comporte conformément à sa spécification.

Afin de faciliter ces preuves, les effets du programme sont reformulés par des propositions de la logique du premier ordre. Ces formules sont obtenues automatiquement grâce au calcul de plus faible pré-condition. Les obligations de preuve ainsi générées par WHY sont donc de simples formules du premier ordre, qui dans le CIC sont modélisées par des objets du type `Prop`, propice à être démontrés dans le système Coq. Plusieurs algorithmes non triviaux ont été certifiés à l'aide de cette méthode (*Find, Quicksort, Heapsort*, algorithme de *Knuth-Morris-Pratt*, *etc*).

L'outil Why a donc pour effet de simplifier le *problème de certification* en transformant des triplets de Hoare en formule du premier ordre. Cette étape à été certifiée

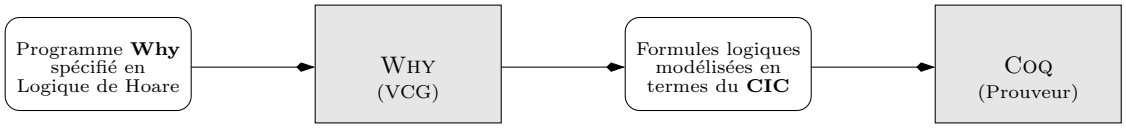


Fig. 2.5 – Génération d'obligations de preuve à partir de programmes **Why** vers le système Coq

en Coq par une preuve de la correction de l'implantation du calcul de plus faible de précondition pour le langage intermédiaire **Why**.

2.2.3 Conclusion sur l'approche du système Why

L'utilisation d'un outil dédié à la certification tel que why permet donc de certifier efficacement qu'un programme impératif (écrit en c notamment) satisfait les comportements attendus et annotés par l'utilisateur. Cependant, le niveau de confiance que l'on peut accorder à cet outil est dépendant à la fois de la correction des outils de traduction vers le langage **Why** et d'une implantation correcte de l'outil why générant les obligations de preuves. Ce que nous entendons par là, c'est que si l'outil permettant de traduire le programme source vers sa représentation **Why** spécifiée est bogué, tout le reste du processus de certification sera erroné. De plus, si les obligations de preuve sont générées de sorte à être prouvées automatiquement, la finalité du verdict repose une nouvelle fois sur la confiance que l'on accorde à ces prouveurs automatiques. Ceux-ci ne sont pas aussi fiables que des outils tels que Coq et il serait souhaitable qu'ils puissent gérerer des objets validables par machine, au moyen d'un de ces outils dignes de confiance, plutot que de répondre par *"Oui"* ou *"Non"*. À notre connaissance, aucune preuve de correction de ces outils n'a été apportée à ce jour.

Pour conclure, cette approche de certification de programmes présente des résultats théoriques très intéressants, et nous indique en quelque sorte la marche à suivre pour certifier efficacement des programmes impératifs. Cependant, la plateforme why, par la complexité des outils qu'elle emploie n'atteint pas en pratique un niveau de confiance optimal. Cela est dû au fait qu'elle suppose de faire aveuglement confiance à des outils de grande taille et complexes.

Nos travaux iront donc dans ce sens : ne pas utiliser d'outils non dignes de confiance, si leurs résultats ne sont pas automatiquement vérifiables par un outil digne de confiance.

2.3 Certification de méta-programmes

Outre les méta-programmes dédiés à la certification tel que why, il existe des méta-programmes dont la certification de leurs résultats peut être cruciale. Par exemple dans le cas de l'avionique, un analyseur statique de programmes peut calculer une propriété sémantique qu'il affirme être satisfaite par un programme de pilotage automatique. Il peut être de la plus haute importance de certifier que cette propriété est effectivement satisfaite par le programme, plutôt que de simplement croire qu'elle l'est, en faisant confiance à cet analyseur qui peut être bogué.

Au lieu de certifier que cette propriété est satisfaite par le programme en utilisant

la plateforme why ou directement à l'aide de l'assistant à la preuve Coq, on pourrait souhaiter certifier l'analyseur lui-même une fois pour toutes, car celui-ci sera sans doute utilisé ultérieurement pour calculer des propriétés sur d'autres programmes. Ainsi, si l'analyseur est certifié comme étant correct par le système Coq, il n'est plus utile de se pencher sur la validité des ses analyses : elle sont toutes correctes car nous avons la garantie que l'analyseur n'est pas bogué et est conforme à sa spécification.

Cette approche est également applicable dans le cadre de la compilation de programmes, d'un langage haut niveau vers un langage machine (ou assembleur). La compilation étant la dernière étape avant d'exécuter un programme écrit, vérifié, certifié dans un langage haut niveau, est une phase critique pour qu'une machine se comporte correctement (*i.e.* conformément à sa spécification). Dans le cas de la compilation, le méta-programme ne calcule pas une propriété sémantique pour un certain programme, mais affirme que deux programmes écrits dans des langages différents ont des sémantiques équivalentes (pour des entrées similaires, ils ont des comportements similaires). Cette approche, en se plaçant au niveau de la compilation des programmes en langage machine, consiste à certifier le compilateur, plutôt que de certifier que chacun des codes compilés qu'il produit, est sémantiquement équivalent au code source correspondant.

Cette technique consistant à certifier les méta-programmes, dans le système Coq, pose principalement deux problèmes : Le premier concerne la difficulté que peut poser la construction de la preuve de correction du méta-programme, dans un système formel comme Coq, où tous les pas de raisonnement sont à expliciter. Le second concerne l'expression de la propriété de correction d'un certain méta-programme : les méta-programmes étant des programmes de grande taille et complexes, il est souvent difficile d'exprimer formellement et de façon concise, leurs comportements attendus.

Nous présentons dans la section qui suit, des travaux portant sur la certification d'un compilateur de programmes impératifs vers du code assembleur. Nous choisissons de présenter ces travaux car ils sont accompagnés d'une étude de cas complète illustrant bien les différentes problèmatiques, théoriques et techniques, que quiconque rencontrera s'il se lance dans la certification d'un autre méta-programme (même autre qu'un compilateur).

2.3.1 Certification de compilateurs

Les programmes impératifs, pour être exécutables, sont compilés vers un langage machine. Si ces programmes ont été certifiés au niveau de leur code source, par exemple grâce à la plateforme why, celle-ci ne précise aucunement comment garantir que le code exécutable est conforme à son code source. La problématique qui est posée ici est différente de la précédente (et de la notre également) dans le sens où même si un programme a été certifié – au niveau de son code source (ce que nous souhaitons faire) – elle *doute* encore de la fiabilité de sa compilation en langage machine : « *Est-ce bien mon code source qui a été certifié, qui est exécuté par la machine ?* » [BRMT05]. En abordant cette question, les auteurs des travaux que nous allons présenter, franchissent un pas supplémentaire vers le plus haut niveau de confiance que l'on peut accorder à l'exécution d'un programme, en faisant l'hypothèse suivante :

> *Dans un futur où les méthodes formelles, telle la certification de programmes, seront suffisament efficaces et fiables pour garantir la correction des programmes, la compilation pourra être le point faible de la chaine qui va de la spécification de ces programme à leurs exécutions.*

On suppose donc ici que la représentation de l'algorithme dans un langage *"haut niveau"* impératif est correct (ce qui n'est pas un problème encore parfaitement résolu), et que la compilation de ce programme peut quant à elle engendrer des erreurs.

Les travaux que nous présentons ont été proposés par Xavier Leroy *et al* [Ler06, Ler09] et ont pour ambition de certifier intégralement un compilateur de plusieurs dizaines de milliers de lignes de code. Certifier un compilateur capable de traiter la classe de programmes visée nécéssite de ne fournir qu'une seule preuve de correction (celle du compilateur) pour garantir la validité de la traduction des programmes visés. Ces travaux sont au coeur du projet CompCert et ont pour but de garantir la fiabilité des programmes impératifs en certifiant leur *équivalence sémantique* avec leur représentation en langage machine, après avoir été compilés :

> *Pour tout code source* p*, si le compilateur transforme* p *en un code machine* p' *sans signaler d'erreur de compilation et si* p *a une sémantique bien définie, alors* p' *a la même sémantique que* p*.*

D'un point de vue technique, les sémantiques opérationelles des deux langages (C et PowerPC) ont été modélisées dans le système Coq. Puis, une preuve de l'équivalence entre tout programme et sa version exécutable fut construite grâce à l'environnement de preuve que Coq fourni :

$$\forall \mathsf{p} \in \text{C}, [\![\mathsf{p}]\!]_{\text{C}} \equiv [\![\mathsf{p}']\!]_{\text{PowerPC}},$$

où p' est la traduction de p dans le langage cible, PowerPC en l'occurrence. La propriété de correction prouvée dans ce projet est donc très spécifique aux problèmes posés par la compilation (préservation de sémantique) et sort du cadre qui nous intéresse, mais la technique utilisée pour construire cette preuve est quant à elle significative de ce qu'implique une telle entreprise. Nous y reviendrons en section 2.3.3.

Avant de s'attarder sur le côté technique de la certification de méta-programmes, nous présentons d'autres travaux portant sur la certification d'analyseurs statiques par interprétation abstraite.

2.3.2 Certification d'interpreteurs abstraits

Les travaux que nous allons présenter ont également pour but de certifier un méta-programme mais en se plaçant au niveau du code source des programmes traités, dans des langages haut niveau. Ceci fut réalisé par David Pichardie, qui dans sa thèse [Pic05] à spécifié en Coq les fondements de l'interprétation abstraite [CC76, CC77], afin de pouvoir certifier des analyses statiques s'appuyant sur cette théorie. Dans ce cadre, un méta-programme est un analyseur qui prend en entrées des programmes, puis calcule des propriétés sémantiques qu'il prétend être satisfaites par les programmes traités. Nous détaillerons l'interprétation abstraite dans le chapitre suivant (*cf* section 3.2.1).

Cette modélisation des composantes de base de l'interprétation abstraite dans le CIC lui permet de prouver toutes les propriétés qu'une certaine analyse statique de programmes doit satisfaire. Ces travaux poursuivent ceux réalisés par David Monniaux durant son DEA [Mon98]. La certification de la correction des analyses nous assure donc que les propriétés qu'une analyse découvre ou vérifie sont bien satisfaites par les programmes traités.

La thèse de D.Pichardie nous montre que la plus grande difficulté pour construire des analyses statiques certifiées réside dans la spécification de ces analyses, et non dans la réalisation de cette spécification. Ainsi, les fonctions génériques qu'il a développées offrent un cadre, permettant à quiconque de construire dans le système Coq une analyse statique de son choix, certifiée : les fonctions utiles à toute analyse statique sont implantées, certifiées et prêtes à être utilisées pour developper n'importe quelle nouvelle analyse. Certifier une nouvelle analyse demandera de prouver quelques propriétés de terminaison, mais cela est moindre à coté d'une approche consistant à prouver cette analyse implantée dans un langage de programmation commun (c, java, Ocaml, *etc*). De ce fait, ces travaux sont quelques peu différents de [Ler06, Ler09] car ils ne visent pas à certifier un unique méta-programme, mais de permettre de certifier à un coup raisonnable toute une classe de méta-programmes, les interpréteurs abstraits.

Au niveau de la réalisation, la correction des méta-programmes est donc prouvée parallèlement à leur développement. Le langage de programmation choisi pour développer ces analyses statiques est celui de Coq. Et pour des raisons d'efficacité, le mécanisme d'*extraction* du système Coq [PM89b] est utilisé pour pouvoir exécuter du code Ocaml. Nous développons dans la section suivante comment le logiciel Coq est mis à contribution pour construire et certifier ces méta-programmes.

2.3.3 Utilisation du système Coq

L'originalité de certifier les méta-programmes parallèlement à leur développement vient du fait que le système Coq, en plus d'être utilisé comme un assistant à la preuve (sa vocation principale), est utilisé comme langage de programmation. Certifier un méta-programme implanté dans un langage de programmation usuel (c, java, Ocaml, *etc*) ne semble pas pertinent car cela nécessiterait de formaliser en Coq la spécification d'une grande partie des milliers d'instructions de ce méta-programme. Par exemple, certifier le compilateur gcc en utilisant la plateforme why ou en utilisant simplement l'assistant à la preuve du système Coq, n'apparaît, en pratique, pas vraiment faisable.

En revanche, du fait que Coq soit à la fois un langage de spécification, de programmation et un vérificateur de types (qui en fait un vérificateur de preuves) il semble intéressant (et les travaux que nous avons présentés dans les deux sections précédentes le prouvent) de l'employer sous deux facettes :

- Coq *comme langage de programmation* : pour développer dans le langage de programmation de Coq le méta-programme que l'on souhaite certifier. Par exemple, le compilateur de programmes c vers le langage assembleur PowerPC.
- Coq *comme assistant à la preuve* : pour certifier la correction de ce méta-programme. Par exemple, le compilateur développé dans le projet CompCert est certifié grâce à l'environnement de preuve du système Coq.

Une chose essentielle à remarquer est que la seconde étape de ce processus de certification de méta-programmes est facilitée par la première du fait que le code du compilateur est directement exprimé dans le cic. Ou plutôt, que dans le système Coq, tout (preuves et programmes) est modélisé par des termes de λ-calcul.

Si l'on s'attarde sur l'efficacité des méta-programmes ainsi développés dans le langage de programmation de Coq, les résultats ne sont pas trés bons car ce langage, encombré par toute la partie logique, est extrêmement lent à l'exécution. Pour combler ce manque d'efficacité, il est possible d'utiliser le mécanisme d'*extraction* [PM89b] du

système Coq pour obtenir le code du méta-programme dans le langage Ocaml, nettement plus rapide à l'exécution. Ce mécanisme d'extraction de code, permet de certifier un programme dans le système Coq, puis d'en supprimer les parties logiques, afin d'en extraire les parties calculatoires, certifiées par construction. Ceci peut sembler difficile car comme nous l'avons dit, tout est λ-terme dans le système Coq. Cependant ces termes sont typés, et les preuves sont de type `Prop`, alors que les programmes sont de type `Set`. Intuitivement, le mécanisme d'extraction utilise ce typage pour ne conserver que les termes de type `Set` et obtenir des programmes Ocaml (de nombreuses subtilités sont presentéss dans la thèse de Christine Paulin-Mohring [PM89b]). Cependant, l'extraction elle-même n'est pas certifiée.

2.3.4 Conclusion sur la certification des méta-programmes

Cette courte présentation autour de la certification des méta-programmes dans le système Coq nous a permis de montrer deux choses : (1) il est possible de certifier un méta-programme dans un système formel fiable tel que Coq, mais (2) certifier un méta-programme *existants et implanté dans un autre langage* n'est pas une entreprise réalisable et demande d'être ré-implanté dans le système Coq.

Cette technique de certification peut être appliquée plus largement et notamment à des procédures de décisions. Les procédures de décision sont souvent utilisées en démonstration automatique pour vérifier des formules laissées en sous-buts par l'utilisateur, dès lors que celles-ci rentrent dans une théorie décidable. La tactique `omega` du système Coq par exemple, implante une procédure de décision de contraintes linéaires en arithmétique de Presburger. C'est à dire que pour toute formule de cette arithmétique, la tactique `omega` est en mesure de construire un terme, vérifié ensuite par Coq, prouvant que cette formule est satisfiable, si elle l'est. Si la formule n'est pas satisfiable, la tactique `omega` échoue et ne retourne aucun terme. Cependant, la tactique `omega` est à utiliser avec modération, car si la fiabilité de son exécution est garantie par le vérificateur de preuves du système Coq, sa compléxité algorithmique est exponentielle et peut parfois terminer avec un temps totalement déraisonnable.

Les prouveurs automatiques employés par la plateforme why, pour prouver les obligations de preuves, utilisent des procédures de décision, mais celles-ci ne sont pas certifiées. Utiliser le système Coq pour certifier ces prouveurs en utilisant l'approche que nous venons de présenter, augmenterait considérablement le niveau de confiance que l'on peut avoir dans ce méta-programme dédié à la certification de programmes impératifs.

Pour conclure, nous pensons qu'à l'avenir la méthode à suivre pour developper un méta-programme, dont la fiabilité des résultats ne doit pas laisser planer le moindre doute, est celle présentée dans cette section, qui consiste à implanter ce méta-programme dans le système Coq, puis à prouver sa correction dans ce même système. Et enfin en extraire un code exécutable performant par le mécanisme d'extraction offert par Coq.

En revanche une chose essentielle que nous montrent les travaux présentés dans cette section est qu'il est plus facile de certifier un méta-programme ou une procédure de décision en l'implantant dans le système Coq, que de certifier ces mêmes outils s'ils étaient implantés dans un langage tel que c.

Une question alors se pose : que faire des méta-programmes déjà implantés dans d'autres langages que Coq ? Les ré-implanter dans le système Coq n'est évidement pas la réponse la plus pertinente, notamment lorsque l'on sait que le développement de réels

méta-programmes peut prendre plusieurs années. En effet, dans la réalité, les outils qui implantent ces méta-programmes comportent des optimisations que l'on ne souhaiterait pas modéliser dans un développement Coq car ils complexifieraient davantage la preuve de correction et feraient perdre l'intérêt des cadres généraux tel que celui proposé par David Pichardie.

Une autre approche, tente de répondre à cette question et consiste à développer des techniques permettant de prouver à un coût réduit chaque résultat donné par un méta-programme donné déjà implanté dans des langages haut-niveau.

2.4 Certification des résultats des méta-programmes

Dans la dernière approche de certification que nous présenterons dans ce chapitre , au lieu de certifier la correction d'un méta-programme MP, on cherche à certifier chacun de ses résultats en générant pour chaque entrée e une preuve de la correction du résultat $\text{MP}(e)$. La problématique que nous posons ici rejoint celle de la plateforme WHY, mais au lieu de construire un méta-programmme dédié à la certification, il s'agit d'étendre des méta-programmmes de type compilateurs, model-checker ou anlayseur statique, de façon à ce qu'ils génèrent automatiquement des certificats de correction de leurs résultats.

2.4.1 Principe des méta-programmes certifiant

L'idée d'augmenter la confiance que l'on peut avoir dans les outils de validation ou de traduction, par une extension de leur code de manière à ce qu'ils certifient leurs résultats, fut initiée par Amir Pnueli *et al* [PSS98], suivie par George Necula [Nec00] dans le cas de la compilation : au lieu de prouver en amont que le compilateur produit toujours un code exécutable qui implante correctement le code source d'un programme, chaque exécution du compilateur est suivie d'une phase de certification qui garantie que le code généré est conforme. Dans le but de rendre cette technique applicable en pratique, les auteurs de [PSS98, Nec00] considèrent que la clé de ce travail passe par une automatisation totale de la génération de ces certificats.

Kedar Namjoshi appliqua ensuite ce principe dans le cas du model-checking [Nam01], pour vérifier des propriétés de sureté exprimées en logique temporelle. Il proposa une extension d'un model-checker pour passer de la simple vérification, répondant « *valide* » ou « *non valide* », de ces propriétés de logique temporelle à leur certification par des arguments formelles. Cependant, ces certificats ne sont pas formallisés dans un système logique modélisant le fondement des mathématiques comme le système Coq et par conséquent sont acceptés par un vérificateur de preuves *ad'hoc*. De plus, la technique présentée dans ces travaux utilise des algorithmes de *jeu à deux joueurs* qui semblent très spécifiques au model-checker considéré. Toutes ces particularités de la stratégie proposée dans [Nam01] nous empêchent de voir dans quelle mesure cette approche pourrait être mise en oeuvre sur d'autres méta-programmes.

Plus récemment ces travaux ont été repris par d'autres [SYY03, Cha06], portant sur des propriétés sémantiques, formalisables en logique du premier ordre dont les preuves peuvent être vérifiées par des systèmes tels que Coq, plus proches de notre problématique. Nous en décrivons les grandes lignes ci-dessous en section 2.4.3. Mais avant nous présentons l'application initiale visée par l'un des deux articles fondateurs [Nec00] de cette technique

de certification par extension de méta-programmes existants, qui aujourd'hui est encore l'application la plus désireuse de telles techniques de certification.

2.4.2 Méta-programmes certifiants pour le Proof-Carrying Code

Le *proof-carrying code* (PCC pour faire court) a été proposé à la fin des années quatre-vingt-dix par George Necula [Nec97], et a débouchée sur l'emergence d'une communauté scientifique à part entière. La principale problématique de cette communauté, se place du point de vue d'un utilisateur de logiciel souhaitant télécharger du code exécutable fiable via un réseau informatique. L'enjeu est de fournir à cet utilisateur un protocole lui permettant d'exécuter ce code reçu par le réseau, en toute sécurité, *i.e.* sans que le code soit bogué ou qu'il puisse atteindre certaines zones mémoires non désirées. Pour répondre à cela, il est proposé à l'utilisateur de télécharger le code exécutable associé d'une preuve que celui-ci respecte un certain nombre de propriétés souhaitées. La figure 2.6 illustre un protocole de proof-carrying code utilisant un générateur d'obligations de preuves (VCG).

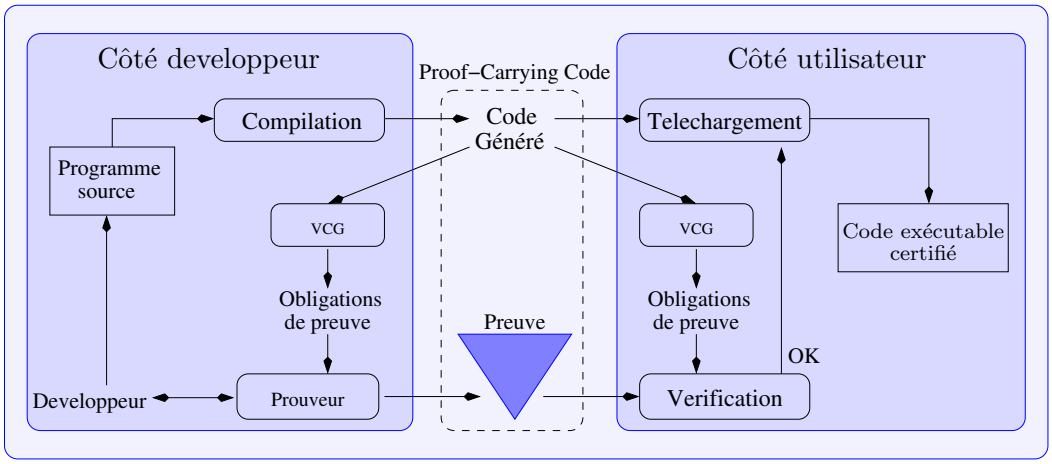


Fig. 2.6 – Schéma classique d'un protocole proof-carrying code

Dans le protocole proposé en figure 2.6 le VCG joue un rôle centrale, car il fournit au développeur les obligations de preuve à fournir pour garantir la fiabilité de son code ; et permet à l'utilisateur de vérifier que les preuves qu'il télécharge avec le code correspondent bien aux propriétés qu'il souhaite. Une difficulté pour le développeur est de construire les preuves qui peuvent être nombreuses. C'est de ce point de vue que les méta-programmes certifiant leurs résultats peuvent être employés, car il est nécessaire de fournir pour chaque code exécutable une preuve de sa correction. Un compilateur certifié n'a pas d'utilité dans ce cadre car l'utilisateur n'a aucune garantie que le développeur a généré son code avec ce compilateur certifié. Un autre problème se pose concernant la fiabilité du protocole dès lors que celui-ci dépend de la correction du VCG. En effet le VCG est un outil de plusieurs milliers de lignes de code, et n'est pas certifié. S'il est bogué et par conséquent génère des obligations de preuves éronnées, les preuves n'auront aucune valeur vis à vis de la bonne exécution du code téléchargé. Afin d'augmenter le niveau de confiance que l'on peut avoir dans la preuve de correction du code exécutable,

Andrew Appel [App01] a proposé le *foundational proof-carrying code*. Contrairement au PCC classique, le FPCC s'abstient d'utiliser un VCG dans un système de type *ad'hoc* et utilise la logique mathématique pour modéliser la sémantique des programmes et construire les preuves de corrections. Ainsi les preuves doivent être construites et vérifiées dans un système tel que Coq, à partir du seul code exécutable. Elvira Albert *et al* proposent une autre version du PCC, où le certificat, au lieu d'être une preuve de correction du programme téléchargé, est une propriété sémantique du programme générée par un interpréteur abstrait [APH04]. Cette solution répond au problème de la génération automatique des certificats mais retrouve la problématique que posait la version initiale du PCC, à savoir qu'il est demandé à l'utilisateur de faire confiance à un outil de grande taille, l'interpréteur abstrait en l'occurrence. Frédéric Besson *et al* [BJP06] ont proposé par la suite une réponse à ce problème en utilisant des interpéteurs abstraits certifiés. En utilisant l'approche proposée par David Pichardie (*cf.* section 2.3.2) l'utilisateur télécharge dès lors, en plus du programme accompagné de son certificat, la preuve de correction de l'interpéteur abstrait, de laquelle il peut extraire (par le mécanisme d'extraction du système Coq) un analyseur statique certifié qui agit ensuite comme un vérificateur de certificat.

Aujourd'hui l'utilisation du proof-carrying code s'est diversifiée et ne porte plus forcément sur du code exécutable. Un utilisateur, ayant à sa disposition un compilateur certifié peut souhaiter télécharger des programmes écrits dans des langages haut-niveaux accompagnés de leur preuve de correction. Utiliser des méta-programmes certifiant trouve une grande utilité du point de vue du développeur, qui peut ainsi certifier son code source automatiquement et joindre la preuve obtenue à son programme. Nous présentons donc maintenant des techniques d'instrumentation permettant à des developpeurs de méta-programmes d'obtenir automatiquement des certificats des résultats de ces méta-programmes.

2.4.3 Génération de preuves guidée par analyse statique

Une autre famille d'approches faisant le lien entre celles présentées plus haut, consiste à étendre les méta-programmes afin que ceux-ci fournissent automatiquement un certificat de leur verdict.

L'idée de cette approche s'inspire l'isomorphisme de Curry-Howard et considère que si un programme a calculé une popriété satisfaite par un autre programme, alors une preuve de cette propriété peut être directement obtenue à partir de ce calcul.

Ces travaux ont été proposés autour du langage Jinja [KN06], un langage de la famille Java et s'inscrit dans la communauté du PCC (*cf.* 2.4.2). Nous consacrons une section à part aux travaux d'Amine Chaieb [Cha06] autour de ce langage, car ceux-ci ont pour idée de base d'étendre un analyseur statique afin que celui-ci génère des certificats pour chacune de ses analyses : considérons qu'une analyse calcule le triplet de Hoare $\{P\}x := e\{Q\}$, alors la méthode qu'il propose permet de construire *automatiquement* et dans *le même temps* une preuve de la formule $P \Rightarrow Q[x/e]$. Les programmes visés par cette *analyse certifiante* sont écrits en *bytecode* Jinja qui est un langage bas niveau. Ces programmes effectuent des affectations sur des structures de données simples, de type variable entière (pas de pointeur, ni de tableau), et s'exécutent séquentiellement.

Étant donné un programme p, écrit dans ce langage, cette approche transforme p en un système de transitions sémantiquement équivalent, exécute une analyse qui associe

à chaque transition du système un invariant exprimé en logique du premier ordre, puis génère une preuve déductive pour chaque invariant. La génération des preuves est guidée par la stratégie de l'analyse et s'appuie sur un calcul de plus faible pré-condition. Il est montré dans [Cha06] comment passer des invariants calculés par un analyseur statique à des obligations de preuve en logique formelle. Cependant, pour prouver les obligations de preuve, il est suggéré d'employer une approche guidée par le code des fonctions de l'analyseur (*syntax-driven approach* en anglais), et cela semble difficilement généralisable.

Ces travaux sont proches de ceux de Sunae Seo *et al* [SYY03] qui suivent le même principe mais en utilisant la logique de Hoare (au lieu d'un calcul de plus faible pré-condition). L'intuition de ces travaux est que *si une propriété concernant la sémantique d'un programme peut être calculée, elle doit pouvoir être démontrée.* En revanche, ils n'expliquent pas non plu clairement *comment* ces preuves peuvent être automatiquement générées en suivant une méthodologie générale. Les auteurs de [SYY03] suggérent à l'utilisateur qui souhaite instrumenter un analyseur statique de développer des stratégies *ad hoc*, permettant de prouver instruction par instruction les invariants calculés par l'analyseur. En d'autre terme, sans préciser comment, leur approche demande de développer une stratégie spécialisée (et donc efficace) permettant de prouver que l'analyse de chaque affectation traitée est correcte, et il en est de même pour les gardes. L'idée est élégante et permet de se dispenser d'utiliser des prouveurs automatiques qui ont souvent une complexité trop lourde pour être exploité en pratique, ou des assistants à la preuve fastidieux à l'usage. Cependant, du fait qu'un travail important d'instrumentation est laissé à la charge de l'utilisateur, sans réellement être guidé, cette approche peut apparaître comme plus difficile à mettre en oeuvre que de prouver les propriétés calculées à l'aide d'un assistant à la preuve.

2.4.4 Conclusion sur les méta-programmes certifiants

Dans le cas de méta-programmes capablent de traiter des programmes écris dans des langages haut niveau, l'approche qui consiste à certifier chacune de ses exécutions, diffère de celle de D.Pichardie, dans le sens où elle n'apporte pas une preuve de correction du méta-programme, mais une preuve de correction pour chaque programme qu'elle traite. Elle diffère également de l'approche de la plateforme WHY, dans la mesure où si WHY certifie aussi le verdict de chaque programme traité, les méta-programmes instrumentés ne demandent pas en revanche de construire une partie des preuves intéractivement avec un outil tiers (tel que Coq), où de faire confiance à des prouveurs automatiques. Cette automatisation plus importante, vient du fait que les méta-programmes instrumentés deviennent des outils de certification spécialisés, alors que WHY est un outil dédié à la certification avec un domaine d'entrée beaucoup plus vaste, se heurtant à des problèmes indécidables.

Pour résumer, le point fort des méta-programmes certifiants leurs résultats est qu'ils ne demandent rien à prouver à l'utilisateur : ni la correction du méta-programme, ni la correction de ses verdicts. Le travail de certification est effectué une fois pour toute par le développeur du méta-programme.

Il réside dans l'extension du méta-programme afin de regrouper et d'assembler les *arguments* lors de son calcul afin de lui permettre de *justifier* son résultat.

2.5 Discussion et direction choisie

Dans ce chapitre, après une brève introduction au calcul des constructions inductives et du système Coq, nous avons présenté why [Fil99, Fil03, FM04, FM07], un méta-programme dédié à la certification de programmes impératifs. L'outil why, qui vise de certifier une large classe de programmes impératifs, écrits dans différents langages, souffre d'un domaine d'entrée trop large et d'une trop grande diversité des propriétés de correction qu'il souhaite pouvoir prouver. Et par conséquent, il souffre d'une automatisation limitée. Pour combler partiellement ce problème, why propose un large éventail de formats d'encodage des obligations de preuve qu'il génère dans les langages de différents prouveurs, afin d'augmenter ses chances de trouver un prouveur capable de prouver les obligations de preuve. Néamoins, dans le cas où ces prouveurs parviendraient à prouver automatiquement toutes les obligations de preuve, aucun de ces prouveurs n'est certifié. Le niveau de confiance de la certification des programmes traités par why n'est dès lors plus optimal.

Nous avons présenté ensuite une technique permettant de certifier des méta-programmes (non dédiés à la certification) dans le système Coq [Ler06, Ler09, Mon98, Pic05]. L'interêt de certifier des méta-programmes, comme les compilateurs ou les interpréteurs abstraits, est d'augmenter le niveau de confiance que l'on peut avoir en leur implantation. Pour réaliser cela, nous avons vu qu'il est préférable d'implanter dans le système Coq le méta-programme que l'on souhaite certifier, afin de prouver plus facillement sa correction. Dans le cas d'interpreteurs abstraits, David Pichardie a developpé une librairie Coq [Pic05] portant sur les bases de l'interprétation abstraite, pour simplifier la conception d'analyseurs statiques certifiés. Cependant ces analyseurs statiques doivent être developpés dans le système Coq pour rendre l'approche efficace. Cela ne permet donc pas la certification de méta-programmes existants, implantés dans d'autres langages (que Coq). Ré-implanter ces méta-programmes dans le système Coq est une solution que l'on souhaiterait éviter.

L'approche qui consiste à certifier chaque résultats d'un méta-programme donné existant [PSS98, Nec00, Nam01] apporte un début de réponse. Elle se fonde sur l'extension de méta-programmes, pour que ceux-ci génèrent automatiquement des certificats de leurs résultats [SYY03, Cha06]. Ici on cherche donc à augmenter le niveau de confiance que l'on peut avoir dans des méta-programmes existants, comprenant de nombreuses heuristiques qui ne jouent aucun rôle dans la correction du résultats ; par exemple, dans le cas des interpréteurs abstraits, les opérateurs d'elargissement [CC92] ne servent qu'à garantir la terminaison de l'analyse. Toutes les parties de l'analyse ne seront donc pas en à prendre en compte pour certifier ses résultats. Vérifier un résultats est (a priori) plus facile que de le calculer, mais la certifcation formelle présente d'autres problèmes, notamment dûs aux contraintes que posent les vérificateurs de preuves tels que Coq.

Nous ne rentrerons donc pas dans le jeu de la comparaison entre les différentes appoches que nous avons présentées car chacune apporte des réponses pertinentes dans des contextes bien différents. Nous justifierons simplement le choix que nous avons fait de suivre celle par *extension de méta-programmes*, du fait qu'elle nous semble nettement moins bien étudiée, et qu'elle apparaît encore difficile à appliquer en pratique. Par "moins bien étudiée", nous voulons dire qu'il nous semble moins clair, à la lecture des articles que nous avons cités, comment cette approche peut être appliquée sur un méta-programme de notre choix. Si ces techniques s'avèrent applicables à un coût modéré

(très inférieur à un devellopement en Coq), un grand nombre de méta-programmes à l'état de prototype pourraient atteindre le niveau de confiance requis pour la validation de systèmes critiques. Les travaux les plus récents à ce sujet portent sur des analyseurs statiques [SYY03, Cha06] et demandent d'instrumenter un partie du code de ces analyseurs trop importantes pour être mis en oeuvre concrètement. Il nous semble que dans de nombreux cas, les fonctions des analyseurs statiques qui necessitent d'être instrumentées pour permettrent à ces analyseurs de générer des preuves de la correction de leurs résultats, peuvent être considérablement réduites. En effet, comme nous l'avons déjà dit, nous pensons qu'il est plus facile de vérifier qu'un invariant est correct que de le calculer. Et par conséquent nous pensons que cette approche est sûrement plus efficace que celle consistant à prouver l'analyseur dans sa totalité.

Le chapitre suivant détaillera cette approche, en raisonnant de façon très générale sur des méta-programmes qui effectuent des calculs d'invariants par analyse statique. Les analyseurs statiques sont de bons candidats à l'instrumentation car toute la partie concernant la recherche des invariants ne sera sans doute pas à instrumenter. Seul compte pour la correction le fait que l'invariant calculé est preservé par les instructions du programme. Nous présenterons ce qui est commun à tout analyseur statique de programmes par interprétation abstraite, et dégagerons l'ensemble minimal de fonctions à instrumenter pour obtenir des analyseurs statiques fournissant des certificats, vérifiables par le système Coq, de la correction de leurs résultats.

Le chapitre 4 étudiera la faisabilité de nos propositions génériques dans le cas très précis d'un prototype d'analyse statique de programmes manipulant des tableaux, développé au laboratoire Verimag.

Chapitre 3

Certification automatique de programmes par instrumentation de méta-programmes

> « *La science, qui a commencé comme la poursuite de la vérité, est en train de devenir incompatible avec la véracité puisque la véracité complète tend de plus en plus à rendre complet le scepticisme.* »
>
> Bertrand Russell

Les travaux présentés dans ce chapitre sont donnés en réponse à une question toute simple que nous nous sommes posés face à la quantité de méta-programmes (model-checkers, compilateurs et analyseurs statiques essentiellement) développés au laboratoire VERIMAG et ailleurs, et dont tout le monde[1] voudraient qu'on leur fasse aveuglement confiance :

> « *Comment certifier formellement la validité des résultats de ces méta-programmes ?* »

Ces résultats ne sont rien de plus que des verdicts concernant le bon fonctionnement de certains programmes, ou bien des propriétés sémantiques calculées sur ces programmes. Prouver la correction de ces méta-programmes en COQ est la première idée qui nous est venue. Face aux dizaines de milliers de lignes qui constituent leur code, nous nous sommes vite découragés. Inspirés par les travaux d'Amir Pnueli et Kedar Namjoshi [PSS98, Nam01] nous sommes partis du principe qu'il est plus efficace et plus réaliste d'étendre un méta-programme existant afin que celui-ci fournisse automatiquement des certificats de ses verdicts, que de certifier directement le code de ce méta-programme.

Nous proposons dans ce chapitre un cadre et une méthodologie qui ont pour finalité de certifier *automatiquement* des propriétés sémantiques de programmes impératifs modélisés par des systèmes de transitions. Ces propriétés sémantiques doivent être exprimables dans une logique formelle modélisable dans le langage du vérificateur de preuves utilisé (le CIC dans notre cas). L'interêt de ce travail est de permettre à chacun de prouver que les verdicts de son méta-programme préféré sont corrects, plutôt que de se limiter à

[1] Leurs développeurs notamment.

croire qu'ils le sont. Cette capacité à pouvoir certifier les résultats des méta-programmes est plus que nécessaire dès lors que l'application visée est critique.

Contributions. Nos contributions présentées dans ce chapite sont d'abord une technique d'instrumentation de code dans le but de justifier les résultats des programmes par des arbres de preuves formelles, vérifiables par machine. La méthodologie que nous proposons s'appuie sur des patrons de preuve (voir définition 3.1.1), qui à l'exécution d'un programme, produiront des preuves qui agiront comme des certificats des résultats calculés. Produire automatiquement des preuves de corrections de programme dans un système tel que Coq est l'un des défis que nous nous sommes fixés. En appliquant cette technique d'instrumentation à des méta-programmes (des programmes qui prennent d'autres programmes en entrées) il est possible d'obtenir des preuves de correction des programmes traités. Les méta-programmes auxquels nous nous intéressons sont les analyseurs statiques par interprétation abstraite et calcul de points fixes. En effet les interpréteurs abstraits sont de bons candidats pour être instrumentés dans le but de certifier leurs résultats, car la plus grande partie de leur stratégie concerne la recherche des invariants et ne nécessite pas d'être instrumentée. Seule la partie concernant la vérification de la stabilité des invariants est utile pour certifier que ces invariants sont corrects [Cha06]. L'instrumentation consiste alors à associer à une partie des fonctions d'un analyseur statique des patrons de preuve qui à l'exécution généreront les certificats. De tels analyseurs instrumentés ne sont pas certifiés puisque leur correction pour toutes leurs entrées possibles n'est pas garantie, néanmoins ils sont capables de justifier par une preuve formelle la correction du résultat de leurs calculs. L'interêt de cette approche est d'augmenter le niveau de confiance d'analyseurs existants, même à l'état de prototypes, sans avoir à certifier leur code qui peut contenir de nombreuse heuristiques. Pour que cette méthodologie soit applicable en pratique, nous avons cherché à limiter le nombre de fonctions à instrumenter. Nous avons dégagé un ensemble restreint de fonctions, commun à tout analyseur, à instrumenter pour qu'un analyseur existant puisse devenir un outil de certification. Nous montrons que notre approche s'applique à toute analyse employant des domaines abstraits dont les éléments s'expriment dans une logique formelle muni d'un système d'inférence. Nous nous attardons plus particulièrement sur la logique du premier ordre dont la signature permet d'exprimer des contraintes d'arithmétique linéaire. Nous illustrons dans ce chapitre notre approche et les problèmes que pose l'instrumentation sur le domaine abstrait des intervalles [CC76]. Le chapitre 4 mettra en pratique les résultats génériques de celui-ci, et présentera l'instrumentation d'un analyseur statique existant non trivial [HP08], calculant des propriétés sémantiques sur le contenu des tableaux de programmes.

La section 3.1 décrit le principe que nous employons pour étendre les programmes afin qu'ils puissent justifier leurs résultats par des certificats formels. Ce principe d'instrumenattion, d'abord illustrée sur un programme très simple et quelconque, sera ensuite précisé sur le cas particulier de l'instrumentation de méta-programmes dont les résultats portent sur la sémantique des programmes qu'ils traitent.

Nous détaillons en section 3.2 cette technique d'instrumentation dans le cadre de l'analyse statique par interprétation abstraite et calcul de point fixe. La section 3.2.1 introduit la théorie de l'interprétation abstraite et l'analyse statique par calcul de point fixe. La section 3.2.3 montre comment une analyse statique peut générer des certificats

de ses résultats en instrumentant un ensemble restreint de fonctions.

3.1 Principe de l'instrumentation de méta-programmes

L'écriture de preuves entièrement formelles est une activité extrêmement fastidieuse et chronophage. Même si les assistants à la preuve présentent beaucoup de caractéristiques intéressantes (comme l'expressivité de leur langage, la fiabilité de leur vérificateur de preuves, etc...), dès lors que les preuves atteignent des tailles trop importantes, ce type de méthodes atteint ses limites ; automatiser devient nécessaire, voire indispensable.

À l'opposé, l'approche consistant à faire valider les programmes au moyen d'autres programmes – des *méta-programmes* – n'apparaît pas suffisamment fiable car elle n'exhibe pas de certificat de cette validation. Il demeure cependant que ces méta-programmes, par leur dimension automatique, offrent une facilité d'utilisation appréciable. L'un de nos objectifs est de préserver cette dimension automatique. De plus, ces programmes au statut particulier (leurs entrées sont des programmes) sont une source d'*"intelligence"* déjà existante et prête à l'utilisation, difficilement méprisable. S'il est possible de les utiliser dans une approche pour la certification de programmes, afin d'économiser la programmation d'algorithmes qu'ils contiennent déjà, nous ne nous en priverons pas.

À la jonction de la preuve assistée par machine et la validation automatique de programmes, existe la démonstration automatique. Démontrer automatiquement des théorèmes exige en pratique de se placer dans une théorie complète (*i.e.* cette théorie n'admet aucun énoncé qu'on ne puisse ni prouver ni réfuter). Malgré la complétude des théories abordées, la démonstration automatique se heurte bien souvent à des problèmes algorithmiques de complexité trop importante (décider la logique propositionnelle est un problème NP-complet par exemple). Une solution pour combler ce manque d'efficacité, consiste à se limiter à la classe des énoncés dont on souhaite démontrer la validité afin de ne pas considérer la théorie dans son ensemble. Les algorithmes peuvent ainsi être définis sur des sous-classes bien *ciblées* de la théorie à laquelle ces sous-classes appartiennent. La méthode que nous proposons pour certifier les programmes va dans ce sens et s'applique sur des méta-programmes existants.

3.1.1 Passage de la validation à la certification grâce à l'instrumentation

Certifier par la preuve, c'est expliquer, justifier, convaincre, se convaincre, comprendre et permettre à l'autre de comprendre. « *Pourquoi un programme a tel comportement ?* » est la question centrale de notre problèmatique. Il existe une infinité d'explications (de preuves) du comportement d'un programme (en fonction de ses entrées). Il est tout à fait possible d'expliquer le *pourquoi* d'une *chose* sans pour autant respecter, reprendre la démarche, le raisonnement, qui y a conduit. Toutefois, s'il est possible de garder une trace du raisonnement aboutissant à une idée, ou plus précisément à l'affirmation qu'un programme satisfait une certaine propriété sémantique, alors l'explication peut être obtenue *quasi* gratuitement : toutes les informations qui lui sont nécessaires sont d'ores et déjà dans le raisonnement initial.

Dans le cadre qui nous intéresse, un certain méta-programme prend un programme en entrée, fait des calculs, puis retourne une propriété qu'il prétend être satisfaite par ce programme (voir figure 3.1(a)). Cette propriété (mathématique, logique, sur les traces,

etc) formalise la sémantique de ce programme et spécifie une partie de ces comportements *invariants*, quelles que soient ses entrées.

Notre volontée n'est pas de prouver la correction du méta-programme mais de prouver que chacune de ses exécutions est correcte. Pour réaliser cela, une approche pourrait consister à developper un outil tiers permettant de construire ces preuves de correction des exécutions du méta-programme. Un tel outil doit être capable de démontrer toutes formules valides appartenant à la classe de propriétés que ce méta-programme est en mesure de considérer. Ceci est très coûteux. Et dans un certain sens revient à reprogrammer une grande partie du méta-programme pour obtenir un tel prouveur automatique.

Au lieu de cela, nous proposons d'*instrumenter* une moindre partie des fonctions du méta-programme en question, celles prenant part à la correction (que nous distinguons de celles dédiées à la terminaison, comme le *widening* dans le cas de l'analyse statique, ou à la précison), de sorte qu'il génère lui même des certificats de ses verdicts : les fonctions instrumentées du méta-programme fournissent une preuve formelle que chacune de leurs exécutions est correcte. Cette technique s'applique à plusieurs niveaux du code du méta-programme, *i.e.* les fonctions appelées par la fonction principale construisent leurs preuves en combinant les preuves des fonctions locales qu'elles appellent. Et ainsi de suite... En *combinant les preuves* produites par les différentes fonctions du méta-programme, la fonction principale du méta-programme construit une preuve de correction de son verdict (Figure 3.1(b)). Cette preuve de correction du verdict constitue au final un certificat de correction du programme traité. Cela achève donc le but initial que nous nous étions fixé.

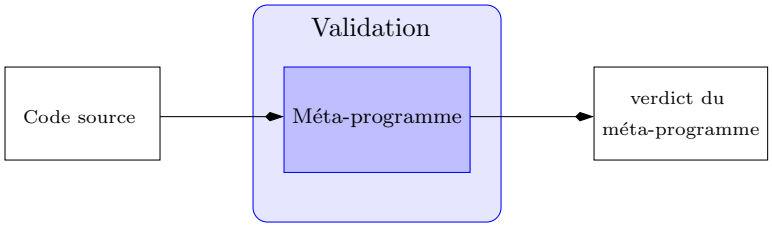
En suivant l'idée initiale d'Amir Pnueli *et al* [PSS98], nous avons généralisé ce principe d'instrumentation afin de générer efficacement des certificats de programmes de manière à pouvoir l'appliquer à des méta-programmes existants et utilisés. L'idée de certifier à la volée les verdicts d'un méta-programme est proche du PCC [Nec97], aux différences près que le PCC (1) utilise un outil tiers (le VCG) afin de génerer des obligations de preuve et (2) ne précise nullement comment construire ces preuves, (3) porte sur un code compilé et (4) n'exploite pas l'intélligence du compilateur par une technique d'instrumentation.

Notre approche peut être vue de deux façons et trouve par conséquent deux types d'utilisation :

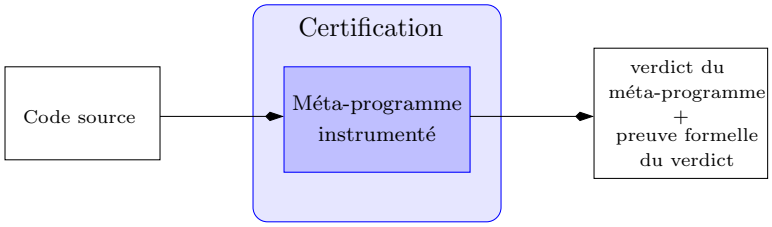
- *Instrumentation de méta-programmes pour certifier automatiquement les programmes qu'ils traitent.*
- *Instrumentation de méta-programmes pour combler un manque de confiance en leurs verdicts : le developpeur du méta-programme peut ainsi utiliser la version instrumentée pour effectuer une campagne de test de son outil, où le vérifiacteur de preuves (*Coq *par exemple) sert d'oracle. Cette utilisation sert plus à déboguer le méta-programme que de certifier les programmes traités.*

3.1.2 Instrumenter pour justifier

L'instrumentation consiste à associer à chaque fonction un ensemble d'arguments formels, (plus ou moins) liés à la définition algorithmique de la fonction, qui lors de son exécution fournira les pas de déduction necessaires à la construction automatique d'une preuve formelle, justifiant le résultat calculé. Dès lors, si notre approche certifie



(a) Vérification formelle de codes sources grâce à un méta-programme



(b) Certification formelle de codes sources grâce à l'instrumenation d'un méta-programme

Fig. 3.1 – Passage de la validation à la certification grâce à l'instrumentation

les programmes traités par le méta-programme instrumenté, en aucun cas elle ne certifie le méta-programme lui même (contrairement aux approches présentées dans [Ler06, Ler09, CJPR05]). C'est pourquoi nous parlons de *justification* de méta-programmes et non de certification. Certifier un méta-programme consiste à garantir son résultat pour n'importe quel programme qui appartient à son domaine d'entrée, en fournissant une unique preuve. La technique que nous proposons est dite *"légère"* et s'avère en pratique plus facile à mettre en œuvre que celles dites *"lourdes"*, qui certifient le code du méta-programme (obtenu par extraction, en général), et se distinguent par les points suivants :

(+) La justification consiste à seulement étendre une partie des fonctions du méta-programme, alors que la certification consiste à prouver la correction dans son intégralité.

(-) La justification demande à ce que chaque preuve des verdicts soit vérifiée alors que la certification n'exige que de vérifier la preuve de correction du méta-programme, une unique fois.

(+) Cette vérification des preuves étant automatisable, par *type-checking* notamment, elle implique que l'intervention humaine dans la justification soit moindre.

(+) Si le méta-programme est bogué, mais retourne parfois des verdicts corrects, la justification pourra malgré tout certifier certains programmes, dans ces cas particuliers, alors que la certification réfutera l'ensemble des verdicts en réfutant la correction du méta-programme.

(+) La justification s'applique à des méta-programmes existants et à des prototypes,

alors qu'il est peu réaliste de certifier des méta-programmes une fois leur implantation achevée.

Les fonctions instrumentées retournent donc un résultat associé d'une justification. Ces justifications ne sont autres que des preuves des propriétés de correction (section 3.1.2.1) des fonctions susmentionnées. Ces preuves, construites dynamiquement sont issues d'une stratégie statique : l'ensemble des pas de déduction nécessaires à la preuve est intégré dans le code des fonctions (section 3.1.2.2).

3.1.2.1 Correction et justification

La correction d'une fonction (ou d'un programme) f s'exprime par deux propriétés P et Q (formalisées par des formules logiques) de ses variables d'entrées et de son résultat relativement à la spécification de f : cela se formalise par le *triplet de Hoare* $\{P\}\mathsf{f}\{Q\}$. Q est une condition que le résultat de f doit satisfaire, alors que P formalise la condition nécessaire pour une exécution correcte de f. Dans de nombreux cas les fonctions peuvent être appelées sans condition particulière. De telles fonctions ont pour pré-condition $P \stackrel{def}{=} \top$.

Pour illustrer la différence entre *certifier et justifier* une fonction, nous considérons dans un premier temps un exemple très simple.

Exemple 3.1.1. *Soit la fonction* $\mathsf{inc}()$*, qui à partir d'un entier naturel* x *calcule et retourne la valeur de* x *plus* 1*. La propriété de correction de cette fonction a pour post-condition* $Q \stackrel{def}{=} 1 \leq x$*, dans le cas où sa pré-condition est* $P \stackrel{def}{=} 0 \leq x$*. La correction de* $\mathsf{inc}(x)$ *s'exprime par le triplet suivant :*

$$\{0 \leq x\}\mathsf{inc}(x)\{1 \leq x\}$$

Ainsi, une preuve de la validité de ce triplet certifie la correction de la fonction inc*.*

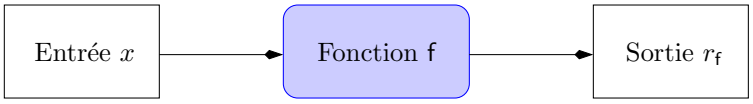
Notre approche, au lieu de certifier la correction de inc *en construisant une preuve du triplet* $\{0 \leq x\}\mathsf{inc}(x)\{1 \leq x\}$*, consiste à intégrer un* patron de preuve *dans le code de* inc*, permettant de construire dynamiquement des preuves de la post-condition pour toutes instances de* x*, lors de l'exécution de* $\mathsf{inc}(x)$*. Cela a pour intérêt de génerer un certificat pour chaque exécution de* inc *(utilisable pour justifier le résultat de fonctions faisant appel à* inc*).*

Définition 3.1.1 (Patron de preuve)**.** *Un patron de preuve est un arbre de dérivation portant sur des variables non instanciées. Un patron de preuve comporte certaines sous-parties variables, comprenant plusieurs dérivations possibles dépendantes de la valeur des variables.*

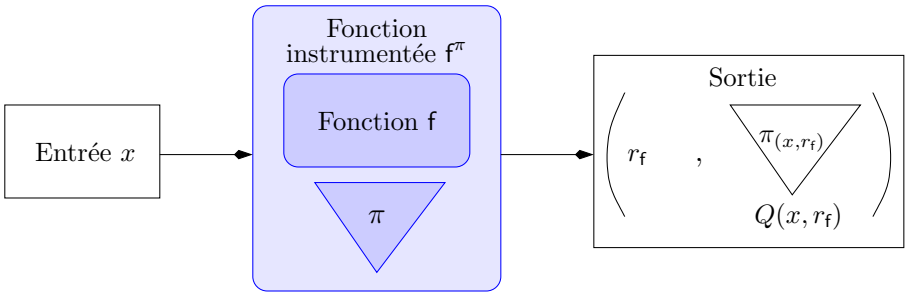
Un patron de preuve peut être vue comme une preuve d'ordre supérieur dont certaines branches sont déterminées dynamiquement. Pour cette raison, un patron de preuve ne peut être vérifié en l'état. La vérification ne se fait qu'après exécution du code qu'il étend, sur la preuve qu'il a permis de générer.

3.1.2.2 Justification de fonctions

La *justification* d'une fonction consiste à retourner un certificat de la correction de son résultat pour toutes ses exécutions correctes. Néanmoins, cela ne garantit en rien que la fonction en question s'exécutera correctement pour toutes ses entrées possibles. En toute généralité, si l'on souhaite justifier une fonction $\mathsf{f}(x_1, \ldots, x_n)$, il nous faut intégrer à son code les arguments nécessaires pour produire une preuve de la correction de r_f quelques soient les valeurs des variables $x_1, \ldots, x_n$. Ces arguments sont regroupés dans un patron de preuve π. En d'autres termes, pour justifier le résultat d'une fonction $\mathsf{f}(x_1, \ldots, x_n)$, retournant un résultat r_f, nous devons fournir un patron de preuve π permettant d'instrumenter f. Nous représentons une fonction f instrumentée par f^π. La correction du résultat de $\mathsf{f}(x_1, \ldots, x_n)$ est exprimée par une propriété $Q(x_1, \ldots, x_n, r_\mathsf{f})$ qui est prouvée par l'arbre de dérivation $\pi_{(x_1,\ldots,x_n,r_\mathsf{f})}$, obtenu à l'exécution grâce au patron de preuve π. Ceci est schématisé en figure 3.2.



(a) L'exécution de la fonction f pour une entrée x, retourne un résultat r_f



(b) L'exécution de la fonction instrumentée f^π pour une entrée x, retourne un résultat r_f et une preuve de $Q(x, r_\mathsf{f})$

Fig. 3.2 – Instrumentation d'une fonction f à l'aide d'un patron de preuve π

Prouver $Q(x_1, \ldots, x_n, r_\mathsf{f})$ nécessite que $Q(x_1, \ldots, x_n, r_\mathsf{f})$ soit une expression exprimable dans la logique du vérificateur de preuves choisi. C'est cette contrainte qui cadre le champs d'application de ce principe de justification. Si le vérificateur de preuves porte sur des théorèmes de la logique du premier ordre, alors la correction des fonctions que nous souhaitons instrumenter doit être exprimable dans la logique du premier ordre. Dans le cas de l'instrumentation de méta-programmes (ce que nous visons), ceux-ci doivent avoir pour résultat une expression logique (logique du premier ordre, logique de Hoare ou autres). Certains analyseurs statiques ont cette caractéristique, nous en donnerons les détails en section 3.2.

Reprenons l'exemple 3.1.1. Certifier la fonction inc consisterait à prouver que pour tout entier naturel x, $r_{\mathsf{inc}} \geq 1$. Nous définissons inc comme suit :

$$\mathsf{inc}(x) \stackrel{def}{=} \mathsf{let}\ r_{\mathsf{inc}} = x + 1\ \mathsf{in}\ r_{\mathsf{inc}}$$

La preuve de correction de cette fonction est donnée ci-dessous :

$$\dfrac{\dfrac{\dfrac{\overbrace{0 \leq x}^{H_1}}{1 \leq x+1}\ {+_c(1)}}{0 \leq x \Rightarrow 1 \leq x+1}\ {\Rightarrow_i(H_1)}}{\{0 \leq x\} r_{\mathsf{inc}} := x + 1 \{1 \leq r_{\mathsf{inc}}\}}\ sem$$

En revanche, l'instrumentation de inc dans le but de justifier chacune de ses exécutions consiste à intégrer un patron de preuve, π, dans le code de inc, afin que π génére une preuve de $1 \leq r_{\mathsf{inc}}$ pour tout x, est beaucoup plus simple. En effet, il suffit pour le vérificateur de preuve de vérifier que le résultat retourné par inc est supérieur à 1.

$$\mathsf{inc}^{\pi}(x) \stackrel{def}{=} \mathsf{let}\ r_{\mathsf{inc}} = x + 1\ \mathsf{in}\ (\pi, r_{\mathsf{inc}})$$

$$avec\ \pi = \left\{\ \overline{1 \leq r_{\mathsf{inc}}}\right.$$

Dans cet exemple trivial, le patron de preuve π permet de générer une preuve très courte (un seul pas de déduction), qui consiste à comparer deux constantes, pour chaque exécution de inc.

Pour illustrer cette distinction entre certifier et justifier, arrêtons nous un court instant sur un exemple un peu plus complexe. Considèrons un programme implantant un *algorithme de tri*, tri(), d'un ensemble d'entier $\{x1, \ldots, x_n\}$ rangé dans un tableau $t[\,]$. Prouver la correction de ce programme n'est pas chose aisée et consiste à montrer que le programme tri, ordonne correctement selon un ordre (croissant ou décroissant) tout tableau d'entiers qui lui est passé en paramètre.

En revanche, justifier les exécutions de ce programme, consiste seulement à instrumenter tri, de manière à ce qu'il génère pour chaque exécution la preuve, nettement plus simple, de $t[1] \leq t[2] \leq \ldots \leq t[n]$. Cela peut s'obtenir en ajoutant au code de tri un test, qui une fois l'algorithme exécuté, compare chaque case du tableau avec celle qui la précède et vérifie ainsi que le tableau est trié. Plutôt que d'instrumenter le code du programme tri, il suffit d'instrumenter ce test, afin de justifier les résultats de tri. Cet exemple nous montre que l'instrumentation laisse beaucoup de libertés et ne se limite pas à doubler le résultat d'une fonction d'un patron de preuve. Des tests instrumentés peuvent être ajoutés si ceux-ci facilitent la justification.

3.1.2.3 Justifications résultats d'un méta-programme

Nous abordons maintenant l'instrumentation de méta-programmes. La particularité des méta-programmes réside dans le fait que leurs entrées sont des programmes, et leurs résultats donnent une expression concernant la sémantique des programmes pris en entrée. Il n'est donc pas étonnant que leur propriété de correction évoque la propriété de correction des programmes qu'ils traitent.

Exemple 3.1.2. *Soit le méta-programme* MP *et sa fonction* sémantique*,* sem*, qui à partir d'une formule ϕ annotée au point de contrôle q du programme traité, et d'une transition $q \xrightarrow{\tau} q'$, calcule et retourne une post-condition ϕ' annotée au point de contrôle q'. La spécification de cette fonction* sem *exprime le fait que pour toute transition τ qu'elle est capable de traiter, le triplet de Hoare $\{\phi\}\tau\{\phi'\}$ est valide. Ainsi, une preuve de la correction de* sem *consiste à prouver que pout toutes transitions τ, le triplet $\{\phi\}\tau\{\phi'\}$ est valide. Indépendamment du fait qu'une telle preuve paraît difficile à trouver, prouver ce triplet pour tout τ et tout ϕ, va bien au delà de ce que nous souhaitons. Il n'est pas utile de prouver la correction de* sem *si seulement celle de la transition τ nous intéresse.*

Notre approche, au lieu de tenter de construire une preuve du triplet $\{P\}\mathsf{sem}(\tau,\phi)\{Q\}$ consiste à intégrer un patron de preuve dans le code de sem, permettant ainsi à $\mathsf{sem}^\pi(\tau,\phi)$ de construire dynamiquement une preuve de $\{\phi\}\tau\{\phi'\}$, pour chacune de ses exécutions.

Afin d'illustrer plus précisément l'instrumentation de méta-programmes, considèrons maintenant que la transition τ de l'exemple précédent soit *gardée* par une certaine condition g. En supposant qu'avant de prendre la transition gardée, le programme ait vérifié une propriété ϕ, l'information la plus évidente à déduire après le passage de cette transition (la condition a été vérifiée) est $\phi \wedge g$. Cela est valable quel que soit le genre du méta-programme (analyseur, model-checker, compilateur, ...), et quelle que soit sa stratégie. Nous souhaitons apporter une preuve de la validité de $\{\phi\}?g\{\phi \wedge g\}$, pour chaque exécution de la fonction sem, et nous proposons d'associer un patron de preuve au code de sem pour automatiser cela.

Dans le cas très général du traitement des transitions gardées qui nous sert d'exemple (où r_{sem} est définie par $\phi \wedge g$), la fonction sem n'a pas de réel contenu calculatoire. Le code de la fonction sem peut être vu comme l'affectation suivante :

$$\mathsf{sem}(?g,\phi) \stackrel{def}{=} r_{\mathsf{sem}} := \phi \wedge g$$

L'instrumentation de la fonction sem consiste à ajouter un patron de preuve π, à cette affectation, afin d'obtenir une preuve de $\{\phi\}?g\{\phi \wedge g\}$ à chaque exécution :

$$\mathsf{sem}^\pi(?g,\phi) \stackrel{def}{=} \mathsf{let}\ r_{\mathsf{sem}} := \phi \wedge g\ \mathsf{in}\ (\pi(g,\phi), r_{\mathsf{sem}})$$

$$avec\ \pi(g,\phi) = \left\{ \dfrac{\dfrac{\dfrac{\vdash_{\mathrm{NJ}} \overbrace{\phi}^{H_1} \quad \vdash_{\mathrm{NJ}} \overbrace{g}^{H_2}}{\vdash_{\mathrm{NJ}} \phi \wedge g}\ \wedge_i}{\vdash_{\mathrm{NJ}} g \Rightarrow \phi \wedge g}\ \Rightarrow_i(H_2)}{\vdash_{\mathrm{NJ}} \phi \Rightarrow (g \Rightarrow \phi \wedge g)}\ \Rightarrow_i(H_1)}{\vdash_{\mathrm{H}} \{\phi\}?g\{\phi \wedge g\}}\ sem \right.$$

Afin d'illustrer le mécanisme de justification après exécution, nous terminons cette section en donnant, relativement à l'exemple ci-dessus, la preuve de correction construite pour une certaine instance de $(?g)$ et ϕ.

Exemple 3.1.3. *Soient x et y deux variables du programme traité, $g \stackrel{def}{=} x > 0$ et $\phi \stackrel{def}{=} y < 0$. L'appel à la fonction instrumentée $\mathsf{sem}^\pi(?(x > 0),(y < 0))$ retourne le*

couple $(\pi_{(x>0,y<0)}, (y < 0 \wedge x > 0))$*, où* $\pi_{(x>0,y<0)}$ *est une preuve du triplet* $\{y < 0\}?x > 0\{y < 0 \wedge x > 0\}$ *obtenue directement en remplaçant* g *et* ϕ *par leurs définitions dans le patron de preuve défini précédemment. On obtient donc la preuve* $\pi_{(x>0,y<0)}$*, donnée ci-dessous, qui garantie la correction du résultat de la fonction* sem^{π}*.*

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{y < 0}^{H_1} \qquad \vdash_{\text{NJ}} \overbrace{x > 0}^{H_2}}{\vdash_{\text{NJ}} y < 0 \wedge x > 0}\wedge_i}{\vdash_{\text{NJ}} x > 0 \Rightarrow y < 0 \wedge x > 0}\Rightarrow_i (H_2)}{\vdash_{\text{NJ}} y < 0 \Rightarrow (x > 0 \Rightarrow y < 0 \wedge x > 0)}\Rightarrow_i (H_1)}{\vdash_{\text{H}} \{y < 0\}?x > 0\{y < 0 \wedge x > 0\}}\,sem$$

On remarque que la structure de cette preuve est la même que celle du patron de preuve $\pi(g, \phi)$, où seules g et ϕ ont été remplacées par leurs définitions.

3.1.3 Construction et vérification des preuves dans le système Coq

Il s'agit ici de répondre à la question, « *comment générer des termes du* CIC *du type de la propriété sémantique à prouver ?* » Générer directement des λ-termes correspondants aux preuves obtenues par instrumentation est une possibilité mais qui s'avère non triviale pour un développeur n'ayant pas un haut niveau de connaissance de la théorie des types et plus particulièrement du calcul des constructions inductives. Une solution alternative consiste à exploiter l'environnement de preuve du système Coq – qui est initialement dédié à la preuve interactive. L'idée est d'utiliser les tactiques de l'environnement de preuve pour assister (non pas le développeur) mais l'outil instrumenté afin de générer des termes du CIC. En effet, une tactique Coq peut implanter un algorithme de recherche de preuve, exécutable dans l'environnement de Coq, qui en cas de succés construit un λ-terme (en cas d'echec de l'algorithme la tactique n'a aucun effet). L'environnement dédié à la preuve assistée du système Coq traduit des séquences de tactiques en des termes du CIC. De plus, chaque raisonnement de la déduction naturelle est modélisé en Coq par une tactique. La solution consiste à traduire les arbres de dérivation, construit à partir des patrons de preuve, sous forme de séquences de tactique. L'assistant à la preuve se chargera de construire les λ-termes à partir de séquences de tactiques que l'instrumentation lui fournira. Ces séquences de tactique sont construites par une traduction directe des raisonnements de la déduction naturelle vers les tactiques correspondantes, ordonnées dans l'ordre inverse de l'arbre de dérivation (de la conclusion vers les hypothèses). La génération de ces scripts s'effectue en parcourant l'arbre de dérivations produit et ne fait appel à aucun algorithme de recherche. La construction des scripts se fait donc en temps constant relativement à l'exécution des fonctions instrumentées, du fait que les arbres de preuves produits ont une structure statiquement définie par les patrons de preuve.

Exemple 3.1.4. *Nous considérons l'arbre de preuve* $\pi_{(x>0,y<0)}$ *donné à l'exemple 3.1.3. Cet arbre est automatiquement traduit en une séquence de tactiques du système* Coq *afin de bénéficier de l'environnement de construction des* λ*-termes puis de vérificateur*

de preuves de Coq *(qui est un vérificateur de type). Le script de tactiques produit pour* $\pi_{(x>0,y<0)}$ *est le suivant :*

`Goal triplet P (guard G) Q.`	$\rightarrow$	$\{y < 0\}?x > 0\{y < 0 \wedge x > 0\}$
`Proof.`		
`apply t_wp;simpl.`	$\rightarrow$	(sem)
`intro H1.`	$\rightarrow$	$(\Rightarrow_i (H_1))$
`intro H2.`	$\rightarrow$	$(\Rightarrow_i (H_2))$
`split.`	$\rightarrow$	$(\wedge_i)$
`exact H1.`	$\rightarrow$	(H_1)
`exact H2.`	$\rightarrow$	(H_2)
`Qed.`		

La tactique `apply t_wp;simpl` *permet de transformer le* λ*-terme* `triplet P (guard G) Q` *de type* `Triplet`*, en un* λ*-terme de type* `Prop` *(cf. sections 2.1.1.2 et 2.1.1.4) :* `0 < y -> x < 0 -> 0 < y / x > 0`*, qui se prouve à l'aide des tactiques* `intro` *et* `split`*. Lorsque les hypothèses sont atteintes, les tactiques* `exact H1` *et* `exact H2` *permettent d'achever la preuve en signalant explicitement à l'environnement de quelles hypothèses il s'agit.*

Le λ*-terme calculé par le système* Coq *en retour de ces séquences de tactiques est le suivant :*

```
t_wp P Q (guard G) (fun (H1 : 0 < y) (H2 : x > 0) => Logic.conj H1 H2)
```

Ce λ*-terme agit comme une preuve du théorème* `triplet P (guard G) Q`*. La validité de cette preuve est ensuite vérifiée par type-checking.*

Toutefois, les tactiques ne sont pas restreintes à des traductions des règles de la déduction naturelle pour construire des termes du CIC. Par exemple, si le développeur de l'outil instrumenté ne souhaite pas gérer les hypothèses dans ses patrons de preuve, il peut utiliser la tactique `auto` qui recherche parmi les hypothèses vivantes celle qui correspond au sous-but courant.

Exemple 3.1.5. *Le script de tactiques correspondant à* $\pi_{(x>0,y<0)}$ *est le suivant, dans le cas où le patron de preuve* $\pi(g, \phi)$ *ne gère pas le nommage des hypothèses :*

`Goal triplet P (guard G) Q.`	$\rightarrow$	$\{y < 0\}?x > 0\{y < 0 \wedge x > 0\}$
`Proof.`		
`apply t_wp;simpl.`	$\rightarrow$	(sem)
`intro.`	$\rightarrow$	$(\Rightarrow_i (H_1))$
`intro.`	$\rightarrow$	$(\Rightarrow_i (H_2))$
`split.`	$\rightarrow$	$(\wedge_i)$
`auto.`	$\rightarrow$	(H_1)
`auto.`	$\rightarrow$	(H_2)
`Qed.`		

L'interêt d'utiliser des tactiques contenant des algorithmes de recherche va au delà

de la gestion des hypothèses. En pratique, il se peut que certaines données utilisées par les méta-programmes ne soient pas calculées (celles-ci sont écrites *en dur* par le développeur de l'outil ou doivent être données par l'utilisateur). C'est la cas de l'analyseur Enkidu dont l'instrumentation sera présentée au chapitre 4 : pour analyser le contenu des tableaux Enkidu utilise des *partitions* des cases de tableaux ; à l'heure actuelle, ces partitions sont fournies dans le code (pour un ensemble de programmes analysables) ou demandées à l'utilisateur (pour d'autres programmes). Si des données non calculées prennent part dans la phase de justification, le développeur de l'instrumentation ne bénéficie d'aucune fonction à instrumenter pour justifier la correction de ces données. Dans ce cas, il est nécessaire d'utiliser des algorithmes de recherche pour combler ces parties non justifiées. Nous détaillerons l'exemple des données non calculées par Enkidu au chapitre suivant en section 4.2.2.2.

Une question se pose dès lors : *où intégrer ces algorithmes de recherche de preuves* ? Deux solutions : on peut (1) soit implanter ces algorithmes de recherche de preuves dans l'instrumentation de l'outil et ceux-ci seront exécutés à l'exécution de l'outil pour achever les arbres de dérivation produit par les patrons de preuve ; (2) soit implanter ces algorithmes de recherche de preuves sous forme de tactiques Coq, qui seront exécutés lors de la phase de construction des λ-termes. Dans ce cas, les arbres de dérivations produits par les patrons de preuve auront à leurs feuilles le nom de ces tactiques.

Avec la première solution, si l'algorithme de recherche de preuves ne termine pas, l'outil instrumenté ne terminera pas non plus et ne retournera ni résultat, ni justification. L'avantage qu'a la seconde solution est que l'exécution du méta-programme ne pourra pas échouer à cause de l'instrumentation, du fait que celle-ci ne diffère de l'exécution du méta-programme original uniquement par l'instanciation des patrons de preuve. La partie manquante de la construction des preuves est alors confiée aux tactiques du système Coq. Dans cette seconde solution, un script de tactiques sera systématiquement généré. Si la tactique contenant l'algorithme de recherche échoue, elle n'aura aucun effet sur le terme en construction et ce dernier ne sera pas reconnu comme une preuve de la propriété à valider. Cependant il sera toujours possible à l'utilisateur de finir la preuve interactivement à l'aide de l'assistant à la preuve (alternative qui est impossible avec la première solution). Ceci n'a rien de désappointant : si l'implantation du méta-programme n'est pas pleinement automatique, la justification peut également ne pas l'être.

Reprenons notre exemple des partitions utilisées par Enkidu. Il est nécessaire dans les preuves de justification de prouver que ces partitions sont valides (qu'elles couvrent bien l'ensemble du tableau). Ces partitions sont formalisées par des disjonctions d'égalités et d'inégalités linéaires. Afin de prouver certains sous-buts exprimés en arithmétique linéaire (non calculés par l'outil) Coq fournit la tactique `omega` qui permet de prouver des formules exprimées en arithmétique de Presburger. Pour justifier l'utilisation de ces formules par l'outil, il est nécessaire d'en prouver la validité. Les patrons de preuve qui instrumentent l'outil devront faire appel à des algorithmes de recherche (telle la tactique `omega`) du fait qu'ils ne bénéficient d'aucun principe calculatoire à instrumenter pour prouver ces formules. Si `omega` ne termine pas ou n'est pas assez rapide, il est toutefois possible de développer une tactique plus efficace, spécialisée pour les formules utilisées par l'outil. Cette solution permet de construire par instrumentation des certificats de taille réduite au détriment d'une vérification plus longue, qui intégre une partie de la construction des preuves. Dans des protocoles tels que le *proof-carrying code* (*cf.* section 2.4.2) où les certificats de correction des programmes sont transmis via un réseau, obtenir

des certificats de taille réduite présente un certain intêret en pratique.

3.1.4 Conclusion

L'avantage d'utiliser des patrons de preuve plutôt que des preuves, réside dans le fait que pour un programmeur, instrumenter un méta-programme est plus simple que de prouver la correction de ce méta-programme. En effet, la structure du méta-programme donne celle des preuves de justification de ses résultats.

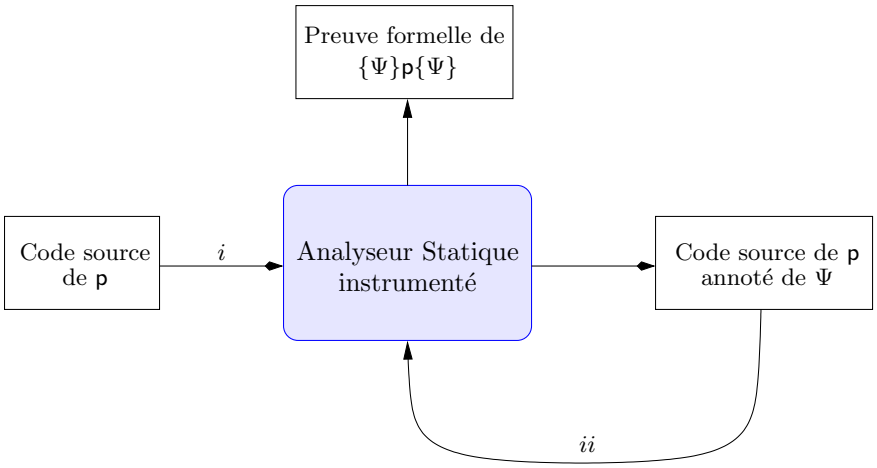
Le second avantage qu'a la justification par rapport à la certification, est qu'une partie seulement des fonctions du méta-programme doit être instrumentée pour certifier les programmes visés. La justification permet la certification mais au niveau inférieur : un méta-programme instrumenté n'est aucunement certifié et peut contenir des bogues, mais l'ensemble des ses verdicts corrects seront quant à eux certifiés grâce aux justifications formelles apportées. La preuve de correction d'un programme traité s'obtient par combinaison de preuves simples produites dynamiquement par l'exécution du méta-programme.

Enfin, il est possible d'ajouter au code d'un méta-programme des tests instrumentés, permettant de produire les justifications souhaitées. Ceci permet de se détacher du code du méta-programme et de justifier ses verdicts grâce à des tests algorithmiquement plus simples. Nous avons vu dans l'exemple du programme de tri, qu'une preuve que le contenu du tableau en sortie est trié peut s'obtenir en instrumentant un simple test qui vérifie que chaque case du tableau contient un entier supérieur à celui contenu dans la case qui la précède. Il est facile de se convaincre que l'instrumentation de l'algorithme de tri (que nous n'avons pas fourni) est plus difficile que celle de ce simple test.

La suite de ce chapitre apportera des précisions à cette brève présentation de l'approche que nous proposons, en s'intéressant à l'instrumentation d'une certaine classe de méta-programmes : les *analyseurs statiques* par *interprétation abstraite* et *calcul de point fixe*. Le chapitre suivant donnera davantage de détails en présentant l'instrumentation d'un certain analyseur statique de *programmes manipulant des tableaux*.

3.2 Instrumentation d'analyses statiques

Nous présentons dans ce qui va suivre une classe de méta-programmes compatibles avec la méthodologie présentée pour générer automatiquement des certificats. Puis, nous détaillerons le principe général de l'application de l'instrumentation à de tels méta-programmes.



Sur le schéma ci-dessus, l'étape (i) consiste à prendre en entrée le code source d'un programme p et à l'analyser (cela passe par une transformation de ce code en un système de transitions) puis à retourner le code source initial annoté d'un invariant Ψ qui caractérise les comportements du programme, définissant ainsi une approximation de sa sémantique. Cette annotation sémantique peut se modéliser par un triplet de Hoare $\{\Psi\}\mathsf{p}\{\Psi\}$.

Notre approche consiste à réinjecter, à l'étape (ii), ce code source annoté dans l'analyseur lui-même, en utilisant sa version instrumentée, afin d'obtenir une justification que la propriété Ψ calculée est une sémantique (un invariant) du programme p. Cette justification se fait sous forme d'un arbre de dérivation du triplet $\{\Psi\}\mathsf{p}\{\Psi\}$. À partir de ce triplet de Hoare, la sémantique de transformation de prédicat conduit à prouver des implications logiques de la forme $\Psi \Rightarrow \Phi$, où Φ est la plus faible pré-condition d'une certaine instruction de p, relativement à Ψ.

Notre objectif est de prouver efficacement ces implications. Développer une stratégie de preuve permettant de construire un arbre de dérivation pour toute formule Ψ et Φ d'une logique du premier ordre, afin de prouver la validité de cette implication, n'est pas chose aisée (décider de la validité d'une formule est un problème NP-difficile pour l'arithmétique de Presburger par exemple [FR74]). L'idée de ces travaux est de construire cet arbre de dérivation en utilisant des patrons de preuve qui seront instanciés à l'exécution de l'analyse statique. Ces patrons de preuve ne sont pas associés à n'importe quelle fonction de l'analyseur statique. Nous montrons qu'il possible dans certains cas de limiter l'instrumentation aux opérateurs ensemblistes du domaine abstrait utilisé par l'analyse. Nous avons identifié ces cas selon la structure des formules retournées par le calcul de plus faible pré-condition : si la formule appartient au domaine abstrait alors nous sommes dans le cas simple et l'arbre de dérivation permettant de prouver l'implication souhaitée sera obtenu par le test d'inclusion instrumenté. Sinon, il faudra effectuer quelques ajustements *ad'hoc*. Nous précisons cela en section 3.2.4 Cet arbre de dérivation doit enfin être validé par un système muni d'un vérificateur de preuves fiable, tel le logiciel Coq, pour pouvoir considérer que Ψ est une approximation correcte de $[\![\mathsf{p}]\!]$, la sémantique de p.

Avant de montrer comment instrumenter un analyseur statique afin qu'il puisse justifier les sémantiques de programmes qu'il calcule (section 3.2.3), nous présentons un résumé de ce qu'est l'analyse statique par interprétation abstraite (section 3.2.1), puis

la classe de domaines abstraits que notre approche permet de considérer (section 3.2.2).

3.2.1 Analyse statique par interprétation abstraite

Il existe bien des approches pour *étudier, vérifier, deviner* les propriétés qui définissent la sémantique d'un programme. La plus simple et la plus intuitive consiste à exécuter le programme puis observer son comportement et ses résultats. Cette approche *dynamique* ne montrera jamais plus qu'*il existe des entrées du programme pour lesquelles il respecte certaines propriétés*. Lorsque l'on sait que les domaines des entrées des programmes sont presque toujours infinis, la limite de cette approche pour inférer la sémantique des programmes devient criante.

Une approche opposée à celle-ci est dite *statique*, elle vise à établir une analyse *universelle* du programme concernant ses variables d'entrées : *pour l'intégralité des domaines d'entrées, l'analyse est valable.* L'analyse statique ne dépend donc pas d'une exécution particulière du programme, et se pratique directement sur la représentation syntaxique du programme, en s'appuyant sur la sémantique de cette représentation (le langage de programmation). L'analyse statique cherche à prendre en compte toutes les exécutions possibles du programme traité pour toutes ses entrées possibles. La difficulté de l'analyse statique est qu'il faut donc représenter par un objet fini, une infinité de comportements. L'inconvénient principal de cette approche est qu'il est, en général, impossible d'analyser des programmes un tant soit peu intéressants (avec des boucles dont la condition d'arrêt n'est pas triviale à évaluer, de la recursivité, du parallélisme, *etc*) du fait qu'elle se heurte à *l'indécidabilité du problème de l'arrêt* [Tur36] : « *il n'existe aucun programme permettant de décider si un autre programme termine ou pas* » Si de telles analyses ne sont même pas capables d'établir qu'un programme termine, comment espérer deviner ou vérifier des propriétés plus complexes, plus intéressantes (théorème de *Rice* [Ric53]) ? Pour contourner cet obstacle, l'interprétation abstraite fait recours à des approximations, afin de rendre les analyses algorithmiquement *décidables*. Ces approximations portent sur la sémantique des programmes analysés et s'appuient sur la théorie de l'interprétation abstraite [CC77].

3.2.1.1 Base de l'interprétation abstraite

L'interprétation abstraite postule que toute sémantique peut être exprimée par un *point fixe* de fonctions monotones sur des structures ordonnées (les treillis particulièrement). La sémantique d'un programme, qui en général n'est pas calculable, peut ainsi être approximée par ce que l'on nomme sa *sémantique abstraite*. L'approximation doit être choisie de manière à : d'une part, se concentrer sur les propriétés pertinentes du programme, et d'autre part, obtenir une sémantique abstraite *calculable*.

L'analyse consiste dès lors à calculer un point fixe de la fonction calculant cette sémantique abstraite. Cela pose la question de savoir si un tel point fixe existe toujours. Nous verrons dans ce qui suit que le théorème de *Knaster-Tarski* (voir théorème 3.2) nous assure que "oui", sous certaines conditions. La base de ces conditions est que la sémantique du programme doit être formalisée par des élements d'un ensemble partiellement ordonné.

Définition 3.2.1 (Ensemble partiellement ordonné)**.** *Un ensemble partiellement ordonné est un couple* $(\mathcal{D}, \sqsubseteq_{\mathcal{D}})$ *où* $\mathcal{D}$ *est un ensemble et* $\sqsubseteq_{\mathcal{D}}$ *est une relation d'ordre partielle telle que :*

- $\sqsubseteq_{\mathcal{D}}$ *est reflexive :* $\forall d \in \mathcal{D}, d \sqsubseteq_{\mathcal{D}} d$
- $\sqsubseteq_{\mathcal{D}}$ *est transitive :* $\forall d_1, d_2, d_3 \in \mathcal{D}, d_1 \sqsubseteq_{\mathcal{D}} d_2 \wedge d_2 \sqsubseteq_{\mathcal{D}} d_3 \Rightarrow d_1 \sqsubseteq_{\mathcal{D}} d_3$
- $\sqsubseteq_{\mathcal{D}}$ *est antisymétrique :* $\forall a_1, a_2 \in \mathcal{D}, a_1 \sqsubseteq_{\mathcal{D}} a_2 \wedge a_2 \sqsubseteq_{\mathcal{D}} a_1 \Rightarrow a_1 = a_2$

Définition 3.2.2 (Treillis, treillis borné et treillis complet)**.** *Un treillis* $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}, \sqcup_{\mathcal{D}}, \sqcap_{\mathcal{D}})$ *est un ensemble partiellement ordonné où chaque couple d'élements admet une borne supérieure et une borne inférieure, telles que :*

- $\sqcup_{\mathcal{D}}$ *calcule une borne supérieure binaire :*
  - $\forall d_1, d_2 \in \mathcal{D}, d_1 \sqsubseteq_{\mathcal{D}} d_1 \sqcup_{\mathcal{D}} d_2 \ \wedge \ d_2 \sqsubseteq_{\mathcal{D}} d_1 \sqcup_{\mathcal{D}} d_2$
  - $\forall d_1, d_2, b \in \mathcal{D}, d_1 \sqsubseteq_{\mathcal{D}} b \wedge d_2 \sqsubseteq_{\mathcal{D}} b \Rightarrow d_1 \sqcup_{\mathcal{D}} d_2 \sqsubseteq_{\mathcal{D}} b$

- $\sqcap_{\mathcal{D}}$ *calcule une borne inférieure binaire :*
  - $\forall d_1, d_2 \in \mathcal{D}, d_1 \sqcap_{\mathcal{D}} d_2 \sqsubseteq_{\mathcal{D}} d_1 \ \wedge \ d_1 \sqcap_{\mathcal{D}} d_2 \sqsubseteq_{\mathcal{D}} d_2$
  - $\forall d_1, d_2, b \in \mathcal{D}, b \sqsubseteq_{\mathcal{D}} d_1 \wedge b \sqsubseteq_{\mathcal{D}} d_2 \Rightarrow b \sqsubseteq_{\mathcal{D}} d_1 \sqcap_{\mathcal{D}} d_2$

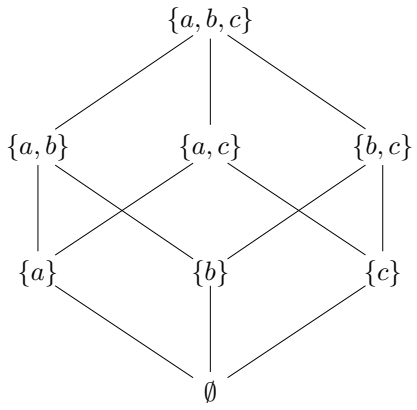
Un treillis borné $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}, \sqcup_{\mathcal{D}}, \sqcap_{\mathcal{D}}, \bot_{\mathcal{D}}, \top_{\mathcal{D}})$ *est un treillis qui admet un plus grand élement* $\top_{\mathcal{D}}$*, et un plus petit élement* $\bot_{\mathcal{D}}$ *tels que :*

- $\forall d \in \mathcal{D}, d \sqsubseteq_{\mathcal{D}} \top_{\mathcal{D}}$
- $\forall d \in \mathcal{D}, \bot_{\mathcal{D}} \sqsubseteq_{\mathcal{D}} d$

Un treillis complet $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}, \bigsqcup_{\mathcal{D}})$ *est un triplet constitué d'un ensemble partiellement ordonné* $(\mathcal{D}, \sqsubseteq_{\mathcal{D}})$*, et* $\bigsqcup_{\mathcal{D}}$ *un opérateur qui calcule la borne supérieure de toute partie* $\mathcal{S}$ *de* $\mathcal{D}$*, tel que :*

- $\forall d \in \mathcal{S}, d \sqsubseteq_{\mathcal{D}} \bigsqcup_{\mathcal{D}} \mathcal{S}$
- $\forall b \in \mathcal{D}, (\forall d \in \mathcal{S}, d \sqsubseteq_{\mathcal{D}} b) \Rightarrow \bigsqcup_{\mathcal{D}} \mathcal{S} \sqsubseteq_{\mathcal{D}} b$

Exemple 3.2.1. *Soit l'ensemble de trois variables,* $E \stackrel{def}{=} \{a, b, c\}$*. L'ensemble des parties de* E*,* $\mathcal{P}(E)$*, muni de la relation d'ordre d'inclusion ensembliste,* $\subseteq$*, forme un treillis complet* $(\mathcal{P}(E), \subseteq, \cup, \cap, \emptyset, \{a, b, c\})$ *:*



Remarques :
- Un treillis complet est forcément borné, avec $\top_{\mathcal{D}} = \bigsqcup_{\mathcal{D}} \mathcal{D}$ et $\bot_{\mathcal{D}} = \bigsqcup_{\mathcal{D}} \emptyset$.
- Un treillis complet possède forcément un opérateur de borne inférieure pour toute partie $\mathcal{S}$ de $\mathcal{D}$, avec $\bigsqcap_{\mathcal{D}} \mathcal{S} = \bigsqcup_{\mathcal{D}} \{b \mid \forall d \in S, b \sqsubseteq_{\mathcal{D}} d\}$.

Définition 3.2.3 (Connexion de Galois)**.** *Soient* $(\mathcal{D}_1, \sqsubseteq_{\mathcal{D}_1}, \bigsqcup_{\mathcal{D}_1})$ *et* $(\mathcal{D}_2, \sqsubseteq_{\mathcal{D}_2}, \bigsqcup_{\mathcal{D}_2})$ *deux treillis complets. Une paire de fonctions* (α, γ)*, où* $\alpha : \mathcal{D}_1 \to \mathcal{D}_2$ *et* $\gamma : \mathcal{D}_2 \to \mathcal{D}_1$*, est une connexion de Galois si elle vérifie la condition suivante :*

$$\forall d_1 \in \mathcal{D}_1, \forall d_2 \in \mathcal{D}_2,\ d_1 \sqsubseteq_{\mathcal{D}_1} \gamma(d_2) \iff \alpha(d_1) \sqsubseteq_{\mathcal{D}_2} d_2$$

Les connexions de Galois sont particulièrement intéressantes, car leur définition est relativement simple et elles vérifient les propriétés énoncées par le théorème 3.1 qui va suivre.

3.2.1.2 Abstraction correcte de sémantiques

Lorsque le domaine sémantique considéré pose des problèmes indécidables ou de compléxité trop élevée, raisonner dans un domaine abstrait est une solution pour résoudre ces problèmes [CC77]. Les réponses apportées dans le *monde abstrait* devront cependant être conservatrices, relativement au *monde concret*, pour pouvoir être exploitées. Autrement dit, les propriétés calculées dans le domaine abstrait doivent être des approximations correctes des propriétés concrètes. Pour garantir cela, le théorème 3.1 nous fournit de précieuses propriétés.

Théorème 3.1 (Propriétés des connexions de Galois)**.** *Soit* (α, γ) *une connexion de Galois entre deux treillis complets* $(\mathcal{D}_1, \sqsubseteq_{\mathcal{D}_1}, \bigsqcup_{\mathcal{D}_1})$ *et* $(\mathcal{D}_2, \sqsubseteq_{\mathcal{D}_2}, \bigsqcup_{\mathcal{D}_2})$*, alors :*

1. α *est une fonction monotone*
2. γ *est une fonction monotone*
3. $\forall d_1 \in \mathcal{D}_1, d_1 \sqsubseteq_{\mathcal{D}_1} \gamma \circ \alpha(d_1)$
4. $\forall d_2 \in \mathcal{D}_2, \alpha \circ \gamma(d_2) \sqsubseteq_{\mathcal{D}_2} d_2$
5. $\forall d_1 \in \mathcal{D}_1, \alpha \circ \gamma \circ \alpha(d_1) = \alpha(d_1)$

6. $\forall d_2 \in \mathcal{D}_2, \gamma \circ \alpha \circ \gamma(d_2) = \gamma(d_2)$

Étant donnée une fonction sémantique (concrète), sem , des programmes vers un domaine concret $\mathcal{C}$, la propriété $[\![\mathsf{p}]\!] \in \mathcal{C}$ – calculée par sem – spécifiant les comportements attendus du programme p, pourra être approximée par une propriété $[\![\mathsf{p}]\!]^\sharp \in \mathcal{A}$, calculée par la fonction sémantique (abstraite) $\mathsf{sem}^\sharp$. Les propriétés abstraites calculées seront correctes si et seulement si $[\![\mathsf{p}]\!] \sqsubseteq_\mathcal{C} \gamma([\![\mathsf{p}]\!]^\sharp)$. Pour exploiter cela, le lien entre les deux domaines doit être formalisé par une connexion de Galois telle que :

$$(\mathcal{C}, \sqsubseteq_\mathcal{C}, \bigsqcup_\mathcal{C}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{A}, \sqsubseteq_\mathcal{A}, \bigsqcup_\mathcal{A})$$

L'utilité d'une fonction $\mathsf{sem}^\sharp$ réside dans le fait que dans la majorité des cas, la fonction sémantique concrète n'est pas calculable, car le domaine concret est trop riche[2]. L'abstraction α formalise une perte d'information et un appauvrissement du domaine concret qui permet de répondre de façon partielle aux problèmes posés. Cependant, d'apres le théorème 3.1, toutes les réponses apportées par la fonction sémantique abstraite, seront des approximations correctes des propriétés sémantiques définies dans le domaine concret (point 3. du théorème 3.1).

3.2.1.3 Illustration : Le domaine des intervalles

Nous prenons, à titre d'exemple de domaine abstrait pour la suite de cette section, le domaine des *intervalles* [CC76]. Ses élements (sur-)approximent les valeurs possibles des variables d'un programme et sont soit l'ensemble vide, noté $\bot_\mathcal{I}$, exprimant que la variable en question n'a pas encore de valeur définie, soit un intervalle dont les bornes peuvent être infinies, signifiant que la variable a une infinité de valeurs possibles :

$$\mathcal{I} \overset{def}{=} \bot_\mathcal{I} \cup \{[\ell, u] \mid \ell \in \mathbb{Z} \cup \{-\infty\} \wedge u \in \mathbb{Z} \cup \{+\infty\} \wedge \ell \leq u\}$$

L'ordre partiel $\sqsubseteq_\mathcal{I}$ sur ce domaine est l'inclusion sur les intervalles. Une variable x d'un programme prend ses valeurs dans $\mathbb{Z}$. Nous formalisons dans la logique du premier ordre une formule $\Phi(x)$ exprimant l'ensemble de ses valeurs possibles, par une certaine partie de $\mathbb{Z}$.

C'est donc tout naturellement qu'une telle propriété concrète sur les valeurs d'une variable peut être *abstraite par le plus petit intervalle* contenant ces valeurs : $\alpha_\mathcal{I}(\Phi(x)) = I(x) \overset{def}{=} [\ell, u]$, où ℓ est la plus petite valeur possible de x et u la plus grande.

Exemple 3.2.2. *Soit* $\Phi(x) \overset{def}{=} 0 \leq x$ *la formule caractérisant l'ensemble infini de valeurs possible de* x *(en l'occurrence,* x a une valeur positive*). L'intervalle correspondant à cette formule est* $I(x) \overset{def}{=} [0, +\infty[$*. Le symbole* ∞ *permet de caractériser syntaxiquement les ensembles infinis.*

Exemple 3.2.3. *Soit* $\Phi(x) \overset{def}{=} \exists k \in \mathbb{Z}, x = 2k$ *la formule caractérisant l'ensemble infini de valeurs possibles de* x *(*x est pair *en l'occurrence). L'intervalle correspondant à*

[2] Du point de vue de l'expressivité

cette formule est $I(x) \stackrel{def}{=}]-\infty, +\infty[$. *On voit ici que le domaine abstrait des intervalles constitue bien une approximation du domaine concret et ne parvient pas à exprimer des propriétés de parité sur les variables du programmes.*

Concrétisation : La concrétisation d'un intervalle $I(x) \stackrel{def}{=} [\ell, u]$ est une formule portant sur x, désignant la partie de $\mathbb{Z}$ qu'il représente : $\gamma_{\mathcal{I}}(I(x)) = \ell \leq x \leq u$ (forme contractée de $\ell \leq x \wedge x \leq u$). En appliquant successivement les fonctions d'abstraction puis de concrétisation, il y a bien une perte potentielle d'information :

$$\forall \Phi, \Phi \Rightarrow \gamma_{\mathcal{I}} \circ \alpha_{\mathcal{I}}(\Phi)$$

Exemple 3.2.4. *Soit* $\Phi(x) \stackrel{def}{=} 0 < x < 5 \vee 10 < x < 15$, *exprimant que la variable* x *a pour valeur un entier compris entre* 0 *et* 5, *ou entre* 10 *et* 15. *L'abstraction de* $\Phi(x)$ *nous donne l'intervalle* $\alpha_{\mathcal{I}}(\Phi(x)) = [0, 15]$ *qui se concrétise en la formule* $\gamma_{\mathcal{I}}([0, 15]) = 0 < x < 15$, *qui est moins précise que* $\Phi(x)$.

Les fonctions $\alpha_{\mathcal{I}}$ et $\gamma_{\mathcal{I}}$ forment une connexion de Galois entre le treillis des intervalles et celui de la logique du premier ordre. Les propriétés calculées dans le domaine abstrait seront des approximations *correctes* – dans le sens où elles ne font que perdre de l'information et que rien de faux ne peut être induit – de la sémantique concrète.

Pour compléter la définition de ce treillis, nous donnons maintenant les opérateurs de borne supérieure et inférieure, puis la relation d'ordre $\sqsubseteq_{\mathcal{I}}$:

$$I_1 \sqcup_{\mathcal{I}} I_2 = \begin{cases} [min(\ell_1, \ell_2), max(u_1, u_2)] \text{ si } I_1 \stackrel{def}{=} [\ell_1, u_1] \text{ et } I_2 \stackrel{def}{=} [\ell_2, u_2] \\ I_1 \text{ si } I_2 = \bot_{\mathcal{I}} \\ I_2 \text{ si } I_1 = \bot_{\mathcal{I}} \end{cases}$$

$$I_1 \sqcap_{\mathcal{I}} I_2 = \begin{cases} [max(\ell_1, \ell_2), min(u_1, u_2)] \text{ si } I_1 \stackrel{def}{=} [\ell_1, u_1] \text{ et } I_2 \stackrel{def}{=} [\ell_2, u_2] \\ \bot_{\mathcal{I}} \text{ sinon} \end{cases}$$

$$I_1 \sqsubseteq_{\mathcal{I}} I_2 = \begin{cases} \top \text{ si } \ell_2 \leq \ell_1 \wedge u_1 \leq u_2 \text{ avec } I_1 \stackrel{def}{=} [\ell_1, u_1] \text{ et } I_2 \stackrel{def}{=} [\ell_2, u_2] \\ \bot \text{ sinon} \end{cases}$$

3.2.1.4 Spécification de sémantiques par calcul de points fixes

Nous abordons désormais les questions que pose le calcul de la sémantique d'un programme. Une solution, pour calculer la sémantique d'un programme, consiste à plonger la fonction d'analyse de programmes dans le cadre des *points fixes de fonction monotones*. En ce sens, la sémantique d'un programme p, dans un treillis $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}, \bigsqcup_{\mathcal{D}})$, sera définie comme la solution de l'équation de point fixe :

$$[\![\mathsf{p}]\!] \stackrel{def}{=} \lambda d \in \mathcal{D}.\ d = (d^0 \sqcup_{\mathcal{D}} \mathsf{sem}(\mathsf{p}, d)) \tag{3.1}$$

où sem est la fonction sémantique qui à partir de la $k-$ième propriété, d^k, calculée et du programme p, calcule la $(k+1)-$ième propriété, d^{k+1}, en tenant compte des effets de chaque transition de p et telle que $d^k \sqsubseteq_{\mathcal{D}} d^{k+1}$.

À la question, « *une telle équation admet-elle une solution ?* » , le théorème 3.2 nous répond oui. Il nous affirme même qu'il en existe plusieurs et que ces solutions forment un ensemble partiellement ordonné.

Théorème 3.2 (Knaster-Tarski)**.** *Soit* $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}, \bigsqcup_{\mathcal{D}})$ *un treillis complet et* $f : \mathcal{D} \to \mathcal{D}$ *une fonction monotone. L'ensemble des points fixes de* f *est un treillis complet* $(\mathcal{D}_f, \sqsubseteq_{\mathcal{D}_f}, \bigsqcup_{\mathcal{D}_f})$ *tel que,*

$$\bot_{\mathcal{D}_f} = \bigsqcap_{\mathcal{D}_f}\{d \in \mathcal{D} \mid f(d) \sqsubseteq_{\mathcal{D}} d\} \qquad et \qquad \top_{\mathcal{D}_f} = \bigsqcup_{\mathcal{D}_f}\{d \in \mathcal{D} \mid d \sqsubseteq_{\mathcal{D}} f(d)\}$$

En conséquence de ce théorème, la sémantique la plus précise qui soit correcte est le plus petit point fixe de l'équation 3.1, soit $\bot_{\mathcal{D}_{\mathsf{sem}}}$.

3.2.1.5 Partitionnement de sémantiques pour les systèmes de transitions

Les programmes que nous souhaitons certifier dans ces travaux, sont des programmes impératifs structurés, que nous représentons par des systèmes de transitions. Cela présente l'avantage de pouvoir partitionner le domaine sémantique, $\mathcal{D}$, d'un programme, selon l'ensemble fini, Q, des points de contrôle du système de transitions correspondant au programme : $\mathcal{D} = \bigcup_{q \in Q} \mathcal{D}_q$. Il est donc possible et correct de remplacer l'équation de point fixe 3.1 par le système d'équations de point fixe suivant :

$$[\![\mathsf{p}]\!] \stackrel{def}{=} \lambda d_1 \ldots \lambda d_n . \bigwedge_{q \in Q} d_q = d_q^0 \sqcup_{\mathcal{D}} (\mathsf{sem}(\mathsf{p}, d_1 \sqcup_{\mathcal{D}} \ldots \sqcup_{\mathcal{D}} d_n) \sqcap_{\mathcal{D}} \mathcal{D}_q) \tag{3.2}$$

où $n = |Q|$ et $d_q \in \mathcal{D}_q$ est défini par $d_q = d \sqcap_{\mathcal{D}} \mathcal{D}_q$.

Si l'on décompose le programme p par rapport à l'ensemble de ses transitions $T \stackrel{def}{=} \{(q, \tau, q') \mid q \xrightarrow{\tau} q'\}$, l'équation peut alors s'écrire de façon plus précise :

$$[\![\mathsf{p}]\!] \stackrel{def}{=} d_{q_0} = \top_{\mathcal{D}} \wedge \lambda d . \bigwedge_{q' \in (Q \setminus q_0)} d_{q'} = \bigsqcup_{(q,\tau,q') \in T} \mathsf{sem}(\tau, d_q) \tag{3.3}$$

Cette décomposition de la sémantique en une conjonction d'invariants est essentielle pour prouver la correction des programmes, transition par transition, comme le suggère la méthode de Floyd-Hoare [Flo67, Hoa69].

La sémantique des programmes peut donc s'exprimer par une conjonction d'invariants, calculés à chaque point de contrôle du programme. Cette sémantique est dite *collectrice* relativement aux états atteignables du système de transitions.

Pour calculer la sémantique d'un tel système d'équations, une technique consiste à itérer la fonction sem sur chaque partie de $\mathcal{D}$, en parallèle, et de manière équitable. Cette technique est nommée *l'itération chaotique* [CC77].

La fonction sem est définie selon des *fonctions de transfert* $\mathsf{sem_g}$ et $\mathsf{sem_a}$, respectivement pour les *gardes* et les *affectations*, qui permettent de définir l'effet de chaque transition du système, mais qui ne sont en général pas calculables. L'important, pour une analyse statique est de définir des fonctions de transfert abstraites $\mathsf{sem_g^\sharp}$ et $\mathsf{sem_a^\sharp}$ permettant de sur-approximer les fonctions de transfert concrètes. Ces approximations des effets des transitions du programme définissent la fonction sémantique abstraite $\mathsf{sem}^\sharp$

et permettent de calculer $[\![\mathsf{p}]\!]^\sharp$. En conséquence, le résultat de l'équation 3.3 peut être approximé par la sémantique abstraite $[\![\mathsf{p}]\!]^\sharp$ définie comme suit :

$$[\![\mathsf{p}]\!]^\sharp \stackrel{def}{=} a_{q_0} = \top_{\mathcal{A}} \wedge \lambda a. \bigwedge_{q' \in (Q \setminus q_0)} a_{q'} = \bigsqcup_{(q,\tau,q') \in T} \mathsf{sem}^\sharp(\tau, a_q) \tag{3.4}$$

Exemple 3.2.5. *Dans le cas des intervalles, les fonctions de transfert sont définies comme suit. Les transitions vont d'un point de contrôle q vers un point de contrôle q' :*

- *Pour des gardes portant sur une variable x, nous ne considèrons que des conditions booléenes définies avec $\{>, \leq\}$ et une constante c :*

$$\mathit{sem}^\sharp_{\mathsf{g}}(x > c, I_q(x)) = I_q(x) \sqcap_{\mathcal{I}} [c+1, +\infty]$$

$$\mathit{sem}^\sharp_{\mathsf{g}}(x \leq c, I_q(x)) = I_q(x) \sqcap_{\mathcal{I}} [-\infty, c]$$

- *Pour les affectations d'une variable x, plusieurs cas sont pris en compte en fonction de la partie droite de l'affectation :*

$$\mathit{sem}^\sharp_{\mathsf{a}}(x := y, I_q(x)) = I_q(y)$$

$$\mathit{sem}^\sharp_{\mathsf{a}}(x := e, I_q(y)) = I_q(y)$$

$$\mathit{sem}^\sharp_{\mathsf{a}}(x := y+z, I_q(x)) = \begin{cases} [\ell_y + \ell_z, u_y + u_z] \ \textit{si}\ I_q(y) \stackrel{def}{=} [\ell_y, u_y] \ \textit{si}\ I_q(z) \stackrel{def}{=} [\ell_z, u_z] \\ \bot_{\mathcal{I}} \ \textit{sinon} \end{cases}$$

$$\mathit{sem}^\sharp_{\mathsf{a}}(x := x + c, I_q(x)) = \begin{cases} [\ell + c, u + c] \ \textit{si}\ I_q(x) \stackrel{def}{=} [\ell, u] \\ \bot_{\mathcal{I}} \ \textit{sinon} \end{cases}$$

$$\mathit{sem}^\sharp_{\mathsf{a}}(x := c, I_q(x)) = \begin{cases} [c, c] \ \textit{si}\ I_q(x) \neq \bot_{\mathcal{I}} \\ \bot_{\mathcal{I}} \ \textit{sinon} \end{cases}$$

3.2.1.6 Calcul de (post-)points fixes abstraits

Si le théorème de *Knaster-Tarski* nous affirme l'existence de points fixes, il ne nous fournit pas une méthode éfficace pour les calculer. En effet, calculer le plus petit point fixe par l'opérateur de borne inférieure sur l'ensemble des élements du domaine qui respecte une certaine condition, semble voué à l'echec dès lors que le domaine est infini. Une approche plus intuitive et, d'un point de vue calculatoire, plus réaliste, consisterait à appliquer la fonction sémantique un nombre de fois suffisant pour atteindre un point fixe. Un tel calcul est possible et le théorème 3.3 (de *Kleene*) nous en donne la garantie [Kle52].

Théorème 3.3 (Points fixes de Kleene)**.** *Soit $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}, \bigsqcup_{\mathcal{D}})$ un treillis complet et $f : \mathcal{D} \to \mathcal{D}$ une fonction monotone. Alors, pour $d \in \mathcal{D}$ un pré-point fixe de f, la séquence d'itérations $f^\delta(d)$ atteint le plus petit point fixe de f, où δ est un nombre ordinal :*

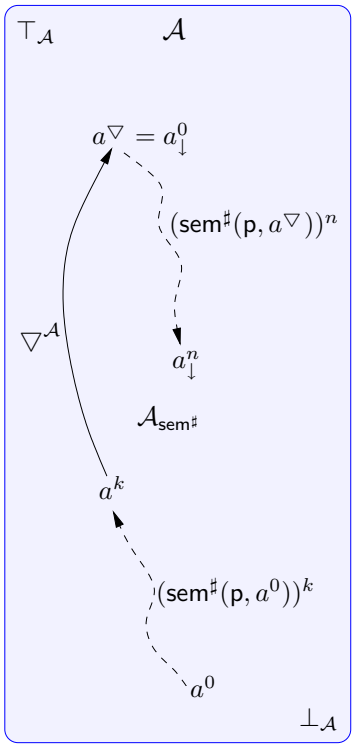
$$d \sqsubseteq_{\mathcal{D}} f(d) \sqsubseteq_{\mathcal{D}} f(f(d)) \sqsubseteq_{\mathcal{D}} \underset{\infty}{\ldots} \sqsubseteq_{\mathcal{D}} f^\delta(d) = f(f^\delta(d)) = \bot_{\mathcal{D}_f}$$

Ceci nous fournit donc une méthode, dans le cas où la fonction sémantique est monotone, convergente vers un point fixe pour des domaines finis. Le symbole ∞, sous les *trois petits points*, signifie que la séquence d'itération peut être infinie dès lors que le domaine du treillis est aussi infini, *i.e.* les *itérations de Kleene* ne peuvent, en général, converger en temps fini. Une solution pour palier ce problème consiste à sur-approximer la limite de la séquence d'itération au moyen d'un opérateur d'élargissement (*widening* en anglais, aussi appelé *accélérarteur*) [CC92], noté $\triangledown^{\mathcal{D}}$ (où $\mathcal{D}$ est le domaine dans lequel il s'applique).

Un opérateur d'élargissement calcule une limite d'une séquence infinie au vue des premiers éléments de cette séquence. Ainsi, l'opérateur d'élargissement s'applique à un certain élément du treillis d^k et à son image par f (soit $f(d^k)$) pour obtenir un d^{k+1} tel que $f(d^k) \sqsubseteq_{\mathcal{D}} d^{k+1}$. Cet *accélérateur* assure ainsi la convergence des itérations de f vers un élément $d^{\triangledown}$. De plus [CC92] établie que cet élément $d^{\triangledown}$ est au pire un *post-point fixe* de f, *i.e.* $\exists d \in \mathcal{D}_f, d \sqsubseteq_{\mathcal{D}} d^{\triangledown}$.

Si nous appliquons ce résultat à l'analyse statique, relativement à la fonction sémantique $\mathsf{sem}^\sharp$ dans un domaine abstrait $\mathcal{A}$, l'opérateur d'élargissement permet de calculer une propriété $a^{\triangledown}$ comme post-point fixe de $\mathsf{sem}^\sharp$. Cet élargissement s'effectue à partir d'un élément $a^k \stackrel{def}{=} (\mathsf{sem}^\sharp(\mathsf{p}, a^0))^k$, où a^0 est un pré-point fixe de la fonction sémantique : $a^{\triangledown} = a^k \triangledown^{\mathcal{A}} \mathsf{sem}^\sharp(\mathsf{p}, a^k)$.

Par définition d'un post-point fixe, la propriété $a^{\triangledown}$ est une sémantique abstraite correcte du programme p, mais peut demeurer trop imprécise pour être exploitée. Pour accroître la précision de l'analyse, il est possible d'améliorer cette approximation en itérant à nouveau la fonction $\mathsf{sem}^\sharp$ à partir de $a^{\triangledown}$.



Cette nouvelle séquence d'itération définie une suite descendante[3] vers le domaine $\mathcal{A}_{\mathsf{sem}^\sharp}$ des points fixes de $\mathsf{sem}^\sharp$ dans $\mathcal{A}$:

$$\begin{cases} a^0_{\downarrow} \stackrel{def}{=} a^{\triangledown} \\ a^{n+1}_{\downarrow} = \mathsf{sem}^\sharp(P, a^n_{\downarrow}) \end{cases}, \text{ telle que } a^{n+1}_{\downarrow} \sqsubseteq_{\mathcal{A}} a^n_{\downarrow},$$

Tous les éléments de cette séquence descendante sont des approximations correctes de $[\![\mathsf{p}]\!]$. Cependant, comme pour la séquence ascendante, le nombre d'itérations nécessaires pour atteindre un point fixe peut être infini. En conséquence, les analyses se contentent en général d'itérer un nombre fixé de fois la fonction sémantique, quitte à ne pas converger vers un point fixe, mais malgré tout vers une approximation correcte. Cette question de précision nous importe peu et est indépendante de notre travail. Notre position, nous le rappelons, est la suivante : Si l'analyse est satisfaisante pour l'utilisateur(*i.e* suffisamment précise pour être exploitée), nous fournissons *une méthologie pour prouver automa-*

[3]Certaines analyse statique utilisent un opérateur de rétrécissement $\triangle$ (*narrowing* en anglais) pour accélerer cette séquence descendante, mais nous n'y ferons pas référence ici.

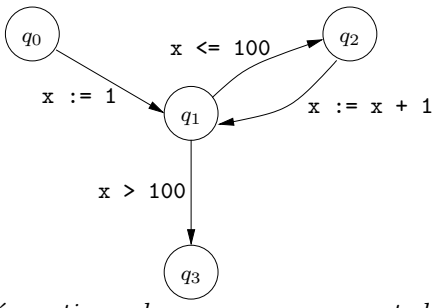
tiquement qu'elle est correcte, si effectivement elle l'est. Si l'analyse n'est pas correcte (du fait que son implantation est boguée par exemple), alors nous n'avons rien à prouver, car prouver des propriétés sémantiques fausses n'a pas de sens. En revanche, un arbre de dérivation sera tout de même construit mais ne sera pas validé par le vérificateur de preuve. L'un des avantages de générer des certificats sous forme d'arbre de dérivation plutôt qu'un verdict *valide/non valide*, est qu'il est possible de cibler dans l'arbre de dérivation les déductions rejetées par le vérificateur de preuves dans le cas où l'analyse est fausse. Cet aspect peut aider pour le débogage.

Dans la suite, nous considèrerons seulement le cas intéressant où l'analyse est correcte, et nous montrerons comment la justifier.

Avant cela, nous revenons sur le calcul de la sémantique abstraite en l'illustrant grâce au domaine des intervalles.

Exemple 3.2.6. *Nous prenons à titre d'exemple, l'analyse d'un programme impératif, calculant les intervalles de variations de ses variables. Le programme en question itère de* 1 *à* 100 *l'incrémentation de la variable* `x` *:*

```
void exemple(){
   int x;
   x = 1;
   while(x <= 100)
      {
      x = x + 1;
      }
}
```



Le domaine abstrait utilisé pour calculer la sémantique de ce programme est donc celui des intervalles. D'après l'équation 3.4, la sémantique de ce programme peut être obtenue en résolvant le système d'équations de point fixe suivant :

$$\begin{cases} I_{q_0} = [-\infty, +\infty] \\ I_{q_1} = \mathit{sem}^\sharp_a(x := 1, I_{q_0}) \sqcup_{\mathcal{I}} \mathit{sem}^\sharp_a(x := x + 1, I_{q_2}) \\ I_{q_2} = \mathit{sem}^\sharp_g(x \leq 100, I_{q_1}) = I_{q_1} \sqcap_{\mathcal{I}} [-\infty, 100] \\ I_{q_3} = \mathit{sem}^\sharp_g(x > 100, I_{q_1}) = I_{q_1} \sqcap_{\mathcal{I}} [101, +\infty] \end{cases}$$

Le pré-point fixe, I^0*, choisi pour résoudre ce système est défini comme suit :*

$$I^0 \stackrel{def}{=} I^0_{q_0} = [-\infty, +\infty] \bigwedge_{k \in \{1,2,3\}} I^0_{q_k} = \bot_{\mathcal{I}}$$

La figure 3.3 décrit l'évolution des propriétés abstraites calculées selon les itérations. Dans le cas sans élargissement (figure 3.3(a)), la propriété abstraite au point de contrôle q_1 *converge à la* $101^{ème}$ *itération* i.e.

$$I^{101}_{q_1} = I^{102}_{q_1} = \mathit{sem}^\sharp_a(x := 1), I^{101}_{q_0} \sqcup_{\mathcal{I}} \mathit{sem}^\sharp_a(x := x + 1), I^{101}_{q_2} = [1, 101].$$

Ainsi, la propriété abstraite en q_3 *qui demeurait indéfinie* ($\bot_{\mathcal{I}}$) *se voit affecter la valeur :*

$$I_{q_3}^{102} = I_{q_1}^{101} \sqcap_{\mathcal{I}} [101, +\infty] = [101, 101].$$

Dans le cas où l'opérateur $\triangledown^{\mathcal{I}}$ est employé (figure 3.3(b)), dès la deuxième itération un post-point fixe est atteint. Cela donne en I^3 et en I^4 des approximations correctes des valeurs concrètes de x, mais dont la précision n'est pas optimale :

$$I_{q_1}^3 = I_{q_1}^4 = [1, +\infty] \quad et \quad I_{q_3}^4 = I_{q_1}^3 \sqcap_{\mathcal{I}} [101, +\infty] = [101, +\infty]$$

En appliquant à nouveau la fonction sémantique sur la propriété abstraite calculée, l'analyse attaque une séquence descendante qui atteindra stabilité en un *pas.*

Évidemment, dans cet exemple très particulier, l'élargissement ne sert qu'à accélerer l'analyse et lui permet de terminer en cinq pas *plutot que* cent deux*. Mais dans beaucoup d'autres cas l'analyse sans élargissement ne termine pas et l'emploi de cet opérateur ne peut être vu comme une simple optimisation.*

	0	1(↗)	2(↗)	⋰	101(↗)	102(↗)
$I_{q_0} =$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$
$I_{q_1} =$	$\bot_{\mathcal{I}}$	$[1, 1]$	$[1, 2]$	$\ldots$	$[1, 101]$	$[1, 101]$
$I_{q_2} =$	$\bot_{\mathcal{I}}$	$[1, 1]$	$[1, 2]$	$\ldots$	$[1, 100]$	$[1, 100]$
$I_{q_3} =$	$\bot_{\mathcal{I}}$	$\bot_{\mathcal{I}}$	$\bot_{\mathcal{I}}$	$\bot_{\mathcal{I}}$	$\bot_{\mathcal{I}}$	$[101, 101]$

(a) Sans élargissement

	0	1(↗)	2(↗)	3 $\triangledown^{\mathcal{I}}$	4(↗)	1(↘)
$I_{q_0} =$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$	$[-\infty, +\infty]$
$I_{q_1} =$	$\bot_{\mathcal{I}}$	$[1, 1]$	$[1, 2]$	$[1, +\infty]$	$[1, +\infty]$	$[1, 101]$
$I_{q_2} =$	$\bot_{\mathcal{I}}$	$[1, 1]$	$[1, 2]$	$[1, 100]$	$[1, 100]$	$[1, 100]$
$I_{q_3} =$	$\bot_{\mathcal{I}}$	$\bot_{\mathcal{I}}$	$\bot_{\mathcal{I}}$	$\bot_{\mathcal{I}}$	$[101, +\infty]$	$[101, 101]$

(b) Avec élargissement et séquence descendante

Fig. 3.3 – Résolution du système de points fixes par itération de *Kleene*

3.2.2 Concrétisation vers la logique mathématique

Nous nous penchons maintenant sur la question suivante : « *comment prouver la correction des propriétés sémantiques calculées par ces analyses statiques ?* » Cette question conduit à s'en poser une autre : « *quelles propriétés sémantiques peut-on prouver en logique mathématique ?* » La logique mathématique et notamment la logique du premier ordre (FOL), est un langage formel, qui comme tout langage (formel ou non), a pour fonction d'exprimer des idées. Les langages formels, contrairement aux langages naturels, ont la particularité d'être accompagnés par des mécanismes de raisonnement rigoureux, fiables et vérifiables par machine, telle que *la déduction naturelle* (pour la FOL). Cela

permet d'exprimer des idées, de les démontrer et de vérifier ces démonstrations avec un niveau de confiance des plus élevés.

Dans cette étude, nous souhaitons prouver des propriétés sur les programmes, en utilisant un langage formel et les systèmes de déduction qui lui sont dédiés. Il en découle que le domaine sémantique concret que nous considérons (pour formaliser les comportements des programmes) doit être suffisamment riche pour nous permettre cela. Nous entendons par là que les comportements des programmes, dont nous tenterons d'établir la validité par des preuves formelles, devront être décrits par des formules logiques et il nous faut choisir une logique qui nous le permette. Le but de notre approche est d'utiliser les calculs effectués par un analyseur dans un domaine abstrait, pour obtenir des preuves de correction de programmes dans la logique employée pour définir la sémantique concrète.

Les propriétés, auxquelles nous allons nous intéresser dans la suite, que les analyses statiques dont nous avons présentées les bases sont capables de calculer, sont dites *de sureté*. Nous nous contenterons de la logique du premier ordre pour formaliser ces propriétés. Par conséquent, nous utiliserons, la *logique de Hoare* (H) pour formaliser la sémantique des programmes, la *logique du premier ordre* (FOL) pour définir les triplets de Hoare, la *sémantique de transformation de prédicats* (*wp*) pour passer des triplets de Hoare à des formules de la FOL , la *déduction naturelle* (NJ) pour construire les preuves validant ces triplets, puis le *calcul des constructions inductives* (CIC) pour modéliser tous ces formalismes, afin de vérifier automatiquement ces preuves dans le système Coq. D'autres logiques existent (logique temporelle [Pnu77], logique linéaire [Gir87], *etc*) et comme nous allons le voir, les abstractions envisageables dépendent directement de la logique choisie pour définir la sémantique dans le domaine concret.

3.2.2.1 Prouver les propriétés sémantiques dans le domaine concret

La logique du premier ordre contient un ensemble (infini) de formules, partiellement ordonné par la relation d'implication ($\Rightarrow$). Cet ensemble ordonné forme un treillis borné ($\text{FOL}, \Rightarrow, \vee, \wedge, \bot, \top$). Ce treillis est complet, car il admet une borne supérieure (ainsi qu'une borne inférieure) pour l'ensemble de ses parties : pour tout ensemble de formules, la disjonction de ces formules est une borne supérieure pour cet ensemble. L'ordre partiel défini par $\Rightarrow$ est semi-décidable dans le sens où il n'existe pas d'algorithme permettant de réfuter, pour deux formules quelconques Φ_1 et Φ_2, que l'une implique l'autre. *A fortiori*, si la signature qui définit la FOL permet de formaliser *l'arithmétique de Peano* ou la *théorie des ensembles*, alors l'ordre partiel défini par $\Rightarrow$ devient indécidable [Tur36]. Par conséquent, pour réaliser une analyse statique de programmes, il faut donc approximer le domaine sémantique défini par la logique du premier ordre, afin de se ramener à des propriétés (au moins) semi-décidables et dont il est aisé de construire un algorithme efficace pour le test d'inclusion, $\sqsubseteq$. En effet, il est plus facile de décider de l'implication de deux formules sur des sous-ensembles de la logique du premier ordre que sur la logique du premier ordre toute entière.

Nous rappelons cependant que notre objectif n'est pas de vérifier que les invariants calculés par l'analyse statique sont corrects, ce qui pourrait être traité avec des techniques de résolution de satisfiabilité de propositions modulo théorie (SMT-*solving* en anglais) [KOSS04, DdM06, dMDS07, Mon09, KS09], mais de générer une preuve formelle de la correction des invariants calculés, qui soit vérifiable par machine. Ainsi notre problématique ne se limite pas à un problème de décision, mais à trouver des algorithmes

générant ces preuves formelles vérifiables par machine.

3.2.2.2 Domaines abstraits considérés

L'intrepretation abstraite permet, en un certain sens, de définir des stratégies algorithmiques pour décider l'implication sur deux formules appartenant à un sous ensemble d'une logique donnée. Si nous souhaitons bénéficier des justifications des opérateurs abstraits dans notre approche pour la certification de programmes, les approximations que nous envisageons doivent ainsi rester dans la logique concrète considérée. Cela signifie que les éléments des domaines abstraits employés par les analyses statiques que nous souhaiterons instrumenter, doivent être également des termes de la logique du premier ordre, mais restreints à une *forme normale* sur laquelle le test d'implication peut être défini par une stratégie calculable et de compléxité (en temps de calcul) minimale.

Les analyses statiques de programmes impératifs que nous visons sont définies sur des domaines abstraits numériques et symboliques, basés sur *l'arithmétique linéaires*. Il existe une multitude de domaines de ce type, avec plus ou moins d'expressivité et un opérateur $\sqsubseteq$ proportionnellement complexe[4]. Pour ne citer que les plus connus : *Signes* [CC76], *Intervalles* [CC76], *Zones et Congruences de zone* [Min02], *Octogones* [Min01, Min06], *Octaèdres* [CC04], *Polyèdres* [CH78], *etc.*

Par exemple, l'opérateur $\sqsubseteq$ sur les polyèdres, pourra décider de l'inclusion entre deux formules linéaires (conjonction de contraintes de la formes $\Sigma_i \delta_{ij} v_i \leq c_j$, où $\delta_{ij} \in \{-1, 0, 1\}$, v_i est une variable du programme et c_j une constante) sur un grand sous ensemble (le plus grand des domaines cités) de la logique du premier ordre. En revanche, la complexité de ce test est polynomiale relativement au nombre de variables manipulées, ce qui n'est pas, en pratique, suffisamment efficace. Si nous choisissons le domaine des intervalles, qui est beaucoup moins expressif, le test d'inclusion admet une complexité linéaire, ce qui est beaucoup plus intéressant en pratique.

D'un point de vue des analyses symboliques que nous allons aborder par la suite le domaine abstrait est celui des *invariants quantifiés* [GMT08, HP08], basé sur les domaines numériques sus-mentionnés et permettant d'exprimer des propriétés sur le contenu des tableaux de programmes.

Ces domaines abstraits respectent la condition que nous posons (α : FOL $\rightarrow \mathcal{A}$, avec $\mathcal{A} \subset$ FOL) et nous permettent donc d'instrumenter les analyses qui se définissent dessus. Cela permet de conserver et d'exploiter les raisonnements construits dans le monde abstrait (par les justifications des opérateurs abstraits) pour prouver la validité des analyses dans le monde concret. La fonction d'abstraction, α, est alors définie comme une *normalisation* (plus un certain codage, parfois) des formules logiques concrètes vers les formules linéaires du domaine abstrait. Nous détaillons cela dans la section suivante en définissant le principe de justification d'une analyse statique par interprétation abstraite et calcul de points fixes.

3.2.3 Justification d'une analyse statique

Nous présentons dans cette section une méthodologie générique permettant à une analyse statique de justifier formellement ses calculs. Cette méthodologie s'appuie sur des patrons de preuve que nous associons à certaines fonctions de l'analyse. La conclusion de

[4] Compléxité algorithmique, nous entendons.

la preuve qui sert à justifier une analyse est un triplet de Hoare spécifiant le programme avec la propriété sémantique calculée par l'analyse. Soit p un programme, et Ψ une propriété calculée par l'analyse sur p. Il faut prouver le triplet $\{\Psi\}\mathsf{p}\{\Psi\}$ pour établir formellement que Ψ est un invariant inductif de p. Comme nous l'avons vu en section 3.2.1.5, cet invariant est partitionné selon les points de contrôle de p. Ainsi la preuve s'effectue point de contrôle par point de contrôle, en parcourant le système de transition représentant p pour montrer que l'invariant est inductif. La base de l'abre de dérivation, commune à toutes justifications, est donné en section 3.2.3.1.

3.2.3.1 Stratégie de preuve pour justifier une analyse statique

La toute première chose que nous souhaitons faire remarquer, est que pour justifier le résultat d'une analyse, toutes les étapes de l'analyse ne sont pas à prendre en considérartion. Seulement les ultimes pas et plus précisément ceux de la séquence descendante – nous supposerons qu'il y a toujours elargissement – nous sont utiles pour obtenir une preuve formelle de la sémantique calculée [Cha06]. Ceci est évident : les étapes qui précedent celles calculant le post-point fixe sont intermédiaires et calculent des propriétés ne caractérisant pas la sémantique du programme p traité. Nous rappellons que les approximations correctes de $[\![\mathsf{p}]\!]$, parmi les élements du domaine abstrait, sont celles satisfaisant l'équation 3.4, *i.e.* les points fixes et les post-points fixes de la fonction $\mathsf{sem}^\sharp$. Dans l'exemple 3.2.6, nous voyons sur la figure 3.3(b) qu'à la deuxième itération, la propriété calculée est fausse (*i.e* n'est pas une approximation de la sémantique du programme `exemple`). Du fait que nous souhaitons *prouver* – et non simplement *calculer* – et qu'il n'est guère envisageable de prouver des sémantiques éronnées, ces étapes ne présentent aucun interêt pour la certification du programme et peuvent être ignorées.

La seconde remarque, afin de rendre ce qui va suivre le plus général possible, est que nous considérerons que l'analyse n'atteint pas un point fixe. En effet, considérer que l'analyse après le calcul du post-point fixe par élargissement ne fait qu'atteindre, par les itérations decendantes, un autre post-point fixe, plus fin, plus précis, est plus général que le cas où elle atteint un point fixe :

> Soient p un programme impératif, et $a^\triangledown$ le *post-point fixe* calculé par l'opérateur d'accélération $\triangledown^{\mathcal{A}}$ de la fonction sémantique $\mathsf{sem}^\sharp : \mathcal{A} \to \mathcal{A}$. D'une part, l'atteignabilité d'un *point fixe* par itération de $\mathsf{sem}^\sharp$ sur p et $a^\triangledown$ n'est pas garantie, comme nous l'avons vu dans les sections précedentes. Cela a pour effet qu'en pratique, après le calcul du post-point fixe $a^\triangledown$, le nombre d'itérations de $\mathsf{sem}^\sharp$ est limité arbitrairement, à n. D'autre part, le fait que $a^n_\downarrow = (\mathsf{sem}^\sharp(\mathsf{p}, a^\triangledown))^n$ soit un post-point fixe, signifie que :
>
> $$a^{n+1}_\downarrow \sqsubseteq_{\mathcal{A}} a^n_\downarrow \sqsubseteq_{\mathcal{A}} a^{n-1}_\downarrow \sqsubseteq_{\mathcal{A}} \ldots \sqsubseteq_{\mathcal{A}} a^\triangledown,$$
>
> alors que si $a^n_\downarrow$ est un point fixe, on a :
>
> $$a^{n+1}_\downarrow \sqsubseteq_{\mathcal{A}} a^n_\downarrow \sqsubseteq_{\mathcal{A}} a^{n-1}_\downarrow \sqsubseteq_{\mathcal{A}} \ldots \sqsubseteq_{\mathcal{A}} a^\triangledown \wedge a^n_\downarrow \sqsubseteq_{\mathcal{A}} a^{n+1}_\downarrow,$$
>
> qui est effectivement moins général.

Remarque. *Les élements de la séquence ascendante n'étant pas pertinents pour justifier le résultat de l'analyse, nous omettons volontairement les $\downarrow$ en indice des élements*

de la séquence descendante, afin de rendre plus claire la lecture de ce qui va suivre : tous les élements du domaine abstrait auxquels nous ferons référence seront des (post-)points fixes issus de la séquence descendante de $\mathsf{sem}^\sharp$.

Ainsi, $(\mathsf{sem}^\sharp(\mathsf{p}, a^\triangledown))^n$ retourne une propriété abstraite a^n, qui n'est pas nécessairement un point fixe, mais qui demeure une approximation *correcte* de la sémantique de p. Lorsque l'analyse termine, a^n est la propriété abstraite calculée et $\Psi \stackrel{def}{=} \gamma(a^n)$ la propriété concrète retournée. Et il nous faut prouver que cette formule est une sémantique correcte pour le programme p, via le triplet de Hoare $\{\Psi\}\mathsf{p}\{\Psi\}$.

Structure globale de la preuve de justification. La méthode que nous suivons pour prouver ce triplet est celle proposée dans [Cha06] et consiste à appliquer une ultime fois la fonction sémantique $\mathsf{sem}^\sharp$ à p, afin de calculer une propriété $a^{n+1} \stackrel{def}{=} \mathsf{sem}^\sharp(\mathsf{p}, a^n)$ (dans le cas où a^n est un point fixe, $a^{n+1} = a^n$).

Soit $\Psi' = \gamma(a^{n+1})$. La preuve du triplet $\{\Psi\}\mathsf{p}\{\Psi'\}$ peut être obtenue automatiquement grâce à l'instrumentation du code de la fonction $\mathsf{sem}^\sharp$ (les détails sont donnés en section 3.2.3.2). En conséquence de la règle *strongest*de la logique de Hoare, la correction de $\{\Psi\}\mathsf{p}\{\Psi\}$ est établie si nous parvenons à fournir une preuve de l'implication $\Psi' \Rightarrow \Psi$, comme l'indique l'arbre de dérivation ci-dessous :

$$\dfrac{\dfrac{}{\vdash_{\text{H}} \{\Psi\}\mathsf{p}\{\Psi'\}}\ \textit{par justification de la fonction } \mathsf{sem}^\sharp \qquad \dfrac{}{\vdash_{\text{NJ}} \Psi' \Rightarrow \Psi}\ \textit{par justification de } \sqsubseteq^\pi_{\mathcal{A}}}{\vdash_{\text{H}} \{\Psi\}\mathsf{p}\{\Psi\}}\ \textit{strongest} \qquad (\Upsilon)$$

Prouver l'implication $\Psi' \Rightarrow \Psi$ avec deux formules quelconques d'arithmétique linéaire s'avère très complexe, décider cette implication étant déjà un problème NP-complet. Cependant, Ψ' et Ψ ne sont pas quelconques. Elles ont un lien calculatoire : $\Psi' \stackrel{def}{=} \gamma \circ \mathsf{sem}^\sharp(a^n)$. De plus nous savons [5] que $a^{n+1} \sqsubseteq_{\mathcal{A}} a^n$. Nous pouvons ainsi obtenir la preuve de $\Psi' \Rightarrow \Psi$ automatiquement grâce à l'instrumentation de $\sqsubseteq_{\mathcal{A}}$. En effet, si $a^{n+1} \sqsubseteq_{\mathcal{A}} a^n$ est vraie, alors une preuve de $\gamma(a^{n+1}) \Rightarrow \gamma(a^n)$ existe et nous l'obtenons grâce à la stratégie de $\sqsubseteq^\pi_{\mathcal{A}}$.

Exemple 3.2.7. *En référence au domaine des intervalles que nous avons défini en section 3.2.1.3, nous donnons l'instrumentation de l'opérateur de comparaison d'intervalles* $\sqsubseteq_{\mathcal{I}}$ *:*

$$I_1 \sqsubseteq^\pi_{\mathcal{I}} I_2 = \begin{cases} (\pi, \top)\ \textit{si}\ \ell_2 \leq \ell_1 \wedge u_1 \leq u_2\ \textit{avec}\ I_1 \stackrel{def}{=} [\ell_1, u_1]\ \textit{et}\ I_2 \stackrel{def}{=} [\ell_2, u_2] \\ (\_, \bot)\ \textit{sinon} \end{cases}$$

$$\textit{avec}\ \pi = \left\{ \dfrac{\dfrac{\vdash_{\text{NJ}} \ell_2 \leq \ell_1 \quad \dfrac{\dfrac{H}{\vdash_{\text{NJ}} \ell_1 \leq x}\ \wedge_e}{\vdash_{\text{NJ}} \ell_2 \leq x}\ \leq_{trans} \qquad \dfrac{\dfrac{\dfrac{H}{\vdash_{\text{NJ}} x \leq u_1}\ \wedge_e \quad \vdash_{\text{NJ}} u_1 \leq u_2}{\vdash_{\text{NJ}} x \leq u_2}\ \leq_{trans}}{\vdash_{\text{NJ}} \ell_2 \leq x \wedge x \leq u_2}\ \wedge_i}{\vdash_{\text{NJ}} \underbrace{\ell_1 \leq x \wedge x \leq u_1}_{\gamma(I_1)} \Rightarrow \underbrace{\ell_2 \leq x \wedge x \leq u_2}_{\gamma(I_2)}}\ \Rightarrow_i \right.$$

[5] Encore et toujours sous l'hypothèse que la fonction sémantique $\mathsf{sem}^\sharp$ se soit correctement exécutée

Concernant l'analyse de l'exemple 3.2.6, celle-ci s'arrête lorsque la stabilité est atteinte au $n-$ième pas de la phase descendante ($n = 1$ en l'occurrence), *i.e.* $\forall q \in Q, I_q^{n+1} \sqsubseteq_{\mathcal{I}} I_q^n = \top$.

Pour certifier cette analyse, il nous faut prouver le triplet $\{\gamma(I^n)\}\texttt{exemple}\{\gamma(I^n)\}$.

D'après l'arbre de dérivation (Υ), cela passe par fournir une preuve de $\gamma(I^{n+1}) \Rightarrow \gamma(I^n)$. Les invariants calculés sont partionnés selon les points de contrôle du système de transitions représentant le programme. Nous pouvons en conséquence prouver cette implication à chaque point de contrôle, au moyen de la version instrumentée de l'opérateur ensembliste $\sqsubseteq_{\mathcal{I}}$. Au point de contrôle q_1, la propriété calculée après élargissement est $I_{q_1}^{\triangledown} \stackrel{def}{=} [1, +\infty]$.

Après une application de la fonction sémantique, la propriété se réduit (toujours pour le point de contrôle q_1) à $[1, 101]$. En appliquant une fois de plus la fonction $\mathsf{sem}^\sharp$, la propriété n'évolue pas (*i.e.* la première itération de la séquence descendante atteint un point fixe, dans cet exemple) et l'implication $\gamma(I_{q_1}^{n+1}) \Rightarrow \gamma(I_{q_1}^n)$ est donc trivialement prouvable. Cependant, le patron de preuve associé à $\sqsubseteq_{\mathcal{I}}$ fournit une preuve valide de cette implication mais plus longue que nécessaire.

Concernant la branche gauche de l'arbre de dérivation (Υ) nous allons voir dans la section suivante comment la générer automatiquement grâce à l'instrumentation de la fonction $\mathsf{sem}^\sharp$.

3.2.3.2 Justification de la fonction sémantique

Pour pouvoir justifier l'analyse en fournissant une preuve de la sémantique calculée, il faut d'abord justifier la fonction sémantique. Cela permet de prouver automatiquement le triplet $\{\Psi\}\mathsf{p}\{\Psi'\}$ dans l'arbre de dérivation (Υ) de la section précédente.

De manière générale, la fonction sémantique abstraite $\mathsf{sem}^\sharp$, définie pour une certaine analyse statique, utilise (au moins) deux *fonctions de transfert* $\mathsf{sem}_{\mathsf{a}}^\sharp()$ et $\mathsf{sem}_{\mathsf{g}}^\sharp()$ pour calculer l'effet des transitions, respectivement d'affectation et de garde. Ces fonctions de transfert définissent les particularités de l'analyse et sont les parties (plus ou moins) compliquées de la fonction sémantique $\mathsf{sem}^\sharp$. Un défi est d'apporter une justification pour chaque résultat de l'analyse, en instrumentant un minimum de code de l'analyse, *i.e.* sans devoir instrumenter les fonctions de transfert en suivant leur définition algorithmique, mais en utilisant des fonctions déjà instrumentées telles que $\sqsubseteq_{\mathcal{A}}^{\pi}$. Nous montrons en section 3.2.4 comment notre approche permet d'éviter dans certains cas l'instrumentation de ces fonctions de transfert et de les justifier au moyen d'une extension minimale. Dans les autres cas, que nous caractériserons, l'instrumentation des fonctions de transfert sera necessaire et le travail à fournir pour justifier une analyse sera un peu plus couteux.

Nous partitionnons les domaines abstraits et concrets selon la structure du programme, sous sa représentation en système de transitions.

En figure 3.4, l'analyseur calcule la sémantique abstraite $a_{q'}^{n+1}$ – au point de contrôle q' – de la transition τ à partir de a_q^n, au moyen de la fonction sémantique $\mathsf{sem}^\sharp$. Le triplet $\{\Psi_q\}\tau\{\Psi'_{q'}\}$ est alors construit par application de la fonction γ sur a_q^n et $a_{q'}^{n+1}$, formalisant ainsi une approximation de la sémantique concrète de τ.

$\Psi'_{q'}$ n'est pas forcement la *plus forte post-condition* de Ψ_q relativement à τ, en raison de l'abstraction effectuée et des contraintes de syntaxe que s'impose l'analyse pour

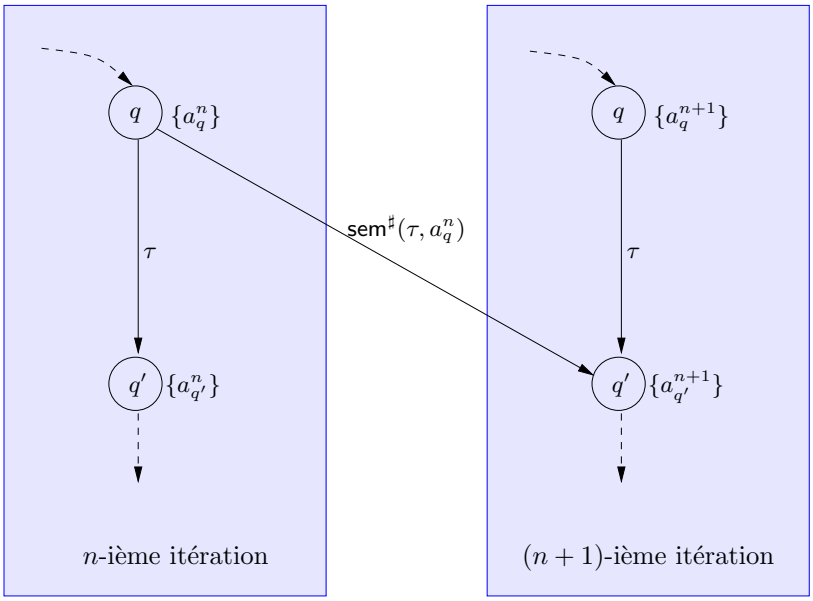


Fig. 3.4 – $(n+1)$-ième pas d'itération de l'analyse du programme entre q et q'

pouvoir terminer. Qu'importe, cette question de précision dépend de l'analyseur et non de notre travail de certification. La stratégie de preuve pour garantir la correction du résultat de cette analyse consiste simplement à calculer une formule $\Phi \stackrel{def}{=} wp(\tau, \Psi'_{q'})$ au moyen d'un calcul de *plus faible pré-condition*[6] [Cha06]. La figure 3.5 illustre cette stratégie de preuve, relativement à la transition donnée en figure 3.4. La correction de ce calcul de plus faible pré-condition est prouvée en amont, une fois pour toutes et nous assure que $\Psi_q \Rightarrow \Phi$ (cf figure 3.5), si l'analyse est correcte.

En somme, si l'analyse est correcte, l'implication est vraie. Si cette implication est vraie, alors une preuve formelle existe (théorème de complétude de Gödel) et le triplet $\{\Psi_q\}\tau\{\Psi'_{q'}\}$ est donc valide.

Prouver – et non seulement vérifier – cette implication entre Ψ_q et Φ est tout l'enjeu de ce travail. Notre idée pour cela, est d'utiliser les calculs de l'analyse. Ceci nous permet de passer d'une théorie du premier ordre *trés vaste*, où nous ne savons guère comment attaquer le problème, à une théorie réduite, définie par $\mathcal{A}$, nous fournissant de nombreux algorithmes déjà implantés. Nous précisons dans la section suivante comment notre principe d'instrumentation nous permet d'obtenir cette preuve automatiquement, que τ soit une garde ou une affectation.

3.2.4 Complétude de l'instrumentation avec patrons de preuve

L'interêt que présente l'utilisation de patrons de preuve, est que la structure des preuves est déterminé statiquement. Ceci garantie qu'une preuve sera toujours générée, car l'exécution de l'analyse ne fait que instancier les variables présentes dans les patrons de preuve. Pour garantir la complétude d'une instrumentation, il faut que celle-ci couvre tous les cas que sait traiter l'analyse. Ces cas dépendent des instructions admises par

[6] Ceci est équivalent aux obligations de preuve de la plate-forme WHY

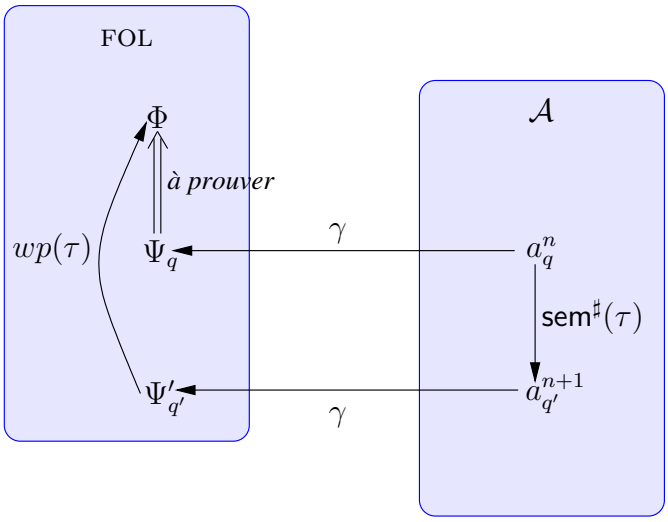


Fig. 3.5 – Stratégie de preuve pour justifier l'analyse

l'analyse et il faut être en mesure d'associer un patron de preuve pour chaque cas traité par les fonctions de transfert. Associer un patron de preuve pour chaque cas traité par les fonctions de transfert, ne signifie pas qu'il faut instrumenter toutes les fonctions de transfert (contrairement aux techniques proposées dans [SYY03, Cha06]). Dans certains cas, les patrons de preuve associés aux opérateurs ensemblistes du domaine abstrait suffiront pour justifier le résultats d'une certaine fonction de transfert. Nous tâcherons d'éclaircir ces différents cas, afin de pouvoir instrumenter tout analyseur statique avec un minimum de patrons de preuve. Pour cela, nous commençons par distinguer l'analyse des gardes, de celle des affectations, puis nous raffinerons dans ces deux types d'instruction les différents cas possibles. Cette analyse de cas est faite en toute généralité, selon la structure de la formule retournée par le calcul de plus faible pré-condition. En fonction de certains cas que nous avons identifiés, nous montrons que l'instrumentation est plus ou moins coûteuse (en terme de quantité de fonctions à instrumenter). Dans chacun des cas nous proposons un ensemble de fonctions à instrumenter afin de garantir la complétude de notre approche.

3.2.4.1 Justifications de l'analyse des affectations

Nous cherchons dans cette section à distinguer les différents cas, selon l'obligation de preuve (une implication logique) retournée par le calcul de plus faible pré-condition pour une affectation donnée. Notre but est de limiter l'instrumentation aux opérateurs ensemblistes du domaine asbtrait pour justifier chaque analyse, afin d'éviter d'instrumenter les fonctions de transfert ou de recourir à un prouveur automatique. Les éléments abstraits que nous considérons dans cette section sont des formules d'arithmétique linéaire sur les variables du programme, mais les raisonnements qui vont suivre sont applicables à des domaines abstraits plus riches, comme nous le verrons au chapitre suivant. Nous considérons également que les parties droites des affectations sont des expressions (non des formules) linéaires, $e(v_1, \ldots, v_m)$, des m variables du programme.

La fonction de transert pour les affectations, $\mathsf{sem}^\sharp_{\mathsf{a}}()$, de manière générale, construit

une formule linéaire appartenant au domaine abstrait, en intégrant l'expression $e(v_1, \ldots, v_m)$ à la formule courante (x' et v' sont les nouvelles valeurs de x et v dans $a_{q'}^{n+1}$) :

$$a_{q'}^{n+1} := \mathsf{sem}_{\mathsf{a}}^{\sharp}(x := e(v_1, \ldots, v_m), a_q^n) \stackrel{def}{=} \alpha(x' = e(v_1, \ldots, v_m)) \bigwedge_{v \in (\{v_1, \ldots, v_m\} \setminus x)} v' = v$$

L'expression $\alpha(x' = e(v_1, \ldots, v_m))$ consiste à tirer la meilleure information de l'égalité entre la variable modifiée et l'expression affectée, tout en se ramenant à un élément du domaine abstrait. Si le domaine abstrait autorise l'égalité alors $\alpha(x' = e(v_1, \ldots, v_m)) \rightarrow x' = e(v_1, \ldots, v_m)$. S'il n'autorise que l'inégalité alors cette égalité se représente par la formule $x' \leq e(v_1, \ldots, v_m) \wedge e(v_1, \ldots, v_m) \leq x'$, qui lui est logiquement équivalente. Une perte d'information peut provenir de la nature de l'expression elle-même : si l'expression est l'addition de deux variables, alors par exemple, dans le cas du domaine abstrait des intervalles, une telle égalité $x = y + z$ se réduit à :

$$y_{min} + z_{min} \leq x \ \wedge \ x \leq y_{max} + z_{max}$$

Si avant l'affectation l'analyse était précise sur les valeurs de y et z, $I_q(y) = [c, c]$ et $I_q(z) = [k, k]$ alors l'analyse en q' sera précise sur x, mais le fait que x soit égale à $y + z$ sera perdu :

$$c + k \leq x \ \wedge \ x \leq c + k, \text{ soit } x = c + k$$

Le problème que pose le manque d'expréssivité du domaine abstrait est de toute autre nature : la prouvabilité. L'approche que nous suivons consiste à calculer la plus faible pré-condition de $\gamma(a_{q'}^{n+1})$ par rapport à $x := e(v_1, \ldots, v_m)$, puis à prouver l'implication suivante [Cha06] :

$$\gamma(a_q^n) \Rightarrow wp(x := e(v_1, \ldots, v_m), \gamma(a_{q'}^{n+1}))$$

Nous proposons pour cela, non pas de développer un algorithme de preuve automatique *ad hoc* ou d'instrumenter la fonction de transfert qui a permis de calculer $a_{q'}^{n+1}$, mais d'utiliser l'instrumentation de $\sqsubseteq_{\mathcal{A}}$. Cet opérateur permet de décider l'inclusion (qui se traduit logiquement par l'implication) de deux formules du domaine abstrait. Pour prouver l'implication ci-dessus en se limitant à l'instrumentation du test d'inclusion du domaine abstrait et d'éviter d'instrumenter la fonction de transfert pour les affectations, il est donc nécessaire que wp ne retourne pas de formule non exprimable dans le domaine abstrait. Cela se formalise par l'égalité suivante :

$$\alpha \circ wp(x := e(v_1, \ldots, v_m), \gamma(a_{q'}^{n+1})) = wp(x := e(v_1, \ldots, v_m), \gamma(a_{q'}^{n+1}))$$

La justification de la fonction $\mathsf{sem}_{\mathsf{a}}^{\sharp}$ s'obtient alors sans l'instrumenter, simplement grâce à l'instrumentation de l'opérateur $\sqsubseteq_{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \rightarrow \mathbb{B}$ (voir exemple 3.2.7 pour une illustration de la preuve retournée par $\sqsubseteq_{\mathcal{I}}^{\pi}$) :

$$\dfrac{\dfrac{par \ \sqsubseteq_{\mathcal{A}}^{\pi}}{\vdash_{\text{NJ}} \gamma(a_q^n) \Rightarrow \gamma(a_{q'}^{n+1})\sigma}}{\vdash_{\text{H}} \{\gamma(a_q^n)\} x := e \{\gamma(a_{q'}^{n+1})\}} \ sem$$

Ceci constitue le cas simple de la justification de l'analyse des affectations. Nous présentons maintenant le cas où wp ne préserve pas les invariants dans $\mathcal{A}$, que nous illustrons sur le domaine des intervalles.

Cas où wp ne préserve pas les invariants dans le domaine abstrait. En revanche, dans le cas où l'affectation est $x := y + z$, la formule calculée par wp n'est pas exprimable dans le domaine des intervalles par exemple. En effet,

$$wp(x := y + z, (y_{min} + z_{min} \leq x \ \wedge \ x \leq y_{max} + z_{max})) = y_{min} + z_{min} \leq y + z \ \wedge \ y + z \leq y_{max} + z_{max}$$

Cela pose un problème. Dans cet exemple, l'implication $\gamma(a_q^n) \Rightarrow wp(x := e(v_1, \ldots, v_m), \gamma(a_{q'}^{n+1}))$ est vraie, mais notre stratégie de preuve utilisant $\sqsubseteq_{\mathcal{A}}^{\pi}$ n'est pas en mesure de le prouver, car $\mathcal{I}$ ne peut pas exprimer des formules du type $v_i + v_j \leq c$ (le domaine abstrait des octogones le peut [Min01]).

Nous sommes donc ici dans un cas où l'analyse en avant ne peut pas être justifiée par une stratégie de preuve en arrière, utilisant simplement l'instrumentation de l'opérateur $\sqsubseteq_{\mathcal{A}}$. La fonction de transfert $\mathsf{sem}_{\mathsf{a}}^{\sharp}$ doit être instrumentée pour produire des preuves de :

$$\gamma(a_q^n) \Rightarrow wp(x := e(v_1, \ldots, v_m), \gamma(a_{q'}^{n+1})).$$

Dans l'exemple des intervalles et de l'affectation, $x := y+z$, une telle instrumentation n'est pas difficile à développer et consiste à ajouter la méta preuve qui va suivre, au code de $\mathsf{sem}_{\mathsf{a}}^{\sharp}$ que nous donnons d'abord. Pour des analyses plus complexe, un tel exercice d'intrumentation de fonction de transfert est nettement plus difficile à developper.

La fonction $\mathsf{sem}_{\mathsf{a}}^{\sharp}$ est définie récursivement sur la longueur de l'expression $e(v_1, \ldots v_m)$ en partie droite. Nous écrivons son code dans un pseudo-langage proche de OCaml :

```
let sem♯a(a^n_q, x := e) ≝ let (π, a^{n+1}_{q'}) =
  let rec compute_min (ex, av) =
    match ex with
    | v → let (v_min, v_max) = extract_inter(v, av) in (π^v_min, v_min)
    | v + e → let (π^v_min, v_min) = compute_min(v, av)
              and (π^e_min, e_min) = compute_min(e, av) in (π^{v+e}_min, v_min + e_min)

  and compute_max(ex, av) =
    match ex with
    | v → let (v_min, v_max) = extract_inter(v, av) in (π^v_max, v_max)
    | v + e → let (π^v_max, v_max) = compute_max(v, av)
              and (π^e_max, e_max) = compute_max(e, av) in (π^{v+e}_max, v_max + e_max)

  in let (π_min, e_min) = compute_min(e, a^n_q)
     and (π_max, e_max) = compute_max(e, a^n_q)
     in (π, update(a^n_q, x, (e_min, e_max)))
```

$$avec\ \pi_{min}^{v} = \left\{ \frac{\gamma(a_q^n)}{v_{min} \leq v} \wedge_e \right. \qquad et \qquad \pi_{max}^{v} = \left\{ \frac{\gamma(a_q^n)}{v \leq v_{max}} \wedge_e \right.$$

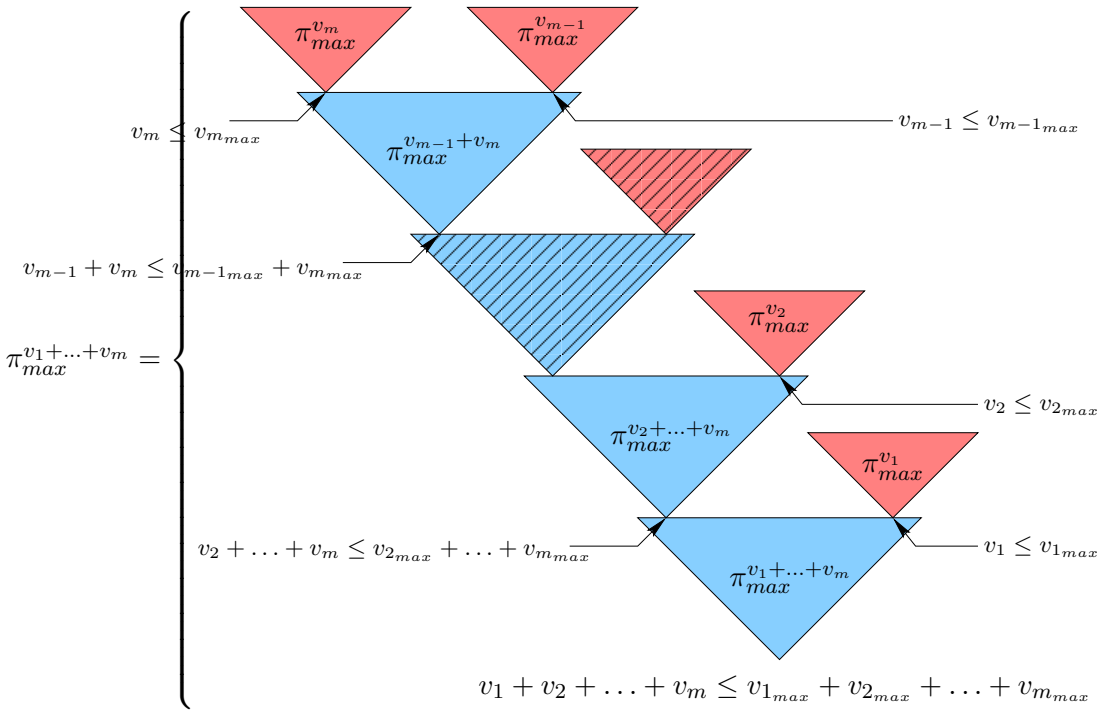
$$et\ \pi^{v+e}_{min} = \left\{ \dfrac{\dfrac{\dfrac{\dfrac{\gamma(a^n_q)}{v_{min} \leq v}\ \wedge_e}{\vdash_{\text{NJ}} v_{min} - v \leq 0}\ +_c(-v)}{\vdash_{\text{NJ}} v_{min} + e_{min} - v \leq e_{min}}\ +_c(e_{min}) \qquad \dfrac{\pi^e_{min}}{\vdash_{\text{NJ}} e_{min} \leq e}}{\dfrac{\vdash_{\text{NJ}} v_{min} + e_{min} - v \leq e}{\vdash_{\text{NJ}} v_{min} + e_{min} \leq v + e}\ +_c(v)}\ \leq_{trans} \right.$$

$$et\ \pi^{v+e}_{max} = \left\{ \dfrac{\dfrac{\pi^e_{max}}{\vdash_{\text{NJ}} e \leq e_{max}} \qquad \dfrac{\dfrac{\dfrac{\gamma(a^n_q)}{v \leq v_{max}}\ \wedge_e}{\vdash_{\text{NJ}} 0 \leq v_{max} - v}\ +_c(-v)}{\vdash_{\text{NJ}} e_{max} \leq v_{max} + e_{max} - v}\ +_c(e_{max})}{\dfrac{\vdash_{\text{NJ}} e \leq v_{max} + e_{max} - v}{\vdash_{\text{NJ}} v + e \leq v_{max} + e_{max}}\ +_c(v)}\ \leq_{trans} \right.$$

$$puis\ \pi = \left\{ \dfrac{\dfrac{\dfrac{\dfrac{\pi_{min}}{\vdash_{\text{NJ}} e_{min} \leq e} \qquad \dfrac{\pi_{max}}{\vdash_{\text{NJ}} e \leq e_{max}} \qquad \dfrac{\overbrace{\gamma(a^n_q)}^{H_1}}{\vdash_{\text{NJ}} \varepsilon}\ \wedge_e}{\vdash_{\text{NJ}} e_{min} \leq e \wedge e \leq e_{max} \wedge \varepsilon}\ \wedge_i}{\vdash_{\text{NJ}} \gamma(a^n_q) \Rightarrow e_{min} \leq e \wedge e \leq e_{max} \wedge \varepsilon}\ \Rightarrow_i(H_1)}{\vdash_{\text{H}} \{\gamma(a^n_q)\}x := e\{\underbrace{e_{min} \leq x \wedge x \leq e_{max} \wedge \varepsilon}_{\gamma(a^{n+1}_{q'})}\}}\ sem \right.$$

Plusieurs remarques :
Les intervalles des autres variables que x, n'étant pas modifiés par l'affectation, sont dans la formule ε de $\gamma(a^{n+1}_{q'})$. Cette formule est une sous-formule de $\gamma(a^n_q)$ (la partie des variables non modifiées) et se prouve triviallement en éliminant les parties non préservées dans $a^{n+1}_{q'}$, par la régle $(\wedge_e)$. La variable x ne peut apparaître dans ε, car l'intervalle contenant x dans a^n_q est retiré par la fonction update, qui construit $a^{n+1}_{q'}$ en mettant à jour le nouvel intervalle de x calculé. Cela est necessaire afin de ne pas introduire de contradiction. Ainsi, le calcul de plus faible pré-condition n'a pas d'effet sur ε.

Afin d'illustrer le mécanisme de construction des preuves en suivant les appels récursifs des fonctions compute_min et compute_max, nous donnons la structure de l'arbre de dérivation construit par la fonction compute_max dans le cas d'une affectation de la forme $x := v_1 + \ldots + v_m$:



Nous appliquons cela à notre exemple où e est l'addition de deux variables. Ici, $m = 2$ et donc $\pi^{v_1}_{max} = \pi^{v_{m-1}}_{max} = y$, et $\pi^{v_2}_{max} = \pi^{v_m}_{max} = z$. Cela nous donne :

$$\pi^{y+z}_{max} = \left\{ \dfrac{\dfrac{\gamma(a^n_q)}{\vdash_{\text{NJ}} z \leq z_{max}}\wedge_e \quad \dfrac{\dfrac{\dfrac{\gamma(a^n_q)}{y \leq y_{max}}\wedge_e}{\vdash_{\text{NJ}} 0 \leq y_{max} - y}+_c(-y)}{\vdash_{\text{NJ}} z_{max} \leq y_{max} + z_{max} - y}+_c(z_{max})}{\dfrac{\vdash_{\text{NJ}} z \leq y_{max} + z_{max} - y}{\vdash_{\text{NJ}} y + z \leq y_{max} + z_{max}}+_c(v)}\leq_{trans} \right.$$

En intégrant π^{y+z}_{min} et en faisant abstraction de ε, cela nous donne le patron de preuve suivant :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\gamma(a^n_q)}^{H_1}}{\vdash_{\text{NJ}} y_{min} \leq y}\wedge_e}{\vdash_{\text{NJ}} y_{min} - y \leq 0}+_c(-y)}{\vdash_{\text{NJ}} y_{min} + z_{min} - y \leq z_{min}}+_c(z_{min}) \quad \dfrac{\vdash_{\text{NJ}} \overbrace{\gamma(a^n_q)}^{H_1}}{\vdash_{\text{NJ}} z_{min} \leq z}\wedge_e}{\vdash_{\text{NJ}} y_{min} + z_{min} - y \leq z}\leq_{trans}}{\vdash_{\text{NJ}} y_{min} + z_{min} \leq y + z}+_c(y) \quad \dfrac{\pi^{y+z}_{max}}{\vdash_{\text{NJ}} y + z \leq y_{max} + z_{max}}}{\vdash_{\text{NJ}} y_{min} + z_{min} \leq y + z \wedge y + z \leq y_{max} + z_{max}}\wedge_i$$
$$\dfrac{\vdash_{\text{NJ}} y_{min} + z_{min} \leq y + z \wedge y + z \leq y_{max} + z_{max}}{\vdash_{\text{NJ}} \gamma(a^n_q) \Rightarrow y_{min} + z_{min} \leq y + z \wedge y + z \leq y_{max} + z_{max}}\Rightarrow_i (H_1)$$
$$\dfrac{\vdash_{\text{NJ}} \gamma(a^n_q) \Rightarrow y_{min} + z_{min} \leq y + z \wedge y + z \leq y_{max} + z_{max}}{\vdash_{\text{H}} \{\gamma(a^n_q)\} x := y + z \{y_{min} + z_{min} \leq x \wedge x \leq y_{max} + z_{max}\}}\ sem$$

Exemple 3.2.8. *Nons considérons un programme ayant quatre variables y, z, t et x.*

Soit la transition $q \xrightarrow{x:=y+z} q'$, *avec* $\gamma(a_q^n) \stackrel{def}{=} \bigwedge \begin{pmatrix} \ell_y \leq y \wedge y \leq u_y \\ \ell_z \leq z \wedge z \leq u_z \\ \ell_t \leq t \wedge t \leq u_t \\ \ell_x \leq x \wedge x \leq u_x \end{pmatrix}$

La propriété calculée en q' *par* $\mathsf{sem}_a^\sharp$ *est donc* $\gamma(a_{q'}^{n+1}) \stackrel{def}{=} \bigwedge \begin{pmatrix} \ell_y \leq y \wedge y \leq u_y \\ \ell_z \leq z \wedge z \leq u_z \\ \ell_t \leq t \wedge t \leq u_t \\ \ell_y + \ell_z \leq x \wedge x \leq u_y + u_z \end{pmatrix}$

On remarque d'abord que les valeurs ℓ_x *et* u_x *qui servaient à définir l'intervalle de* x *en* q, *n'apparaissent plus en* q'. *Celles-ci ont été effacées afin de ne pas contredire le nouvel intervalle calculé pour* x. *Puis on remarque que les intervalles de* y, z *et* t *sont inchangés.*

Aprés le calcul de plus faible pré-condition, la formule que nous devons prouver est :

$$wp \circ \gamma(a_{q'}^{n+1}) \stackrel{def}{=} \bigwedge \begin{pmatrix} \ell_y \leq y \wedge y \leq u_y \\ \ell_z \leq z \wedge z \leq u_z \\ \ell_t \leq t \wedge t \leq u_t \\ \ell_y + \ell_z \leq y + z \wedge y + z \leq u_y + u_z \end{pmatrix}$$

Cette formule est prouvée en deux étapes :

1. *Les intervalles de* y, z *et* t *sont prouvés trivialement par égalité syntaxique, car ils apparaissent à l'identique dans les deux formules. Cette partie n'apparaît pas dans le patron de preuve ci-dessus.*

2. *L'intervalle de* $y + z$ *introduit par la substitution sur* x, *induite par l'affectation* $x := y + z$, *est prouvé grâce à l'instanciation des valeurs de* $y_{min}, z_{min}, y_{max}$ *et* z_{max}, *par* ℓ_y, ℓ_z, u_y *et* u_z, *dans le patron de preuve donné précédemment.*

Le cas que nous venons de présenter, où le calcul de la plus faible pré-condition retourne une formule non exprimable dans le domaine abstrait, est défavorable pour mettre en pratique cette technique d'instrumentation d'analyseurs statiques. En effet, le fait de devoir instrumenter les fonctions de transfert est coûteux, et se traite au cas par cas, sans pouvoir bénéficier d'une méthodologie générale. Cependant, si espérer pouvoir toujours tomber dans le premier cas, où le calcul de plus faible pré-condition retourne une formule du domaine abstrait semble déraisonnable, une troisième situation peut être rencontrée : le calcul de plus faible pré-condition retourne une formule n'appartenant pas au domaine abstrait mais pour autant exprimable dans le domaine abstrait. C'est à dire que par une normalisation de $wp \circ \gamma(a_{q'}^{n+1})$, il est possible de construire une formule equivalente appartenant au domaine abstrait. La justification de l'analyse dans ce cas sera obtenue par la seule intrumentation du test d'inclusion, au prix de cette normalisation de l'obligation de preuve retournée par wp. Cette normalisation peut être justifiée soit de manière déductive par un patron de preuve (*cf.* figure 3.6(a)) après que le calcul de plus faible pré-condition ait été appliqué (comme pour la traduction des gardes dans le domaine abstrait), soit de manière constructive en l'intégrant à la définition de la fonction wp (*cf.* figure 3.6(b)). Nous donnerons un exemple de ce cas au chapitre suivant en section 4.2.3.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\text{NJ}} \gamma(a_q^n) \Rightarrow \alpha(\gamma(a_{q'}^{n+1})\sigma)}\ par \sqsubseteq_{\mathcal{A}}^{\pi}}{\vdots}}{\textit{justification de la normalisation}}}{\dfrac{\vdots}{\vdash_{\text{NJ}} \gamma(a_q^n) \Rightarrow \gamma(a_{q'}^{n+1})\sigma}}}{\vdash_{\text{H}} \{\gamma(a_q^n)\}x := e\{\gamma(a_{q'}^{n+1})\}}\ sem$$

(a) La normalisation est effectuée après le calcul de plus faible pré-condition et donc justifié par un patron de preuve.

$$\dfrac{\dfrac{}{\vdash_{\text{NJ}} \gamma(a_q^n) \Rightarrow \alpha(\gamma(a_{q'}^{n+1})\sigma)}\ par \sqsubseteq_{\mathcal{A}}^{\pi}}{\vdash_{\text{H}} \{\gamma(a_q^n)\}x := e\{\gamma(a_{q'}^{n+1})\}}\ sem$$

(b) La normalisation est intégrée à wp et donc justifiée par la preuve de correction de wp.

Fig. 3.6 – Deux possibilités pour justifier l'analyse des affectations dans le cas où wp retourne une formule n'appartenant pas au domaine abstrait, mais exprimable dedans

3.2.4.2 Justifications de l'analyse des gardes

Le traitement des gardes se résume, de manière générale, à intégrer l'information obtenue par *le passage de la garde* à la propriété courante. Ainsi, $a_q^n \wedge \alpha(g)$ est l'information la plus précise que peut calculer $\mathsf{sem}_{\mathsf{g}}^{\sharp}(a_q^n, g)$.

La représentation abstraite de la conjonction est définie par $\alpha(\Phi_1 \wedge \Phi_2) \stackrel{def}{=} \alpha(\Phi_1) \sqcap_{\mathcal{A}} \alpha(\Phi_2)$, où $\alpha(\Phi_1) \sqcap_{\mathcal{A}} \alpha(\Phi_2)$ est la plus grande borne inférieure des formules $\alpha(\Phi_1)$ et $\alpha(\Phi_2)$. Formellement, cela signifie (entre autres) que $\alpha(\Phi_1) \sqcap_{\mathcal{A}} \alpha(\Phi_2) \sqsubseteq_{\mathcal{A}} \alpha(\Phi_1)$ et $\alpha(\Phi_1) \sqcap_{\mathcal{A}} \alpha(\Phi_2) \sqsubseteq_{\mathcal{A}} \alpha(\Phi_2)$.

Concernant les domaines abstraits que nous considérons, la conjonction est un opérateur admis par la syntaxe des éléments abstraits. Cela implique que dans certains cas, $\alpha(\Phi_1) \sqcap_{\mathcal{A}} \alpha(\Phi_2) \stackrel{def}{=} \alpha(\Phi_1) \wedge \alpha(\Phi_2)$. Toutefois, il arrive que $\alpha(\Phi_1) \sqcap_{\mathcal{A}} \alpha(\Phi_2)$ soit normalisée vers une formule logiquement équivalente à $\alpha(\Phi_1) \wedge \alpha(\Phi_2)$ mais non syntaxiquement identique. En effet, les domaines abstraits que nous considéront sont fermés pour la conjonction, *i.e.* la conjonction est abstraite sans perte d'information, au sens suivant[7] :

$$\forall a_1, a_2 \in \mathcal{A}, \gamma(a_1) \wedge \gamma(a_2) \Rightarrow \gamma(a_1 \sqcap_{\mathcal{A}} a_2).$$

Pour revenir au cas des gardes, nous pouvons en déduire que la conjonction de la pré-condition et de l'abstraction de la garde implique leur intersection :

$$\gamma(a_q^n) \wedge \gamma \circ \alpha(g) \Rightarrow \gamma(a_q^n \sqcap_{\mathcal{A}} \alpha(g))$$

Comme les domaines que nous considérons forment des sous-classes complètes de la logique du premier ordre, les formules vraies sont prouvables.

Nous en concluons qu'une preuve de cette implication existe. Nous allons maintenant expliquer sur un exemple comment l'obtenir par l'instrumentation de l'opérateur $\sqcap_{\mathcal{A}}$.

[7] L'implication réciproque est valide quelque soit le domaine $\mathcal{A}$, d'après la correction de $\sqcap_{\mathcal{A}}$.

Exemple 3.2.9. *Nous illustrons l'instrumentation de $\sqcap_{\mathcal{A}}$ sur le domaine des intervalles. L'opérateur d'intersection est défini en section 3.2.1.3, nous en donnons maintenant sa version instrumentée :*

$$I_1 \sqcap^{\pi}_{\mathcal{I}} I_2 = \begin{cases} (\pi, [max(\ell_1, \ell_2), min(u_1, u_2)]) \text{ si } I_1 \stackrel{def}{=} [\ell_1, u_1] \text{ et } I_2 \stackrel{def}{=} [\ell_2, u_2] \\ (\_, \bot) \text{ sinon} \end{cases}$$

$$\text{avec } \pi = \left\{ \dfrac{\dfrac{\dfrac{H_1}{\vdash_{\text{NJ}} \ell_{max} \leq x}\,\wedge_e \quad \dfrac{H_1}{\vdash_{\text{NJ}} x \leq u_{min}}\,\wedge_e}{\vdash_{\text{NJ}} \ell_{max} \leq x \wedge x \leq u_{min}}\,\wedge_i}{\vdash_{\text{NJ}} \underbrace{\ell_1 \leq x \wedge x \leq u_1}_{\gamma(I_1)} \wedge \underbrace{\ell_2 \leq x \wedge x \leq u_2}_{\gamma(I_2)} \Rightarrow \underbrace{\ell_{max} \leq x \wedge x \leq u_{min}}_{\gamma(I_1 \sqcap_{\mathcal{I}} I_2)}}\,\Rightarrow_i (H_1) \right.$$

$$\text{où } \ell_{max} \stackrel{def}{=} max(\ell_1, \ell_2) \text{ et } u_{min} \stackrel{def}{=} min(u_1, u_2)$$

Pour finir la preuve de correction de l'analyse d'une garde, il reste à démontrer que $\gamma(a^{n+1}_{q'})$ est une sur-approximation de $\gamma(a^n_q \sqcap_{\mathcal{A}} \alpha(g))$. La preuve de $(\gamma(a^n_q \sqcap_{\mathcal{A}} \alpha(g)) \Rightarrow \gamma(a^{n+1}_{q'})$ peut s'obtenir par la relation d'ordre instrumentée $(\sqsubseteq^{\pi}_{\mathcal{A}})$ du domaine abstrait, car $a^n_q \sqcap_{\mathcal{A}} \alpha(g) \in \mathcal{A}$, et $a^{n+1}_{q'} \in \mathcal{A}$ également.

Nous donnons désormais le patron de preuve que nous intégrons dans le code de $\mathsf{sem}^{\sharp}_{\mathsf{g}}$, afin d'obtenir une justification de ses calculs.

$$\dfrac{\dfrac{\dfrac{\overbrace{\vdash_{\text{NJ}} \gamma(a^n_q)}^{H_1} \quad \dfrac{\vdash_{\text{NJ}} \overbrace{g}^{H_2} \quad \dfrac{\text{par } \alpha(g)^{\pi}}{\vdash_{\text{NJ}} g \Rightarrow \gamma \circ \alpha(g)}}{\vdash_{\text{NJ}} \gamma \circ \alpha(g)}\,\Rightarrow_e}{\vdash_{\text{NJ}} \gamma(a^n_q) \wedge \gamma \circ \alpha(g)}\,\wedge_i \quad \begin{array}{c} \text{puis par } \sqsubseteq^{\pi}_{\mathcal{A}} \\ \vdots \\ \text{par } \sqcap^{\pi}_{\mathcal{A}} \end{array} \dfrac{}{\vdash_{\text{NJ}} (\gamma(a^n_q) \wedge \gamma \circ \alpha(g)) \Rightarrow \gamma(a^{n+1}_{q'})}}{\vdash_{\text{NJ}} \gamma(a^{n+1}_{q'})}\,\Rightarrow_e}{\vdash_{\text{NJ}} g \Rightarrow \gamma(a^{n+1}_{q'})}\,\Rightarrow_i [H_2]}{\vdash_{\text{NJ}} \gamma(a^n_q) \Rightarrow (g \Rightarrow \gamma(a^{n+1}_{q'}))}\,\Rightarrow_i [H_1]}{\vdash_{\text{H}} \{\gamma(a^n_q)\}?g\{\gamma(a^{n+1}_{q'})\}}\,sem$$

Dans de nombreux cas nous aurons même l'égalité entre $a^{n+1}_{q'}$ et $a^n_q \sqcap_{\mathcal{A}} \alpha(g)$, mais il se peut, pour être général, que l'analyse approxime $\gamma(a^n_q \sqcap_{\mathcal{A}} \alpha(g))$ pour des raisons techniques.

En conclusion, nous savons que par l'instrumentation de $\sqcap^{\pi}_{\mathcal{A}}$ et de $\sqsubseteq^{\pi}_{\mathcal{A}}$ une preuve du triplet $\{\gamma(a^n_q)\}?g\{\gamma(a^{n+1}_{q'})\}$ sera systématiquement générée, grâce au patron de preuve ci-dessus.

Exemple 3.2.10. *Nous illustrons la génération d'une telle preuve en reprenant l'exemple 3.2.6. Le point de contrôle q_2 est accessible à partir de q_1 si la garde $g \stackrel{def}{=}$ `x <= 100` est vérifiée.*

La génération de preuve se fait lorsque la stabilité est atteinte, après n itérations descendantes : au point de contrôle q_2 l'invariant calculé est $I^n_{q_2} \stackrel{def}{=} [1, 100]$. L'itération

supplémentaire que nous effectuons pour générer la preuve n'améliore pas la précison de l'invariant calculé et $I_{q_2}^{n+1} \stackrel{def}{=} [1, 100]$. *Au point de contrôle* q_1 *l'invariant est* $I_{q_1}^{n} \stackrel{def}{=} [1, 101]$. *Le triplet à prouver est le suivant :*

$$\{\underbrace{1 \leq x \leq 101}_{\gamma(I_{q_1}^n)}\}\texttt{x <= 100}\{\underbrace{1 \leq x \leq 100}_{\gamma(I_{q_2}^{n+1})}\}$$

La preuve est générée par la version instrumentée de $\mathsf{sem}_g^\sharp$ *qui instancie le patron de preuve donnée par les valeurs de* $I_{q_1}^n$ *et* $I_{q_2}^{n+1}$. *Cela nous donne l'arbre de dérivation suivant :*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}H_1\\ \vdots\\ \vdash_{\text{NJ}} 1 \leq x \leq 101 \wedge x \leq 100\end{matrix} \quad \begin{matrix}H_2\\ \vdots\\ \vdash_{\text{NJ}} x \leq 100\end{matrix}}{\vdash_{\text{NJ}} 1 \leq x \leq 101 \wedge x \leq 100}\wedge_i \quad \dfrac{\text{par } \sqcap_{\mathcal{I}}^\pi \text{ ou } \sqsubseteq_{\mathcal{I}}^\pi}{\vdash_{\text{NJ}} (1 \leq x \leq 101 \wedge x \leq 100) \Rightarrow 1 \leq x \leq 100}}{\vdash_{\text{NJ}} 1 \leq x \leq 100}\Rightarrow_e}{\vdash_{\text{NJ}} x \leq 100 \Rightarrow 1 \leq x \leq 100}\Rightarrow_i [H_2]}{\vdash_{\text{NJ}} 1 \leq x \leq 101 \Rightarrow (x \leq 100 \Rightarrow 1 \leq x \leq 100)}\Rightarrow_i [H_1]}{\vdash_{\text{H}} \{1 \leq x \leq 101\}\,\texttt{?(x <= 100)}\{1 \leq x \leq 100\}}\,sem$$

Dans cet exemple, l'abstraction ne fait pas perdre d'information à la garde : $\alpha(x \leq 100) = [-\infty, 100]$ *et* $\gamma([-\infty, 100]) = x \leq 100 = g$. *En conséquence la branche gauche de l'arbre est triviale à construire, et l'appel à la fonction* α^π *n'a pas d'utilité.*

Concernant la branche droite, $\sqsubseteq_{\mathcal{I}}^\pi$ *construira une preuve de* $(1 \leq x \leq 101 \wedge x \leq 100) \Rightarrow 1 \leq x \leq 100$, *en prenant en partie gauche deux intervalles sur* x. *En revanche, si l'algorithme ne traite qu'un intervalle par variable du programme, alors cette implication ne peut être prouvée directement par* $\sqsubseteq_{\mathcal{I}}^\pi$. *Il faut au préalable* normaliser *la propriété abstraite afin qu'elle ne contienne qu'un intervalle par variable. Ainsi, le plus petit intervalle sur* x *que l'on peut déduire à partir de* $[1, 101]$ *et* $[-\infty, 100]$, *est l'intersection* $[1, 101] \sqcap_{\mathcal{A}} [-\infty, 100] = [1, 100]$. *Cet intervalle est identique à* $I_{q_2}^{n+1}$, *calculé par la fonction* $\mathsf{sem}_g^\sharp$ *lors de l'analyse.*

Ceci n'est pas étonnant, $\mathsf{sem}_g^\sharp$ *est simplement défini en calculant l'intersection entre le garde et la propriété abstraite courante. Dans ce cas, l'opérateur* $\sqsubseteq_{\mathcal{I}}^\pi$ *ne sert à rien et la preuve vient directement de la version instrumentée de* $\sqcap_{\mathcal{A}}$.

3.3 Conclusion

Nous avons présenté dans ce chapitre une technique d'instrumentation s'appuyant sur des patrons de preuve. Ces patrons de preuve permettent à un analyseur statique de programmes de produire automatiquement des justifications de ses résultats, sous formes de preuves formelles vérifiables par le système Coq. Produire automatiquement des preuves de corrections de programme dans un système formel basé sur la théorie des types, tel que Coq, est l'un des défis que nous nous étions fixés. De plus l'utilisation de patrons de preuve garantit la complétude, dès lors que chaque fonction de transfert d'un analyseur donné comprend une stratégie de justification.

Nous avons montré que contrairement à ce que proposent [SYY03, Cha06] il n'est pas toujours utile d'instrumenter les fonctions de transfert des analyseurs statiques, ou

d'utiliser des prouveurs automatiques pour prouver les obligations de preuve retournées par le calcul de plus faible précondition. Par l'utilisation de patrons de preuve, l'instrumentation est au pire aussi coûteuse que l'implantation des fonctions de transfert de l'analyse dans un langage haut niveau. Mais dans de nombreux cas, elle est même plus facile. Si [SYY03, Cha06] ont donné les bases permettant d'arguer qu'il est possible de faire générer à un analyseur des certificats de leurs résultats, nous avons fourni les éléments pour rendre cette approche applicable à un coût raisonnable. En effet, nous avons mis en avant le fait que dès lors que le calcul de plus faible pré-condition préserve les post-conditions dans le domaine abstrait utilisé par l'analyse, l'instrumentation se limite aux opérateurs ensemblistes du domaine ambrait. Cela présente une contribution importante du point de vue de la mise en application de cette approche de certification par instrumentation.

Comme nous allons le voir dans le chapitre suivant, l'analyseur Enkidu [HP08] a été instrumenté en développant seulement des patrons de preuve pour une partie des opérateurs ensemblistes ($\sqsubseteq$ et $\sqcap$) d'un domaine abstrait du contenu des tableaux de programmes et la fonction d'abstraction (α) des expressions affectées et des gardes dans ce domaine. Cette étude de cas montrera de façon extrêmement limpide que le fait d'éviter d'instrumenter les fonctions de transfert présente un interêt majeur. Les fonctions de transfert contiennent les spécifités de chaque analyseur et s'appuient sur de nombreuses heuristiques qui sont implantées par de longs et complexes morceaux de code. Instrumenter seulement une partie des opérateurs des domaines abstraits rend l'instrumentation, en plus d'être réduite, portable d'un analyseur statique à un autre utilisant les mêmes domaines abstraits.

D'un point de vue plus fondamentale, on pourrait se demander : « *si la classe des propriétés sémantiques considérées appartient à une logique décidable et complète, pourquoi ne pas prouver à l'aide d'un prouveur automatique les invariants calculés par l'analyse ?* »

Cette solution est suggerée dans [Cha06] pour achever des obligations de preuves exprimées en arithmétique notamment. Cela est possible, mais dans le cas de domaines abstraits plus complexes, l'utilisation de tels prouveurs automatiques n'est plus efficace car un prouveur aura en général une expressivité plus importante que les invariants calculés par l'analyse et donc des performances souvent médiocres. Il faudrait en conséquence ré-implanter une partie de l'analyse dans un prouveur, ce qui force un travail de programmation complétement redondant. L'instrumentation par utilisation de patrons de preuve transforme chaque analyseur en un prouveur hyper spécialisé (pour prouver les propriétés calculables par l'analyseur) avec des performances optimales. Et c'est de cela qu'il s'agit ici : permettre aux analyseurs statiques de générer des certificats de leurs résultats avec la meilleure efficacité possible, pour rendre l'instrumentation applicable et utilisable en pratique.

Chapitre 4

Étude de cas : Instrumentation de l'analyseur statique Enkidu

> « *Si je suggérais qu'entre la Terre et Mars se trouve une théière de porcelaine en orbite elliptique autour du Soleil, personne ne serait capable de prouver le contraire pour peu que j'aie pris la précaution de préciser que la théière est trop petite pour être détectée par nos plus puissants télescopes. Mais si j'affirmais que, comme ma proposition ne peut être réfutée, il n'est pas tolérable pour la raison humaine d'en douter, on me considérerait aussitôt comme un illuminé. Cependant, si l'existence de cette théière était décrite dans d'anciens livres, enseignée comme une vérité sacrée tous les dimanches et inculquée aux enfants à l'école, alors toute hésitation à croire en son existence deviendrait un signe d'excentricité et vaudrait au sceptique les soins d'un psychiatre à une époque éclairée ou de l'Inquisition en des temps plus anciens.* »
>
> Bertrand Russell

L'objectif de ce chapitre est de développer l'instrumentation d'une analyse statique de programmes basée sur un domaine abstrait non trivial. Derrière ce travail technique, notre volonté est d'évaluer dans quelle mesure l'approche que nous avons proposée au chapitre précédent, s'appuyant sur des patrons de preuve, est applicable en pratique pour des analyseurs non triviaux. Nous nous sommes ainsi intéressés à Enkidu qui implante une analyse statique de programmes manipulant des tableaux, proposée par Mathias Péron et Nicolas Halbwachs [HP08]. L'analyse des accès aux tableaux par les programmes, qui permet de garantir qu'ils en respectent les bornes a été l'une des premières motivations de l'interprétation abstraite [CC76]. Il ne s'agit pas de cela ici mais d'une analyse du contenu des tableaux. Les tableaux introduisent des phénomènes d'alias et peuvent représenter un grand nombre de variables, ce qui rend l'analyse de leur contenu très difficile.

La méthode suivie par Enkidu se décompose en deux analyses consécutives comme proposé dans [GRS05] : la première analyse permet de partitionner les tableaux en un ensemble de tranches ; la seconde (qui est celle qui nous intéresse) associe à chaque tranche une propriété vérifiée par le contenu des cases de tableaux appartenant à cette tranche. Cette analyse se base sur une sémantique abstraite collectrice et d'atteignabilité.

Les programmes traités par Enkidu sont représentés par des systèmes de transitions étiquetés par des affectations ou des gardes. L'analyse calcule et associe une propriété invariante du contenu des tableaux à chaque point de contrôle d'un programme. Les propriétés associées à chaque tranche du tableau sont structurées par quatre domaines abstraits (trois instances du domaine des zones [Min04] et le domaine des propriétés de tableaux [HP08, GMT08]) que nous présenterons en section 4.1. L'analyse s'effectue par un calcul de post-point fixe, en utilisant un opérateur d'élargissement, suivi d'une séquence descendante permettant d'améliorer la précision de ce post-point fixe. Comme nous l'avons vu au chapitre précédent, ces étapes de l'analyse ne sont pas pertinentes pour apporter des certificats aux invariants calculés. Seule la dernière itération nous intéresse, lorsque Enkidu a atteint stabilité. Nous donnons une instrumentation permettant de certifier, point de contrôle par point de contrôle, que les invariants calculés sont corrects. Pour cela, il nous faudra spécifier chaque transition du système $q \xrightarrow{\tau} q'$ par un triplet de Hoare, où la pré-condition sera l'invariant calculé au point de contrôle q et la post-condition l'invariant calculé au point de contrôle q'. L'instrumentation s'appuiera sur des patrons de preuve, comme proposé au chapitre précédent.

Enkidu distingue trois types de transitions : les affectations d'indices, les affectations de tableaux ou de variables scalaires et les transitions gardées par des expressions booléenes. Les affectations d'indices de tableaux sont traitées séparément car si elles ne modifient pas le contenu des tableaux, elles en modifient la représentation que l'analyse en fait (section 4.2.1). Un défi pour achever ce travail est de parvenir à rendre Enkidu capable de certifier chacun des ses calculs en instrumentant un minimum de code : se limiter aux opérateurs ensemblistes du domaine abstrait et éviter les fonctions de transfert.

Contributions. Nos contributions présentées dans ce chapitre sont l'instrumentation des opérateurs ensemblistes des domaines abstraits des zones et des propriétés de tableaux, et l'exploitation de ces opérateurs instrumentés pour permettre à Enkidu de générer des certificats de ses résultats. Une difficulté rencontrée pour permettre à Enkidu de justifier ses calculs vient du fait que pour les instructions d'affectation le calcul de plus faible pré-condition ne préserve pas les invariants dans le domaine des propriétés de tableaux. Nous avons réussi cependant à éviter l'instrumentation des fonctions de transfert qui permettent d'analyser l'effet des affectations. Dans le cas de l'affectation des variables d'indice nous avons instrumenté un test d'inclusion plus générique que celui du domaine des propriétés de tableaux implanté dans Enkidu. Et dans le cas des affectations de tableaux nous avons développé des patrons de preuve permettant de justifier la traduction des formules retournées par le calcul de plus faible pré-condition dans le domaine des propriétés de tableaux, afin de pouvoir utiliser l'instrumentation de ce même test d'inclusion. L'atout notable de limiter l'instrumentation aux opérateurs ensemblistes, est que si les développeurs d'Enkidu modifient les fonctions de transfert de l'analyse (pour en améliorer la précison par exemple) l'instrumentation sera préservée. De plus, toute autre analyse utilisant le domaine abstrait des propriétés de tableaux (ou celui des zones) peut bénéficier de notre travail. L'instrumentation de l'analyseur Enkidu a été implantée en Ocaml, et nous avons grâce à cela certifié différents programmes manipulant des tableaux (recherche du maximum d'un tableau, copie des éléments d'un tableau dans un autre, tri par insertion, *etc.*).

La suite de ce chapitre donne l'ensemble de patrons de preuve permettant d'obtenir

des preuves formelles de correction des programmes analysés, par une instrumentation des opérateurs ensemblistes des domaines abstraits utilisés par Enkidu (section 4.1). Nous présentons en section 4.2 deux approches pour justifier l'analyse de l'affectation des indices de tableaux. L'une instrumente la fonction de transfert, l'autre un test d'inclusion plus générique. Cela nous a permis de mesurer le coût de l'instrumentation des fonctions de transfert et nous a renforcés dans l'idée qu'il est important de limiter l'instrumentation aux opérateurs ensemblistes. Les sections 4.3 et 4.4 présentent la justification de l'affectation des tableaux et des transitions gardées à l'aide de l'instrumentation des seuls opérateurs ensemblistes et de la fonction permettant d'exprimer les parties droites des affectations ainsi que les gardes dans le domaine abstrait des propriétés de tableaux. Puis, nous présentons quelques points de notre implantation et les expérimentations effectuées en section 4.5

4.1 Domaines abstraits de l'analyseur Enkidu

Enkidu effectue une abstraction permettant de se focaliser sur le contenu des tableaux et représente ceux-ci en utilisant un ensemble de formules $\varphi_1, \varphi_2, \ldots$ pour spécifier les cases des tableaux, associées à un ensemble de formules $\psi_1, \psi_2, \ldots$, qui spécifient leur contenu. À cela s'ajoute une formule ϕ fournissant des informations de contexte sur les variables d'indices du programme. Ces formules sont structurées en trois domaines abstraits. Un quatrième domaine est défini sur les trois autres et permet de spécifier le contenu d'un tableau à un point de contrôle du programme. Ce domaine est également structuré par une relation d'ordre partiel et ses élements constituent les *valeurs abstraites* de la sémantique approximée d'un programme traité. Ces variables abstraites constituent des *propriétés de tableaux* du programme en ses différents points de contrôle et ont la forme [BMS06] suivante :

$$\Psi \stackrel{def}{=} \phi \wedge \forall \ell, \bigwedge_{k=1}^{n} \varphi_k(\ell) \Rightarrow \psi_k(\ell)$$

où $\phi \in \mathscr{I}$ (*le contexte*), $\varphi_k(\ell) \in \mathscr{T}$ (*une tranche*) $\psi_k(\ell) \in \mathscr{P}$ (*une propriété de tranche*), et $\Psi \in \mathscr{A}$ (*la propriété de tableaux*). Les éléments des domaines $\mathscr{I}$, $\mathscr{T}$ et $\mathscr{P}$ expriment des différences de potentiel sous forme de systèmes d'égalités et d'inégalité et sont définis tous les trois (mais sur des ensembles de variables différents, voir section 4.1.2.1), dans l'implantation actuelle d'Enkidu, dans le *domaine abstrait des zones* [Min04], que nous présentons immédiatement.

4.1.1 Le domaine abstrait des zones

Les zones ont été introduites pour la vérification des automates temporisés. Le contrôle de ces automates est contraint, au niveau des états, par des invariants sur des horloges de la forme $x \bowtie k$, où $\bowtie \in \{<, \leq, =, \geq, >\}$ et k est une constante. Lorsque l'on souhaite effectuer une analyse d'accessibilité sur ces automates [Dil89] on est amené à considérer dans quel intervalle se trouve une horloge et la différence de deux horloges, *i.e.* des contraintes de la forme $(x_1 - x_2 \bowtie k)$. On appelle les contraintes de cette forme des contraintes de potentiel.

Les contraintes de potentiel ont trouvé une grande utilité pour l'analyse de programmes impératifs pour exprimer des invariants de boucle notamment. Le code suivant : `x := 1; while (x ≤ y) do x := x + 1`, a pour invariant en tête de boucle la propriété $1 \leq x \leq y + 1$. Nous pouvons représenter cet invariant sous forme d'une conjonction de contraintes de potentiel : $0 - x \leq -1 \ \wedge \ x - y \leq 1$.

Ces contraintes de potentiel peuvent également exprimer des propriétés de la forme $A[i] \leq A[i+1]$ signifiant qu'un tableau A est ordonné dans l'ordre croissant sur les cases indicées par i et $i+1$. Si nous souhaitons étendre cette propriété à une partie plus grande du tableau A, les contraintes de potentiel peuvent encore être employées : combinées avec une quantification universelle et une implication elles permettent d'exprimer qu'un tableau est trié : $\forall \ell, 1 \leq \ell \leq n \Rightarrow A[\ell] \leq A[\ell + 1]$. Nous reviendrons sur l'utilité des contraintes de potentiel dans la section 4.1.2.

4.1.1.1 Système de contraintes de zone

Un système de contraintes de zone est une conjonction d'inégalités sur un ensemble de variables, exprimant des différences de potentiel entre chaque couple de variables de cet ensemble.

Définition 4.1.1 (Contrainte de zone)**.** *Une contrainte de zone est soit une contrainte de potentiel $x_i - x_j \leq c$ où $x_i, x_j \in V$ et $c \in \mathbb{Z}$, soit une contrainte d'intervalle $x \in [c_1, c_2]$, où $c_1, c_2 \in \mathbb{Z}$, que l'on exprimera comme une constrainte de potentiel à l'aide de la constante 0 : $x - 0 \leq c_2$ et $0 - x \leq -c_1$.*

Exemple 4.1.1.

Nous donnons ci-contre un exemple de système de contraintes de zones C, sur l'ensemble de trois variables $V = \{x_1, x_2, x_3\}$.

$$\begin{cases} 3 \leq x_1 \leq 6 \\ -1 \leq x_2 \leq 1 \\ x_3 \leq x_1 \\ x_2 + 1 \leq x_3 \end{cases}$$

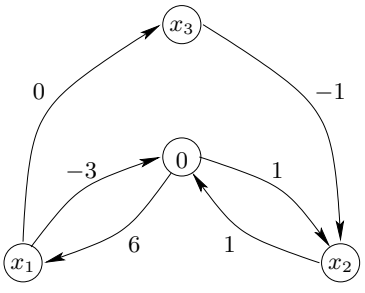
Un ensemble des points de $\mathbb{Z}^n$ satisfaisant un système de contraintes de zone est appellé une zone. On représente une zone sous forme d'un graphe de potentiels.

Exemple 4.1.2.

L'ensemble C de contraintes de zone définies sur $V = \{x_1, x_2, x_3\}$ de l'exemple 4.1.1 :

$$\begin{cases} 3 \leq x_1 \leq 6 \\ -1 \leq x_2 \leq 1 \\ x_3 \leq x_1 \\ x_2 + 1 \leq x_3 \end{cases}$$

Le graphe de potentiels ci-dessous représente le système de contraintes C :



Dans cet exemple, on observe un arc $(x_2, x_3, -1)$ *signifiant que la distance de* x_3 *à* x_2 *est inférieure ou égale à* -1.

Définition 4.1.2 (Graphe de potentiels)**.** *Un graphe de potentiels est un graphe orienté étiqueté, dont l'ensemble de sommets est* $S \stackrel{def}{=} V \cup \{0\} = \{x_1, \ldots, x_n, 0\}$. *Soit* C *un système de contraintes de zone, nous notons* $\mathcal{G}(C) = (S, E)$ *le graphe de potentiels représentant* C, *avec :*

$$E \subseteq S \times S \times \mathbb{Z} \text{ tel que } E = \{(x_i, x_j, k) \mid (x_i - x_j \leq k) \in C\}$$

Un arc $e \stackrel{def}{=} (x_i, x_j, k) \in E$ *se note également* $x_j \xrightarrow{k} x_i$ *et signifie intuitivement que* « la distance potentielle de x_j à x_i est inférieure ou égale à k » .

Théorème 4.1 (Inconsistance d'un ensemble de contraintes de zone)**.** *Soit* C *un ensemble de contraintes de zone,* C *est inconsistant* i.e. *le système d'inégalités linéaires que définit* C *n'est pas satisfaisable, si le graphe de potentiels* $\mathcal{G}(C)$ *contient un cycle de poids négatif :*

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

La preuve de ce théorème peut se trouver dans [Pra77a].

4.1.1.2 Implantation des contraintes de zone par des matrices

Cette présentation des systèmes de contraintes de zone étant faite, nous présentons maintenant le domaine abstrait des zones, dont les élements, les conjonctions de contraintes, sont en pratique représentés par des *matrices de différences bornées* (*difference bound matrices* en anglais, abrégées DBMs) [Dil89].

Définition 4.1.3 (Matrice de différences bornées)**.** *Soit* C *un système de contraintes de zone sur* V. *On définit la matrice de différences bornées représentant* C *comme la matrice d'adjacence de* $\mathcal{G}(C)$. *Cette matrice est donc carrée, de taille* $((|V|+1)\times(|V|+1))$, *et ses élements prennent leurs valeurs dans* $\mathbb{Z}$.

Exemple 4.1.3. *Nous donnons ci-dessous en exemple un ensemble de contraintes et sa matrice de bornes correspondante.*

L'ensemble C *de contraintes de zone définies sur* $V = \{x_1, x_2, x_3\}$ *de l'exemple 4.1.1 :*

$$\begin{cases} 3 \leq x_1 \leq 6 \\ -1 \leq x_2 \leq 1 \\ x_3 \leq x_1 \\ x_2 + 1 \leq x_3 \end{cases}$$

La DBM m, *représentant l'ensemble de contraintes* C *:*

	0	x_1	x_2	x_3
0	$+\infty$	-3	1	$+\infty$
x_1	6	$+\infty$	$+\infty$	$+\infty$
x_2	1	$+\infty$	$+\infty$	-1
x_3	$+\infty$	0	$+\infty$	$+\infty$

On remarque sur la matrice de différences bornées de cet exemple, que toutes les contraintes de zone non spécifiées dans l'ensemble C, *ont une distance potentielle inférieure*

ou égale à plus l'infini. Cette distance potentiellement infinie exprime la non-information dont on dispose entre les variables concernées.

4.1.1.3 Normalisation des dbms

Remarquez qu'une même zone, vide ou non, peut être représentée par différentes matrices de différences bornées : sur l'exemple 4.1.3, si la matrice de différences bornées exprimait en plus la contrainte $x_3 \leq 6$, celle-ci représenterait la même zone, du fait que cette matrice exprime déjà que $x_3 \leq x_1$ et $x_1 \leq 6$ et par transitivité de l'inégalité, on peut déduire que $x_3 \leq 6$. Pour faciliter la comparaison d'ensembles de contraintes, on utilise représentation normalisée (ou forme canonique) d'une zone par une matrice de différences bornées. Outre le fait que l'analyse statique que nous présentons dans les sections suivantes effectue cette normalisation, elle permet de décider de l'inconsistance et de l'inclusion des ensembles de contraintes. Prenons pour exemple la zone représentée par la DBM m de l'exemple précédent et celle représentée par la DBM m' exprimant seulement que $x_3 \leq 6$ (*i.e.* $m'_{30} = 6$ et $m'_{ij} = +\infty$ pour tout couple $(i,j) \neq (3,0)$). Pour trouver que l'ensemble de contraintes représenté par m' est inclus dans celui représenté m, il faut se rendre compte que m représente une zone où $x_3 \leq 6$ également. Ceci sera explicité par la normalisation de m : à partir d'une DBM m, trouver la DBM $\tilde{m}$ qui explicite toute contrainte de zone implicite dans m. Une telle DBM normalisée, qui représente la même zone que m, est unique. Chacune de ses bornes sur $x_i - x_j$ se définit comme le plus court chemin de x_j à x_i dans le graphe de potentiels que représente m. Les plus courts chemin d'un graphe sont correctement définis s'il n'existe pas de cycle de poids strictement négatif dans ce graphe. Dans le cas où il existe un cycle de poids négatif la forme normale de m est $\bot_{zone}$.

Définition 4.1.4 (Forme normale d'une DBM)**.** *Soit m une DBM et $\mathcal{G}$ son graphe de potentiels. Un chemin du sommet x au sommet y dans $\mathcal{G}$ est une séquence $\langle x, \ldots, y \rangle$. La forme normale de m, notée $\tilde{m}$, est :*

- $\tilde{m} = \bot_{zone}$ *si $\mathcal{G}$ contient un cycle de poids négatif,*
- $\begin{cases} \tilde{m}_{ii} = 0 \\ \tilde{m}_{ij} = \min_{\langle i=\ell_1,\ell_2,\ldots,\ell_m=j\rangle} \sum_{k=1}^{m-1} m_{\ell_k \ell_{k+1}} \text{ si } i \neq j \end{cases}$

En pratique, on utilise l'algorithme de Floyd-Warshall pour calculer la fermeture des plus courts chemins (figure 4.1). La complexité de cet algorithme est cubique dans le nombre de variables.

Cet algorithme a l'avantage de résoudre aussi le cas où m représente un ensemble de contraintes inconsistant. En effet, après son appel, tester la présence de cycles strictement négatifs revient à vérifier si sur la diagonale de la matrice il existe une valeur strictement négative. Ainsi, l'algorithme de normalisation consiste à appeler l'algorithme de Floyd-Warshall qui retourne une DBM m' où les plus courts chemins entre chaque variable ont été calculés. Si dans cette nouvelle DBM il existe $i \in [0, n]$ tel que $m'_{ii} < 0$ il faut retourner $\bot$ sinon on peut retourner m'.

Exemple 4.1.4. *Nous donnons ci-dessous la version normalisée de l'ensemble de*

```
for  k ← 0  to  n
  { for  i ← 0  to  n
    { for  j ← 0  to  n
      { if  (m_ij > m_ik + m_kj)
        then  m_ij ← m_ik + m_kj
      }
    }
  }
```

Fig. 4.1 – L'algorithme de Floyd Warshall calculant la fermeture des plus courts chemins d'un graphe orienté représenté par la matrice m

contraintes de zone de l'exemple 4.1.3 et sa matrice de différences bornées correspondante :

Une partie de l'ensemble C normalisé de contrainte de zone définies sur $V = \{x_1, x_2, x_3\}$:

$$\begin{cases} 3 \leq x_1 \leq 6 \\ -1 \leq x_2 \leq 1 \\ x_3 \leq x_1 \\ x_2 + 1 \leq x_3 \\ 0 \leq x3 \leq 6 \\ x_1 - x_3 \leq 6 \\ x_1 - x_2 \leq 7 \\ \dots \end{cases}$$

La DBM *m, normalisée représentant l'ensemble de contraintes C normalisé :*

	0	x_1	x_2	x_3
0	2	-3	1	0
x_1	6	6	7	6
x_2	1	-2	5	-1
x_3	6	0	7	6

4.1.1.4 Instrumentation de la normalisation des dbms

Nous donnons ci-dessous une partie de l'instrumentation de l'algorithme de Floyd-Warshall permettant d'obtenir une justification formelle de la normalisation d'une DBM. Cette justification consiste à prouver que l'ensemble de contraintes initial implique l'ensemble de contraintes obtenu par normalisation. Nous donnons le patron de preuve associé au test de l'algorithme de Floyd-Warshall qui minimise la valeur du coefficient m_{ij} qui code la borne supérieure (le plus petit des majorants) de la distance de x_i à x_j pour un chemin passant par une variable x_k :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{x_i - x_k \leq m_{ik} \wedge x_k - x_j \leq m_{kj}}^{H}}{\vdash_{\text{NJ}} x_i - x_k \leq m_{ik}} \wedge_e}{\vdash_{\text{NJ}} x_i - x_j \leq m_{ik} + (x_k - x_j)} +_c(x_k - x_j)}{\vdash_{\text{NJ}} x_i - x_j - m_{ik} \leq x_k - x_j} +_c(-m_{ik}) \quad \dfrac{\vdash_{\text{NJ}} \overbrace{x_i - x_k \leq m_{ik} \wedge x_k - x_j \leq m_{kj}}^{H}}{\vdash_{\text{NJ}} x_k - x_j \leq m_{kj}} \wedge_e}{\vdash_{\text{NJ}} x_i - x_j - m_{ik} \leq m_{kj}} \leq_{trans}}{\vdash_{\text{NJ}} x_i - x_j \leq m_{ik} + m_{kj}} +_c(m_{ik})}{\vdash_{\text{NJ}} (x_i - x_k \leq m_{ik} \wedge x_k - x_j \leq m_{kj}) \Rightarrow x_i - x_j \leq m_{ik} + m_{kj}} \Rightarrow_i [H]$$

Pour une exécution particulière, la preuve se construit selon les itérations de l'algorithme par assemblage d'instances du patron de preuve ci-dessus. La taille de la preuve dépend donc du nombre de fois où le test, `if` $(m_{ij} > m_{ik}+m_{kj})$, est évalué positivement.

Remarque. *En pratique, pour réduire la taille des preuves, on prouve un lemme* DBM$_{trans}$ *qui agit ensuite comme une règle de déduction :*

$$\frac{\vdash_{\text{NJ}} x_i - x_k \leq m_{ik} \qquad \vdash_{\text{NJ}} x_k - x_j \leq m_{kj}}{\vdash_{\text{NJ}} x_i - x_j \leq m_{ik} + m_{kj}}\ \text{DBM}_{trans}$$

Le patron de preuve donné précédemment pour justifier la normalisation d'une DBM*, peut donc avoir une structure de taille réduite en utilisant la règle* DBM$_{trans}$ *dont la validité a été vérifiée en amont :*

$$\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{x_i - x_k \leq m_{ik} \wedge x_k - x_j \leq m_{kj}}^{H}}{\vdash_{\text{NJ}} x_i - x_k \leq m_{ik}}\wedge_e \quad \dfrac{\vdash_{\text{NJ}} \overbrace{x_i - x_k \leq m_{ik} \wedge x_k - x_j \leq m_{kj}}^{H}}{\vdash_{\text{NJ}} x_k - x_j \leq m_{kj}}\wedge_e}{\vdash_{\text{NJ}} x_i - x_j \leq m_{ik} + m_{kj}}\ \text{DBM}_{trans}}{\vdash_{\text{NJ}} (x_i - x_k \leq m_{ik} \wedge x_k - x_j \leq m_{kj}) \Rightarrow x_i - x_j \leq m_{ik} + m_{kj}}\Rightarrow_i [H]$$

Ceci étant vu, il ne nous reste plus qu'à définir et instrumenter les opérateurs du domaine abstrait des zones. Pour plus de détails concernant le domaine des zones, le lecteur pourra se référer à la présentation exhaustive de ce domaine dans [Min04].

4.1.1.5 Opérateurs du domaine des zones

Si on veut savoir précisemment si une zone représentée par une matrice m est incluse dans celle représentée par une matrice m', il suffit de comparer les bornes de la forme normale de m et les bornes de m' : si ces dernières sont toutes plus grandes que les premières alors l'inclusion est vérifiée.

Pour calculer la borne supérieure de deux zones représentées par m et m', l'opération consiste simplement à prendre point par point la borne la plus faible, non pas de m et m', mais de leurs formes normales afin de ne perdre aucune contrainte implicite. L'opérateur de borne inférieure, quant à lui, prend point par point le coefficient le plus petit et n'a pas besoin de faire appel à la normalisation de ses opérandes pour être précis :

- $m \sqsubseteq_{zone} m' \iff (\tilde{m} = \bot_{zone}) \vee (\forall i, j \in [1, n],\ \tilde{m}_{ij} \leq m'_{ij})$
- $m \sqcap_{zone} m' = \forall i, j \in [1, n], \min(m_{ij}, m'_{ij})$
- $m \sqcup_{zone} m' = \forall i, j \in [1, n], \max(\tilde{m}_{ij}, \tilde{m}'_{ij})$

4.1.1.6 Instrumentation des opérateurs du domaine des zones

Souhaitant obtenir une justification déductive des résultats de ces opérateurs, nous instrumentons les opérateurs définis précédement. Pour les patrons de preuve que nous allons définir, nous considérons dans chaque cas deux systèmes de contraintes C et C' définis par :

$$C \stackrel{def}{=} \bigwedge_{\forall i,j \in |V|} x_i - x_j \leq m_{ij} \quad \text{et} \quad C' \stackrel{def}{=} \bigwedge_{\forall i,j \in |V|} x_i - x_j \leq m'_{ij}.$$

Pour l'inclusion de C dans C', il s'agit de prouver que $C \Rightarrow C'$. Pour justifier les calculs des opérateurs de bornes inférieure et supérieure de C et C', il faut prouver que pour chaque coefficient m''_{ij} retourné par $\sqcap_{zone}$, $C \wedge C' \Rightarrow x_i - x_j \leq m''_{ij}$ et pour chaque coefficient m''_{ij} retourné par $\sqcup_{zone}$, $C \vee C' \Rightarrow x_i - x_j \leq m''_{ij}$.

Instrumentation du test d'inclusion. La justification de l'opérateur d'inclusion consiste simplement à tester l'inégalité entre chaque coefficient de la DBM m correspondante à C, avec celle de la DBM m' correspondante à C'. C et C' contiennent chacune $n+1 \times n+1$ contraintes si $|V| = n$, dont nous souhaitons prouver l'implication. Le patron de preuve global pour justifier l'inclusion de C et C' est le suivant :

$$\dfrac{\dfrac{\begin{array}{ccc}\vdots & \cdots & \vdots \\ \vdash_{\text{NJ}} c'_1 & & \vdash_{\text{NJ}} c'_{n+1\times n+1}\end{array}}{\vdash_{\text{NJ}} c'_1 \wedge c'_2 \wedge \ldots \wedge c'_{n+1\times n+1}}\wedge_i}{\vdash_{\text{NJ}} \underbrace{(c_1 \wedge c_2 \wedge \ldots \wedge c_{n+1\times n+1})}_{C} \Rightarrow \underbrace{(c'_1 \wedge c'_2 \wedge \ldots \wedge c'_{n+1\times n+1})}_{C'}}\Rightarrow_i [H]$$

Pour une certaine contrainte $c_{i\times j}$, le patron de preuve est le suivant :

$$\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{c_1 \wedge \ldots \wedge x_i - x_j \leq m_{ij} \wedge \ldots \wedge c_{n+1\times n+1}}^{H}}{\vdash_{\text{NJ}} x_i - x_j \leq m_{ij}}\wedge_e \quad \dfrac{\textit{vérifiée par le système Coq}}{\vdash_{\text{NJ}} m_{ij} \leq m'_{ij}}}{\vdash_{\text{NJ}} x_i - x_j \leq m'_{ij}}\leq_{trans}}{\vdash_{\text{NJ}} (x_i - x_j \leq m_{ij}) \Rightarrow (x_i - x_j \leq m'_{ij})}\Rightarrow_i [H]$$

Les inégalités entre deux constantes de la forme $m_{ij} \leq m'_{ij}$ sont vérifiées automatiquement par le système Coq.

Instrumentation de l'opérateur de borne inférieure. L'opérateur de borne inférieure dans le domaine des zones s'applique sur deux systèmes de contraintes de zone, C et C', et consiste à conserver le minimum des deux coefficients pour chaque couple de variables (*cf.* section 4.1.1.5). L'instrumentation de cet opérateur consiste alors à intégrer le test suivant accompagné de patrons de preuve différents selon le résultat de ce test à l'exécution, qui permet de justifier cette conservation du minimum à partir de la conjonction $C \wedge C'$, pour chaque couple de variables x_i et x_j :

`si` $\min(m_{ij}, m'_{ij}) = m_{ij}$
`alors` $\pi =$

$$\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{C \wedge C'}^{H}}{\vdash_{\text{NJ}} C}\wedge_e}{\vdash_{\text{NJ}} x_i - x_j \leq m_{ij}}\wedge_e}{\vdash_{\text{NJ}} (C \wedge C') \Rightarrow x_i - x_j \leq m_{ij}} \Rightarrow_i [H]$$

`si` $\min(m_{ij}, m'_{ij}) = m'_{ij}$
`alors` $\pi =$

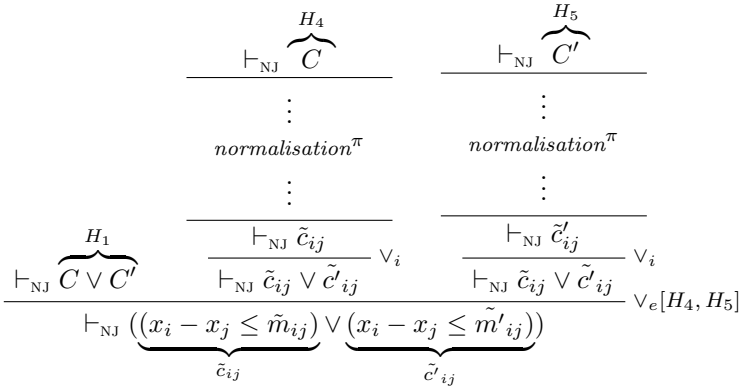
$$\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{C \wedge C'}^{H}}{\vdash_{\text{NJ}} C'}\wedge_e}{\vdash_{\text{NJ}} x_i - x_j \leq m'_{ij}}\wedge_e}{\vdash_{\text{NJ}} (C \wedge C') \Rightarrow x_i - x_j \leq m'_{ij}} \Rightarrow_i [H]$$

Instrumentation de l'opérateur de borne supérieure. L'application de l'opérateur de borne supérieure, sur C et C', signifie qu'il faut tenir compte de *l'une ou de l'autre* et s'interprète logiquement par $C \vee C'$. L'opérateur de borne supérieure se distingue des deux autres opérateurs du fait que le connecteur logique disjonctif ($\vee$) n'appartient pas à la syntaxe du domaine des zones (contrairement à l'implication et la conjonction). Par conséquent, le résultat est une sur-approximation de la traduction logique $C \vee C'$. De plus, l'opérateur de borne supérieure normalise les matrices de bornes m et m', afin d'obtenir la sur-approximation la plus précise possible. Cette phase de normalisation des contraintes de zone est prise en compte dans la justification de l'opérateur.

Nous donnons ci-dessus le patron de preuve qui permet d'instrumenter l'opérateur de borne supérieure pour deux variables x_i et x_j, avec $\tilde{c}_{ij} \stackrel{def}{=} (x_i - x_j \leq \tilde{m}_{ij})$, $\tilde{c'}_{ij} \stackrel{def}{=} (x_i - x_j \leq \tilde{m'}_{ij})$. Soit $max := \max(\tilde{m}_{ij}, \tilde{m'}_{ij})$ le coefficient calculé par $\sqcup_{zone}$, nous donnons la première étape du patron de preuve :

$$\dfrac{\dfrac{\vdash_{\text{NJ}} \tilde{c}_{ij} \vee \tilde{c}'_{ij} \qquad \dfrac{\vdash_{\text{NJ}} \overbrace{x_i - x_j \leq \tilde{m}_{ij}}^{H_2 \stackrel{def}{=} \tilde{c}_{ij}} \qquad \vdash_{\text{NJ}} \tilde{m}_{ij} \leq max}{\vdash_{\text{NJ}} x_i - x_j \leq max}\leq_{trans} \qquad \dfrac{\textit{idem avec } \tilde{m'}_{ij} \text{ et } H_3}{\vdash_{\text{NJ}} x_i - x_j \leq max}\leq_{trans}}{\vdash_{\text{NJ}} x_i - x_j \leq max}\vee_e[H_2, H_3]}{\vdash_{\text{NJ}} (C \vee C') \Rightarrow x_i - x_j \leq \underbrace{\max(\tilde{m}_{ij}, \tilde{m'}_{ij})}_{max}} \Rightarrow_i [H_1]$$

Il reste donc à prouver la disjonction $\tilde{c}_{ij} \vee \tilde{c'}_{ij}$. Pour cela, nous utilisons de nouveau la règle d'élimination de la disjonction à partir de l'hypothèse H_1. Les deux sous-buts générés par l'application de $\vee_e$ se prouvent grâce à la justification de la fonction de normalisation (*cf* section 4.1.1.4) :



Dans cette partie du patron de preuve, la version instrumentée de l'opérateur de normalisation est employée pour déduire $\tilde{c}_{ij}$ de C et $\tilde{c}'_{ij}$ de C'. En pratique la justification de la normalisation d'un système de contraintes C (via sa DBM m correspondante) retourne une preuve du système de contraintes $\tilde{C}$ à partir de l'hypothèse C. Il faut donc ensuite sélectionner dans $\tilde{C}$ la contrainte $\tilde{c}_{ij}$. Ceci s'effectue au moyen de la règle de déduction $\wedge_e$ et n'apparaît pas dans le patron de preuve ci-dessus (pour des raisons de lisibilité).

4.1.1.7 Conclusion sur le domaine abstrait des zones

Nous avons présenté le domaine abstraits des zones [Min04], puis les patrons de preuve permettant d'instrumenter les opérateurs du domaine dans le but d'apporter des justifications à chacun de leurs résultats. L'analyseur Enkidu utilise le domaine des zones pour calculer des propriétés sur le contenu des tableaux. L'instrumentation de ces opérateurs nous sera utile pour instrumenter l'analyseur Enkidu et pourra être réutilisé pour toute autre analyse s'appuyant sur le domaine abstrait des zones.

4.1.2 Domaine abstrait du contenu des tableaux $\mathscr{A}$

Dans ce qui suit, nous décrivons le domaine abstrait permettant à Enkidu d'exprimer des propriétés relationnelles sur le contenu des tableaux manipulés par un programme impératif. Ces propriétés de tableaux sont des implications logiques [BMS06] de la forme :

$$\forall \ell, \varphi(\ell, i_1, \ldots, i_m) \Rightarrow \psi(A_1[\ell + k_1], \ldots, A_p[\ell + k_p], x_1, \ldots, x_q)$$

où $i_1, \ldots, i_m$ sont des variables d'indice de tableaux, $A_1, \ldots, A_p$ des tableaux, $k_1, \ldots, k_p$ des constantes dans $\mathbb{Z}$, $x_1, \ldots, x_q$ des variables scalaires du programme analysé, et les formules φ et ψ définissent des systèmes de contraintes de zone.

Exemple 4.1.5. *Pour exprimer qu'un tableau est initialisé à une certaine constante k, l'analyse retournera la propriété de tableaux $\forall \ell, 1 \leq \ell \leq n \Rightarrow A[\ell] = k$. Dans cet exemple, $\varphi \stackrel{def}{=} 1 \leq \ell \leq n$ et $\psi \stackrel{def}{=} A[\ell] = k$, et 1 et n sont les bornes du tableau $A[\,]$.*

De manière à être précise, l'analyse s'autorise pour une propriété de tableaux calculée à définir plusieurs formules de ce type, selon différentes zones du tableau, jugées pertinentes.

Exemple 4.1.6. *Si l'on considère un programme qui range dans un tableau toutes les valeurs inférieures à une constante k à gauche d'un indice i, la valeur k en i, et toutes les valeurs supérieures à k à droite i. La propriété de tableaux retournée par l'analyse sera :*

$$\forall \ell, \bigwedge \begin{pmatrix} 1 \leq \ell \leq i-1 \Rightarrow A[\ell] \leq k \\ \ell = i \Rightarrow A[\ell] = k \\ i+1 \leq \ell \leq n \Rightarrow A[\ell] \geq k \end{pmatrix}$$

Cette formule contient trois contraintes de zone φ_1, φ_2 et φ_3 (représentant respectivement les cases du tableau qui précèdent $A[i]$, la case $A[i]$ et les cases qui suivent $A[i]$), qui spécifient l'ensemble des cases du tableau entre 1 et n (voir figure 4.2). Ces trois contraintes de zone forment une partition *du tableau $A[\,]$.*

Définition 4.1.5 (Partition d'un tableau)**.** *Une partition P d'un tableau, est un ensemble fini de formules $\{\varphi_p\}_{p \in P}$ portant sur les variables d'indice du programme et une variable logique quantifiée ℓ, telles que :*

$$\forall p, p' \in P, (p \neq p') \Rightarrow \neg(\varphi_p \wedge \varphi_{p'}) \ \wedge \ 1 \leq \ell \leq n \Rightarrow \bigvee_{p \in P} \varphi_p$$

Chaque φ_p de la partition P est appelée une « tranche » du tableau.

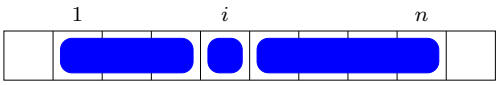


Fig. 4.2 – Représentation graphique de la partition $\{\varphi_1, \varphi_2, \varphi_3\}$

Domaine des partitions. Ce domaine est noté $\mathcal{D}_{part}$ et a pour éléments toutes les partitions définissables par des contraintes de zone sur $\mathscr{I}$. L'opérateur de borne supérieure de $\mathcal{D}_{part}$ est défini par l'union des intersections de chaque couple de tranches dont l'intersection n'est pas vide. L'opérateur de borne inférieure n'est pas donné ici car il ne sera pas utilisé dans la suite.

Définition 4.1.6 (Opérateurs ensembliste du domaine des partitions)**.**
$\forall P_1, P_2 \in \mathcal{D}_{part}$:
- $P_1 \sqsubseteq_{part} P_2 \iff \forall \varphi_q \in P_2, \ \exists \varphi_p \in P_1, \ \varphi_q \sqsubseteq_{zone} \varphi_p$
- $P_1 \sqcup_{part} P_2 \quad = \bigcup \{\varphi_p \sqcap_{zone} \varphi_q \mid \varphi_p \in P_1 \ \wedge \ \varphi_q \in P_2 \ \wedge \ \varphi_p \sqcap_{zone} \varphi_q \neq \bot_{zone}\}$

Lorsque Enkidu analyse un programme, la partition des tableaux de ce programme est définie automatiquement par une pré-analyse. Cette partition est identique à chaque itération de l'analyse et pour chaque point de contrôle. Le choix de la partition est inspiré de [GRS05] et consiste à intégrer toute case de tableau modifiée ou testée, afin de préserver un maximum d'informations sur ces cases. Les tranches obtenues selon ce principe sont ensuite raffinées selon les informations fournies par les contextes aux différents

points de contrôle. Les autres cases sont regroupées en tranches de part et d'autre des cases affectées ou testées. Les auteurs de [GRS05, HP08] ont l'idée que l'on peut déduire bien plus aisément du code source d'un programme comment le programmeur a structuré ses tableaux que ce qu'il met dedans[1].

Exemple 4.1.7. *Le programme impératif suivant initialise un tableau A de n cases à une constante k :*

```
void exemple(int A[n]){
  int i = 1;
  while (i <= n)
    {
     A[i] := k;
     i := i + 1;
    }
}
```

La partition déterminée par la pré-analyse est $\{(\ell < i), (\ell = i), (\ell > i)\}$. Elle prend soin de distinguer la case indicée par i, du fait de l'affectation $A[i] := k$.

Avec cette partition, Enkidu *calcule la propriété de tableaux* $i = n + 1 \wedge \forall \ell, \bigwedge \begin{pmatrix} \ell \leq i - 1 \Rightarrow A[\ell] = k \\ \ell = i \Rightarrow \top \\ i + 1 \leq \ell \Rightarrow \top \end{pmatrix}$ *au point de sortie du programme.*

Nous donnons maintenant un exemple de programme plus compliqué que Enkidu sait traiter. Pour ce genre de programmes la pré-analyse ne consiste pas seulement à découper le tableau selon les cases testées ou modifiées.

Exemple 4.1.8. *Le programme suivant défini un algorithme de tri par ordre croissant d'un tableau A. L'algorithme implanté par ce programme est nommé* le tri par insertion

[1] Je ne partage pas ce point de vue car il me semble que pour définir une analyse statique de programmes, il est aussi difficile de trouver le "bon" domaine abstrait que définir l'analyse en elle-même. D'autant plus que quand le domaine abstrait doit être automatiquement déduit du code source à analyser, définir un algorithme qui permet de trouver ce "bon" domaine me semble peut-être la chose la plus difficile. Cependant, ces deux problèmes sont orthogonaux et du point de vue de la certification, seule l'analyse du contenu des tableaux nous intéresse.

```
void InsertionSort(int A[n]){
  i : = 2;
  while(i <= n)
    {
     x := A[i];
     j := i-1;
     while (1 <= j & A[j] > x)
       {
        A[j+1] := A[j];
        j := j-1
       }
     A[j+1] := x;
     i := i + 1
    }
  }
```

Dans le cas du tri par insertion, seules deux cases du tableau A sont testées ou modifiées : $A[j]$ et $A[j+1]$. La partition est calculée selon ces deux expressions sur l'indice j et est dans sa version brute l'ensemble de tranches : $\{(\ell < j), (\ell = j), (\ell = j+1), (\ell > j+1)\}$. Cette partition n'évoque pas l'indice i car la case $A[i]$ ne satisfait pas les conditions requises par le principe que nous avons mentionné. Cependant, l'indice i est essentiel pour l'algorithme et l'analyse se doit d'intégrer cet indice à la partition du tableau A. La partition raffinée selon i et utilisée par la deuxième phase de l'analyse (du contenu de tableaux) comporte dix tranches et est donnée ci-dessous :

$$\left\{ \begin{array}{l} \varphi_1 \stackrel{def}{=} (1 = \ell < j < i), \varphi_2 \stackrel{def}{=} (2 \leq \ell < j < i), \\ \varphi_3 \stackrel{def}{=} (1 = \ell = j < i), \varphi_4 \stackrel{def}{=} (2 \leq \ell = j < i), \\ \varphi_5 \stackrel{def}{=} (1 = \ell = j+1 = i), \varphi_6 \stackrel{def}{=} (2 \leq \ell = j+1 < i), \varphi_7 \stackrel{def}{=} (2 \leq \ell = j+1 = i), \\ \varphi_8 \stackrel{def}{=} (1 \leq j+1 < \ell < i), \varphi_9 \stackrel{def}{=} (1 \leq j+1 < \ell = i), \varphi_{10} \stackrel{def}{=} (1 \leq j+1 < i < \ell) \end{array} \right\}$$

Le choix de la partition est crucial pour la pertinence des propriétés découvertes par l'analyse car il influe directement sur la précision de l'analyse. Cependant, la justification de l'analyse est indépendante du choix de la partition au sens où les propriétés calculées doivent être valides quelle que soit la partition choisie. Seule la correction des invariants calculés nous importe. Si la partition choisie n'est pas pertinente et conduit à une analyse trop imprécise, cela n'est pas de notre ressort. Ce que nous voulons, c'est simplement certifier que l'analyse est correcte, qu'elle soit précise ou non. Pour ces raisons, nous nous dispensons de détailler les algorithmes permettant de définir les partitions (car ceux-ci sont loin d'être triviaux et font l'objet des travaux actuels de Valentin Perrelle et Nicolas Halbwachs). Une fois que la partition est fixée, elle conditionne tout le reste de l'analyse, et par conséquent l'instrumentation que nous souhaitons définir en tient compte. À chaque point de contrôle du programme, l'analyse calcule une conjonction d'implications : une par tranche de la partition du tableau. Toutes les cases d'une même tranche partagent la même propriété de tranche, *i.e.* la même spécification de leur contenu. En outre, chaque propriété de tableaux contient une formule ϕ appelée le *contexte*, définissant les positions des variables d'indice entre les bornes des tableaux.

D'autres analyses du contenu des tableaux de programmes ont des partitions qui peuvent évoluer à chaque itération de l'analyse [GMT08], et par conséquent utilisent des domaines abstraits plus expressifs.

4.1.2.1 Instances du domaine des zones utilisées par $\mathscr{A}(P)$

Nous détaillons ci-dessous le rôle des trois instances du domaine abstrait des zones (*cf.* section 4.1), avant de définir le domaine abstrait des propriétés de tableaux, à partir

duquel sont calculés les invariants des programmes traités.

Contextes d'une propriété de tableaux. Le contexte d'une propriété de tableaux est un système de contraintes qui fournit des informations sur la position des indices des tableaux du programme analysé. Les contextes appartiennent au domaine $\mathscr{I}$ qui est un sous-ensemble du domaine des zones portant uniquement sur les variables d'indice du programme.

Exemple 4.1.9. *Soit i l'unique variable d'indice d'un programme manipulant des tableaux et n la borne supérieure du tableau :*

$$\phi \stackrel{def}{=} (1 \leq i) \wedge (i \leq n)$$

Le contexte $\phi \in \mathscr{I}$ permet de spécifier que la variable d'indice i est entre les bornes du tableau.

Nous donnons un second exemple, plus compliqué, impliquant deux variables d'indice :

Exemple 4.1.10. *Soit i et j deux variables d'indice de tableaux dans un programme p donné et n la borne supérieure du tableau :*

$$\text{Soit } \phi \stackrel{def}{=} (j \leq n) \wedge (j \leq i) \wedge (i \leq n) \wedge (2 \leq n) \wedge (2 \leq j) \wedge (2 \leq i)$$

Le contexte ϕ traduit les positions relatives de i et j comme illustré en figure 4.3.

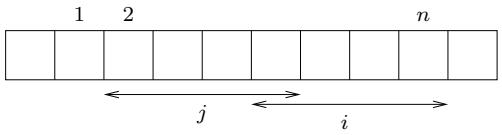


Fig. 4.3 – Représentation graphique du contexte de l'exemple 4.1.10

Tranches. Une tranche est un système de contraintes sur la variable logique (quantifiée) ℓ et les variables d'indice utilisées pour définir le contexte. Une tranche représente une ou plusieurs cases d'un tableau et appartient au domaine $\mathscr{T}$, qui est un sous-ensemble du domaine des zones portant uniquement sur les variables d'indice des programmes analysés et la variable logique ℓ.

Exemple 4.1.11. *Relativement au contexte de l'exemple 4.1.10, nous définissons la tranche suivante :*

$$\varphi \stackrel{def}{=} (\ell = j) \wedge (\ell \leq i - 1)$$

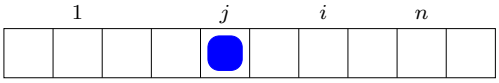


Fig. 4.4 – La tranche φ relativement au contexte ϕ de l'exemple 4.1.10

φ est une tranche de la propriété de tableaux à un certain point de contrôle du programme. La case représentée est donc celle indicée par j lorsque j est inférieur à i. Cette tranche est φ_3 dans l'exemple 4.1.8 et est colorée en bleu dans le schéma de la figure 4.4.

L'ensemble des tranches d'une propriété de tableaux constitue la partition P des tableaux du programme analysé.

Propriétés de tranche. Une propriété de tranche est un système de contraintes sur les cases de tableaux du programme. Dans une propriété de tranche, les tableaux sont indicés par des expressions arithmétiques contenant seulement la variable ℓ et des constantes entières. Les variables scalaires du programme peuvent apparaître dans les propriétés de tranche, mais pas les indices. Les propriétés de tranche appartiennent au domaine $\mathscr{P}$, qui est un sous-ensemble du domaine des zones portant uniquement sur les tableaux et les variables scalaires des programmes analysés. Pour chaque tranche de la partition, l'analyseur lui associe une propriété de tranche qui spécifie le contenu de cette tranche.

Exemple 4.1.12. *Nous définissons la propriété de tranche ψ associée à la tranche φ de l'exemple 4.1.10 :*

$$\psi \stackrel{def}{=} (A[\ell] \leq x) \wedge (A[\ell - 1] \leq A[\ell])$$

ψ décrit le contenu du tableau $A[\,]$ sur la tranche $\varphi \stackrel{def}{=} (\ell = j) \wedge (\ell \leq i - 1)$. Elle exprime littéralement que toutes les cases $A[\ell]$, pour ℓ dans la tranche φ sont inférieures ou égales à une certaine variable scalaire x, et qu'elles sont supérieures ou égales à la case qui les précède.

Dans l'exemple 4.1.12, on pourrait se demander pourquoi utiliser cette variable ℓ alors que la propriété peut être définie plus simplement par $(A[j] \leq x) \wedge (A[j - 1] \leq A[j] - 1)$. Cette représentation du contenu des tableaux au moyen de la variable quantifiée ℓ prend tout son sens dans l'expression du contenu d'un ensemble de cases non spécifiquement indicées :

Exemple 4.1.13. *Soient le contexte $(1 \leq i) \wedge (i \leq n)$ de l'exemple 4.1.9 et la tranche $(1 \leq \ell \leq i - 1)$. Si l'on associe à cette tranche la propriété de tranche $A[\ell] = 0$, on peut ainsi exprimer que l'ensemble des cases qui précèdent $A[i]$ ont pour valeur 0.*

Propriété de tableaux. Une propriété de tableaux est une formule donnant une représentation du contenu des tableaux du programme à un point de contrôle q donné.

Cette représentation du contenu est relative à un découpage du tableau, sa partition P. Ainsi les propriétés de tableaux n'appartiennent pas à un domaine abstrait mais à une *famille de domaines asbtraits* paramétrée par une partition P :

Définition 4.1.7 (Domaine abstrait des propriétés de tableaux)**.** *L'ensemble des domaines abstraits du contenu des tableaux diffère par la partition P sur laquelle les tableaux sont représentés :*

$$\mathscr{A}(P) = \bot_{\mathscr{A}} \cup \{\phi \wedge \forall \ell, \bigwedge_{k=1}^{|P|} \varphi_k \Rightarrow \psi_k \mid \phi \in \mathscr{I}, \forall \varphi_k \in P, \psi_k \in \mathscr{P}\}$$

où $\bot_{\mathscr{A}}$ s'exprime logiquement par $\bot$, que nous traduisons par le terme `False` dans le système Coq.

Exemple 4.1.14. *Relativement au contexte de l'exemple 4.1.9 et à la partition $P \stackrel{def}{=} \{(1 \leq \ell \leq i-1), (\ell = i), (i+1 \leq \ell \leq n)\}$, la propriété de tableaux suivante*

$$\Psi \stackrel{def}{=} (1 \leq i) \wedge (i \leq n) \wedge \forall \ell \bigwedge \begin{pmatrix} 1 \leq \ell \leq i-1 & \Rightarrow & A[\ell] \leq k \\ \ell = i & \Rightarrow & A[\ell] = k \\ i+1 \leq \ell \leq n & \Rightarrow & A[\ell] \geq k \end{pmatrix},$$

spécifie le contenu du tableau A au point de contrôle q du programme p*. Les propriétés de tranche de Ψ expriment que les cases inférieures à i contiennent des valeurs inférieures à k, que la case d'indice i contient une valeur égale à k et que les autres cases du tableau contiennent des valeurs supérieures à k.*

Nous remarquons que ces domaines abstraits ont une structure appartenant à la logique du premier ordre, et par conséquent, nous pourrons justifier les analyses d'Enkidu en travaillant directement sur les élements abstraits.

Les *invariants* calculés par l'analyseur Enkidu sont des ensembles de propriétés de tableaux dénotant le contenu des tableaux à chacun des points de contrôle du pogramme p. Soit Q l'ensemble des points de contrôle de p et Ψ_q la propriété de tableaux au point de contrôle q, l'invariant approximant la sémantique de p est l'ensemble $\{(q, \Psi_q) \mid q \in Q\}$.

4.1.2.2 Opérateurs du domaine abstrait des propriétés de tableaux

Les trois domaines de base $\mathscr{I}$, $\mathscr{T}$ et $\mathscr{P}$ sont des instances du domaine des zones ; par conséquent, les opérateurs de ces domaines sont ceux définis à la section 4.1.1.5. Nous présentons maintenant les opérateurs des propriétés de tableaux, qui, bien évidemment, utilisent les opérateurs du domaine des zones. Soient deux propriétés de tableaux du domaine abstrait $\mathscr{A}(P)$, où $|P| = n$:

$$\Psi \stackrel{def}{=} \phi \wedge \forall \ell, \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k \quad \text{et} \quad \Psi' \stackrel{def}{=} \phi' \wedge \forall \ell, \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi'_k.$$

Les trois opérateurs ensemblistes du domaine $\mathscr{A}(P)$ sont définis comme suit :

Définition 4.1.8 (Opérateurs du domaine des propriétés de tableaux)**.**
$\forall P \in \mathcal{D}_{part}, \forall \Psi, \Psi' \in \mathscr{A}(P)$:

- $\Psi \sqsubseteq_{\mathscr{A}(P)} \Psi' \iff \begin{cases} \Psi = \bot_{\mathscr{A}} \\ \vee \\ (\phi \sqsubseteq_{zone} \phi' \wedge \forall k \in [1,n], \psi_k \sqsubseteq_{zone} \psi'_k) \end{cases}$
- $\Psi \sqcup_{\mathscr{A}(P)} \Psi' = \begin{cases} \Psi & si\ \Psi' = \bot_{\mathscr{A}} \\ \Psi' & si\ \Psi = \bot_{\mathscr{A}} \\ (\phi \sqcup_{zone} \phi') \wedge \forall \ell \bigwedge_{k=1}^{n} \varphi_k \Rightarrow (\psi_k \sqcup_{zone} \psi'_k) & sinon \end{cases}$
- $\Psi \sqcap_{\mathscr{A}(P)} \Psi' = \begin{cases} \bot_{\mathscr{A}} & si\ \Psi = \bot_{\mathscr{A}}\ ou\ \Psi' = \bot_{\mathscr{A}} \\ (\phi \sqcap_{zone} \phi') \wedge \forall \ell \bigwedge_{k=1}^{n} \varphi_k \Rightarrow (\psi_k \sqcap_{zone} \psi'_k) & sinon \end{cases}$

La relation $\sqsubseteq_{\mathscr{A}(P)}$ est basée sur le test d'inclusion du domaine des zones et s'interprète en logique par l'implication $\Rightarrow$. Ainsi deux propriétés de tableaux définies sur la partition P, Ψ_1 et Ψ_2, seront ordonnées $(\Psi_1 \sqsubseteq_{\mathscr{A}(P)} \Psi_2)$ ssi $\Psi_1 \Rightarrow \Psi_2$.

4.1.2.3 Instrumentation du test d'inclusion dans $\mathscr{A}(P)$

Justifier les résultats de l'analyse exige de formuler le sens de ces opérateurs dans notre système d'inférence. En premier lieu, cela nécessite de mettre en évidence leur propriété de correction. Ensuite, il nous faut définir les patrons de preuve de ces propriétés de correction permettant de les prouver automatiquement quelques soient les arguments qui leur sont passés. Le test d'inclusion dans $\mathscr{A}(P)$, tout comme $\sqsubseteq_{zone}$, correspond à l'implication logique. En revanche, le domaine ne se limite pas à des conjonctions de formules arithmétiques. Nous décrivons ci-dessous la stratégie de preuve employée pour justifier les résultats de cet opérateur.

Le test $\Psi \sqsubseteq_{\mathscr{A}(P)} \Psi'$ doit retourner dans sa version instrumentée une preuve de $\Psi \Rightarrow \Psi'$, si le résultat du test $\Psi \sqsubseteq_{\mathscr{A}(P)} \Psi'$ est positif. La décidabilité de $\sqsubseteq_{\mathscr{A}(P)}$ dépend uniquement de la décidabilité de $\sqsubseteq_{zone}$. En effet, le problème $\sqsubseteq_{\mathscr{A}(P)}$ se décompose et se résoud de la façon suivante :

1. Prouver que $\phi \Rightarrow \phi'$;
2. Pour chaque tranche φ_k de la partition P, prouver que la propriété de tranche $\psi_k \in \Psi$ implique la propriété de tranche $\psi'_k \in \Psi'$, soit $\psi_k \Rightarrow \psi'_k$.

Ces deux sous-buts sont automatiquement prouvables grâce à l'instrumentation du test d'inclusion $\sqsubseteq_{zone}$ sur le domaine des zones (voir section 4.1.1.6). Nous donnons, pour conclure cette section, en figure 4.5, le patron de preuve qui permet de justifier chaque résultat retourné par $\sqsubseteq_{\mathscr{A}(P)}$.

Nous observons sur le patron de preuve de la figure 4.5 que la stratégie de justification proposée ci-dessus est explicitée. Ce patron de preuve fait appel aux justifications apportées par $\psi_k \sqsubseteq_{zone} \psi'_k$ pour chaque tranche φ_k de P et $\phi \sqsubseteq_{zone} \phi'$. Le reste de ce patron de preuve n'est qu'un *jeu* de dérivation permettant de se ramener à ces sous-buts.

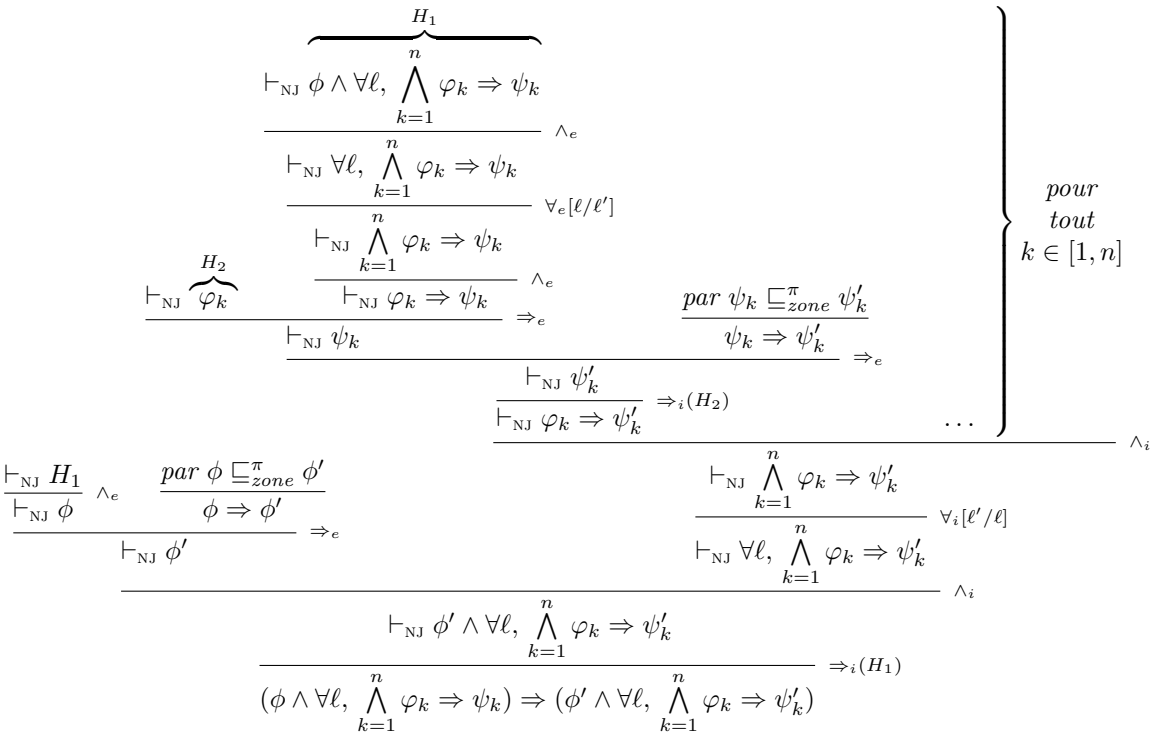


Fig. 4.5 – Patron de preuve pour la justification de $\sqsubseteq_{\mathscr{A}(P)}$

4.2 Modification des indices de tableaux

Nous nous intéressons dans cette section aux instructions d'affectation des variables d'indice de tableaux, de la forme $i := j + k$ où $k \in \mathbb{Z}$ et i et j sont deux variables d'indice de tableaux. Enkidu distingue les variables d'indice des variables scalaires et les affectations du type $A[\langle \textit{expression} \rangle] := \langle \textit{expression sur les indices} \rangle$ (par exemple $A[i] := i + 1$) ou du type $\langle \textit{indice} \rangle := \langle \textit{expression sur les tableaux} \rangle$ (par exemple $i := A[i + 1]$) ne sont pas permises. Ces restrictions sur le langage d'entrée de l'analyseur facilitent le traitement des affectations à l'aide du domaine abstrait des propriétés de tableaux et ce langage demeure complet au sens de Turing (les restrictions sur la structure de données ne changent rien à l'expressivité du langage). Malgré ces restrictions fortes sur les expressions d'indice de tableaux, la modification des variables d'indice introduit des problèmes complexes tant pour l'analyse que pour la justification de l'analyse.

4.2.1 Gestion des décalages d'indices dans l'analyse et sa justification

Le décalage des indices pose la plus grande difficulté, tant pour l'analyse que pour la justification de l'analyse. En effet, pour une certaine analyse, une partition P est fixée en amont et l'analyse calculera des propriétés de tableaux relativement à cette partition. Modifier un indice i ne change pas le contenu des tableaux mais modifie le contexte ϕ de la propriété de tableaux au point de contrôle courant. Cependant, ϕ n'est pas la seule formule qui doit être mise à jour par cette modification de l'indice i. En effet, les cases représentées par les tranches φ_k dont la définition dépend de i, ne sont plus les mêmes

après la modification de i (voir figure 4.6). En conséquence, les propriétés de tranche ψ_k associées aux φ_k ne sont plus appropriées et l'analyse se doit de les mettre à jour également. C'est cette étape que nous allons détailler dans cette section.

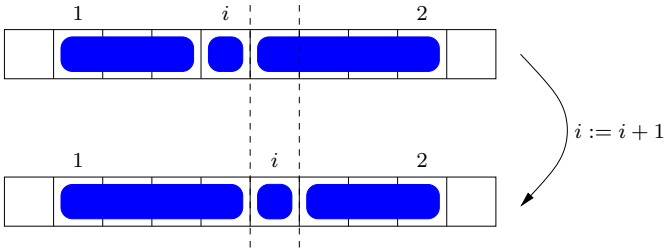


Fig. 4.6 – Représentation schématique de l'effet de la modification d'un indice

En figure 4.6, le tableau est découpé selon la partition $P \stackrel{def}{=} \{\varphi_1 = (\ell < i), \varphi_2 = (\ell = i), \varphi_3 = (\ell > i)\}$. À chacune de ces tranches est associée une propriété de tranche ψ_1, ψ_2 et ψ_3 respectivement. L'incrémentation de l'indice i a pour effet que la propriété de tranche ψ_2 n'est plus correcte pour la case désignée par φ_2. En effet, après l'incrémentation de i, la case désignée par φ_2 a pour propriété de tranche ψ_3. La tranche φ_3 n'est pas affectée par cette modification de i et préserve la propriété qui lui était associée avant l'affectation. Concernant φ_1, on remarque sur le schéma de la figure 4.6 que les cases qu'elle représente maintenant sont celles représentées par φ_1 et φ_2 avant la modification de i. En conséquence, la nouvelle propriété associée à φ_1 doit vérifier que $\varphi_1 \Rightarrow \psi_1 \vee \psi_2$. Malheureusement, la disjonction n'est pas un connecteur admis par le domaine abstrait des propriétés de tranche (le domaine des zones en l'occurrence). L'analyse devra donc calculer une propriété ψ_1', qui sera ensuite associée à φ_1, qui capte l'information contenue dans la formule $\psi_1 \vee \psi_2$. Pour cela, l'analyse emploie l'opérateur de borne supérieure $\sqcup_{zone}$, tel que $\psi_1' = \psi_1 \sqcup_{zone} \psi_2$. Du point de vue de la correction de l'analyse de la modification d'un indice, il n'est pas simple de générer automatiquement une preuve de la correction de ce nouvel invariant calculé. Nous présentons maintenant les problèmes rencontrés pour justifier ce traitement de l'affectation des indices.

4.2.1.1 Problématique pour générer la preuve de correction

Notre stratégie de preuve s'appuie sur un calcul de plus faible pré-condition. Il serait appréciable de pouvoir utiliser le test d'inclusion instrumenté, $\sqsubseteq^{\pi}_{\mathscr{A}(P)}$, pour générer une preuve de la formule obtenue par l'application du calcul de plus faible pré-condition sur la propriété de tableaux en post-condition de cette transition.

$$\frac{\textit{non prouvable avec } \sqsubseteq^{\pi}_{\mathscr{A}(P)} \textit{ car } wp(i := i+1, \Psi') \textit{ n'est pas définie sur } P}{\dfrac{\Psi \Rightarrow wp(i := i+1, \Psi')}{\{\Psi\} i := i+1 \{\Psi'\}} \Rightarrow_i [H]}$$

Ci-dessus, on remarque que l'analyse calcule une propriété de tableaux Ψ' qui, comme convenu, appartient au domaine abstrait $\mathscr{A}(P)$. Le calcul de plus faible pré-condition, $wp(i := i+1, \Psi')$, ne modifie dans Ψ' que les tranches (car les propriétés de tranche ne

contiennent pas d'occurrence de i, seulement ℓ pour les indices de tableaux), et fait donc basculer cette propriété de tableaux dans le domaine $\mathscr{A}(P')$. Cela nous conduit à prouver une implication dont les parties gauche et droite n'appartiennent pas au même domaine. Comme dans l'exemple trivial des intervalles (*cf* section 3.2.4.1), le test d'inclusion intrumenté n'est pas applicable pour prouver cette implication.

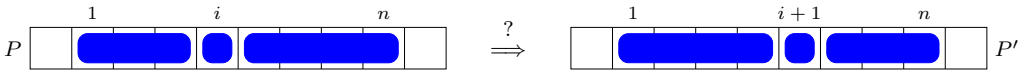


Fig. 4.7 – Problème posé par le calcul de plus faible pré-condition standard : les partitions des parties gauche et droite de l'implication ne sont pas les mêmes.

4.2.1.2 Solutions pour justifier l'analyse de la modification des indices

Du fait que le test d'inclusion intrumenté n'est pas applicable pour justifier la fonction de transfert de la modification des indices, une solution consiste à suivre l'approche proposée dans [SYY03, Cha06]. Cela revient à développer une stratégie *ad hoc*, permettant de construire, pour chaque modification d'indice traitée par l'analyse, une preuve de $\Psi \Rightarrow wp(i := i+1, \Psi')$. Pour ne pas partir aléatoirement, [Cha06] suggère d'instrumenter la fonction qui permet à l'analyse de gérer la modification des indices, car celle-ci peut servir d'*oracle*. Pour évaluer la faisabilité de cette approche, qui consiste à développer une stratégie de preuve *ad hoc*, nous nous sommes soumis à cet exercice. Cela nous permet de constater qu'instrumenter les fonctions de transfert est coûteux en temps de développement et que les preuves obtenues sont de taille importante. Nous présentons cette fonction puis son instrumentation en section 4.2.2.1 et 4.2.2.2.

Dans un second temps, nous avons exploré une solution qui consiste à générer une preuve de $\Psi \Rightarrow wp(i := i+1, \Psi')$ en transformant le problème de sorte qu'il puisse être traité uniquement à l'aide des opérateurs du domaine abstrait. Nous faisons cet effort dans le soucis de factoriser les fonctions à instrumenter, pour rendre ce travail réaliste et applicable à un coût raisonnable, en nous limitant aux opérateurs ensemblistes. Comme nous l'avons vu, $\sqsubseteq^{\pi}_{\mathscr{A}(P)}$ ne convient pas. Néanmoins, il est possible de modifier le calcul de plus faible pré-condition ($\tilde{wp}$ sur la figure 4.8) de sorte qu'il retourne une formule équivalente à $wp(i := i+1, \Psi')$ mais exprimée sur la partition $P'' \stackrel{def}{=} P \sqsubseteq_{part} P''$. Cette modification présente l'intêret d'avoir – certes une implication entre deux propriétés de tableaux définies sur des partitions différentes – une plus faible pré-condition définie sur une partition plus fine que P. Pour prouver l'implication $\Psi \Rightarrow \tilde{wp}(i := i+1, \Psi')$ (voir figure 4.8) il est dès lors possible d'utiliser une version plus générale du test d'inclusion sur le domaine des propriétés de tableaux, proposée dans [GMT08]. Ce test d'inclusion plus générique permet de comparer des propriétés de tableaux définies sur des partitions différentes, à condition que ces partitions soient ordonnées selon $\sqsubseteq_{part}$. Les détails de cette solution sont donnés en section 4.2.3.

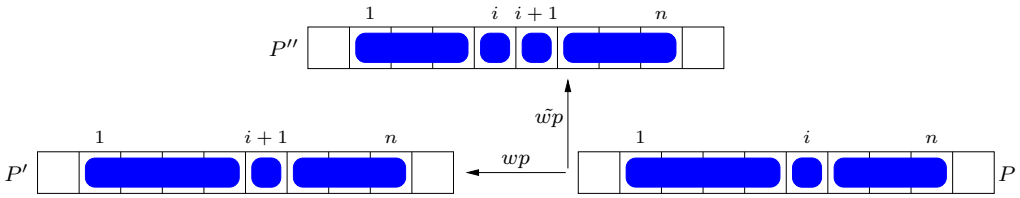


Fig. 4.8 – Partitions créées par les calculs de plus faible pré-condition

4.2.2 Justification de l'analyse de la modification des indices par une stratégie *ad hoc*

Dans ce qui suit, nous présentons la fonction utilisée par l'analyse, qui relativement à une propriété de tableaux donnée, calcule la meilleure propriété de tranche, pour une tranche donnée (qui n'appartient pas forcément à la partition du programme analysé). Étant donnée une propriété de tableaux $\Psi \stackrel{def}{=} \phi \wedge \forall \ell, \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k$ et une tranche φ, exprimée sur une partie des variables du contexte ϕ, le but de cette fonction est de calculer la propriété la plus précise possible, ψ, sur le contenu de la tranche φ, à partir des informations contenues dans Ψ. Soit ExtractProperty la fonction qui implante ce calcul ($\mathsf{ExtractProperty}(\Psi, \varphi) = \psi$), nous définissons sa spécification par le triplet de Hoare suivant :

$$\{\Psi \wedge \varphi\}\psi := \mathsf{ExtractProperty}(\Psi, \varphi)\{\Psi \Rightarrow (\varphi \Rightarrow \psi)\}$$

4.2.2.1 Algorithme du calcul de meilleure information

Soit $\Psi \stackrel{def}{=} \phi \wedge \forall \ell, \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k \in \mathscr{A}(P)$ la propriété de tableaux au point de contrôle courant de l'analyse. Nous définissons la partition P par l'ensemble $\{\varphi_1, \varphi_2, \ldots, \varphi_n\}$ de tranches. Nous décrivons l'algorithme de la fonction ExtractProperty en trois étapes :

1. *Calcul des tranches en intersections avec φ :* L'analyse détermine, par un calcul d'intersection dans $\mathscr{T}$, le sous-ensemble de P ($\mathcal{I}$) des tranches qui *couvrent* la tranche φ. Pour cela, l'analyse effectue une succession de *tests d'intersection vide*, avec $\mathcal{I} \stackrel{def}{=} \emptyset$ initialement :
   $$\boxed{\textit{pour tout } \varphi_k \in P, \textit{ si } \varphi_k \sqcap_{zone} \varphi \neq \bot_{zone} \textit{ alors } \mathcal{I} := \varphi_k \cup \mathcal{I}}$$

2. *Récupération des propriétés de tranche correspondantes :* L'analyse construit ensuite l'ensemble $\mathcal{P}$ des propriétés de tranche correspondantes aux tranches de l'ensemble $\mathcal{I}$. Cela se fait aussi par un parcours des contraintes de P, avec $\mathcal{P} \stackrel{def}{=} \emptyset$ initialement :
   $$\boxed{\textit{pour tout } \varphi_k \in P, \textit{ si } \varphi_k \in \mathcal{I} \textit{ alors } \mathcal{P} := \psi_k \cup \mathcal{P}}$$

3. *Calcul de la meilleure information sur la tranche donnée :* Ce calcul se termine en définissant une propriété ψ comme *la plus petite borne supérieure* (au sens de $\sqsubseteq_{zone}$) des propriétés de $\mathcal{P}$. Cette propriété est d'un point de vue logique, la plus forte propriété déductible à partir de Ψ. ψ vaut initialement $\bot_{zone}$:

$\boxed{\textit{pour tout } \psi_k \in \mathcal{P},\ \psi := \psi \sqcup_{zone} \psi_k}$

L'analyse associe ψ à la contrainte φ, pour former la *nouvelle propriété de tranche* $(\varphi \Rightarrow \psi)$.

Exemple 4.2.1. *Pour illustrer le fonctionnement de l'algorithme de* ExtractProperty *nous reprenons l'exemple de la figure 4.6 où un tableau est découpé selon la partion* $P \stackrel{def}{=} \{\varphi_1 = (\ell < i), \varphi_2 = (\ell = i), \varphi_3 = (\ell > i)\}$. *En considérant que la propriété de tableaux au point de contrôle qui précède l'instruction* $i := i+1$ *soit* $\Psi \stackrel{def}{=} \phi \forall \ell \bigwedge \begin{pmatrix} \varphi_1 \Rightarrow \psi_1 \\ \varphi_2 \Rightarrow \psi_2 \\ \varphi_3 \Rightarrow \psi_3 \end{pmatrix}$, *l'analyse, grâce à la fonction* ExtractProperty, *calcule la post-condition* $\Psi' \stackrel{def}{=} \phi \forall \ell \bigwedge \begin{pmatrix} \varphi_1 \Rightarrow \psi'_1 \\ \varphi_2 \Rightarrow \psi'_2 \\ \varphi_3 \Rightarrow \psi'_3 \end{pmatrix}$.
Pour calculer la propriété de tranche ψ'_1 *la fonction de transfert va chercher la meilleure information sur la tranche* $\varphi_1[i/i+1]$ *soit* $(\ell < i+1) \equiv (\ell \leq i)$. *Nous détaillons ce calul en suivant les trois étapes de l'algorithme :*

1. *La première étape détermine que l'ensemble* $\mathcal{I}$ *de tranches de* P *qui intersecte la tranche cherchée est* $\{\varphi_1, \varphi_2\}$. φ_3 *n'est pas retenue car* $(\ell > i) \sqcap_{zone} (\ell \leq i) = \bot_{zone}$
2. *La seconde consiste à récupérer dans la propriété de tableaux* Ψ *les propriétés de tranche qui correspondent aux tranches de* $\mathcal{I}$. *Soit* $\mathcal{P}$ *l'ensemble* $\{\psi_1, \psi_2\}$. *Cela signifie que sur la tranche* $(\ell \leq i)$ *les cases vérifient la propriété* $\psi_1 \vee \psi_2$.
3. *La dernière étape consiste à appoximer la propriété* $\psi_1 \vee \psi_2$ *au moyen de l'opérateur de borne supérieure car le connecteur logique* $\vee$ *n'est pas admis dans la syntaxe des propriétés de tranche. Ainsi,* $\psi'_1 \stackrel{def}{=} \psi_1 \sqcup_{zone} \psi_2$

Dans le cas où l'ensemble $\mathcal{P}$ *contient plus de deux propriétés de tranche, le calcul de la propriété sur la tranche cherchée se fait récursivement :* $((\psi_1 \sqcup_{zone} \psi_2) \sqcup_{zone} \psi_3) \sqcup_{zone} \ldots$

4.2.2.2 Principe de l'instrumentation du calcul de meilleure information

Justifier le calcul de meilleure information n'impose pas de prouver que l'information calculée (la propriété de tranche ψ) soit la meilleure qu'elle puisse être (au sens de l'implication logique). Dans cette étude, *seule la correction nous importe*. La *qualité* (en terme de précison) de l'analyse est indépendante de notre travail : notre objectif est d'apporter un certificat pour chaque résultat que produit l'analyse, sans regard sur la valeur de ce résultat. Ainsi savoir si ψ est la plus forte propriété déductible de Ψ relativement à φ (*i.e* $\forall \psi' \in \mathscr{P}, (\Psi \Rightarrow (\varphi \Rightarrow \psi')) \Rightarrow (\psi \Rightarrow \psi'))$, n'est pas de notre ressort. L'analyse le prétend, nous certifions juste que ψ est correcte, en fournissant une preuve de $\Psi \Rightarrow (\varphi \Rightarrow \psi)$.

Les parties arithmétiques, des patrons de preuve que nous avons définis pour instrumenter cette fonction, qui ne rentrent pas dans le cadre du domaine abstrait des zones, sont prouvées à l'heure actuelle au moyen des tactiques `omega` et `intuition` du système Coq. Si nous n'avons pas pu construire de patrons de preuve pour justifier que les partitions utilisées par Enkidu sont valides, c'est parce que celles-ci ne sont pas issues d'un calcul dans la version présente d'Enkidu. Ces partitions sont fournies dans le code pour un ensemble restreint de programmes. De ce fait, nous ne disposons pas

de fonction à instrumenter pour justifier la validité des partitions. L'analyseur n'étant pas parfaitement automatique, l'automatisation des justifications en est atteinte. Cette solution n'est évidemment pas satisfaisante, mais nous a permis de tester plus rapidement la correction des preuves générées par notre instrumentation globale de l'outil. Une solution *ad hoc* complète consisterait à développer des patrons de preuve, ou des procédures de décision certifiées, permettant de prouver la validité de ces partitions. La tactique `intuition` a quant à elle été employée pour prouver l'insatisfiabilité de deux tranches pour justifier les tests d'intersection vide de la première étape de la fonction ExtractProperty. Cela nous a permis de ne pas utiliser les justifications de l'opérateur d'intersection puis de l'algorithme de normalisation pour justifier la correction de ces tests. Les instrumentations de ces algorithmes ne sont pas optimisées et fournissent des justifications de taille trop importante pour des sous-buts aussi simples à prouver. La tactique `intuition` est aujourd'hui moins coûteuse.

Principe général. En utilsant l'introduction de l'implication par deux fois, notre but revient à prouver ψ sous les hypothèses Ψ et φ.

La méthode ici, n'est pas de partir de la formule à prouver et de construire un arbre de dérivation en *remontant* jusqu'à atteindre les hypothèses – comme ce peut être le cas dans les approches interactives – mais de suivre les calculs de l'algorithme afin de bénéficier de sa stratégie. Pour cela, il nous faut représenter logiquement les ensembles fournis (la partition P) à ou calculés ($\mathcal{I}$ et $\mathcal{P}$) par l'algorithme pour déterminer ψ. Dans un premier temps, par correspondance avec l'algorithme de ce calcul, il nous faut déduire la représentation de P. Considérons que $\mathcal{I} \stackrel{def}{=} \{\varphi_1, \varphi_2, \ldots, \varphi_n\}$, sa représentation en logique du premier ordre est la disjonction $\bigvee_{k=1}^{n} \varphi_k$. Cette disjonction doit être prouvée si l'on veut l'utiliser dans l'arbre de dérivation. Pour cela, nous faisons appel à la tactique `omega`. Pour justifier la première étape du calcul, nous souhaitons ensuite déduire une représentation logique de l'ensemble $\mathcal{I}$, à partir de φ et Ψ. En effet, l'analyse construit $\mathcal{I}$ en éliminant toutes les contraintes de P n'ayant pas d'intersection avec φ. Considérons que $P \stackrel{def}{=} \{\varphi_1, \varphi_2, \ldots, \varphi_m\}$. Sa représentation logique n'est autre que $\bigvee_{k=1}^{m} \varphi_k$. Ceci constitue la première étape de la justification :

$$\textit{déduction de } \bigvee_{k=1}^{n} \varphi_k, \textit{ puis } ((\varphi \wedge \bigvee_{k=1}^{n} \varphi_k) \Rightarrow \bigvee_{k=1}^{m} \varphi_k).$$

La deuxième étape relie les tranches à leur propriété. Il nous faut, pour chaque tranche de $\mathcal{I}$, déduire la propriété qui lui correspond à partir de Ψ, puis déduire la représentation de l'ensemble $\mathcal{P}$ (*i.e.* $\bigvee_{k=1}^{m} \psi_k$) :

$$(\Psi \wedge \bigvee_{k=1}^{m} \varphi_k) \Rightarrow \bigvee_{k=1}^{m} \psi_k).$$

La dernière étape a pour objectif de justifier le calcul de ψ à partir de $\mathcal{P}$. Nous utilisons la justification de l'opérateur $\sqcup_{zone}$ pour prouver :

$$(\bigvee_{k=1}^{m} \psi_k) \Rightarrow \psi$$

Pour récapituler, l'idée de l'instrumentation s'appuie sur une correspondance forte avec l'algorithme de ExtractProperty (nous insistons là-dessus car cela n'est pas le cas pour les fonctions de transfert des autres instructions qui peuvent être justifiées grâce à l'instrumentation des seuls opérateurs ensemblistes du domaine abstrait des propriétés de tableaux) et s'effectue en trois étapes, par assemblage de trois patrons de preuve :

$$\underbrace{((\varphi \wedge \bigvee_{k=1}^{n} \varphi_k) \Rightarrow \bigvee_{k=1}^{m} \varphi_k)}_{\text{étape 1}} \wedge \underbrace{((\Psi \wedge \bigvee_{k=1}^{m} \varphi_k) \Rightarrow \bigvee_{k=1}^{m} \psi_k)}_{\text{étape 2}} \wedge \underbrace{((\bigvee_{k=1}^{m} \psi_k) \Rightarrow \psi)}_{\text{étape 3}}$$

Remarque. *Les preuves que nous obtenons par ce principe de justifiaction (via les patrons de preuve) ont une structure* statiquement déterminée*, indépendamment des valeurs de Ψ, φ et ψ. Celles-ci sont fournies par* Enkidu *lors de l'exécution. La difficulté de notre travail réside dans la construction de preuves valides pour toutes instances de ces formules Ψ, φ et ψ. De ce fait, la taille des preuves est fixe et par conséquent non-optimale. Pour exemple, dans le cas où $\psi \stackrel{def}{=} \top_{zone}$, il n'est pas utile de fournir une preuve formelle avec une telle structure pour justifier son calcul. Pourtant, si nous n'*optimisons *pas en définissant des patrons de preuve pour un certain nombre de cas particuliers identifiés, et notamment celui-ci, la preuve de $\top_{zone}$ qui sera obtenue aura tout de même la structure de dérivation (en trois étapes).*

4.2.2.3 Details de l'instrumentation du calcul de meilleure information

Dans le cas où la tranche φ cherchée n'est pas définie au moyen d'une égalité, plusieurs contraintes peuvent être en intersection avec φ. Il nous faut fournir, pour chaque *test d'intersection vide* positif, réalisé par l'analyse (à l'etape 1 de l'algorithme), une *preuve de réfutation*. Cela nous permet de justifier la sélection des *propriétés* utilisées pour le calcul de ψ. Pour chaque tranche $\varphi_k \in P$, si $\varphi_k \sqcap_{zone} \varphi = \bot_{zone}$ alors φ_k n'est pas conservée pour la seconde étape. Intuitivement, cela signifie que « *les tranches du tableau définies par φ_k et φ n'ont pas de case commune* » . D'un point de vue *logique*, cela revient à prouver $\varphi_k \wedge \varphi \Rightarrow \bot$ pour chaque contrainte satisfaisant ce test. Dans l'implantation actuelle de l'instrumentation d'Enkidu, ces preuves de réfutation sont obtenues au moyen de la tactique `intuition` du système Coq. Nous développons ci-dessous comment ces preuves de réfutation sont utilisées pour garder l'ensemble complémentaire de tranche ($\mathcal{I}$ en l'occurrence) puis nous donnons le patron de preuve associé à la première étape de la fonction ExtractProperty.

Étape 1 : Déduction des tranches couvrantes. Cette première étape se traduit par la déduction d'une formule $\bigvee_{k=1}^{m} \varphi_k$ formalisant $\mathcal{I}$ en logique du premier ordre (nous rappelons que nous dénotons par m le nombre de contraintes de $\mathcal{I}$, *i.e.* qu'il y a m contraintes dans Ψ qui ont une intersection non-vide avec φ). L'idée de la preuve est de partir du plus grand ensemble couvrant φ (dans les bornes du tableau), en l'occurrence la partition P traduit par $\bigvee_{k=1}^{n} \varphi_k$, et de le réduire en retirant pas à pas chaque tranche de

P qui n'a pas d'intersection avec la tranche φ cherchée. Il nous faut ainsi prouver dans un premier temps la formule $\bigvee_{k=1}^{n} \varphi_k$. Cette formule est une caractérisation de l'ensemble des entiers naturels, découpée relativement aux variables d'indice du programme. Elle s'evalue donc systématiquement à $\top$ et sa preuve nous est donnée par la tactique `omega` de l'assistant à la preuve Coq.

Exemple 4.2.2. *Soient i une variable du programme `p` analysé et $\Psi \stackrel{def}{=} \phi \wedge (\forall \ell, (\varphi_1 \Rightarrow \psi_1) \wedge (\varphi_2 \Rightarrow \psi_2) \wedge (\varphi_3 \Rightarrow \psi_3))$ la propriété de tableaux au point de contrôle courant, avec $\phi \stackrel{def}{=} 1 < i < n$, $\varphi_1 \stackrel{def}{=} \ell < i$, $\varphi_2 \stackrel{def}{=} \ell = i$ et $\varphi_3 \stackrel{def}{=} i < \ell$.*

La disjonction $(\varphi_1 \vee \varphi_2 \vee \varphi_3) \equiv \ell \in \mathbb{N}$, i.e. *que la disjonction des trois contraintes est équivalente à la formule $\ell \in \mathbb{N}$ mais* découpée *selon i. Le contexte ϕ sert à restreindre l'ensemble des entiers naturels dénoté par cette disjonction aux bornes du tableau, 1 et n en l'occurrence.*

La suite de la première étape consiste à déduire $\bigvee_{k=1}^{m} \varphi_k$ à partir de la disjonction déjà établie ($\bigvee_{k=1}^{n} \varphi_k$), au moyen d'utilisations récursives ($(n-1)-(m+1)$ applications en figure 4.9) de la règle d'*élimination de la disjonction* et des *preuves de réfutation* pour les tranches non conservées par l'algorithme ($\varphi_n, \ldots \varphi_{m+1}$ en figure 4.9), comme le montre le patron de preuve donné en figure 4.9.

Étape 2 : Déduction des propriétés associées aux tranches. La deuxième étape du calcul de meilleure information consiste à calculer les propriétés de $\mathcal{P}$ correspondant aux tranches de $\mathcal{I}$ (exprimées en logique du premier ordre par $\bigvee_{k=1}^{m} \varphi_k$). La justification de cette étape consiste à déduire sous une même formule l'ensemble des propriétés de $\mathcal{P}$ à partir de la formule déduite lors de la première étape. La formule représentant l'ensemble $\mathcal{P}$, a pour formalisation logique $\bigvee_{k=1}^{m} \psi_k$: le connecteur disjonctif est encore employé en raison des diverses propriétés de tranche potentiellement affectables à la tranche spécifiée par φ. Intuitivement le raisonnement, en reprenant depuis la première étape, peut être exprimé comme suit :

> « *Si je suis dans φ_1 alors j'ai la propriété ψ_1, et si je suis dans φ_2 alors j'ai la propriété $\psi_2, \ldots$, et si je suis dans φ_m alors j'ai la propriété ψ_m. Or, si je suis dans φ alors je suis dans $\bigvee_{k=1}^{m} \varphi_k$, i.e. je suis soit dans φ_1, soit dans $\varphi_2, \ldots$, soit dans φ_m. Donc, si je suis dans φ, alors j'ai la propriété ψ_1 ou j'ai la propriété $\psi_2, \ldots$ ou j'ai la propriété ψ_m* » .

Cette preuve se construit au moyen d'applications récursives de l'élimination de la disjonction et des formules $\bigvee_{k=1}^{m} \varphi_k$ et Ψ.

Il faut ensuite, à partir de φ_m et Ψ, déduire la propriété ψ_m, puis en déduire la disjonction de propriétés représentant $\mathcal{P}$, au moyen de la règle d'inférence $\vee_i$ (figure 4.10(a)). Le même patron de preuve est appliqué pour les tranches restantes, en $(m-1)$

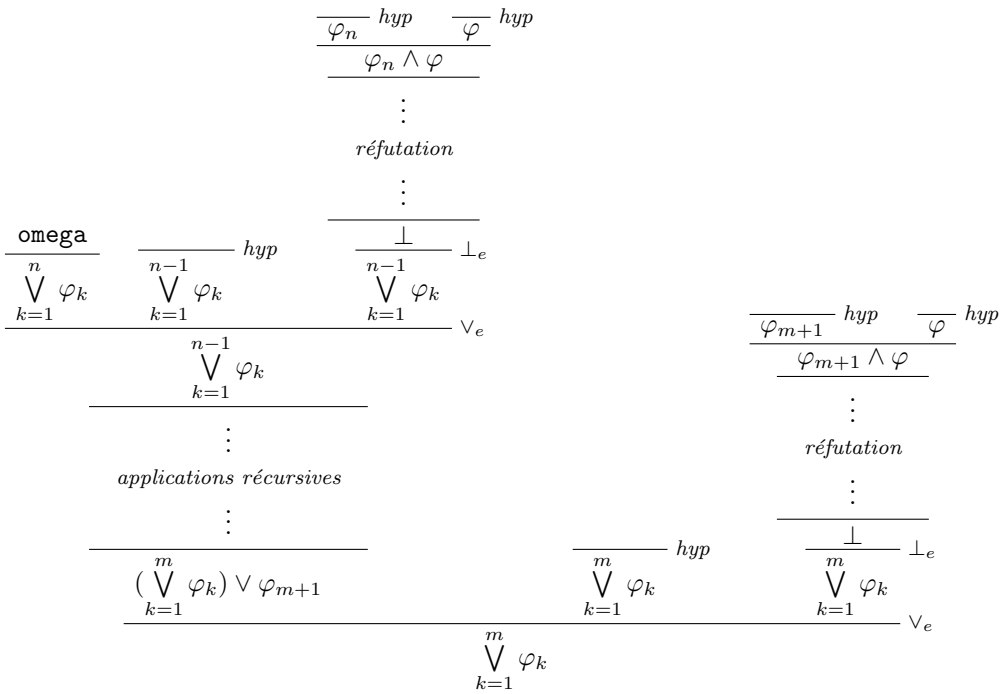


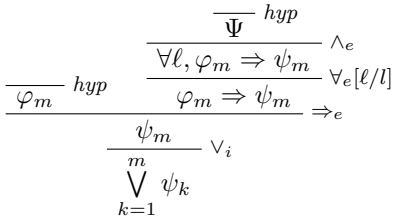
Fig. 4.9 – Patron de preuve permetant de justifier la première étape du calcul de meilleure information

applications recursives (figure 4.10(b)). Les preuves justifiant la seconde étape ont la structure donnée en figure 4.10(c)).

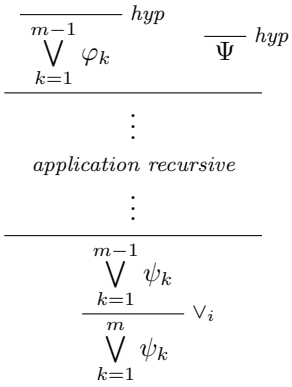
Remarque. *Les hypothèses* $\bigvee_{k=1}^{m-1} \varphi_k$ *et* φ_m*, utilisées dans ce patron de preuve, sont obtenues par l'élimination de la disjonction* $((\bigvee_{k=1}^{m-1} \varphi_k) \vee \varphi_m) = (\bigvee_{k=1}^{m} \varphi_k)$ *précédemment prouvée (à la première étape de l'algortithme de* ExtractProperty*).*

Étape 3 : Déduction de la propriété de tranche en φ. La dernière étape de la justification de la fonction ExtractProperty(Ψ, φ) consiste à déduire le résultat de cette fonction à partir de la formule déduite à la seconde étape. Le patron de preuve de la propriété retournée, $\psi \stackrel{def}{=} \bigsqcup_{k=1}^{m} \psi_k = (\bigsqcup_{k=1}^{m-1} \psi_k) \sqcup_{zone} \psi_m$, s'appuie sur l'instrumentation de l'opérateur $\sqcup_{zone}$ et la règle d'inférence $\vee_e$, toujours en $(m-1)$ applications récursives (Figure 4.11). La disjonction *éliminée* lors de la dernière application de la règle $\vee_e$ $((\bigwedge_{k=1}^{m-1} \psi_k) \vee \psi_m$ dans le patron de preuve donné en figure 4.11) est déduite de la seconde étape.

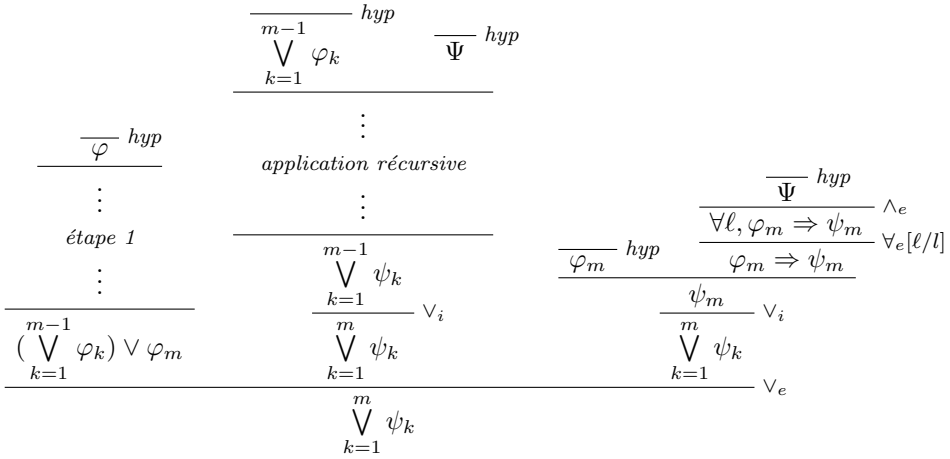
Pour finir, il suffit d'introduire la tranche φ utilisée à la première étape, pour déduire et justifier la *propriété de tranche* calculée $(\varphi \Rightarrow \psi)$. La structure complète de la preuve



(a) Patron de preuve utilisé pour prouver la disjonction de m propriétés de tranche



(b) Applications récursives du patron de preuve pour $m-1$ propriétés de tranche



(c) Patron de preuve global de la disjonction des propriétés de tranche

Fig. 4.10 – Justification de la seconde étape de l'algorithme

permettant de justifier la correction d'une exécuton de la fonction $\mathsf{ExtractProperty}(\Psi, \varphi)$ est donnée en figure 4.12.

4.2.3 Justification de l'analyse de l'affectation des indices par modification du calcul de plus faible pré-condition

Nous avons vu en section 4.2.1.1 qu'une grande partie du problème pour générer une preuve justifiant ce traitement de l'affectation des indices, vient du calcul de plus faible pré-condition qui ne préserve pas les propriétés de tableaux dans le domaine abstrait $\mathscr{A}(P)$ (la partition des tableaux n'est plus la même). Plutôt que de développer une stratégie *ad hoc* complexe, comme nous l'avons fait dans la section précédente, nous pensons qu'il est préférable d'utiliser un système d'inférence correct (comme la logique de Hoare et la déduction naturelle) et de limiter l'instrumentation aux opérateurs ensemblistes du domaine abstrait : ceux-ci sont utilisés pour justifier plusieurs fonctions

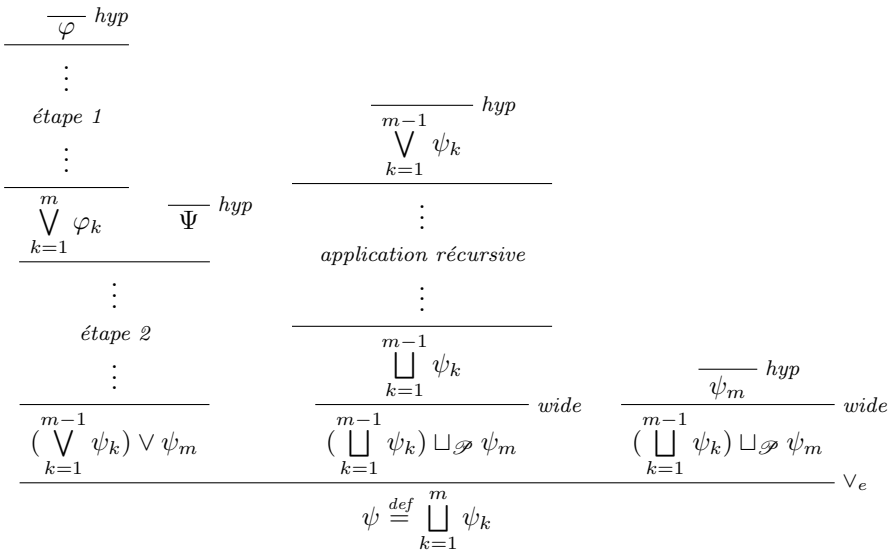


Fig. 4.11 – Justification de la troisième étape de l'algorithme

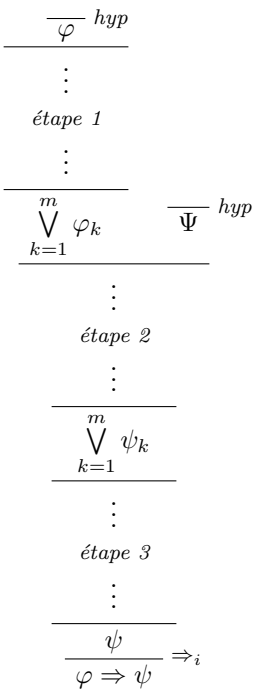


Fig. 4.12 – Patron de preuve global justifiant le calcul de meilleure information

de l'analyse et il semble intéressant, d'un point de vue pratique, de *factoriser* le code instrumenté pour justifier les résultats de l'analyse. De plus, ces opérateurs instrumentés pourront être réutilisés dans le cadre d'une autre analyse statique s'appuyant sur le même domaine abstrait. Nous proposons donc de modifier le calcul de plus faible pré-condition de sorte qu'il retourne une propriété de tableaux exprimée dans une partition plus fine que P. Cela dans le but d'utiliser l'instrumentation d'un opérateur ensembliste pour construire une preuve de l'implication retournée par wp. La modification du calcul de plus faible pré-condition consiste à développer une fonction $\tilde{wp}$ retournant une propriété de tableaux non pas dans le domaine abstrait $\mathscr{A}(P')$, avec $P' \stackrel{def}{=} P[i/i+1]$ (*cf.* section 4.2.1.1), mais dans $\mathscr{A}(P'')$, où $P'' \stackrel{def}{=} P \sqcup_{part} P'$. L'important est que la propriété de tableaux retournée par $\tilde{wp}$ soit logiquement équivalente à celle qui était retournée par wp, mais exprimée dans une partition plus fine, avec plus de tranches. L'interêt d'exprimer la plus faible pré-condition dans cette partition P'' vient du fait que $P \sqsubseteq_{part} P''$. En effet, [GMT08] propose un test d'inclusion plus générique que celui proposé dans [HP08] qui permet de comparer des propriétés de tableaux étant définies sur des partitions différentes. Ce test permet de décider l'inclusion entre deux propriétés de tableaux $\Psi_1 \in P_1$ et $\Psi_2 \in P_2$ si $P_1 \sqsubseteq_{part} P_2$. Utiliser ce test d'inclusion instrumenté pour générer les obligations de preuve retournées par ce calcul de plus faible pré-condition modifié nous dispense d'instrumenter la fonction ExtractProperty. Nous en présentons l'instrumentation en section 4.2.3.2.

4.2.3.1 Modification du calcul de plus faible pré-condition

Il s'agit ici de modifier la fonction wp sur les indices, de manière à ce qu'elle retourne une propriété de tableaux exprimée dans une partition plus fine que P', qui inclue également P. Un candidat évident est $P'' \stackrel{def}{=} P \sqcup_{part} P'$, par définition de $\sqcup_{part}$ (*cf.* définition 4.1.6). L'utilité de cette modification est que si la propriété de tableaux à prouver est exprimée dans une partition plus fine que P alors il existe un test d'inclusion générique $\sqsubseteq_{\mathscr{A}_{gen}}$ permettant de comparer des propriétés de tableaux définies sur ces deux partitions :

$$\forall \Psi \in \mathscr{A}(P), \forall \Psi'' \in \mathscr{A}(P''), (P \sqsubseteq_{part} P'') \Rightarrow (\Psi \sqsubseteq_{\mathscr{A}_{gen}} \Psi'' \textit{ est décidable})$$

Notre idée est d'intégrer dans wp le calcul de la partition P'' à partir de P (fixée) et de P' (obtenue par l'application du wp classique sur P). Ce calcul est réalisé au moyen de $\sqcup_{part}$ qui utilise $\sqcap_{zone}$. Ces deux opérateurs doivent être implantés dans le système Coq pour pouvoir modifier le calcul de plus faible pré-condition comme nous le souhaitons. La définition de ce calcul de plus faible pré-condition est donnée ci-dessous :

On considère en post-condition d'une instruction de la forme $i := j + k$ une propriété de tableaux $\Psi' \in \mathscr{A}(P) \stackrel{def}{=} \phi' \wedge \forall \ell, \bigwedge_{k=1}^{|P|} \varphi_k \Rightarrow \psi'_k$.

$$\tilde{wp}(i := j + k, \Psi') \stackrel{def}{=} \phi'[i/j+k] \wedge \forall \ell, \bigwedge_{q=1}^{|P''|} \varphi'_q \Rightarrow \psi'_q,$$

$$\text{telle que } P'' \stackrel{def}{=} P \sqcup_{part} P[i/j+k] \text{ et } \begin{cases} \forall \varphi'_q \in P'', \forall \varphi'_k \in P[i/j+k] \\ \quad \text{si } \varphi'_q \sqsubseteq_{zone} \varphi'_k, \\ \quad \text{alors } \psi'_q := \psi'_k \end{cases}$$

Exemple 4.2.3. *Nous reprenons l'exemple des figures 4.6 et 4.8.*

À partir de l'invariant en post-condition $\Psi' \stackrel{def}{=} \phi' \wedge \forall \ell, \bigwedge \begin{pmatrix} (\ell < i) \Rightarrow \psi'_1 \\ (\ell = i) \Rightarrow \psi'_2 \\ (\ell > i) \Rightarrow \psi'_3 \end{pmatrix}$*, le nouveau calcul de plus faible pré-condition retourne la propriété de tableaux appartenant au domaine* $\mathscr{A}(P'')$ *:*

$$\tilde{wp}(i := i+1, \Psi') = \phi'[i/i+1] \wedge \forall \ell, \bigwedge \begin{pmatrix} (\ell < i) \Rightarrow \psi'_1 \\ (\ell = i) \Rightarrow \psi'_1 \\ (\ell = i+1) \Rightarrow \psi'_2 \\ (\ell > i+1) \Rightarrow \psi'_3 \end{pmatrix}$$

au lieu de la propriété de tableaux du domaine $\mathscr{A}(P')$ *:*

$$wp(i := i+1, \Psi') = \phi'[i/i+1] \wedge \forall \ell, \bigwedge \begin{pmatrix} (\ell < i+1) \Rightarrow \psi'_1 \\ (\ell = i+1) \Rightarrow \psi'_2 \\ (\ell > i+1) \Rightarrow \psi'_3 \end{pmatrix}$$

Contrairement à la partition P'*, la partition* P'' *est plus fine que la partition* P*, dans le sens où* $P \sqsubseteq_{part} P''$*. Pour chaque tranche* φ_q *de* P''*, il existe une tranche* φ_p *de* P *telle que* $\varphi_q \sqsubseteq_{zone} \varphi_p$*. En effet, sur cet exemple* $(\ell < i) \sqsubseteq_{zone} (\ell < i)$*,* $(\ell = i) \sqsubseteq_{zone} (\ell = i)$*,* $(\ell = i+1) \sqsubseteq_{zone} (\ell > i)$ *et* $(\ell > i+1) \sqsubseteq_{zone} (\ell > i)$*.*

Dans $\tilde{wp}(i := i+1, \Psi')$*, la propriété de tranche associée aux tranches* $(\ell < i)$ *et* $(\ell = i)$ *est celle associée à la tranche* $(\ell = i)$ *dans* Ψ'*, soit* ψ'_1*, car* $(\ell < i) \sqsubseteq_{zone} (\ell = i)[i/i+1]$ *et* $(\ell = i) \sqsubseteq_{zone} (\ell = i)[i/i+1]$*.*

4.2.3.2 Test d'inclusion générique de $\mathscr{A}$

Pour justifier l'analyse de la modification des indices de tableaux, après avoir retouché le calcul de plus faible pré-condition, nous proposons d'utiliser l'instrumentation d'un test d'inclusion plus générique que $\sqsubseteq_{\mathscr{A}(P)}$. Ce test d'inclusion provient d'un domaine abstrait plus expressif que celui utilisé par Enkidu [GMT08].

Définition 4.2.1 (Opérateur du test d'inclusion générique)**.** *Soient deux propriétés de tableaux appartenant respectivement aux domaines abstraits* $\mathscr{A}(P)$ *et* $\mathscr{A}(P')$ *où* $|P| = n_1$ *et* $|P'| = n_2 \geq n_1$*, tel que* $P \sqsubseteq_{part} P'$ *:*

$$\Psi \stackrel{def}{=} \phi \wedge \forall \ell, \bigwedge_{p=1}^{n_1} \varphi_p \Rightarrow \psi_p \quad \text{et} \quad \Psi' \stackrel{def}{=} \phi' \wedge \forall \ell, \bigwedge_{q=1}^{n_2} \varphi'_q \Rightarrow \psi'_q.$$

Le test d'inclusion générique est défini comme suit :

$$\Psi \sqsubseteq_{\mathscr{A}_{gen}} \Psi' \iff \begin{cases} \Psi = \bot_{\mathscr{A}} \\ \vee \\ (\phi \sqsubseteq_{zone} \phi' \wedge \forall q \in [1, n'], \exists p \in [1, n], \varphi'_q \sqsubseteq_{zone} \varphi_p \wedge \psi_p \sqsubseteq_{zone} \psi'_q) \end{cases}$$

La définition de ce test se différencie de $\sqsubseteq_{\mathscr{A}(P)}$ du fait que les deux propriétés de tableaux comparées ne sont pas définies sur les mêmes partitions. En conséquence, pour une propriété de tranche $\psi'_q \in \Psi'$, il n'existe pas forcément une propriété $\psi_q \in \Psi$ définie

sur la même tranche φ'_q. De fait, l'opérateur recherche une tranche φ_p plus large (*i.e.* comprenant plus de cases) que φ'_q, telle que *tout ce qui est vrai sur la tranche* φ'_q*, l'est également dans* φ_p.

On peut modéliser ce raisonnement par l'arbre de dérivation suivant :

$$\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\varphi'_q}^{H_2} \qquad \dfrac{\sqsubseteq^{\pi}_{zone}}{\vdash_{\text{NJ}} \varphi'_q \Rightarrow \varphi_p}}{\vdash_{\text{NJ}} \varphi_p} \qquad \vdash_{\text{NJ}} \overbrace{\varphi_p \Rightarrow \psi_p}^{H_1}}{\vdash_{\text{NJ}} \psi_p}\Rightarrow_e \qquad \dfrac{\sqsubseteq^{\pi}_{zone}}{\vdash_{\text{NJ}} \psi_p \Rightarrow \psi'_q}}{\vdash_{\text{NJ}} \psi'_q}\Rightarrow_e}{\vdash_{\text{NJ}} \varphi'_q \Rightarrow \psi'_q}\Rightarrow_i (H_2)}{\vdash_{\text{NJ}} (\varphi_p \Rightarrow \psi_p) \Rightarrow (\varphi'_q \Rightarrow \psi'_q)}\Rightarrow_i (H_1)$$

Deux implications ne sont pas démontrées dans cet arbre de dérivation, et sont laissées à la charge de $\sqsubseteq^{\pi}_{zone}$: $\varphi'_q \Rightarrow \varphi_p$ et $\psi_p \Rightarrow \psi'_q$. Si le résultat de $\Psi \sqsubseteq_{\mathscr{A}_{gen}} \Psi'$ est positif, ces implications sont vraies et donc démontrables par $\sqsubseteq^{\pi}_{zone}$, (*cf* définition 4.2.1).

Une question subsiste : une telle tranche φ_p existe-t-elle pour toute tranche φ'_q ? La réponse est oui car $P \sqsubseteq_{part} P'$ (*cf* définition 4.1.6). Comment la trouver ? Il suffit de parcourir l'ensemble des tranches de P'. De plus, une telle tranche φ_p est unique pour chaque φ'_q car par définition les tranches d'une partition ne s'intersectent pas entre elles.

Le patron de preuve permettant d'instrumenter ce test d'inclusion générique est donné en figure 4.13.

Grâce à l'instrumentation de ce test d'inclusion, nous pouvons donc générer une preuve du triplet $\{\Psi\} i := e \{\Psi'\}$ en modifiant le calcul de pré-condition, de manière à ce qu'il produise des propriétés de tableaux exprimées dans une partition P'' plus fine que P (la partition fixée pour l'analyse). Cela présente deux avantages : (1) de ne pas avoir à instrumenter le code d'une fonction complexe comme ExtractProperty ; (2) de ne pas avoir à prouver des formules d'arithmétique n'étant pas dans les domaines abstraits employés par l'analyse (comme prouver que P est une partition par exemple, *cf.* section 4.2.2.3).

4.2.4 Conclusion sur la justification de l'analyse des affectations d'indices de tableaux

Nous avons proposé dans cette section deux instrumentations de l'analyse des affectations d'indices de tableaux. Le contenu des tableaux est spécifié selon une partition définie en amont, en fonction des indices de tableaux. La problématique posée par cette analyse vient du fait que la modification d'un indice modifie également l'ensemble des propriétés calculées pour chaque tranche de la partition. En effet, après l'affectation $i := i + 1$, la case désignée par la tranche $\ell = i$ n'est plus la même. Il en découle que la propriété qui était associée à cette tranche n'est plus appropriée et doit être mise à jour en fonction de l'information que l'analyse possède sur la tranche couvrant la case indicée par $i+1$. Pour cela, l'analyse utilise un calcul de meilleure information (ExtractProperty) en une tranche donnée n'appartenant pas forcément à la partition P, qui lui permet

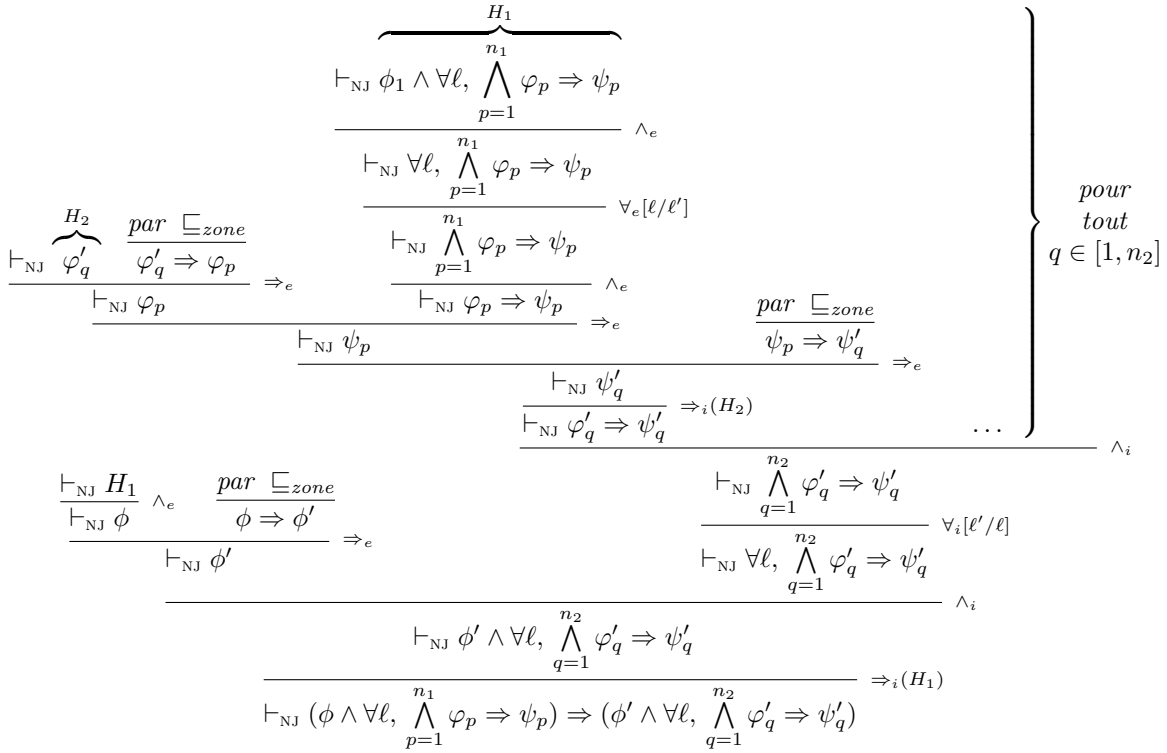


Fig. 4.13 – Patron de preuve pour la justification de $\sqsubseteq_{\mathscr{A}_{gen}}$

de calculer une post-condition Ψ' où les propriétés de tranche sont mises à jour. Du point de vue de la justification, le problème vient du fait que l'application du calcul de plus faible pré-condition sur cette post-condition Ψ' la transforme en une propriété de tableaux exprimée dans une partition différente. Le test d'inclusion instrumenté ne peut donc être utilisé pour prouver le sous-but retourné par *wp*, car l'implication que nous sommes conduits à prouver comporte deux formules n'appartenant pas au même domaine (*cf.* chapitre précédent, section 3.2.4.1).

La première solution que nous avons proposée, consiste à instrumenter le code de la fonction ExtractProperty utilisée par l'analyse pour mettre à jour les propriétés de tranche. L'instrumentation consiste alors à associer à chaque étape de cette fonction un patron de preuve spécifique. Cette solution est très *ad hoc* et nous a pris beaucoup de temps pour être réalisée car il fallu comprendre dans les moindres détails la fonction ExtractProperty. De plus, en suivant cette solution, il est nécessaire de prouver que P est une partition ce qui n'est pas faisable à l'aide des seuls opérateurs ensemblistes du domaine abstrait des propriétés de tableaux. Enkidu ne vérifie jamais que les partitions qui lui sont fournies par la pré-analyse (*cf.* section 4.1.2) sont bien des ensembles couvrant l'intégralité des tableaux. Il faut donc fournir une stratégie permettant de prouver que chaque ensemble de tranches utilisé par Enkidu est une partition. Nous avons pour cela utilisé la tactique `omega` du système Coq, qui rend la vérification des certificats peu efficace (*cf.* section 4.5)

L'autre solution que nous avons proposé, s'appuie sur une modification du calcul de

plus faible pré-condition et du test d'inclusion. Ces modifications sont très raisonnables et permettent de justifier l'analyse sans développer des patrons de preuve pour les fonctions utilisées par l'analyse. En effet, un seul patron de preuve est utilisé et a l'avantage d'être celui associé à l'un des opérateurs ensemblistes du domaine abstrait. Ceci va dans le sens de la méthodologie présentée au chapitre précédent : utiliser des preuves réflexives pour formaliser la sémantique axiomatique des programmes (garantissant la validité de wp), puis instrumenter les opérateurs du domaine abstrait pour prouver les obligations de preuve retournées par wp. Le test d'inclusion générique [GMT08] utilisé pour la justification de cette fonction de transfert sera utilisé pour justifier le reste de l'analyse au détriment de $\sqsubseteq_{\mathscr{A}(P)}$.

4.3 Modification des tableaux

Cette section présente la fonction de transfert permettant à l'analyse de calculer l'effet d'une affectation de tableau sur une propriété de tableaux Ψ, à un point de contrôle q du programme p. Il en résulte une propriété de tableaux Ψ' associée au point de contrôle q', si l'affectation lie q à q' dans le système de transitions du programme p. Du point de vue de la certification des propriétés de tableaux calculées par cette fonction de transfert, notre stratégie consiste toujours à appliquer un calcul de plus faible pré-condition. Encore une fois, le calcul de plus faible pré-condition ne retourne pas des formules appartenant au domaine abstrait $\mathscr{A}(P)$. La substitution effectuée par wp introduit dans la plus faible pré-condition des cases de tableaux indicées directement par les variables d'indice du programme et non par ℓ. Cependant, plutôt que d'instrumenter le code de la fonction de transfert pour en justifier les résultats, nous proposons une nouvelle fois une stratégie plus maligne et plus pratique qui consiste à utiliser le test d'inclusion entre les propriétés de tableaux. Pour cela, il nous faut traduire les formules retournées par le calcul de plus faible pré-condition en des propriétés de tableaux et justifier cette traduction par un arbre de dérivation. Cette traduction est réalisée par l'analyse et lui permet d'exploiter les expressions en partie droite des affectations (dans le cas où celles-ci contiennent des cases de tableaux) en les traduisant par des propriétés de tableaux. Ainsi, pour justifier l'analyse concernant l'affectation des tableaux, il nous suffit de développer une stratégie permettant de déduire les formules retournées par wp à partir de propriétés de tableaux équivalentes. La suite de la justification est gérée par $\sqsubseteq^{\pi}_{\mathscr{A}_{gen}}$.

4.3.1 Interprétation abstraite de l'affectation d'un tableau

Enkidu n'autorise pas les tableaux de tableaux. De ce fait, nous considérons en toute généralité qu'une affectation de tableau est de la forme $A[\langle expression \rangle] := \langle expression \rangle$ (par exemple $A[i + c] := A[j + d]$). Soient une affectation de cette forme, reliant le point de contrôle q à q' du programme p et $\Psi \in \mathscr{A}(P)$ la propriété de tableaux calculée par Enkidu en q. Du fait que le langage des *expressions* diffère de celui des propriétés de tableaux, l'analyse doit traduire l'expression en partie droite (que nous appellerons e) dans $\mathscr{A}(P)$ afin de pouvoir exploiter l'information que e contient (au moyen des opérateurs définis sur le domaine abstrait $\mathscr{A}(P)$). De même, la case du tableau affectée (nous prendrons pour exemple $A[i + c]$ où c est une constante et i une variable d'indice)

doit être traduite en une tranche afin d'être integrée à la propriété de tableaux Ψ' calculée.

4.3.1.1 Traduction de l'affectation vers le domaine abstrait des propriétés de tableaux.

Nous souhaitons exprimer les informations contenues par l'expression e dans les domaines utilisés par l'analyse et savoir à quelle tranche de P ces informations doivent être affectées. La tranche en question est celle qui couvre la case $A[i+c]$.

Traduction de l'expression affectée. L'expression e, si elle contient une case de tableau, induit une tranche $\varphi_e \in \mathscr{T}$. De cette nouvelle tranche est calculée une propriété $\psi_e \in \mathscr{P}$, au moyen du calcul de meilleure information appliqué à Ψ et φ_e (*cf.* section 4.2.2.1) :

$$\psi_e = \mathsf{ExtractProperty}(\Psi, \varphi_e).$$

Exemple 4.3.1. *Soit l'affectation* $(A[i+c] := A[j+d])$. *L'expression* e *est ici* $A[j+d]$ *et dénote la case indicée* $j+d$ *du tableau* A. *Cette expression induit la tranche* $\varphi_{A[j+d]} \stackrel{def}{=} (\ell = j+d)$. *La propriété de tranche* $\psi_{A[j+d]}$, *sera calculée par* $\mathsf{ExtractProperty}(\Psi_q, \varphi_{A[j+d]})$ *et sera ensuite associée à la tranche qui couvre* $A[i+c]$ *dans la propriété de tableaux* Ψ'.

Si e contient n cases de tableaux, l'analyse construit n propriétés de tranche où chacune est traitée comme dans le cas précédent. Chacune de ces propriétés de tranche sera ensuite associée à la tranche qui la couvre dans Ψ.

Si e ne contient que des variables scalaires, alors il n'y a pas de traduction à effectuer. L'analyse se contente d'exprimer l'égalité entre $A[i+c]$ et e.

Traduction de la case de tableau modifiée. La case de tableau modifiée $A[i+c]$ induit elle aussi une *tranche*, $\ell = i+c$, qui, de par la construction des partitions utilisées par Enkidu, existe déjà dans P (*cf.* section 4.1.2). La propriété associée à cette tranche devra donc être modifiée, en lui attribuant la propriété de tranche calculée en φ_e.

Exemple 4.3.2. *Soit l'affectation* $(A[i+c] := A[j+d])$. *La variable* $A[i+c]$ *dénote la case* $i+c$ *du tableau* A *et induit la tranche* $\varphi_{A[i+c]} \stackrel{def}{=} (\ell = i+c)$ *qui appartient déjà à* P.

La suite de cette section détaille comment l'analyse calcule la propriété qui correspond à $\varphi_{A[i+c]}$, après l'affectation $(A[i+c] = e)$, pour calculer l'assertion Ψ' au point de contrôle q'.

4.3.1.2 Fonction de transfert de l'affectation de tableau

Nous décrivons dans ce qui suit comment l'analyseur Enkidu capte l'effet de l'affectation d'un tableau pour calculer une propriété de tableaux au point de contrôle qui suit cette affectation. Nous présentons l'algorithme qui effectue ce traitement en figure

4.14 pour une tranche φ_k de la partition P. L'algorithme général consiste à appliquer ce traitement à chaque tranche de P. De par la construction des partitions utilisées par Enkidu, nous savons que la tranche $\varphi_{A[i+c]}$ induite de la case de tableau modifiée appartient à P, *i.e.* il existe dans P une tranche φ_k représentant exactement la case indicée par $i + c$. Il se peut cependant qu'une certaine propriété de tranche $\psi_{k'}$ formalise une information sur le contenu de la case $A[i + c]$, sans pour autant que la tranche $\varphi_{k'}$ soit définie par $\ell = i + c$

Exemple 4.3.3. *Soit l'affectation $A[i] := A[j]$. On considère ici la propriété de tranche $i + 1 \leq \ell \Rightarrow A[\ell - 1] \leq A[\ell]$. Dans le cas où ℓ est égal à $i + 1$, ce qui est possible concernant cette tranche, la case désignée par $A[\ell - 1]$ est $A[i]$. La propriété de tranche doit prendre en compte cette modification de $A[i]$, car il se peut que la propriété de tranche $i + 1 \leq \ell \Rightarrow A[\ell - 1] \leq A[\ell]$ ne soit plus vraie au point de contrôle suivant, selon la valeur de $A[j]$.*

Pour cette raison notamment, l'analyse des affectations de tableaux n'est pas triviale et doit considérer les *effets par translation* de la modification d'une case de tableau sur la propriété de tableaux au point de contrôle courant. La fonction de tranfert doit donc construire la nouvelle propriété de tableaux au point de contrôle q', en gérant tous ces cas que nous énumérons ci-dessous pour une tranche φ_k de P :

1. $\varphi_k \stackrel{def}{=} \ell = i + c$. Nous savons qu'il existe une unique tranche de P qui vérifie ce cas. Enkidu effectue une *affectation forte* et associe alors la *propriété de tranche* ψ_e à φ_k ($\psi'_k = \psi_e$ autrement dit).
2. La tranche $\varphi_k \neq \ell = i + c$, mais ψ_k formalise tout de même le contenu de $A[i + c]$, *i.e.* $A[\ell + z] \in \psi_k$, tel que $\ell + z = i + c$ soit satisfiable (selon la définition de φ_k). Dans ce cas, Enkidu effectue une *affectation faible* et associe la *propriété de tranche* $\psi_e \sqcup_{zone} \psi_k$ à φ_k dans Ψ'.
3. La tranche φ_k n'est pas *concernée* par l'affectation. C'est le cas trivial, où Enkidu retourne une propriété de tranche identique à celle au point de contrôle q, ψ_k en l'occurrence ($\psi'_k = \psi_k$ autrement dit).

$$\Psi' = \mathsf{sem}^{\sharp}_{\mathsf{a}}(\Psi, A[i + c] := e) = \begin{cases} (1) & si\ (\varphi_k \stackrel{def}{=} \ell = i + c)\ alors\ (\varphi_k \Rightarrow \psi_e) \\ (2) & si\ \exists z \in \mathbb{Z}, A[\ell + z] \in \psi_k \wedge \varphi_k \sqcap_{zone} (\ell = i + c - z) \neq \bot_{zone} \\ & alors\ (\varphi_k \Rightarrow \psi_k \sqcup_{zone} \psi_e) \\ (3) & (\varphi_k \Rightarrow \psi_k)\ sinon \end{cases}$$

Fig. 4.14 – Algorithme de la fonction de transfert de la modification d'un tableau

De nombreux détails techniques sont volontairement omis concernant cette fonction de transfert et peuvent être trouvés dans [HP08], car ils ne sont pas pertinents pour la justification de l'analyse des affectations de tableaux. En partant du principe qu'il est plus facile de vérifier un résultat que de le calculer, il paraît important de ne pas avoir à rentrer dans les détails de l'analyse pour en apporter une justification. Ceci dans le but de rendre l'instrumentation la plus simple possible.

4.3.2 Justification de l'analyse de l'affectation de tableau

Nous cherchons ici à définir une stratégie de preuve, la plus simple possible, pour justifier la fonction de transfert de l'affectation de tableaux. Comme mentionné au chapitre précédent (*cf.* section 3.2.4.1), si cette stratégie de preuve peut seulement utiliser l'instrumentation du test d'inclusion du domaine abstrait considéré, le travail à fournir pour obtenir des justifications formelles se résume à presque rien. Pour cela, il est nécessaire que le calcul de plus faible pré-condition préserve la post-condition dans le domaine abstrait *i.e.* $wp(A[i+c] := e, \Psi') \in \mathscr{A}(P)$. Malheureusement, ce n'est pas le cas : le calcul de plus faible pré-condition pour l'affectation des tableaux introduit des égalités, des non-égalités et, par substitution de la variable modifiée, des expressions de cases de tableaux non indicées par la variable logique ℓ . Cependant, il serait tout de même appréciable de ne pas avoir à instrumenter cette fonction de transfert par un patron de preuve *ad'hoc*, en suivant l'algorithme de cette fonction. Comme dans le cas de la modification des indices de tableaux, il est possible d'utiliser l'instrumentation du test d'inclusion plus générique, proposé dans [GMT08], à condition d'avoir justifié la traduction des expressions affectées vers le domaine des propriétés de tableaux.

Nous développons la problématique posée par la justification de la fonction de transfert de l'affectation de tableaux et la solution proposée.

4.3.2.1 Principe de la justification

Il s'agit ici de prouver la correction des triplets de la forme $\{\Psi\}A[i+c] := e\{\Psi'\}$, où Ψ est la propriété de tableaux au point de contrôle q, Ψ' la propriété de tableaux au point de contrôle q' et $A[i+c] := e$ l'étiquette de la transition qui va de q à q'.

Prouver de tels triplets de Hoare est facilité par le calcul de plus faible pré-condition défini en section 2.1.2.2. Si ce calcul de plus faible pré-condition ne préserve pas la post-condition dans le domaine des propriétés de tableaux, il conduit à prouver des sous-buts qui sont soit triviaux, soit très proches de la pré-condition Ψ. En effet, du fait que wp explicite les alias de tableaux, la preuve à construire consiste simplement à prouver à partir de Ψ soit des propriétés de tranche portant sur une unique case de tableau (si e est de la forme $A[j+d]$) soit une variable scalaire (si e est une variable scalaire). Nous nous attardons tout particulièrement sur le cas où l'expression e affectée est une variable de tableaux de la forme $A[j+d]$. Les cas où e est une expression contenant une variable scalaire se traite similairement, mais de façon encore plus directe.

Utilisation du wp pour les affectations de tableau. L'application du calcul de plus faible pré-condition sur une affectation de la forme $A[\langle expression \rangle] := A[\langle expression \rangle]$ (par exemple $A[i+c] := A[j+d]$) modifie dans la post-condition Ψ' toute occurrence de A par la fonction $(\lambda\ell.\ \texttt{if}\ \ell = i+c\ \texttt{then}\ A[j+d]\ \texttt{else}\ A[i+c])$. Pour que ce traitement des affectations de tableaux ait du sens, il faut se rappeler que les tableaux seront modélisés par des fonctions des entiers vers les entiers dans le système Coq (qui nous servira de vérificateur de preuves). Cette substitution des occurrences de A, par la fonction que nous venons de donner, est systématiquement suivie de la tactique `case`, qui permet de considérer les deux cas possibles du test `if` $\ell = i+c$. Il s'ensuit deux sous-buts à montrer pour chaque occurrence du tableaux A dans Ψ', l'une où le test est vérifié, l'autre où le test est faux. Pour simplifier la représentation de ce traitement dans les patrons de preuve que nous allons définir pour justifier l'analyse de l'affectation de

tableaux, nous regrouperons en un pas de déduction les substitutions de A dans Ψ et la décomposition des cas. Pour exemple, la plus faible pré-condition pour une affectation $A[i+c] := A[j+d]$ et la post-condition $A[j] = k$ est la formule ci-dessous, qu'il faudra prouver à partir de la pré-condition Ψ :

$$\left(\begin{array}{l} (j = i + c) \Rightarrow A[j+d] = k \\ \wedge \\ (j \neq i + c) \Rightarrow A[j] = k \end{array}\right)$$

La première implication de la conjonction correspond au cas où il y a un alias entre $i + c$ et j. La seconde, au cas où la case $A[j]$ n'est pas affectée par la modification de $A[i + c]$.

Dans le cadre de l'analyseur Enkidu, les preuves de ces deux sous-buts sont relativement simples car les propriétés de tableaux fournissent toutes les informations nécessaires pour déterminer si $j = i + c$ est satisfiable ou non. De par la construction des partitions sur lesquelles sont définies les propriétés de tableaux, il existe obligatoirement une unique tranche qui précise le contenu de la case de tableau affectée. En conséquence l'un des deux sous-buts aura un ensemble de prémisses insatisfiables, et l'autre sera prouvable directement à partir de l'une des propriétés de tranche de Ψ.

Exemple 4.3.4. *Nous faisons l'hypothèse dans cet exemple que l'expression affectée est une case de tableau et que l'analyse est précise sur cette case. Nous nous intéressons au contenu de la case $A[j]$ après une modification de la case $A[i]$. Le contenu de la case $A[j]$ est spécifié par la propriété de tranche $\ell = j \Rightarrow \psi'(A[\ell])$. Une chose importante, est que dans l'analyseur* Enkidu *les tranches sont toujours définies sur l'ensemble des variables d'indices. Ainsi, le contenu de la case $A[j]$ sera en réalité représenté par les propriétés associées aux trois tranches $(\ell = j \wedge \ell < i), (\ell = j \wedge \ell = i), (\ell = j \wedge \ell > i)$. Cette accommodation simplifie grandement, à la fois l'analyse et la justification de l'analyse.*

Soit le triplet $\{\Psi\}A[i] := A[j+d]\{\forall \ell, (\ell < i \wedge \ell = j) \Rightarrow A[\ell] = k\}$, le calcul de plus faible pré-condition nous mène à prouver la formule suivante :

$$\Psi \Rightarrow \left(\begin{array}{l} \ell = i \Rightarrow ((\ell < i \wedge \ell = j) \Rightarrow A[j+d] = k) \\ \wedge \\ \ell \neq i \Rightarrow ((\ell < i \wedge \ell = j) \Rightarrow A[\ell] = k) \end{array}\right)$$

Le premier sous-but contient les prémisses $\ell = i$ et $\ell < i$ qui sont antinomiques et, par conséquent, ce sous-but est trivialement prouvable. Le second contient en prémisse une non-égalité qui n'est pas contradictoire avec la tranche $(\ell < i \wedge \ell = j)$. Cependant cette non-égalité $\ell \neq i$ spécifie le fait qu'il n'y a pas d'alias possible entre $A[i]$ et $A[j]$ et donc que cette propriété de tranche est identique à celle contenue dans Ψ. La preuve est donc également triviale.

On considère maintenant le cas où la post-condition est affectée par la modification de $A[i]$. Soit le triplet $\{\Psi\}A[i] := A[j+d]\{\forall \ell, (\ell = i \wedge \ell = j) \Rightarrow A[\ell] = k\}$, le calcul de plus faible pré-condition nous mène à prouver la formule suivante :

$$\Psi \Rightarrow \left(\begin{array}{l} \ell = i \Rightarrow ((\ell = i \wedge \ell = j) \Rightarrow A[j+d] = k) \\ \wedge \\ \ell \neq i \Rightarrow ((\ell = i \wedge \ell = j) \Rightarrow A[\ell] = k) \end{array}\right)$$

Ici, c'est le second sous-but qui est trivialement prouvable, par la contradiction entre $\ell \neq i$ et $(\ell = i \wedge \ell = j)$. Le premier, quant à lui, est prouvable par la pré-condition Ψ car nous avons supposé que Ψ était précise sur la tranche $\ell = j + d$ $(\ell = j + d \in P)$.

Si l'on souhaite être davantage général et considérer que $\ell = j + d \notin P$, alors l'information sur le contenu de la case $A[j + d]$, permettant de définir la post-condition, a été calculée par la fonction ExtractProperty*$(\Psi, (\ell = j + d))$. Comme nous l'avons vu en section 4.2.1.2, nous avons à notre disposition une stratégie permettant de justifier les résultats de cette fonction de calcul de meilleure information à l'aide de $\sqsubseteq_{\mathscr{A}_{gen}}$. Pour achever la preuve, il suffit donc de greffer la justification apportée par l'exécution de $\sqsubseteq^{\pi}_{gen}$ au premier sous-but. Les détails des preuves de ces exemples seront donnés dans la section suivante.*

Si en pratique l'analyseur Enkidu rencontre des cas plus simples à traiter qu'en théorie, c'est parce que la stratégie développée dans Enkidu évite de construire des propriétés de tableaux contenant des alias, afin de se faciliter la tâche. La pré-analyse qui permet de déterminer la partition est à l'origine de cette facilité : les tranches sont définies de sorte à toujours être précises sur les cases de tableaux affectées. Cependant, si l'analyse évite les alias classiques entre $A[i]$ et $A[j]$ quand $i = j$ par exemple, nous avons vu en section 4.3.1.2, détaillant la fonction de transfert de l'affectation de tableaux, qu'il est possible qu'une propriété de tranche contiennent des informations sur le contenu d'une case n'appartenant pas à sa tranche respective. Par exemple, si l'on considère la tranche $\ell = i$, la propriété $A[\ell] = A[\ell + 1]$ exprime également une information sur le contenu de la case indicée par $i+1$. Nous détaillons cela dans la section suivante à travers les trois cas possibles distingués par l'analyse.

4.3.2.2 Détails de la justification

Une chose importante à remarquer dans le cas où l'affectation est de la forme $A[i + c] := A[j+d]$ est que la tranche $\ell = j+d$ ne désigne qu'une seule case du tableau A. Cela a pour conséquence qu'il existe une unique tranche de P, φ_p, telle que $(\ell = j+d) \sqsubseteq_{zone} \varphi_p$. La version intrumentée du test d'inclusion générique (*cf.* section 4.2.3.2) peut donc être employée pour prouver la formule $\Psi \Rightarrow \forall \ell, (\ell = j+d) \Rightarrow \psi'(A[\ell])$. Les patrons de preuve que nous allons donner consistent donc à faire le lien entre la formule retournée par wp et une formule qui puisse être prouvée par $\sqsubseteq^{\pi}_{gen}$. Ces patrons de preuve formalisent et justifient la traduction de l'expression $\psi'_k(A[j + d])$ en la propriété de tranche $(\ell = j + d) \Rightarrow \psi'_k(A[\ell])$.

Une seconde remarque concerne l'algorithme de la fonction de transfert utilisée par Enkidu : à partir d'une propriété de tableaux Ψ, il n'y a que trois façons de calculer les propriétés de tranche de Ψ'. Nous détaillons ici l'instrumentation de cette fonction de transfert en distinguant ces trois cas.

Pour des raisons de clarté nous ne considérons qu'une seule propriété de tranche de Ψ' en post-condition. Le cas général consiste simplement à traiter chaque propriété de tranche selon le cas dans lequel elle se trouve. La validité des propriétés de tranche de Ψ' se prouvent indépendamment les unes des autres (car elles sont liées par le connecteur conjonctif) au moyen d'applications de la règle d'inférence $\wedge_i$.

Justification du cas 1. Dans le cas numéro un de l'algorithme de la fonction de transfert, où la tranche mise à jour désigne directement la case modifiée, il ne peut y avoir d'alias car la propriété de tranche *parle* explicitement de la case modifiée. Ainsi, l'explicitation des alias par le calcul de plus faible pré-condition conduit à prouver : (1) en branche gauche que la propriété de tranche ψ'_k vient de l'information que contient la Ψ sur la case indicée par $j+d$ du tableau $A[\,]$. L'égalité introduite par *wp* ne sert à rien dans cette branche de la preuve. La propriété de tranche ψ'_k est calculée par Enkidu grâce à la fonction $\mathsf{ExtractProperty}(\Psi, \ell = j + d)$, que nous justifions par l'instrumentation du test d'inclusion $\sqsubseteq_{\mathscr{A}_{gen}}$. Pour cela, il nous suffit de déduire $\psi'_k(A[j + d])$ à partir de $\forall \ell, \ell = j + d \Rightarrow \psi'_k(A[\ell])$; (2) en branche droite une contradiction, avec les deux hypothèses $\ell \neq i+c$ (venant du calcul de plus faible pré-condition) et $\ell = i+c$ désignant la tranche à laquelle on s'intéresse. C'est précisément ici que le calcul de plus faible pré-condition nous facilite la tâche, en introduisant cette contradiction :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{j+d=j+d}\,{=_{refl}} \quad \dfrac{\dfrac{\textit{par } \sqsubseteq^{\pi}_{\mathscr{A}_{gen}}}{\vdash_{\text{NJ}} \forall \ell, \ell = j+d \Rightarrow \psi'_k(A[\ell])}}{\vdash_{\text{NJ}} j+d = j+d \Rightarrow \psi'_k(A[j+d])}\,{\forall_e[\ell/j+d]}}{\vdash_{\text{NJ}} \psi(A[j+d])}\,{\Rightarrow_e}}{\vdash_{\text{NJ}} \ell = i+c \Rightarrow \psi'_k(A[j+d])}\,{\Rightarrow_i}}{\vdash_{\text{NJ}} \ell = i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[j+d]))}\,{\Rightarrow_i} \qquad \begin{array}{l}\textit{par la contradiction}\\ \textit{des hypothèses}\\ \ell \neq i+c \wedge \ell = i+c\end{array}}{\vdash_{\text{NJ}} \left(\begin{array}{l}\ell = i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[j+d]))\\ \wedge\\ \ell \neq i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[\ell+z]))\end{array}\right)}\,{\wedge_i}$$

$$\dfrac{\vdash_{\text{NJ}} \forall \ell, \left(\begin{array}{l}\ell = i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[j+d]))\\ \wedge\\ \ell \neq i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[\ell+z]))\end{array}\right)}{\vdash_{\text{NJ}} \Psi \Rightarrow \forall \ell, \left(\begin{array}{l}\ell = i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[j+d]))\\ \wedge\\ \ell \neq i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[\ell+z]))\end{array}\right)}\,{\forall_i}$$

$$\dfrac{\vdash_{\text{NJ}} \Psi \Rightarrow \forall \ell, \left(\begin{array}{l}\ell = i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[j+d]))\\ \wedge\\ \ell \neq i+c \Rightarrow (\ell = i+c \Rightarrow \psi'_k(A[\ell+z]))\end{array}\right)}{\vdash_{\text{NJ}} \{\Psi\}A[i+c] := A[j+d]\{\forall \ell, \ell = i+c \Rightarrow \psi'_k(A[\ell])\}}\,{\Rightarrow_i}\,{sem}$$

Dans le patron de preuve ci-dessus, il reste donc à prouver $\forall \ell, \ell = j + d \Rightarrow \psi(A[\ell])$, et nous affirmons que cette preuve peut être obtenue par $\sqsubseteq^{\pi}_{\mathscr{A}_{gen}}$. En effet, cette propriété de tranche porte sur une unique case du tableau A, $A[j + d]$ en l'occurrence, et il existe donc une tranche $\varphi_k \in P$ telle que $\ell = j + d \sqsubseteq_{zone} \varphi_k$.

Exemple 4.3.5. *Nous considérons dans cet exemple l'affectation $A[i] := A[j + d]$ et la propriété de tranche en post-condition $\forall \ell, (\ell = i \wedge \ell = j) \Rightarrow (A[\ell] = k)$. Il nous faut donc prouver le triplet $\{\Psi\}A[i] := A[j + d]\{\forall \ell, (\ell = i \wedge \ell = j) \Rightarrow (A[\ell] = k)\}$, pour justifier formellement la correction de l'analyse. L'application du calcul de plus faible pré-condition nous conduit à prouver la formule ci-dessous, où l'alias possible entre $A[\ell]$ (issue de la post-condition) et $A[i]$ (la variable modifiée) est explicité par une conjonction d'implications :*

$$\dfrac{\vdash_{\text{NJ}} \Psi \Rightarrow \forall \ell, \left(\begin{array}{l} \ell = i \Rightarrow (\ell = i \wedge \ell = j \Rightarrow (A[j+d] = k)) \\ \wedge \\ \ell \neq i \Rightarrow (\ell = i \wedge \ell = j \Rightarrow (A[\ell] = k)) \end{array} \right)}{\vdash_{\text{NJ}} \{\Psi\} A[i] := A[j+d] \{\forall \ell, (\ell = i \wedge \ell = j) \Rightarrow A[\ell] = k\}}\ sem$$

Cela nous conduit à prouver en branche droite, que si $\ell \neq i$ alors $(\ell = i \wedge \ell = j) \Rightarrow (A[\ell] = k)$ était vraie au point de contrôle q. Ceci est trivialement prouvable, sans exploiter la propriété de tableaux calculée au point de contrôle q, en mettant en avant une antinomie entre les prémisses de cette double implication, $\ell \neq i$ et $(\ell = i \wedge \ell = j)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\ell = i \wedge \ell = j}^{H_3}}{\vdash_{\text{NJ}} \ell = i}\wedge_e \quad \dfrac{\vdash_{\text{NJ}} \overbrace{\ell \neq i}^{H_2}}{\vdash_{\text{NJ}} \ell = i \Rightarrow \bot}\neg_{def}}{\vdash_{\text{NJ}} \bot}\Rightarrow_e}{\vdash_{\text{NJ}} A[\ell] = k}\bot_e}{\vdash_{\text{NJ}} (\ell = i \wedge \ell = j) \Rightarrow (A[\ell] = k)}\Rightarrow_i (H_3)}{\vdash_{\text{NJ}} \ell \neq i \Rightarrow (\ell = i \wedge \ell = j \Rightarrow (A[\ell] = k))}\Rightarrow_i (H_2)$$

En branche gauche, lorsque $\ell = i$, il nous faut prouver la formule $(\ell = i \wedge \ell = j \Rightarrow (A[j+d] = k)$ obtenue par substitution de $A[\ell]$ par $A[j+d]$ dans la post-condition. Cette preuve est obtenue par instanciation du patron de preuve permettant de justifier la traduction de cette formule dans le domaine des propriétés de tableaux, puis par le test d'inclusion instrumenté entre Ψ et $\forall \ell, (\ell = j + d) \Rightarrow A[\ell] = k$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{j+d=j+d}=_{refl} \quad \dfrac{\dfrac{}{\vdash_{\text{NJ}} \forall \ell, \ell = j+d \Rightarrow (A[\ell] = k)}\ par\ \sqsubseteq^{\pi}_{\mathscr{A}_{gen}}}{\vdash_{\text{NJ}} (j+d = j+d) \Rightarrow (A[j+d] = k)}\forall_e[\ell/j+d]}{\vdash_{\text{NJ}} A[j+d] = k}\Rightarrow_e}{\vdash_{\text{NJ}} (\ell = i \wedge \ell = j) \Rightarrow (A[j+d] = k)}\Rightarrow_i (H_5)}{\vdash_{\text{NJ}} \ell = i \Rightarrow (\ell = i \wedge \ell = j \Rightarrow (A[j+d] = k))}\Rightarrow_i (H_4)$$

Prouver la validité de $\forall \ell, (\ell = j + d) \Rightarrow (A[\ell] = k)$ revient à justifier le calcul de meilleure information sur la tranche $\ell = j + d$ dans la propriété de tableaux $\Psi \in \mathscr{A}(P)$. Nous avons vu que, pour cela, il était possible d'utiliser le test d'inclusion générique instrumenté du fait que pour la tranche $\ell = j + d$, il existe obligatoirement dans P une tranche φ_k telle que $\ell = j + d \sqsubseteq_{zone} \varphi_k$.

Justification du cas 2. Le cas numéro deux est plus complexe, car il traite le cas d'un alias par translation. C'est à dire que la propriété de tranche en post-condtion contient l'expression $A[\ell + z]$ et $\ell + z = i + c$ est satisfiable. Malgré tout, la solution pour justifier le traitement de l'analyse des affectations de tableaux dans ce cas peut être résolue une fois de plus à l'aide de $\sqsubseteq^{\pi}_{\mathscr{A}_{gen}}$. En effet, si nous sommes dans le cas numéro deux de la fonction de transfert, la post-condition a été calculée au moyen de la fonction $\mathsf{ExtractProperty}(\Psi, (\ell = j + d))$, qui nous a rendu une justification de son calcul en la preuve de $\Psi \Rightarrow ((\ell = j+d) \Rightarrow \psi')$. Il nous suffit donc de conduire le patron de preuve pour atteindre une telle formule et exploiter la justification de $\mathsf{ExtractProperty}(\Psi, (\ell = j + d))$, qui provient du patron de preuve associé à $\sqsubseteq_{\mathscr{A}_{gen}}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{j+d=j+d}\,{=_{refl}} \quad \dfrac{\dfrac{\text{par } \sqsubseteq^{\pi}_{\mathscr{A}_{gen}}}{\vdash_{\text{NJ}} \forall \ell, \ell = j+d \Rightarrow \psi'_k(A[\ell])}}{\vdash_{\text{NJ}} j+d = j+d \Rightarrow \psi'_k(A[j+d])}\,{\forall_e[\ell/j+d]}}{\vdash_{\text{NJ}} \psi(A[j+d])}\,{\Rightarrow_e}}{\vdash_{\text{NJ}} \varphi_k \Rightarrow \psi'_k(A[j+d])}\,{\Rightarrow_i}}{\vdash_{\text{NJ}} \ell + z = i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[j+d]))}\,{\Rightarrow_i} \qquad \text{idem au cas 3}}{\vdash_{\text{NJ}} \left(\begin{array}{l} \ell + z = i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[j+d])) \\ \wedge \\ \ell + z \neq i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[\ell+z])) \end{array}\right)}\,{\wedge_i}$$

$$\dfrac{\vdash_{\text{NJ}} \forall \ell, \left(\begin{array}{l} \ell + z = i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[j+d])) \\ \wedge \\ \ell + z \neq i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[\ell+z])) \end{array}\right)}{\vdash_{\text{NJ}} \Psi \Rightarrow \forall \ell, \left(\begin{array}{l} \ell + z = i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[j+d])) \\ \wedge \\ \ell + z \neq i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[\ell+z])) \end{array}\right)}\,{\Rightarrow_i}$$

$$\dfrac{\vdash_{\text{NJ}} \Psi \Rightarrow \forall \ell, \left(\begin{array}{l} \ell + z = i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[j+d])) \\ \wedge \\ \ell + z \neq i + c \Rightarrow (\varphi_k \Rightarrow \psi'_k(A[\ell+z])) \end{array}\right)}{\vdash_{\text{NJ}} \{\Psi\} A[i+c] := A[j+d] \{\forall \ell, \varphi_k \Rightarrow \psi'_k(A[\ell+z])\}}\,{sem}$$

La branche gauche est similaire au cas numéro un et donc prouvée grâce à $\sqsubseteq^{\pi}_{\mathscr{A}_{gen}}$. Pour la branche droite, le cas où il n'y a pas d'alias (cela est explicité par la non-égalité $\ell + z \neq i + c$), la preuve est similaire au cas numéro trois, que nous développons immédiatement après l'exemple 4.3.6 :

Exemple 4.3.6. *Pour le cas numéro deux nous considérons la même affectation que dans l'exemple 4.3.5, soit $A[i] := A[j+d]$, mais en post-condition la propriété de tranche $\forall \ell, (\ell < i \wedge \ell = j) \Rightarrow (A[\ell+1] \leq k)$. Il nous faut donc prouver le triplet $\{\Psi\} A[i] := A[j+d] \{\forall \ell, (\ell < i \wedge \ell = j) \Rightarrow (A[\ell+1] \leq k)\}$, pour justifier formellement la correction de l'analyse. L'application du calcul de plus faible pré-condition nous conduit à prouver la formule ci-dessous, où l'alias possible entre $A[\ell+1]$ (issue de la post-condition) et $A[i]$ (la variable modifiée) est explicité par une conjonction d'implications :*

$$\dfrac{\vdash_{\text{NJ}} \Psi \Rightarrow \forall \ell, \left(\begin{array}{l} \ell + 1 = i \Rightarrow (\ell < i \wedge \ell = j \Rightarrow (A[j+d] \leq k)) \\ \wedge \\ \ell + 1 \neq i \Rightarrow (\ell < i \wedge \ell = j \Rightarrow (A[\ell+1] \leq k)) \end{array}\right)}{\vdash_{\text{NJ}} \{\Psi\} A[i] := A[j+d] \{\forall \ell, (\ell < i \wedge \ell = j) \Rightarrow A[\ell+1] \leq k\}}\,{sem}$$

Dans cette configuration le calcul de plus faible pré-condition n'introduit pas d'implication dont les prémisses sont contradictoires.

La branche droite, où $A[\ell+1]$ n'a pas été substituée par $A[j+d]$ car il n'y a pas d'alias entre $A[\ell+1]$ et la case modifiée $A[i]$, la preuve se construit en déduisant de la pré-condition Ψ (la propriété de tableaux au point de contrôle q) la propriété de tranche $(\ell < i \wedge \ell = j) \Rightarrow \psi$. Nous sommes ici dans le cas d'une affectation faible *où la tranche $\ell < i \wedge \ell = j$ n'est pas entièrement affectée par la modification de $A[i]$ (voir section 4.3.1.2) et $A[\ell+1] \leq k = \psi \sqcup_{zone} \psi_{A[j+d]}$. Par définition de l'opérateur $\sqcup_{zone}$, nous savons que $\psi \sqsubseteq_{zone} A[\ell+1] \leq k$. Et par conséquent la propriété de tranche $(\ell < i \wedge \ell) = j \Rightarrow (A[\ell+1] \leq k)$ peut être déduite de Ψ via l'instrumentation de $\sqsubseteq_{zone}$:*

$$\dfrac{\vdash_{\text{NJ}} \overbrace{\ell < i \wedge \ell = j}^{H_3} \qquad \dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\Psi}^{H}}{\vdash_{\text{NJ}} \forall \ell, \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k}\wedge_e}{\vdash_{\text{NJ}} \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k}\forall_e}{\vdash_{\text{NJ}} (\ell < i \wedge \ell = j) \Rightarrow \psi}\wedge_e}{\vdash_{\text{NJ}} \psi}\Rightarrow_e \qquad \dfrac{\textit{par } \sqsubseteq^{\pi}_{zone}}{\vdash_{\text{NJ}} \psi \Rightarrow A[\ell+1] \leq k}$$

$$\dfrac{\dfrac{\vdash_{\text{NJ}} A[\ell+1] \leq k}{\vdash_{\text{NJ}} (\ell < i \wedge \ell = j) \Rightarrow (A[\ell+1] \leq k)}\Rightarrow_i (H_3)}{\vdash_{\text{NJ}} \ell + 1 \neq i \Rightarrow ((\ell < i \wedge \ell = j) \Rightarrow (A[\ell+1] \leq k))}\Rightarrow_i (H_2)$$

La branche gauche consiste à prouver la formule $\ell + 1 = i \Rightarrow ((\ell < i \wedge \ell = j) \Rightarrow (A[j + d] \leq k))$*. De la même manière que dans la justification du cas numéro un de l'analyse, il suffit de justifier la traduction de* $A[j + d] \leq k$ *dans* $\mathscr{A}(P)$ *puis d'utiliser* $\sqsubseteq^{\pi}_{\mathscr{A}_{gen}}$ *pour achever la preuve.*

Justification du cas 3. Dans le cas numéro trois, la tranche considérée n'est aucunement modifiée par l'affectation du tableau $A[\,]$. La propriété de tranche ψ'_k est donc identique à celle au point de contrôle précédent, *i.e.* $\psi'_k = \psi_k$: La preuve se sépare en deux branches, dues aux deux implications générées par le calcul de plus faible précondition. La branche gauche correspond au cas où la tranche $\ell+z = i+c$ est insatisfiable avec la tranche φ_k. La branche droite correspond à la déduction de Ψ de la propriété de tranche $\varphi_k \Rightarrow \psi_k$.

$$\dfrac{\dfrac{\textit{par la contradiction des hypothèses } \ell + z = i + c \wedge \varphi_k \qquad \dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\Psi}^{H}}{\vdash_{\text{NJ}} \forall \ell, \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k}\wedge_e}{\vdash_{\text{NJ}} \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k}\forall_e}{\vdash_{\text{NJ}} \varphi_k \Rightarrow \psi_k(A[\ell+z])}\wedge_e}{\vdash_{\text{NJ}} \ell + z \neq i + c \Rightarrow (\varphi_k \Rightarrow \psi_k(A[\ell+z]))}\Rightarrow_i}{\vdash_{\text{NJ}} \left(\begin{array}{l} \ell + z = i + c \Rightarrow (\varphi \Rightarrow \psi_k(A[j+d])) \\ \wedge \\ \ell + z \neq i + c \Rightarrow (\varphi \Rightarrow \psi_k(A[\ell+z])) \end{array}\right)}\wedge_i$$

$$\dfrac{\dfrac{\vdash_{\text{NJ}} \forall \ell, \left(\begin{array}{l} \ell + z = i + c \Rightarrow (\varphi_k \Rightarrow \psi_k(A[j+d])) \\ \wedge \\ \ell + z \neq i + c \Rightarrow (\varphi_k \Rightarrow \psi_k(A[\ell+z])) \end{array}\right)}{\vdash_{\text{NJ}} \Psi \Rightarrow \forall \ell, \left(\begin{array}{l} \ell + z = i + c \Rightarrow (\varphi_k \Rightarrow \psi_k(A[j+d])) \\ \wedge \\ \ell + z \neq i + c \Rightarrow (\varphi_k \Rightarrow \psi_k(A[\ell+z])) \end{array}\right)}\forall_i}{\vdash_{\text{NJ}} \{\Psi\}A[i+c] := A[j+d]\{\forall \ell, \varphi_k \Rightarrow \psi_k(A[\ell+z])\}}\Rightarrow_i \textit{sem}$$

Exemple 4.3.7. *Pour illustrer la justification de ce dernier cas, nous considérons encore l'affectation* $A[i] := A[j+d]$ *mais avec la propriété de tranche* $\forall \ell, (\ell < i \wedge \ell = j) \Rightarrow (A[\ell+1] \leq k)$ *en post-condition. Il nous faut donc prouver le triplet* $\{\Psi\}A[i] := A[j+d]\{\forall \ell, (\ell > i \wedge \ell = j) \Rightarrow (A[\ell] \geq k)\}$ *pour justifier formellement la correction de l'analyse. L'application du calcul de plus faible pré-condition nous conduit à prouver la formule ci-dessous, où l'alias possible entre* $A[\ell]$ *(issue de la post-condition) et* $A[i]$ *(la variable modifiée) introduit une contradiction entre les prémisses de cette implication :*

$$\frac{\vdash_{\text{NJ}} \Psi \Rightarrow \forall \ell, \left(\begin{array}{l} \ell = i \Rightarrow (\ell > i \wedge \ell = j \Rightarrow (A[j+d] \geq k)) \\ \wedge \\ \ell \neq i \Rightarrow (\ell > i \wedge \ell = j \Rightarrow (A[\ell] \geq k)) \end{array} \right)}{\vdash_{\text{NJ}} \{\Psi\}A[i] := A[j+d]\{\forall \ell, (\ell > i \wedge \ell = j) \Rightarrow A[\ell+1] \geq k\}}\ sem$$

Ainsi, la branche gauche de l'arbre consiste à prouver une contradiction entre $\ell = i$ *et* $(\ell > i \wedge \ell = j)$ *de la même façon que pour la branche droite du cas numéro un.*

La branche droite justifie le fait que de q *à* q'*, la tranche* $(\ell > i \wedge \ell = j)$ *n'a pas été modifiée. En d'autres termes la propriété de tranche* $(\ell > i \wedge \ell = j) \Rightarrow (A[\ell] \geq k)$ *peut être déduite de la pré-condition* Ψ *:*

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\Psi}^{H}}{\vdash_{\text{NJ}} \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k}\wedge_e}{\vdash_{\text{NJ}} \forall \ell, \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi_k}\forall_e}{\vdash_{\text{NJ}} (\ell > i \wedge \ell = j) \Rightarrow (A[\ell] \geq k)}\wedge_e}{\vdash_{\text{NJ}} \ell \neq i \Rightarrow ((\ell > i \wedge \ell = j) \Rightarrow (A[\ell] \geq k))}\Rightarrow_i$$

4.3.3 Conclusion sur la justification de l'analyse des affectations de tableaux

Lorsque nous avons entrepris d'instrumenter Enkidu, nous pensions que la justification de la fonction de transfert des affectations de tableaux concentrerait tous nos efforts. En réalité, ceci fut plutôt aisé et ce pour deux raisons : la première, est que la structure des propriétés de tableaux permet à Enkidu de lever une grande partie des alias entre les différentes cases d'un même tableau, sans grande difficulté. Avec le recul, ceci peut paraître évident, car les analyses sont construites sur des domaines abstraits choisis de sorte à ce que les invariants puissent être calculés avec la plus grande facilité. La deuxième est que le calcul de plus faible pré-condition est un outil de raisonnement extrêmement puissant, qui permet de capter une grande partie des traitements de l'analyse, sous la forme d'une unique fonction, faisant le lien entre la logique de Hoare et la logique du premier ordre. La fonction *wp* explicite sous forme de conjonctions d'implications logiques les alias possibles entre les cases de tableau de la post-condition et la case de tableau modifiée. Cela permet de prouver une partie de ces conjonctions grâce à des contradictions évidentes entre les prémisses des implications. L'autre partie, où la

case modifiée est spécifiée dans la post-condition, se prouve facilement grâce à la substitution effectuée par *wp*. Dans ces seconds cas, il nous suffit de justifier la traduction de la formule résultante de la substitution des occurrences de la case de tableau modifiée, par une propriété de tableaux équivalente.

En résumé, une grande partie de la fonction de transfert est justifiée par la fonction *wp*. Le reste, de par la structure des propriétés de tableaux, est géré sans difficulté par le test d'inclusion instrumenté. Le travail d'instrumentation ne consiste alors qu'à faire le lien entre la formule produite par *wp* et celle prouvable par $\sqsubseteq^{\pi}_{gen}$. Cela revient à justifier la traduction de l'expression e (d'une affectation de la forme $A[i + c] := e$) dans $\mathscr{A}(P)$.

Nous avons présenté la justification des trois cas que la fonction de transfert distingue, en fournissant des patrons de preuve permettant de garantir la génération d'une preuve de correction pour chacun de ces cas. Nous avons illustré la justification de ces trois cas sur des exemples simples où les propriétés de tranche ne contiennent qu'une seule occurrence du tableau modifié. Dans des cas plus complexes, comme pour les programmes de tri où les propriétés de tranche sont de la forme $A[\ell] \leq A[\ell + 1]$, le calcul de plus faible pré-condition génère une conjonction de quatre implications, au lieu de deux dans les exemples présentés. Ces quatre implications sont prouvées en utilisant les différents patrons de preuve que nous venons de présenter et cela demande quelques efforts techniques supplémentaires.

De manière générale, si une propriété de tranche contient n occurrences du tableau modifié, le calcul de plus faible pré-condition retournera 2^n implications. Cette explosion des cas est un problème pratique sur lequel nous ne nous sommes pas penchés car, dans les programmes que Enkidu sait traiter, les occurrences d'un même tableau sont limitées à deux. Une perspective serait de trouver des techniques d'optimisation via des procédures de simplification de formules justifiées afin de limiter cette explosion des sous-buts retournés par *wp*.

Pour finir, nous n'avons pas abordé dans cette section le cas de l'affectation des variables scalaires. Celles-ci se traitent de la même façon que les tableaux, sauf qu'il n'y a pas d'alias possible. Il suffit de mettre à jour toutes les propriétés de tranche qui contiennent la variable scalaire modifiée.

4.4 Transitions gardées

Cette section présente l'analyse des transitions gardées par Enkidu et la stratégie de justification que nous lui avons apportée. Une garde est une condition que l'état du programme doit satisfaire pour passer d'un point de contrôle q à un autre point de contrôle q'. Ici, les états du programme sont approximés par des propriétés de tableaux. Comme pour les affectations, afin d'exploiter l'information contenue par le passage d'une transition gardée, Enkidu traduit l'expression de cette garde g dans le domaine $\mathscr{A}(P)$. La suite de l'analyse consiste à modifier la propriété de tableaux courante au point de contrôle q, Ψ en l'occurrence, en tenant compte du fait que g est vérifié par l'état du programme au point de contrôle q'. Une fois l'analyse effectuée et la post-condition calculée, notre travail consiste à apporter une stratégie à l'algorithme de la fonction de transfert des gardes afin qu'elle puisse justifier la correction des post-conditions calculées.

Le passage d'une transition gardée par une expression g se spécifie par le triplet de Hoare $\{\Psi\}?g\{\Psi'\}$ où Ψ et Ψ' sont les pré et post-condition de cette transition. Pour

prouver la validité de ce triplet, nous utilisons le calcul de plus faible pré-condition qui nous conduit à prouver la formule $g \Rightarrow \Psi'$ à partir de la pré-condition Ψ :

$$\frac{\vdash \Psi \Rightarrow (g \Rightarrow \Psi')}{\vdash \{\Psi\}?g\{\Psi'\}}\ sem$$

Nous présentons dans un premier temps, la fonction de transfert utilisée par Enkidu pour analyser les transitions gardées, puis notre stratégie de justification qui s'appuie une nouvelle fois sur l'instrumentation des opérateurs ensemblistes du domaine $\mathscr{A}(P)$.

4.4.1 Analyse des transitions gardées

Le langage des gardes autorisé par Enkidu est constitué des connecteurs `and` et `or` et des relations booléenes $=$ et $\leq$ entres les variables du programmes et/ou des constantes. Dans un cadre purement logique, la plus forte post-condition que l'on puisse déduire à partir d'une garde g et d'une pré-condition Ψ, est $\Psi \wedge g$ (*cf.* section 3.2.4.2). Cependant, $\Psi \wedge g$ n'appartient pas au domaine abstrait des propriétés de tableaux utilisé par Enkidu et il lui faut donc calculer une propriété de tableaux $\Psi' \stackrel{def}{=} \phi' \wedge \forall \ell, \bigwedge_{k=1}^{n} \varphi_k \Rightarrow \psi'_k$ (potentiellement) plus faible (au sens de l'implication logique) que $\Psi \wedge g$. Pour cela, Enkidu commence par exprimer g dans $\mathscr{A}(P)$.

4.4.1.1 Traduction de la garde vers le domaine asbtrait des propriétés de tableaux

De manière à exploiter l'information fournie par le passage d'une transition gardée, celle-ci doit être exprimée dans le domaine abstrait $\mathscr{A}(P)$. Cela est effectué par la fonction d'abstraction $\alpha()$, dont nous donnons le principe maintenant.

Fonction d'abstraction des gardes. Nous nous penchons d'abord sur les termes de base des gardes, que nous appelons les *gardes simples*, avant d'aborder le traitement des constructions conjonctives et disjonctives. Compte tenu de la structure du domaine $\mathscr{A}(P)$, les gardes portant sur des variables scalaires ou d'indices sont abstraites sous forme de contraintes de zone. Cette traduction de ce type de garde est trivial et ne présente aucune perte d'information, du fait que les expressions de garde sont construites avec les seules relations booléenes $=$ et $\leq$.

Pour les tableaux, l'abstraction consiste à traduire l'expression sous la forme d'une propriété de tranche, comme pour l'affectation des tableaux lorsque que la partie droite est également un tableau (*cf.* section 4.3.1.1). Par exemple, la garde $A[i] = k$ sera traduite par la propriété de tranche $(\ell = i) \Rightarrow (A[\ell] = k)$. De par la construction des partitions utilisé par Enkidu, nous savons que la tranche $\ell = i$ appartient à P. Si la garde comprend deux expressions de tableaux, elle sera traduite par une conjonction de deux propriétés de tranche : par exemple la garde $A[i] \leq B[i+1]$ sera traduite par la conjonction de propriétés de tranche $\bigwedge \begin{pmatrix} (\ell = i) \Rightarrow A[\ell] \leq B[\ell+1] \\ (\ell = i+1) \Rightarrow A[\ell-1] \leq B[\ell] \end{pmatrix}$. Ensuite, afin de ramener la garde dans $\mathscr{A}(P)$, il suffit d'associer à toutes les autres tranches de P auxquelles la garde ne fait pas référence la propriété de tranche $\top_{zone}$. Dans l'exemple que nous venons de donner, la propriété de tableaux correspondant à la garde $g \stackrel{def}{=} A[i] \leq B[i+1]$ est :

$$\alpha(g) = \Psi_g \stackrel{def}{=} \phi \wedge \forall \ell \bigwedge \begin{pmatrix} (\ell = i) \Rightarrow A[\ell] \leq B[\ell + 1] \\ (\ell = i + 1) \Rightarrow A[\ell - 1] \leq B[\ell] \end{pmatrix} \bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone},$$

où $P' \stackrel{def}{=} P \setminus \{(\ell = i), (\ell = i + 1)\}$, et le contexte ϕ est celui de Ψ, la propriété de tableaux en pré-condition.

Concernant les gardes composées de plusieurs gardes simples, par des disjonctions ou des conjonctions, aucune traduction n'est effectuée. Toutes les gardes simples sont asbtraites séparément, puis les conjonctions et les disjonctions sont gérées directement par la fonction de transfert que nous définissons maintenant.

4.4.1.2 Fonction de transfert des transitions gardées

Une fois la traduction de la garde effectuée, l'analyse est d'une grande simplicité et consiste simplement à calculer la borne inférieure de Ψ et Ψ_g, via l'opérateur $\sqcap_{\mathscr{A}(P)}$ donné en section 4.1.8. Ainsi, pour une garde simple g (sans `and`, ni `or`) la post-condition est la propriété de tableaux Ψ' calculée comme suit :

$$\Psi' = \mathsf{sem}_{\mathsf{g}}^{\sharp}(\Psi, g) = \Psi \sqcap_{\mathscr{A}(P)} \alpha(g)$$

Pour les gardes composées avec le connecteur `and`, la fonction de transfert est appliquée séquentiellement :

$$\Psi' = \mathsf{sem}_{\mathsf{g}}^{\sharp}(\Psi, g_1 \ \texttt{and} \ g_2) = \mathsf{sem}_{\mathsf{g}}^{\sharp}(\mathsf{sem}_{\mathsf{g}}^{\sharp}(\Psi, g_1), g_2)$$

Cela revient à diviser chaque transition gardée de la forme $q \xrightarrow{g_1 \ \texttt{and} \ g_2} q'$, par la séquence de transitions $q \xrightarrow{g_1} q'' \xrightarrow{g_2} q'$. Le connecteur `and` n'a pas de réelle utilité dès lors qu'il peut être remplacé par une séquence de transition et nous en ferons abstraction concernant la justification.

Pour les gardes composées avec le connecteur `or`, la fonction de transfert utilise l'opérateur de borne supérieure $\sqcup_{\mathscr{A}(P)}$:

$$\Psi' = \mathsf{sem}_{\mathsf{g}}^{\sharp}(\Psi, g_1 \ \texttt{or} \ g_2) = \mathsf{sem}_{\mathsf{g}}^{\sharp}(\Psi, g_1) \sqcup_{\mathscr{A}(P)} \mathsf{sem}_{\mathsf{g}}^{\sharp}(\Psi, g_2)$$

Dans le cas des gardes composées avec `or`, si les composantes ne portent pas sur la même variable, l'analyse retournera systématiquement Ψ. En effet, pour une garde de la forme g_1 `or` g_2, si la garde g_1 contraint Ψ sur une de ces tranches et que g_2 contraint Ψ sur une autre tranche alors, par l'application de $\sqcup_{\mathscr{A}(P)}$, on reviendra sur les deux tranches aux propriétés de tranche calculées avant le passage de cette transition gardée. Comme l'analyse ne peut décider laquelle des composantes de la garde a été satisfaite par l'état du programme, elle ne peut en tirer aucune information. En revanche, si les différentes composantes portent sur la même variable, l'application de l'opérateur $\sqcup_{\mathscr{A}(P)}$ permettra d'intégrer à Ψ la borne supérieure de ces composantes.

4.4.2 Justification de l'analyse des gardes

La justification des gardes traitées par Enkidu suit le patron de preuve générique donné en section 3.2.4.2. D'après ce patron de preuve, il nous faut apporter pour chaque exécution une preuve de $g \Rightarrow \alpha(g)$ et une autre pour $\Psi \wedge \alpha(g) \Rightarrow \Psi'$. Nous donnons dans les sections suivantes les patrons de preuve permettant de prouver ces deux sous-buts du patron de preuve générique pour la justification de l'analyse des gardes.

4.4.2.1 Justification de la traduction des gardes

Les gardes portant sur des variables d'indice ou scalaires n'ont pas à être traduites car elles appartiennent déjà au domaine abstrait des zones. Concernant les tableaux, il suffit d'exprimer la case de tableau testée par une propriété de tranche équivalente. Nous considérons une expression booléene g construite avec $=$ ou $\leq$, portant sur la case $i+c$ du tableau $A[\,]$, $g(A[i+c])$. La traduction de cette garde en une propriété de tableaux sera justifiée par le patron de preuve donné ci-dessous, qui permet de prouver pour toute garde de la forme $g(A[i+c])$ qu'elle implique la propriété de tableaux $\alpha(g(A[i+c])) \stackrel{def}{=} \phi \wedge \forall \ell, (\ell = i+c) \Rightarrow g(A[\ell]) \wedge \bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone}$, où $P' \stackrel{def}{=} P \stackrel{def}{=} P \setminus \{(\ell = i+c)\}$:

$$
\dfrac{
\dfrac{\dfrac{\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{\ell' = i+c}^{H_2} \qquad \vdash_{\text{NJ}} \overbrace{g(A[i+c])}^{H_1}}{\vdash_{\text{NJ}} g(A[\ell'])} =_e}{\vdash_{\text{NJ}} (\ell' = i+c) \Rightarrow g(A[\ell])} \Rightarrow_i (H_2) \qquad \vdash_{\text{NJ}} \bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone}}{\vdash_{\text{NJ}} (\ell' = i+c) \Rightarrow g(A[\ell']) \bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone}} \wedge_e}{\vdash_{\text{NJ}} \forall \ell, (\ell = i+c) \Rightarrow g(A[\ell]) \bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone}} \forall_i \qquad \dfrac{\vdash_{\text{NJ}} \Psi}{\vdash_{\text{NJ}} \phi} \wedge_e}{\vdash_{\text{NJ}} \phi \wedge \forall \ell, (\ell = i+c) \Rightarrow g(A[\ell]) \bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone}} \wedge_e
$$

$$
\dfrac{\vdash_{\text{NJ}} \phi \wedge \forall \ell, (\ell = i+c) \Rightarrow g(A[\ell]) \bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone}}{\vdash_{\text{NJ}} g(A[i+c]) \Rightarrow \underbrace{(\phi \wedge \forall \ell, (\ell = i+c) \Rightarrow g(A[\ell]) \bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone})}_{\alpha(g(A[i+c]))}} \Rightarrow_i (H_1)
$$

Dans le patron de preuve ci-dessus, deux sous-buts ne sont pas pouvés : le premier est la propriété de tableaux Ψ en pré-condition de la transition. Or (cela n'apparaît pas dans ce patron de preuve) la propriété de tableaux Ψ est une hypothèse introduite juste après l'application du *wp* sur le triplet $\{\Psi\}?g\{\Psi'\}$ (*cf.* section 3.2.4.2). Le second sous-but à prouver est la conjonction de propriété de tranche $\bigwedge_{k=1}^{|P'|} \varphi_k \Rightarrow \top_{zone}$), sur toutes les tranches de P autres que $\ell = i+c$. Ces propriétés de tranche sont toutes $\top_{zone}$, qui s'exprime en logique par $\top$. Ces propriétés sont trivialement vraies et il n'y a donc rien à prouver pour justifier l'analyse.

4.4.2.2 Justification de la fonction de transfert des gardes

Dans le cas d'une garde simple g, l'algorithme de la fonction de transfert consiste simplement à appliquer l'opérateur de borne inférieure sur Ψ et $\alpha(g)$. Pour justifier cela, il nous suffit d'instrumenter l'opérateur $\sqcap_{\mathscr{A}(P)}$ dont nous redonnons la définition pour deux propriétés de tableaux quelconques Ψ_1 et Ψ_2 :

$$\Psi_1 \sqcap_{\mathscr{A}(P)} \Psi_2 \quad = \quad \begin{cases} \bot_{\mathscr{A}} & si\ \Psi_1 = \bot_{\mathscr{A}}\ ou\ \Psi_2 = \bot_{\mathscr{A}} \\ (\phi_1 \sqcap_{zone} \phi_2) \wedge \bigwedge_{k=1}^{n} \varphi_k \Rightarrow (\psi_{1_k} \sqcap_{zone} \psi_{2_k}) & sinon \end{cases}$$

Dans le cas où ni Ψ_1 ni Ψ_2 ne sont absurdes (ce qui est le cas qui nous intéresse), nous lui associons le patron de preuve donné en figure 4.15. Ce patron de preuve consiste à prouver pour chaque tranche φ_k, que $\psi_{1_k} \wedge \psi_{2_k} \Rightarrow \psi_{1_k} \sqcap_{zone} \psi_{2_k}$. Ces sous-buts sont automatiquement prouvés grâce aux justifications apportées par $\sqcap^{\pi}_{zone}$. De la même façon, il nous faut prouver que $\phi_1 \wedge \phi_2 \Rightarrow \phi_1 \sqcap_{zone} \phi_2$, ce qui est également fourni par $\sqcap^{\pi}_{zone}$. Le reste du patron de preuve, comme pour celui du test d'inclusion, n'est qu'un *jeu* de dérivations permettant de se ramener à ces sous-buts en remontant vers les hypothèses, $\Psi_1 \wedge \Psi_2$ en l'occurrence.

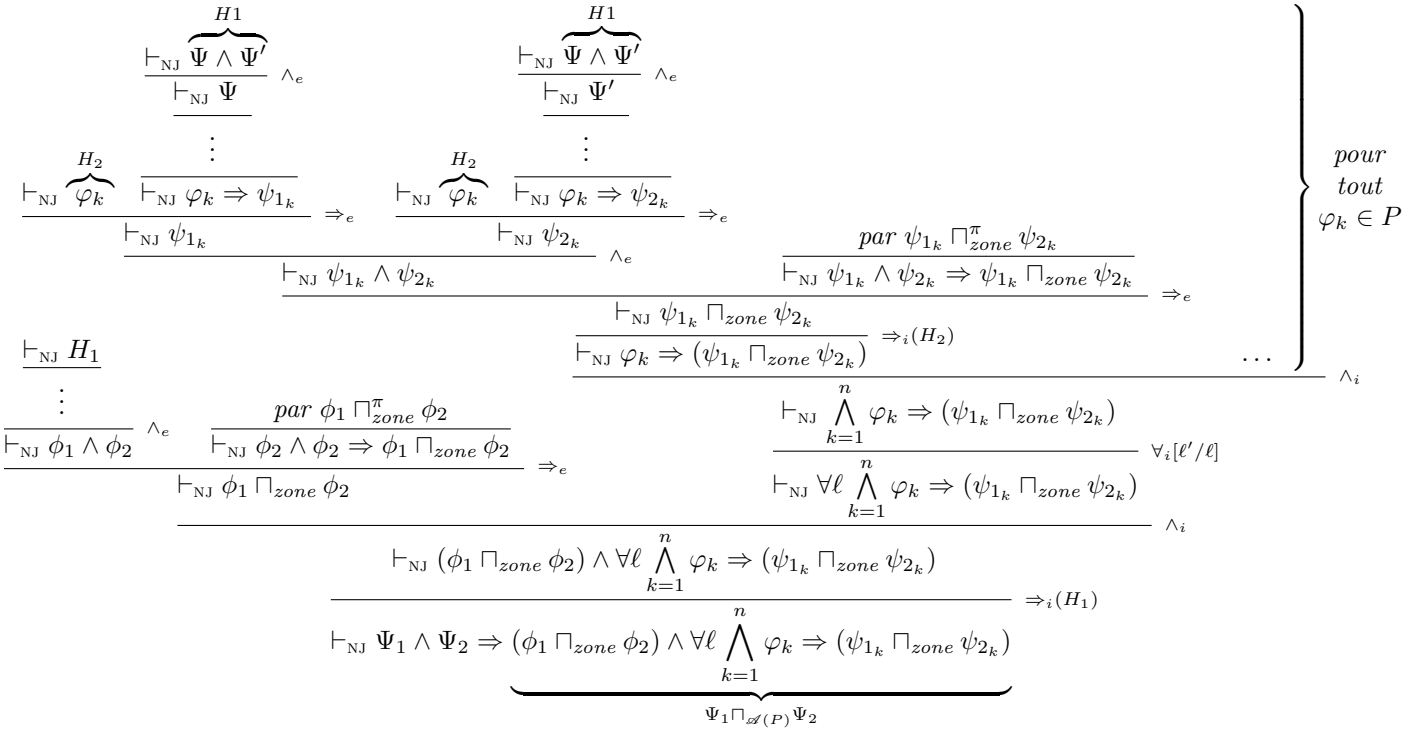


FIG. 4.15 – Patron de preuve pour la justification de $\sqcap_{\mathscr{A}(P)}$

Pour justifier le traitement des gardes simples, nous pouvons donc utiliser $\sqcap^{\pi}_{\mathscr{A}(P)}$ pour prouver le sous-but $\Psi \wedge \alpha(g) \Rightarrow \Psi'$, du fait que $\Psi' \stackrel{def}{=} \Psi \sqcap_{\mathscr{A}(P)} \alpha(g)$.

Nous abordons maintenant la question des gardes composées avec les connecteurs `and` et `or`.

Justification des gardes conjonctives. Dans le cas des conjonctions, nous avons vu que l'analyse appliquait séquentiellement le traitement pour chaque garde simple qui compose g. Cela se justifie également séquentiellement en prouvant chaque garde simple séparément les unes des autres.

Justification des gardes disjonctives. Le cas de la disjonction ne peut être décomposé par une séquence de transitions gardées. Cependant, l'analyse construit la post-condition en utilisant l'opérateur de borne supérieure, $\sqcup_{\mathscr{A}(P)}$, qui induit une perte d'information. Ainsi, si l'on considère une garde disjonctive de la forme g_1 `or` g_2 et une post-condition $\Psi' \stackrel{def}{=} \mathsf{sem}^\sharp_{\mathsf{g}}(\Psi, g_1 \texttt{ or } g_2)$, il s'avère que $(\Psi \sqcap_{\mathscr{A}(P)} \alpha(g_1)) \sqsubseteq_{\mathscr{A}(P)} \Psi'$ et $(\Psi \sqcap_{\mathscr{A}(P)} \alpha(g_2)) \sqsubseteq_{\mathscr{A}(P)} \Psi'$, par définition de $\sqcup_{\mathscr{A}(P)}$. Pour justifier l'analyse des gardes dans le cas d'une disjonction, nous commençons par interpréter le connecteur `or` par la disjonction logique $\vee$. Cette interprétation du connecteur `or` est captée par la fonction wp :

$$\frac{\vdash_{\text{NJ}} \Psi \Rightarrow ((g_1 \vee g_2) \Rightarrow \Psi')}{\vdash_{\text{H}} \{\Psi\}?(g_1 \texttt{ or } g_2)\{\Psi'\}}\ sem$$

Pour compléter la preuve, il nous suffit de prouver que $g_1 \vee g_2 \Rightarrow \alpha(g_1) \vee \alpha(g_2)$, puis que $(\Psi \sqcap_{\mathscr{A}(P)} \alpha(g_1)) \Rightarrow \Psi'$ et $(\Psi \sqcap_{\mathscr{A}(P)} \alpha(g_2)) \Rightarrow \Psi'$, par élimination de la disjonction sur la formule $\alpha(g_1) \vee \alpha(g_2)$.

Prouver $g_1 \vee g_2 \Rightarrow \alpha(g_1) \vee \alpha(g_2)$ consiste simplement à prouver que $g_1 \Rightarrow \alpha(g_1)$ et $g_2 \Rightarrow \alpha(g_1)$:

$$\dfrac{\dfrac{\vdash_{\text{NJ}} \overbrace{g_1 \vee g_2}^{H_1} \qquad \dfrac{\dfrac{\vdots}{\vdash_{\text{NJ}} \alpha(g_1)}}{\vdash_{\text{NJ}} \alpha(g_1) \vee \alpha(g_2)} \vee_i \qquad \dfrac{\dfrac{\vdots}{\vdash_{\text{NJ}} \alpha(g_2)}}{\vdash_{\text{NJ}} \alpha(g_1) \vee \alpha(g_2)} \vee_i}{\vdash_{\text{NJ}} \alpha(g_1) \vee \alpha(g_2)} \vee_e(H_2, H_3)}{\vdash_{\text{NJ}} (g_1 \vee g_2) \Rightarrow \alpha(g_1) \vee \alpha(g_2)} \Rightarrow_i (H_1)$$

Prouver $\alpha(g_1)$ avec g_1 en hypothèse consiste à appliquer le patron de preuve permettant de justifier la traduction des gardes dans le domaine des propriétés de tableaux dans le cas des gardes simples. Il en est de même pour g_2.

Pour finir, prouver l'implication $(\Psi \sqcap_{\mathscr{A}(P)} \alpha(g_1)) \Rightarrow \Psi'$ est encore une fois similaire au cas des gardes simples que nous avons présenté plus haut. Il suffit d'utiliser l'opérateur de borne inférieure instrumenté $\sqcap^{\pi}_{\mathscr{A}(P)}$. Il en est de même pour g_2.

4.4.3 Conclusion sur la justification de l'analyse des gardes

La justification de l'analyse des gardes dans le cas d'Enkidu est conforme à la justification générique des gardes que nous avons donnée dans le chapitre précédent. Pour pouvoir garantir que chaque analyse sera justifiée, il nous a fallu instrumenter l'opérateur de borne inférieure du domaine $\mathscr{A}(P)$ et fournir un patron de preuve pour justifier l'asbtraction des gardes simples dans le domaine abstrait.

4.5 Développement de l'instrumentation d'Enkidu

Nous détaillons dans cette section le travail de développement que nous a demandé de fournir l'instrumentation d'Enkidu, puis les performances de cette version instrumentée sur un ensemble de programmes connus. Concernant le développement de l'instrumentation, une incommodité rencontrée fut de comprendre puis d'adapter un code qui n'était pas le notre. Enkidu utilise un certain nombre de classes Ocaml avec une longue chaîne de dépendances et chaque modification du type d'une fonction a engendré des modifications en cascade d'autres fonctions. Les fonctions dont nous avons modifié le type sont celles à qui nous avons associé un patron de preuve. Dans le cas de l'instrumentation d'un analyseur, réalisée par le développeur[2] de cet analyseur, ces difficultés et les pertes de temps qui en ont découlé seront évitées, et notre méthodologie serait encore plus facilement applicable.

Du point de vue des expérimentations, le nombre de programmes traités est réduit mais permet de voir des différences de performances considérables selon la taille du programme et de la partition utilisée. Pour les programmes où les performances ne sont pas satisfaisantes, un certain nombre d'optimisations peuvent être apportées sur des points que nous avons clairement identifiés. La version actuelle d'Enkidu instrumentée utilise à différents endroits la tactique `omega` qui ralentit fortement la construction des λ-terme (qui est incluse dans la vérification des preuves). De plus, un grand nombre de buts triviaux ne sont pas reconnus par cette implantation et sont prouvés à l'aide de patrons de preuve inutilement grands. Cependant, cette version permet de générer automatiquement des preuves de programmes non triviaux qui manipulent des tableaux, ce qui est en soit un résultat non-négligeable et original, même si ces preuves ne sont pas encore optimales.

4.5.1 Implantation

Enkidu, implanté en Ocaml, est encore à l'état de prototype et comprend de nombreuses heuristiques qui limitent la diversité des programmes analysables. Pour des raisons de compatibilité, nous avons implanté en Ocaml les patrons de preuve permettant d'instrumenter chacune des fonctions utiles pour justifier l'analyse des transitions qu'Enkidu sait traiter. Dans notre implantation, les patrons de preuve sont définis par un type Ocaml qui permet de modéliser tous les raisonements formels de la déduction naturelle et de la logique de Hoare. À l'exécution, les preuves sont générées dans un type Ocaml, qui sera ensuite traduit en un script de tactique Coq comme nous l'avons proposé en section 3.1.3. Une classe Ocaml a été définie à ce titre et peut être utilisée à nouveau pour l'instrumentation de n'importe quel autre analyseur statique. Cette classe, nous permettant de modéliser les raisonnements logiques dans un type Ocaml puis de les traduire sous forme de tactiques du système Coq, est d'environ 2000 lignes et n'est pas spécifique à Enkidu. Du point de vue des patrons de preuve développés pour instrumenter l'analyseur Enkidu, le volume est d'environ 800 lignes de code (pour un analyseur qui n'en fait pas loin de 10000). Parmis ces 800, lignes n'est comprise que l'instrumentation *ad hoc* que nous avons proposée pour justifier l'analyse de la modification des indices (*cf.* section 4.2.2.2). La seconde version utilisant le test d'inclusion

[2]Une condition nécessaire est que ce développeur ait un minimum de notion en logique et en preuve de programmes.

générique n'est pas encore implantée. Concernant la permière version de la justification des indices, une partie des sous-buts est prouvée grâce à la tactique `omega`, qui permet de prouver automatiquement la validité de formules simples exprimées en arithmétique de Presburger. Du fait que dans la version actuelle d'Enkidu les partitions ne sont pas calculées, nous n'avions pas de fonction à instrumenter pour prouver que chaque P est une partition du tableau. Le fait que P soit une partition (*i.e.* $\varphi_1 \vee \ldots \vee \varphi_n$ est valide) a donc été prouvé grâce à la tactique `omega`. L'utilisation de cette tactique nous a permis de tester plus rapidement notre instrumentation d'Enkidu. Cependant si au niveau de la génération des scripts Coq, les performances sont acceptables, elles le sont moins concernant le temps de vérification des preuves. L'exécution de la tactique `omega`, qui a lieu à la vérification des preuves, est extrêmement lente comme l'indique le tableau d'expérimentation donné en figure 4.16. Dans le cas de l'algorithme find, la tactique `omega` ne termine pas car la partition comprend trop de tranches.

4.5.2 Expérimentation

Nous donnons en figure 4.16 une table qui indique les performances de la version instrumentée d'Enkidu sur les quelques exemples que l'implantation de l'analyse sait traiter à ce jour.

	$\#\tau$	$\lvert P\rvert$	analyse *(second)*	invariant calculé	gp *(second)*	tp *(kilo byte)*	vp *(second)*
array_copy(A,B)	3	3	0.042	$\forall \ell,\ A[\ell] = B[\ell]$	0.014	9.2	2.97
max_search(A)	6	4	0.284	$\forall \ell,\ A[\ell] \leq \mathtt{m}$	0.028	15	4.03
sentinel(A,x)	3	9	0.461	$A[\mathtt{i}] = \mathtt{x} \ \wedge\ \forall \ell < \mathtt{i},\ A[\ell] \neq \mathtt{x}$	0.21	61	8.67
first_non_null(A)	8	13	4.782	$A[\mathtt{i}] \neq 0 \ \wedge\ \forall \ell < \mathtt{i},\ A[\ell] = 0$	2.58	100	26.54
insert_sort(A[1..n])	11	10	6.047	$\forall \ell,\ 2 \leq \ell \leq \mathtt{n} \Rightarrow A[\ell] \leq A[\ell - 1]$	2.75	145	31.54
find(A[1..n],x)	13	14	31.697	$\forall \ell,\ \bigwedge \begin{pmatrix} \ell = \mathtt{i} \Rightarrow A[\ell] = \mathtt{x} \\ 1 \leq \ell < \mathtt{i} \Rightarrow A[\ell] < \mathtt{x} \\ \mathtt{i} < \ell \leq \mathtt{n} \Rightarrow \mathtt{x} \leq A[\ell] \end{pmatrix}$	5.92	548	41.22

gp = génération des preuves ; tp = taille des preuves ; vp = vérification des preuves.

Fig. 4.16 – Temps d'analyse, de génération des preuves, de vérification des preuves et taille des preuves produites par la version instrumentée d'Enkidu

Les invariants calculés donnés en figure 4.16 sont les propriétés de tableaux calculées aux points de contrôle finaux de ces programmes avec des partitions simplifiées. Les preuves, en revanche, portent sur l'ensemble des points de contrôle avec les partitions réelles. Comme nous l'avons déjà fait remarquer, ces preuves contiennent de nombreuses branches inutilement grandes pour des sous-buts triviaux qu'il serait utile de reconnaître. Ce genre d'optimisations, tout comme la justification de l'analyse de la modification des indices en utilisant un calcul de plus faible pré-condition retournant des propriétés de tableaux exprimées sur une partition affinée et un test d'inclusion générique (*cf.* section 4.2.3), conduiront dans une prochaine version à de meilleures performances. Ce qui nous importait en premier lieu était de voir si notre méthodologie etait applicable en pratique sur des analyseurs non triviaux. L'expérience conduite sur Enkidu montre qu'il est possible de faire générer à un analyseur des preuves de programmes vérifiables par Coq, au prix d'une implantation réalisée dans un temps limité et raisonnable.

4.6 Conclusion

Nous avons instrumenté Enkidu dans le but de lui faire générer des justifications de ses analyses sous forme de preuves déductives vérifiées par le système Coq. Au delà de ce résultat technique, qui révèle que l'instrumentation est réalisable en pratique, nous avons comparé l'efficacité de notre approche avec celle proposée dans [SYY03, Cha06]. Cette dernière n'est pas optimale, car elle demande d'instrumenter les fonctions de transfert alors que cela n'est pas nécessaire dans le cas d'Enkidu. Dans le cas de l'analyse des modifications des indices de tableaux, le calcul de plus faible pré-condition ne préserve pas les invariants dans le domaine abstrait. Cependant, en affinant la partition des tableaux, nous avons tout de même pu utiliser l'instrumentation d'un test d'inclusion (plus générique) pour justifier les résultats de cette fonction de transfert sans avoir à l'instrumenter. Le code de cette fonction de transfert est d'une grande complexité et éviter de l'instrumenter présente un intérêt majeur. En effet, de manière à évaluer les différentes approches, nous avons tout de même instrumenté la fonction de transfert pour les affectations d'indices de tableaux. En présentant ces deux instrumentations consécutivement, nous avons montré de façon très limpide que l'instrumentation des fonctions de transfert demande un gros travail de développement, aussi coûteux que l'implantation de la fonction de transfert elle-même. De plus, utiliser le test d'inclusion pour justifier l'analyse de la modification des indices factorise les fonctions à instrumenter, car ce même test est utilisable pour justifier l'analyse des affectations de tableaux.

Cette étude de cas autour d'Enkidu a permis de soulever un certain nombre de problèmes, difficiles à mettre en avant en restant à un niveau trop général. Dans la perspective de l'instrumentation d'autres analyseurs statiques, nos travaux apportent des éléments permettant de déterminer en amont quel est l'ensemble minimal de fonctions à instrumenter pour que ces analyseurs puissent générer des certificats de leurs résultats. Pour un certain type de transitions qu'un analyseur sait traiter, nous avons identifié trois cas selon les formules retournées par le calcul de plus faible pré-condition : (1) la formule appartient au domaine abstrait et la preuve est obtenue par une instanciation des patrons de preuve associés aux seuls opérateurs ensemblistes du domaine ; (2) la formule n'appartient pas au domaine abstrait mais il est possible de la normaliser vers le domaine abstrait et de finir la preuve comme dans le premier cas ; (3) la formule n'appartient pas au domaine abstrait et il n'est pas possible de la normaliser vers le domaine abstrait. Il n'est donc pas possible dans le troisième cas de se passer des informations apportées par les fonctions de transfert, et celles-ci doivent être instrumentées.

Une étude intéressante serait d'appliquer cette technique d'instrumentation basée sur des patrons de preuve sur un analyseur statique industriel comme astrée [CCF+05, CCF+06, CCF+07]. Cela permettrait de savoir si cette technique d'instrumentation est vraiment efficace, ou si le cas d'Enkidu était un cas favorable. Une chose nous semble sûre, c'est que dans tous les cas, il est moins difficile d'instrumenter un analyseur (y compris astrée) que de le certifier avec l'assistance de Coq.

Implanter un analyseur comme Enkidu est un travail de thèse (ceci fut réalisé par Mathias Péron) et demande de gros efforts en terme de developpement et d'optimisation pour avoir des performances raisonnables. Implanter Enkidu en Ocaml, sans se soucier de la correction ni de la terminaison pour chacune de ses exécutions, est plus facile que de le certifier en Coq où il est en plus nécessaire d'apporter des preuves de correction de l'ensemble de ses fonctions (même si cela peut être facilité par le travail

de David Pichardie [Pic05], un certain nombre de preuves sont à fournir). D'ailleurs, en instrumentant Enkidu, nous avons mis en évidence de nombreux bogues que nous avons communiqués à Mathias Péron et que ce dernier à ensuite pu corriger. Rien que du point de vue du débogage, l'instrumentation d'analyseurs statiques présente un interêt majeur. Les bogues trouvés dans Enkidu montrent de plus qu'un analyseur peut retourner des résultats corrects, sans pour autant que le code de cet analyseur le soit totalement. Certifier un tel analyseur nécessite que celui-ci soit exempt de bogues, alors que l'instrumentation est plus permissive. D'un point de vue pratique cela n'est pas négligeable. En effet, l'instrumentation permet de transformer des outils d'analyse dont la correction n'est pas garantie, en des outils de certification atteignant un niveau de confiance des plus élevés.

Chapitre 5

Vers la certification d'un protocole de communication pour systèmes multi-tâches

Dans ce chapitre, nous allons étudier comment un analyseur statique instrumenté peut être mis à contribution pour certifier des programmes utilisés dans des applications critiques. Les *systèmes critiques* auxquels nous allons nous intéresser, ajoutent des dimensions supplémentaires aux programmes que nous avons jusqu'ici considérés : les systèmes sont concurrents, synchronisés et paramétrés. Ces dimensions rendent la certification de programmes beaucoup plus complexe, dûs à des couches sémantiques supplémentaires, étendant celle des programmes impératifs. Certifier automatiquement de tels systèmes paramétrés n'est pas à notre porté à l'heure actuelle. Ces systèmes sont difficiles (voire impossibles) à analyser et à valider par le calcul et en conséquence extrêmement difficiles à certifier par la preuve formelle. De l'autre côté, de nombreuse branches de leur preuve de correction sont redondantes, longues et fastidieuses, ce qui rend la certification de tels systèmes à l'aide d'un assistant à la preuve difficilement réalisable. Cependant la construction d'une partie de ces branches peut être automatisée avec des outils instrumentés tel Enkidu. Ce chapitre a donc pour objet d'étudier comment des outils de certification automatiques peuvent faciliter la construction de preuves de correction de systèmes critiques et complexes. Si nous faisons le choix d'appliquer nos techniques de certification à de tels programmes, c'est par une volonté de sortir des études de cas académiques, et confronter nos résultats à de réels problèmes rencontrés dans l'industrie. Certifier la correction de l'implantation du tri par insertion est intéressant et nous a demandé beaucoup de travail pour réaliser cela automatiquement, mais est peu utile en soit. Ce que nous visons, c'est de certifier des programmes pour applications embarquées et nous présentons comment un outil d'analyse pourtant à l'état de prototype, mais instrumenté, peut servir à cela.

5.1 Contexte et motivations

Le contexte de cette étude est un peu particulier car il fait appel à des paradigmes de programmation que nous n'avons pas encore évoqués. Les systèmes que nous allons considérer sont programmés dans des *langages de programmation synchrones*, tels que

simulink ou lustre [Hal93b]. Ces langages sont utilisés pour concevoir des systèmes critiques et en interaction avec leur environnement, embarqués dans l'avionique, l'aeronautique, le ferroviaire et l'automobile notamment [Hal93a]. La recherche en vérification et validation de ces systèmes critiques a pour but de permettre aux industries de concevoir des programmes fiables, avec la certitude qu'ils auront les comportements attendus à l'exécution. Sans rentrer dans les détails du paradigme de la programmation synchrone, nous rappelons seulement que les programmes synchrones peuvent être vus comme des automates déterministes de Mealy. Ils s'exécutent périodiquement selon une *horloge logique* et produisent à chaque instant logique des *sorties* qui constituent un *flôt de données*. Les sorties sont calculées par *les fonctions du programme* et dépendent de *l'état courant du système* et des *variables d'entrées* (voir figure 5.1).

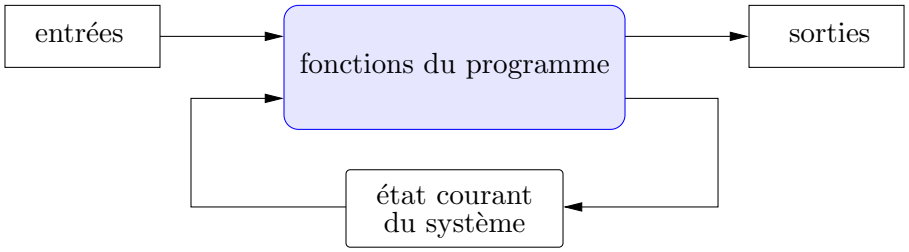


Fig. 5.1 – Programme synchrone sous forme d'automate de Mealy

Un programme synchrone peut être vu commme un ensemble de tâches où chaque tâche est déclenchée puis s'exécute un nombre fixé (selon chaque entrée) de fois entre deux instants logiques. Les tâches communiquent entre elles et s'exécutent en concurrence, *i.e.* une tâche prioritaire peut en interrompre une autre en accédant au microprocesseur afin d'être exécutée. Cette concurrence entre les tâches est gérée par un système d'exploitation (que nous appelons l'`os`) qui accorde aux tâches l'accés au *microprocesseur* selon un ordre total de priorité. L'ensemble (`os` + tâches) constitue le *système multi-tâches* (voir section 5.2.1). Cette décomposition des applications synchrones sous forme de systèmes multi-tâches communicantes facilite la construction et la génération automatique des programmes synchrones à partir d'une modèlisation dite de *haut-niveau* qui fait abstraction des temps d'exécution notamment. Utiliser des modèles de *haut niveau*, permet aux programmeurs de se concentrer sur le flôt de données des tâches et leurs interactions. Par exemple, les modèles ne tiennent pas compte des temps de calculs nécessaires aux tâches pour produire leurs sorties (voir section 5.2.1.4). La gestion du temps est dès lors reportée à l'implantation du système. Si le système d'exploitation utilise un ordonnancement des tâches avec interruptions, des différences de flôts de données entre le modèle haut-niveau et son implantation peuvent survenir (voir section 5.3.1). Ces aspects doivent être gérés à l'implantation du système par un protocole permettant de préserver le flôt de données du modèle haut niveau à son code exécutable. Cette propriété du protocole dépend des *priorités* et de l'*ordonnancement*, deux notions essentielles dans la conception de tels applications synchrones, nous y reviendrons dans les sections suivantes. Notre objectif est de certifier la correction de ce protocole : prouver formellement que les données échangées entre les tâches à l'exécution seront conformes à la modélisation de haut-niveau.

Contributions. Du point de vue de la certification, cette décomposition des applications synchrones en systèmes multi-tâches permet de raisonner sur des programmes

impératifs. En effet, les tâches sont implantées par des programmes rentrant naturellement dans le paradigme de la programmation impérative. La dimension synchrone des applications considérées est alors exprimée par les exécutions successives de ces tâches. Ce que nous voulons certifier, c'est le comportement global du système multi-tâches. Ce qui nous importe dès lors, c'est la façon dont les tâches *communiquent* et *interagissent* entre elles. Par contre les calculs que les tâches font pour traiter et produire les données ne sont pas considérés comme pertinents ici. *Est-ce que les tâches s'échangent les bonnes valeurs conformément à la modélisation haut-niveau du système ?* est la question qui attire notre attention. Une implantation du système multi-tâches préservant le flôt de données du modèle haut-niveau est proposée dans [SC04, TSSC05, STC06]. Les auteurs proposent un protocole qui utilise un espace mémoire restreint et qui garantit que l'implantation du système est conforme à la modélisation de haut niveau, au sens où elles ne sont pas distinguables d'un point de vue du flôt de données. Nous donnons une modélisation du système de communication en section 5.2, puis nous proposons en section 5.3 une implantation du protocole défini dans [SC04]. Concernant la certification de cette implantation, notre objectif est d'utiliser l'analyseur statique intrumenté Enkidu pour construire automatiquement une partie de la preuve de correction d'un tel protocole. L'emploi d'Enkidu a nécessité quelques simplifications du système multi-tâches que nous avons justifiées formellement. Le but de ces simplifications est d'amener l'implantation du protocole dans le domaine d'entrée des programmes analysables par Enkidu. La section 5.4 présente l'automatisation de la génération de la preuve de correction de ce protocole qui permet de préserver à l'exécution le flôt de données défini par la modélisation haut niveau du système.

5.2 Modélisation du système réactif multi-tâches

Nous présentons dans cette section le système multi-tâches sur lequel le protocole sera employé ainsi que les propriétés sémantiques de ce système. Ces propriétés concernent les communications et les interruptions entre les tâches (voir section 5.2.2). Il est important de détailler ces points car établir la correction du protocole consiste à prouver que le flôt de données du système au niveau de sa modélisation sera preservée à l'exécution malgré les interruptions. À l'exécution, les temps de calculs imposent des délais minimums, quant aux données lues par certaines tâches, que la modélisation haut niveau doit prévoir pour que le protocole puisse préserver le flôt de données. Ces restrictions concernant la *fraicheur* des données lues par certaines tâches sont présentées en section 5.2.2.3 et sont essentielles pour définir la propriété de correction du protocole que nous donnons en section 5.4.

5.2.1 Modèles du système multi-tâches avec évènements

Nous dénotons par V l'ensemble des variables du système. Pour notre étude, la valeur des variables n'est pas importante, seules nous importent les flôts de données, *i.e.* les égalités entre la sortie d'une tâche écrivain et l'entrée d'une tâche lectrice à chaque instant de l'exécution du système. Nous utilisons une représentation symbolique permettant d'indiquer le nombre d'affectations qu'a subi une certaine variable. Grâce à cela nous pouvons modéliser le fait que la n-ième valeur d'une variable v_1, notée v_1^n (indépendamment de cette valeur) soit égale à la m-ième valeur d'une autre variable v_2,

notée v_2^m, *i.e.* $v_1^n = v_2^m$. Cette représentation s'inspire de la représentation SSA (*Single Static Assignment*) [Pop06] utilisée notamment en compilation et en analyse statique. L'ensemble des valeurs (symboliques) du système est défini par $\mathcal{V} = V \times \mathbb{N}$ et nous noterons v^n la valeur symbolique (v, n).

Nous dénotons par T l'ensemble des tâches du système. Soit $\mathcal{T}_i \in T$ une tâche, nous notons $\mathcal{T}_i^n$ la n-ième exécution de cette tâche dans le système. Une tâche lit des entrées, effectue des calculs, et écrit ses résultats dans une variable de sortie. À chaque tâche $\mathcal{T}_i$ sont associées deux variables : x_i la variable représentant ses entrées et y_i la variable repésentant le résultat de ses calculs donné en sortie :

- nous définissons par x_i^n la valeur lue à la n-ième occurrence de la tâche $\mathcal{T}_i$.
- nous définissons par y_i^n la valeur écrite à la n-ième occurrence de la tâche $\mathcal{T}_i$.
- y_i^0 représente la valeur de sortie initiale (indéfinie) de la tâche $\mathcal{T}_i$.

5.2.1.1 Exécution d'une tâche

L'exécution d'une tâche $\mathcal{T}_i$ est modélisée par trois *événements* a_i, d_i, f_i correspondant respectivement au déclenchement de la tâche, au début et à la fin de son exécution. Les propriétés sémantiques que nous définissons par la suite exigent de distinguer les occurrences de ces événements. L'ensemble des événements du système est dénoté par E. Nous définissons par e^n la n-ième occurrence de l'événement e lors d'une exécution du système. Ainsi, a_i^n (*resp* d_i^n, f_i^n) correspond à la n^{eme} occurrence du déclenchement (*resp*, début et fin) de la tâche $\mathcal{T}_i$. Les occurrences des événements sont définies (similairement aux valeurs symboliques des variables) dans $E \times \mathbb{N}$.

5.2.1.2 Traces d'exécution

Une trace d'exécution, σ, du système constitué de $|T|$ tâches concurrentes et communicantes, est une séquence potentiellement infinie d'événements provenant d'une ou plusieurs tâches. L'ensemble $\Sigma(T)$ des traces d'exécution possibles pour un système multi-tâches sur l'ensemble de tâches T, correspond au *langage hors-contexte* [Cho59] caractérisé par la grammaire suivante :

$$S \rightarrow a_i; S; d_i; S; f_i; S \mid \varepsilon, \qquad \text{avec } i \in [1, |T|]$$

où la trace de longueur 0 est dénotée par ε (la trace vide).

Pour l'instant, nous pouvons envisager les traces d'exécution commes des séquences d'evénements associées aux tâches du système, avec pour unique contrainte, le respect de l'ordre $a_i; d_i; f_i$ pour une tâche $\mathcal{T}_i$ donnée.

Exemple 5.2.1. *Soient les tâches $\mathcal{T}_i$ et $\mathcal{T}_j$. Les séquences d'événements suivantes sont des séquences d'exécution possibles du système :*

$$a_i^1; d_i^1; f_i^1;\ a_j^1; d_j^1; f_j^1$$
$$a_i^1;\ a_j^1; d_j^1; f_j^1;\ d_i^1; f_i^1$$
$$a_i^1; d_i^1; f_i^1;\ a_j^1; d_j^1; f_j^1;\ a_i^2; d_i^2; f_i^2$$
$$a_i^1; d_i^1; f_i^1;\ a_j^1; d_j^1;\ a_i^2; d_i^2; f_i^2;\ f_j^1$$

Dans les deux premières traces d'exécution, les deux tâches s'exécutent une seule fois, sans interruption après leur activation. Cependant, dans la seconde les événements des deux tâches s'entrelacent les uns aux autres. Dans la troisième et la quatrième trace d'exécution, la tâche $\mathcal{T}_i$ s'exécute deux fois et ses événements s'entrelacent avec les événements de $\mathcal{T}_j$ dans la quatrième seulement. Dans cette dernière, la tâche $\mathcal{T}_i$ suspend l'exécution de $\mathcal{T}_j$ lors de sa seconde activation (a_i^2). $\mathcal{T}_j$ reprend son exécution après la fin de celle de $\mathcal{T}_i$ (f_i^2). Nous reviendrons sur cette notion d'interruption en section 5.2.2.1.

Une représentation des traces d'exécution sous forme de diagramme est plus intélligilible. Les événements sont présentés chronologiquement sur une flèche horizontale représentant l'axe du temps. Les flèches verticales décrivent les signaux émis par l'environnement, c'est à dire le déclenchement des tâches (les événements a). La troisième trace d'exécution de l'exemple 5.2.1 est donnée sous forme de diagramme en figure 5.2 et 5.4. Si une trace d'exécution est représentée par deux diagrammes, c'est qu'il existe deux modèles pour un tel système multi-tâches. Un premier modèle haut niveau, dit idéal, ne tient pas compte du temps d'exécution des tâches et permet de se focaliser sur les fonctionnalités du système. Le second modèle, dit réaliste, se rapproche de l'implantation du système et ne peut ignorer le temps d'exécution des tâches. Les traces d'exécution dans ce modèle réaliste doivent garantir une exécution du système conforme au modèle idéal malgré les contraintes d'implantation.

5.2.1.3 Modèle idéal

La *modélisation idéale* du système considère une exécution des tâches en *temps zéro*. Ce modèle simplifié de l'exécution des tâches permet de raisonner sur l'ordonnancement des tâches en faisant abstraction des temps de calcul. Les événements a_i^k, d_i^k et f_i^k sont alors confondus.

$$\forall i, k \in \mathbb{N}, a_i^k = d_i^k = f_i^k$$

Nous donnons en figure 5.2 la première trace d'exécution de l'exemple 5.2.1, $a_i^1; d_i^1; f_i^1;\ a_j^1; d_j^1; f_j^1$, en l'occurrence.

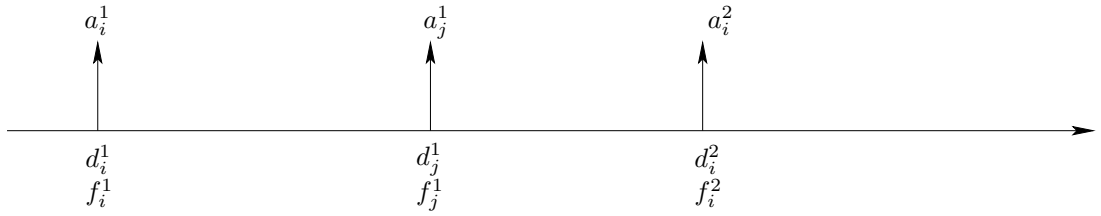


Fig. 5.2 – Exemple de diagramme dans le modèle idéal

À chaque événement a_i, la tâche $\mathcal{T}_i$ lit les entrées x_i, calcule et produit les sorties y_i. L'immédiateté des calculs fait qu'il n'y a pas d'entrelacement possible entre les tâches dans ce modèle. Les traces d'exécution deux et quatre de l'exemple 5.2.1 ne sont donc pas possibles dans ce modéle.

5.2.1.4 Modèle réaliste avec contraintes d'implantation

Le *modèle réel* du système considère au contraire un temps de calcul pour chaque événement et autorise plus de traces d'exécution que le modèle idéal. La figure 5.3 illustre la k-ième exécution dans le modèle réel de la tâche $\mathcal{T}_i$ déclenchée par l'événement a_i^k.

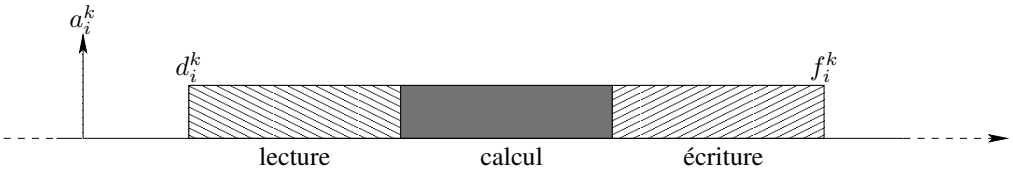


Fig. 5.3 – Opérations effectuées lors de l'exécution d'une tâche

Les temps de lecture, de calcul et d'écriture ne sont pas connus. Nous savons simplement qu'une tâche commence à lire les entrées à l'instant d et qu'elle a calculé et écrit ses résultats à l'instant f. Les calculs effectués par les tâches et la façon dont les valeurs de sorties sont produites ne sont pas significatifs pour les propriétés de correction qui vont nous intéresser par la suite et qui ne concernent que les flôts de données.

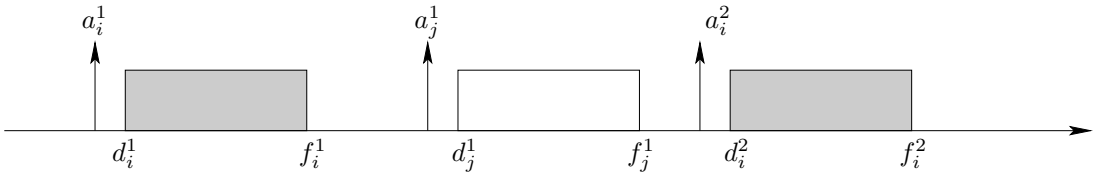


Fig. 5.4 – Diagramme de la trace d'exécution $a_i^1; d_i^1; f_i^1;\ a_j^1; d_j^1; f_j^1;\ a_i^2; d_i^2; f_i^2$ dans le modèle réaliste

La figure 5.4 est une représentation par un diagramme de la troisième trace d'exécution de l'exemple 5.2.1 ($a_i^1; d_i^1; f_i^1; a_j^1; d_j^1; f_j^1; a_i^2; d_i^2; f_i^2$ en l'occurrence). Le diagramme de la figure 5.4 présente deux exécutions de la tâche $\mathcal{T}_i$, déclenchées par les événements a_i^1 et a_i^2 et une exécution de la tâche $\mathcal{T}_j$, déclenchée par a_j^1. La longueur des rectangles symbolise la durée des exécutions, bornée par les événements d_i et f_i de même exposant, et la hauteur des rectangles symbolise leur niveau de priorité. Comme nous n'avons pas encore défini cette notion, nous représentons pour l'instant les exécutions des tâches sans tenir compte des priorités, et les rectangles ont tous la même hauteur. Enfin, le temps de latence entre l'activation (signalée par a) d'une tâche et le début de son exécution correspond au temps necéssaire à l'os pour gérer l'attribution des ressources à la tâche activée.

Nous définissons maintenant un prédicat de *précédence*, noté $\prec$, dans la signature de notre logique qui nous permet de raisonner sur les ordonnancements des événements du système. Cela nous permet de définir la *schédulabilité* d'une trace d'exécution, une propriété essentielle des systèmes réactifs.

Définition 5.2.1 (Précédence)**.** *Soient deux événements du système e_1 et e_2, nous dirons que e_1 précède e_2 si e_1 apparaît avant e_2 dans une trace d'exécution du système. Nous formalisons cela de la façon suivante :*

$$\forall \sigma_1, \sigma_2 \in \Sigma, \sigma_1 \stackrel{def}{=} (e_1; \sigma_2; e_2) \Rightarrow e_1 \prec_\sigma e_2$$

Définition 5.2.2 (Schedulabilité)**.** *Une trace d'exécution est* schedulable *si au plus une instance d'une tâche donnée est active à tout instant. Autrement dit, l'exécution d'une tâche ne peut être de nouveau déclenchée tant que l'exécution courante de cette même tâche n'a pas terminé. Nous formalisons la schedulabilité d'une trace d'exécution σ par la contrainte suivante :*

$$\forall \sigma \in \Sigma, \forall i, k \in \mathbb{N}, (a_i^{k+1} \in \sigma) \Rightarrow a_i^k \prec_\sigma d_i^k \prec_\sigma f_i^k \prec_\sigma a_i^{k+1}$$

Dans la suite de ce chapitre nous considérons uniquement des systèmes respectant la propriété de schédulabilité. Le protocole que nous allons certifier ne s'applique qu'à des systèmes schédulables. Cette propriété est vérifiable dès lors que les temps de calculs ont été fixés [SC04].

5.2.2 Communications inter-tâches avec priorités statiques

Nous décrivons maintenant comment implanter le système sur un os à gestion des interruptions basées sur des priorités statiques. Cela signifie que l'ordre de priorité défini sur l'ensemble des tâches est fixé en amont et ne peut être modifié dynamiquement pendant l'exécution du système. Le déclenchement des tâches est géré par l'environnement qui n'est pas contrôlé par l'os. Par conséquent, une tâche plus prioritaire peut suspendre temporairement l'exécution d'une tâche de priorité inférieure. Nous faisons l'hypothèse que les interruptions, si elles peuvent survenir à tout moment, ne peuvent en revanche contrarier la propriété de schédulabilité du système.

5.2.2.1 Priorités et interruptions

Pour implanter un ensemble de tâches communicantes avec interruption $T \stackrel{def}{=} \{\mathcal{T}_1, \ldots, \mathcal{T}_n\}$, le système exige que les tâches soient ordonnées selon un ordre total de priorité. Nous définissons à cet effet une fonction injective $p : T \to \mathbb{N}$, permettant d'ordonner les tâches selon l'ordre des entiers naturels.

$p(\mathcal{T}_j) < p(\mathcal{T}_i)$ signifie : « $\mathcal{T}_j$ *a une priorité plus faible que* $\mathcal{T}_i$ » . Une tâche avec une priorité plus élevée peut en interrompre une autre. Au niveau des événements, pour deux tâches $\mathcal{T}_i$ et $\mathcal{T}_j$ telles que $p(\mathcal{T}_j) < p(\mathcal{T}_i)$, cela implique les contraintes suivantes :

1. $\forall k, p \in \mathbb{N}, a_i^k \prec_\sigma f_j^p \Rightarrow f_i^k \prec_\sigma f_j^p$
2. $\forall k, p \in \mathbb{N}, a_i^k \prec_\sigma d_j^p \Rightarrow f_i^k \prec_\sigma d_j^p$
3. $\forall k, p \in \mathbb{N}, d_i^k \prec_\sigma f_j^p \Rightarrow f_i^k \prec_\sigma f_j^p$
4. $\forall k, p \in \mathbb{N}, d_i^k \prec_\sigma d_j^p \Rightarrow f_i^k \prec_\sigma d_j^p$

Ces quatre contraintes formalisent le fait que la tâche $\mathcal{T}_i$, du fait de sa priorité supéreure, finira de s'exécuter avant la tâche $\mathcal{T}_j$, qu'elle l'interrompe juste après son décléchement (entre a_j et d_j) ou pendant son exécution (entre d_j et f_j). Nous donnons en figure 5.5 un automate définissant les traces d'exécution bien formées pour deux tâches $\mathcal{T}_i$ et $\mathcal{T}_j$ telles que $p(\mathcal{T}_j) < p(\mathcal{T}_i)$. L'état "init" de l'automate correspond aux instants où aucune tâche n'est active et donc à l'état initial du système en particulier. À partir de cet état deux événements sont possibles : a_i ou a_j signalant le décléchement de l'une des deux tâches. Les états successeurs de l'automate respectent la séquentialité des événements

a, d, f, la schédulabilité et la priorité supérieure de $\mathcal{T}_i$ qui lui permet d'interrompre $\mathcal{T}_j$ à tout instant. Cependant, si $p(\mathcal{T}_j) < p(\mathcal{T}_i)$, l'activation des tâches est aléatoire et a_j peut être déclenché à tout instant et interrompre les exécutions de $\mathcal{T}_i$. Néanmoins les exécutions de $\mathcal{T}_j$, signalées par d_j et f_j, ne pourront avoir lieu si $\mathcal{T}_i$ est activée ou en train de s'exécuter. Lorsque les deux tâches ont fini de s'exécuter et n'ont pas été de nouveau déclenchées, l'automate revient à l'état "init". En résumé, cet automate est le produit des automates correspondants à $\mathcal{T}_i$ et $\mathcal{T}_j$ où les transitions ne respectant pas les contraintes de priorité ont été suprimées.

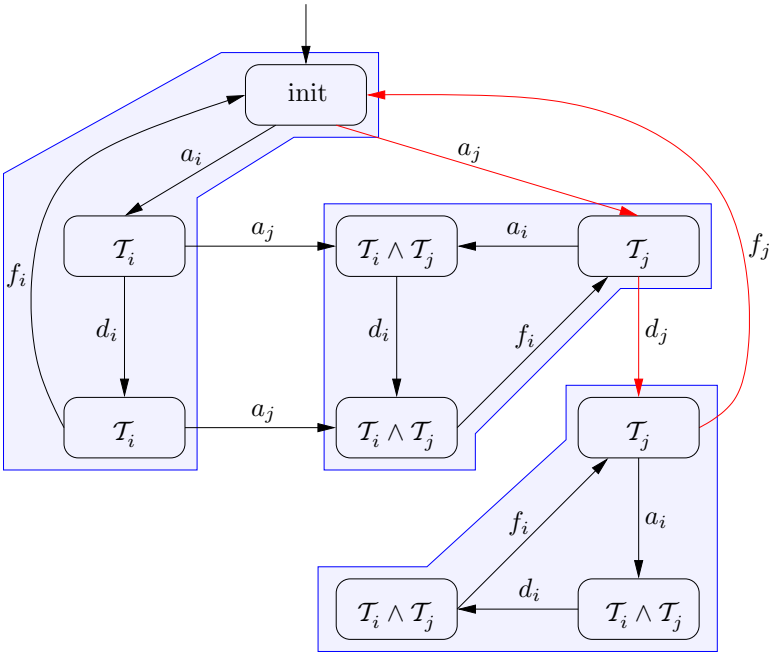


Fig. 5.5 – Automate définissant les traces d'exécution possibles entre deux tâches en tenant compte des priorités

En figure 5.5, les exécutions sans interruption de $\mathcal{T}_i$ sont colorées en bleu. De par sa priorité supérieure, les événements associés aux exécutions de $\mathcal{T}_i$ ne peuvent s'entrelacer qu'avec a_j. L'unique exécution non interrompue de $\mathcal{T}_j$ est représentée par les flêches rouges. Dans tous les autres cas, $\mathcal{T}_j$ doit attendre la fin de l'exécution de $\mathcal{T}_i$ pour (finir de) s'exécuter. En effet, les événements d_j et f_j ne peuvent avoir lieu que si la tâche $\mathcal{T}_i$ n'a pas encore été déclenché ou si $\mathcal{T}_i$ a fini d'être exécutée.

Exemple 5.2.2. *Soient deux tâches $\mathcal{T}_i$ et $\mathcal{T}_j$ telles que $p(\mathcal{T}_j) < p(\mathcal{T}_i)$. Nous reprenons les diagrammes donnés en figures 5.2 et 5.4, mais en avançant temporellement l'arrivé de la deuxième exécution de la tâche $\mathcal{T}_i$, de sorte que $\mathcal{T}_i$ interrompe $\mathcal{T}_j$:*

L'événement a_i^2 a pour effet d'interrompre l'exécution de $\mathcal{T}_j^1$, ce qui n'était pas le cas dans les exemples précendents. $\mathcal{T}_j^1$ reprend son exécution à l'instant f_i^2 où $\mathcal{T}_i^2$ finit son exécution. De ce fait, l'événement f_i^2 précède chronologiquement l'événement f_j^1 (à l'inverse de la figure 5.2).

Exemple 5.2.3. *Nous regardons maintenant un exemple où la tâche ayant la plus*

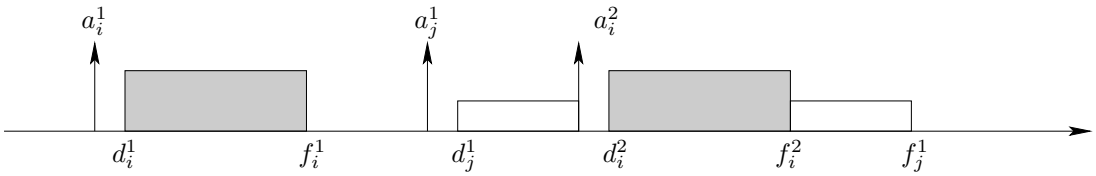


Fig. 5.6 – Exemple de trace d'exécution avec interruption dans le cas $p(\mathcal{T}_j) < p(\mathcal{T}_i)$

grande priorité est $\mathcal{T}_j$, c'est à dire que $p(\mathcal{T}_i) < p(\mathcal{T}_j)$. Nous considérons que la tâche $\mathcal{T}_i$ s'exécute pour la k-ième fois et la tâche $\mathcal{T}_j$ pour la p-ième :

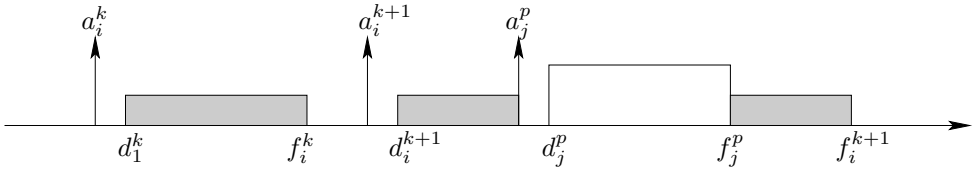


Fig. 5.7 – Exemple de trace d'exécution avec interruption dans le cas $p(\mathcal{T}_i) < p(\mathcal{T}_j)$

Le diagramme présente deux occurrences successives de la tâche $\mathcal{T}_i$ déclenchées par les événements a_i^k et a_i^{k+1}. $\mathcal{T}_i^{k+1}$ est interrompu par le déclenchement de l'exécution de la tâche $\mathcal{T}_j$ à l'instant a_j^p. Ce qui a pour conséquence que f_j^p précède chronologiquement f_i^{k+1}.

5.2.2.2 Communications entre les tâches

Les communications entre les tâches sont modélisées par un graphe de flôt de données $G \in T^2 \times \{-1, 0\}$, où un triplet $(\mathcal{T}_i, \mathcal{T}_j, \delta) \in G$ modélise une communication de la tâche i vers la tâche j .

Si $\delta = 0$, nous représentons le triplet $(\mathcal{T}_i, \mathcal{T}_j, c)$ par $\mathcal{T}_i \rightarrow \mathcal{T}_j$ précisant le sens du flôt de données et signifiant que : « *$\mathcal{T}_j$ lit les dernières données calculées et émises par $\mathcal{T}_i$* » . Si $\delta = -1$, nous représentons le triplet $(\mathcal{T}_i, \mathcal{T}_j, c)$ par $\mathcal{T}_i \xrightarrow{-1} \mathcal{T}_j$ signifiant que : « *$\mathcal{T}_j$ lit les avant-dernières données calculées et émises par $\mathcal{T}_i$* » . Du fait des priorités et des interruptions il n'est pas possible à l'implantation de garantir qu'un écrivain fournira toujours la valeur la plus fraîche à une tâche lectrice si cette dernière a une priorité plus élevée que l'écrivain. En revanche si on accepte de se contenter de l'avant-dernière donnée produite alors le protocole proposé dans [SC04] apporte une solution d'implantation qui permet de préserver les flôts de données définis dans le modèle idéal.

Exemple 5.2.4. *Nous nous intéressons à deux tâches $\mathcal{T}_i$ et $\mathcal{T}_j$ liées dans le graphe G par l'arc $\mathcal{T}_i \rightarrow \mathcal{T}_j$. Nous considérons que la valeur lue par $\mathcal{T}_j$ est celle écrite lors de la dernière occurrence de $\mathcal{T}_i$.*

Le diagramme de la figure 5.8 présente une communication non perturbée par des interruptions. L'événement a_j^p déclenche l'exécution de la tâche $\mathcal{T}_j$. Elle lit la valeur symbolique y_i^k, produite par $\mathcal{T}_i^k$, disponible à l'instant f_i^k. L'exécution de $\mathcal{T}_j^p$ se déroule normalement et produit une valeur de sortie que nous ne représentons pas sur le diagramme, pouvant être lue par d'autres tâches.

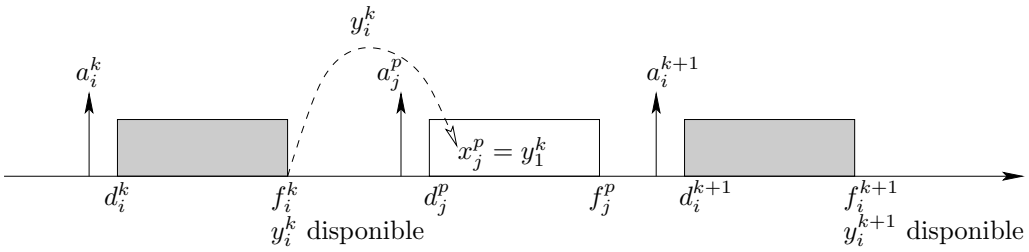


Fig. 5.8 – Exemple de communication entre deux tâches

5.2.2.3 Restriction des délais de communication

Nous avons illustré le principe de communication entre les tâches. Une question se pose si nous souhaitons intégrer les interruptions avec priorités statiques à ce système de communication :

> « *Est-il toujours possible pour une tâche de lire les dernières données produites par une autre ?* »

Nous répondons à cette question par un contre-exemple. Nous reprenons l'exemple illustré en figure 5.7 avec l'ordre de priorité $p(\mathcal{T}_i) < p(\mathcal{T}_j)$, et nous y intégrons une communication de la forme $\mathcal{T}_i \rightarrow \mathcal{T}_j$. La figure 5.9 illustre le fait qu'il n'est pas possible dans certains cas de garantir un délai de communication nul entre les tâches.

Exemple 5.2.5. *En considérant que la valeur lue par $\mathcal{T}_j$ est celle écrite lors de la dernière occurrence de l'exécution de $\mathcal{T}_i$, nous observons des différences de flôts de donées entre le modèle idéal et son implantation du fait des interruptions.*

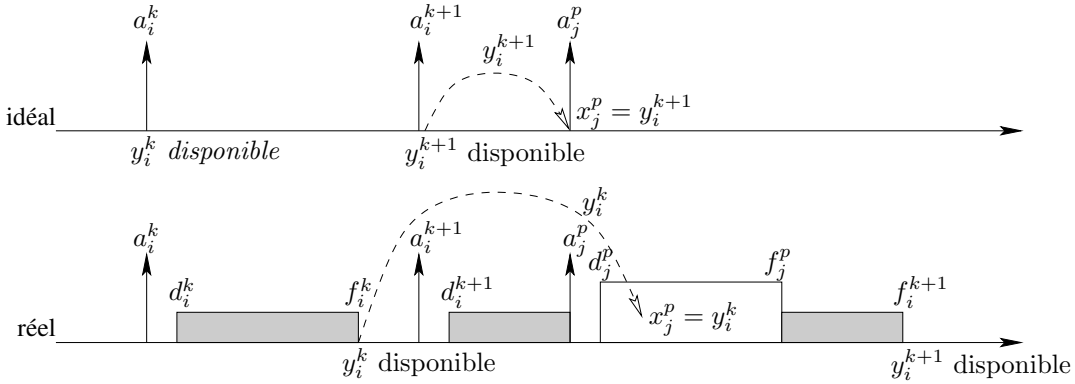


Fig. 5.9 – Exemple illustrant une différence sémantique entre le modèle idéal et son implantation

Du fait que $\mathcal{T}_j$ peut interrompre l'exécution de $\mathcal{T}_i$, nous observons une différence de comportement entre le modèle idéal et le modèle réel. Lorsque $\mathcal{T}_j^p$ démarre son exécution à l'instant d_j^p, la valeur disponible produite par $\mathcal{T}_i$ la plus récente n'est pas la même dans les deux modèles. Dans le modèle idéal, où les calculs se font en temps nul, il n'y a pas d'interruption possible et la valeur la plus fraîche *est y_i^{k+1} ; elle est immédiatement disponible dès l'activation de $\mathcal{T}_i^{k+1}$. En revanche, nous remarquons que dans le modèle*

réaliste f_j^p précède f_i^{k+1}. À l'instant f_j^p la valeur la plus récente écrite par $\mathcal{T}_i$ est donc y_i^k. La valeur lue par $\mathcal{T}_j$ ne peut donc pas être celle écrite lors de l'exécution la plus récente de $\mathcal{T}_i$.

L'exemple précédent montre que, pour permettre l'implantation de ce système de communication, nous devons prendre en compte les délais de communication dans les cas où une tâche lit des valeurs émises par des tâches de priorité inférieure. Toutefois, nous allons montrer par la suite que la *tâche lectrice* peut toujours prendre en entrée les valeurs produites lors de la pénultième occurrence de l'arrivée de la *tâche écrivain*.

5.3 Préservation de la sémantique grâce au DBP protocole

Nous présentons dans cette section le DBP protocole, proposé par Paul Caspi *et al* dans [SC04, TSSC05]. Afin de pouvoir implanter les communications entre les tâches le protocole alloue des tampons mémoires, que nous nommerons *buffers*, où les tâches lisent et écrivent respectivement leurs données d'entrée et de sortie. Le protocole comporte deux fonctions : (1) l'une exécutée aux déclenchements des tâches, leur permettant de mettre à jour les adresses mémoire où elles pourront écrire (ou lire, selon les cas) leurs données ; (2) l'autre exécutée entre les événements d et f, permettant aux tâches de lire leurs entrées à l'adresse mémoire fournie par la première fonction, puis de calculer et écrire leurs données de sortie.

Si nous voulons préserver les mêmes flôts de données entre les modèles idéals et réalistes, les communications doivent être dirigées par le déclenchement des tâches et non par leurs exécutions. En effet, si nous souhaitons déterminer quelle est la dernière valeur produite par une tâche $\mathcal{T}_i$ relativement à une exécution de la tâche $\mathcal{T}_i$, seuls les événements de déclenchement des tâches sont identiques dans les deux modèles. Les événements de déclenchement permettent de comparer ainsi les communications entre la modélisation idéale et son implantation et de formaliser la préservation des flôts de données par le protocole.

5.3.1 Problème avec une implantation "simple"

L'implantation la plus simple consiste à allouer un unique tampon mémoire pour chaque couple de tâches communicantes, partagé par les deux tâches, où l'une écrit, et l'autre lit.

Exemple 5.3.1. *Soit $\mathcal{T}_i$ et $\mathcal{T}_j$ deux tâches telles que $p(\mathcal{T}_j) < p(\mathcal{T}_i)$ et $\mathcal{T}_i \rightarrow \mathcal{T}_j$,* i.e. *les valeurs lues par $\mathcal{T}_j$ sont celles écrites lors de l'exécution de $\mathcal{T}_i$ la plus récente. Nous donnons en figure 5.10 une communication entre ces deux tâches, afin d'illustrer cette implantation simple.*

Un second exemple illustre les limites de cette implantation utilisant un simple buffer (un tampon à une place), dans le cas où la tâche écrivain $\mathcal{T}_j$ interrompt $\mathcal{T}_i$ entre son déclenchement par l'os et son exécution.

Exemple 5.3.2. *Soit $\mathcal{T}_i$ et $\mathcal{T}_j$ deux tâches telles que $p(\mathcal{T}_j) < p(\mathcal{T}_i)$ et $\mathcal{T}_i \rightarrow \mathcal{T}_j$. La*

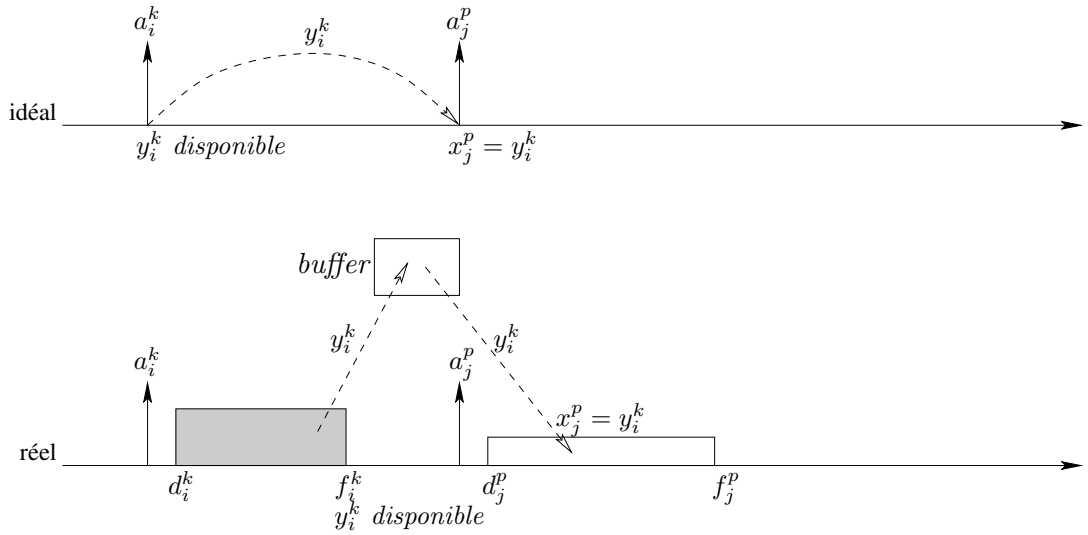


Fig. 5.10 – Exemple de communication avec un simple buffer

figure 5.11 montre une communication entre ces deux tâches qui illustre les problèmes de préservation du flôt de données entre les deux modèles, dans le cas d'une implantation simple avec un buffer à une place mémoire.

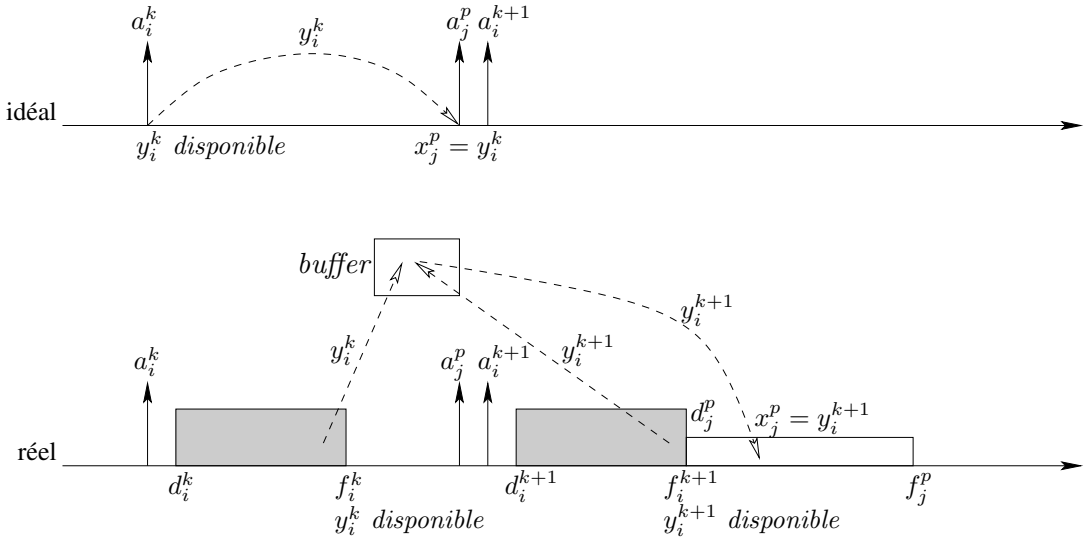


Fig. 5.11 – Problème de préservation du flôt de données causé par une interruption de la tâche écrivain

En figure 5.11, lorsque l'interruption survient, à l'instant a_i^{k+1}, nous observons que $\mathcal{T}_j^p$ est déclenchée, mais n'a pas commencé son exécution. Conformément à la sémantique idéale, la valeur que $\mathcal{T}_j^p$ doit lire est y_i^k : à l'instant a_j^p, y_i^k est mémorisée dans le buffer. Cependant, la $k+1$-ième exécution de $\mathcal{T}_i$ vient modifier le buffer en écrasant y_i^k par y_i^{k+1}. Lorsque $\mathcal{T}_j^p$ démarre son exécution, y_i^k est perdue et le flôt de données du modèle idéal n'est pas préservé par cette implantation du système. Le fait que la tâche $\mathcal{T}_j$ lise à l'exécution des valeurs plus fraîches que prévue par la modélisation idéale du système

provoque une modification (même avantageuse, en terme de fraîcheur) du flôt de données qui rend tout raisonnement dans le modèle idéal utilisé en conception inapproprié.

Remarque. *Si l'interruption survient après que $\mathcal{T}_j^p$ ait débuté son exécution (a_i^{k+1} arrive entre les événements d_j^p et f_j^p) cela ne change rien au problème. Comme nous ne pouvons pas savoir si $\mathcal{T}_j^p$ a entièrement lu les données contenues dans le buffer, nous devons considérer le cas pessimiste où ce n'est pas le cas et en conséquence que le flôt de données ne sera pas préservé. Ainsi, les interruptions entre les événements d et f d'une certaine tâche, sont confondues avec celles entre a et d, comme nous en avons donnée un exemple en figure 5.11.*

5.3.2 Principe du DBP protocole

Nous expliquons le principe du DBP protocole à travers des exemples illustrant le fonctionnement du protocole dans des cas où il n'y a que deux tâches. L'implantation du protocole dans le langage C est donnée en figure 5.16. Le protocole utilise des *buffers* mémoires afin de pouvoir conserver les valeurs produites par chaque tâche. Cette mémorisation est essentielle pour pouvoir préserver les flôts de données correspondant aux traces d'exécution du modèle idéal. Comme nous l'avons vu précédemment, l'ordre de priorité entre les tâches induit deux schémas de communication. Il est possible de considérer les communications où des tâches lisent des valeurs produites par des tâches de priorité plus élevée, et les communications où des tâches lisent des valeurs produites par des tâches de priorité plus faible. Cette décomposition du système a tout son sens du point de vue du problème de préservation des flôts de données car les valeurs lues (en terme de fraîcheurs) ne sont pas les mêmes selon l'ordre de priorité (*cf.* section 5.2.2.3). Le protocole utilise des tampons mémoires à deux *cases*, appelées *double-buffer* (voir figure 5.17) pour une communication entre deux tâches. Il se pose dès lors le problème de *où lire et où écrire* parmi ces deux cases ? Le protocole utilise, pour chaque double-buffer, deux pointeurs indiquant les adresses où une tâche doit lire et l'autre doit écrire. Ces pointeurs prennent leurs valeurs dans $\mathbb{B} \stackrel{def}{=} \{0, 1\}$ (voir figure 5.15) du fait que le tampon mémoire alloué pour un couple écrivain-lecteur ne contient que deux cases. L'évolution des pointeurs d'adresse est dirigée par les événements d'activation des tâches. Les tâches enregistrent l'adresse courante à leur arrivé, écrivent à cette dernière et lisent à l'autre case du buffer (dont l'adresse est la négation booléene de l'adresse courante). Nous illustrons ce principe sur la trace d'exécution donnée à l'exemple 5.3.3, qui nous servira de référence pour illustrer le fonctionnement du protocole.

Exemple 5.3.3. *Soit deux tâches $\mathcal{T}_i$ et $\mathcal{T}_j$ telles que $p(\mathcal{T}_j) < p(\mathcal{T}_i)$. Nous considérons la trace d'exécution suivante où nous ne précisons pas le sens de la communication entre ces deux tâches :*

Le diagramme de la figure 5.12 présente trois occurrences de l'exécution de la tâche $\mathcal{T}_i$, déclenchées par les événements a_i^k, a_i^{k+1} et a_i^{k+2} et deux occurrences de l'exécution de la tâche $\mathcal{T}_j$, déclenchées par les événements a_j^p et a_j^{p+1}. La tâche $\mathcal{T}_i$ a plus forte priorité que $\mathcal{T}_j$. Nous observons que la $(k+1)$-ième exécution de $\mathcal{T}_i$ interrompt $\mathcal{T}_j^p$.

Nous n'avons pas défini dans cet exemple le sens du flôt de données entre ces deux

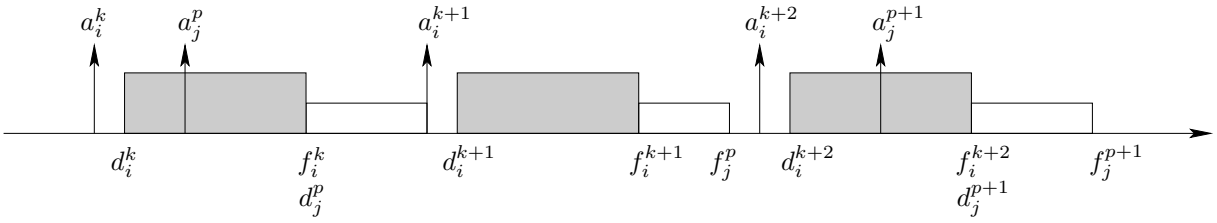


Fig. 5.12 – Exemple de trace d'exécution utilisé pour illustrer le fonctionnement du protocole

tâches. Selon le flôt de données, le système a des comportements bien distincts. Nous illustrons ces deux cas par deux schémas de communication, que nous nommons *high to low* où $\mathcal{T}_i \rightarrow \mathcal{T}_j$ et *low to high* où $\mathcal{T}_j \xrightarrow{-1} \mathcal{T}_i$, sur la trace d'exécution donnée en exemple 5.3.3. Pour faciliter la lecture des diagrammes, nous représentons l'exécution de chaque tâche sur des lignes distinctes (voir figures 5.13 et 5.14).

5.3.2.1 Communication dans le sens *high to low*

Considérons le cas où $\mathcal{T}_i \rightarrow \mathcal{T}_j$. Le protocole utilise deux pointeurs d'adresse : *awh* (*address writer high*) pour la tâche écrivain de plus grande priorité et *arl* (*address reader low*) pour la tâche lectrice de priorité inférieure. La figure 5.13 donne un exemple d'exécution du système à l'aide du protocole. L'évolution des pointeurs d'adresse est surlignée en bleu.

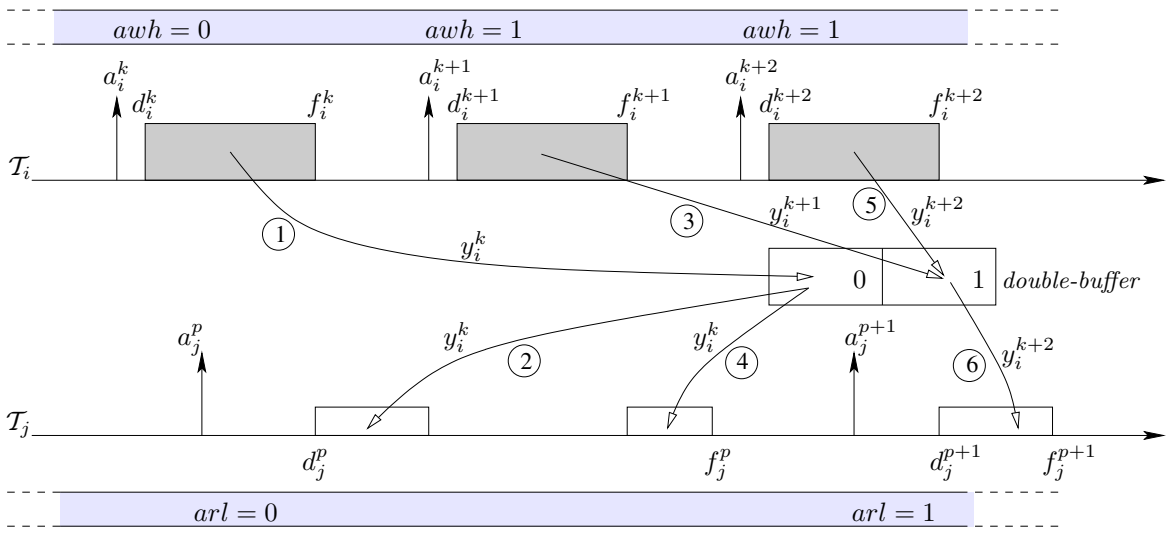


Fig. 5.13 – Exemple de communication où la tâche écrivain est prioritaire

La figure 5.13 illustre le fonctionnement du protocole sur l'exemple précédent en six étapes :

1. $\mathcal{T}_i^k$ démarre son exécution et écrit à l'adresse courante (la case 0 du double-buffer, *i.e.* $awh = 0$).
2. $\mathcal{T}_j^p$ est activée, le protocole fait basculer le pointeur d'adresse de lecture sur l'adresse courante d'écriture ($arl = 0$). Cependant, $\mathcal{T}_j$ n'ayant pas la plus forte

priorité doit attendre la fin de l'exécution de $\mathcal{T}_i^k$, signalée par f_i^k, pour s'exécuter. $\mathcal{T}_j^p$ utilise la même adresse que $\mathcal{T}_i^k$ pour lire la dernière valeur produite par $\mathcal{T}_i$ et respecte donc le flôt de données défini dans le modèle idéal.

3. a_i^{k+1} interrompt l'exécution de $\mathcal{T}_j^p$. Cette nouvelle activation de $\mathcal{T}_i$ fait basculer le pointeur d'adresse où elle doit écrire ($awh = 1$ désormais) de sorte que $\mathcal{T}_i^{k+1}$ ne puisse modifier les données que $\mathcal{T}_j$ est en train de lire.
4. $\mathcal{T}_j^p$ reprend son exécution et peut de nouveau accéder à y_i^k enregistré à la case 0 du buffer (arl vaut toujours 0).
5. Une nouvelle exécution de $\mathcal{T}_i$ est déclenchée à l'instant a_i^{k+2}, $\mathcal{T}_i^{k+2}$ en l'occurrence, et écrase la valeur écrite lors de son occurrence précédente, y_i^{k+1}. En effet, du fait qu'il n'y a pas eu de nouvelle arrivé de la tâche $\mathcal{T}_j$ le protocole ne modifie pas la valeur du pointeur d'adresse d'écriture courante. La valeur y_i^{k+1} n'est lue par aucune exécution de $\mathcal{T}_j$ car elle n'est la valeur produite par $\mathcal{T}_i$ la plus fraîche ni pour la p-ième, ni pour la $p+1$-ième exécution de $\mathcal{T}_j$.
6. L'arrivé de l'événements a_j^{p+1}, qui signale l'activation de $\mathcal{T}_j^{p+1}$, fait de nouveau basculer le pointeur de lecture ($arl = 1$) et permet à $\mathcal{T}_j$ d'accèder à y_i^{k+2} durant son exécution. La $p+1$-ième exécution de $\mathcal{T}_j$ lit bien la dernière valeur produite par $\mathcal{T}_i$.

Le protocole consiste donc à gérer des pointeurs d'adresse permettant aux tâches de communiquer en utilisant un tampon mémoire à seulement deux cases. L'algorithme permettant de mettre à jour ces pointeurs d'adresse est donné en figure 5.15 : la procédure `os(i)` (*resp.* `os(j)`) qui implante cet algorithme sera exécutée à chaque déclenchement de l'événement a_i (*resp.* a_j). Les pointeurs d'adresse sont implantés par deux tableaux à deux dimensions, `awh[][]` et `arl[][]`, dont les cases `awh[i][j]` et `arl[j][i]` représentent les variables awh et arl de la figure 5.13. Cette procédure gère également les pointeurs d'adresse utilisés dans le cas d'une communication où la tâche écrivain a une priorité plus faible que son lecteur.

5.3.2.2 Communication dans le sens *low to high*

Considérons maintenant le cas où $\mathcal{T}_j \xrightarrow{-1} \mathcal{T}_i$, en inversant le sens de communication entre les tâches $\mathcal{T}_i$ et $\mathcal{T}_j$. Le protocole utilise à nouveau deux pointeurs d'adresse : awl (*adress writer low*) pour la tâche écrivain de plus faible priorité et arh (*adress reader high*) pour la tâche lectrice de priorité supérieure. Ces pointeurs sont implantés dans le code du protocole donné en figure 5.15 par les cases de tableaux `awl[j][i]` et `arh[i][j]`. La figure 5.14 donne un exemple d'exécution du système à l'aide du protocole dans le cas *low to high* :

1. A la k-ième occurrence de la tâche $\mathcal{T}_i$, la dernière donnée produite par $\mathcal{T}_j$ est y_j^{p-1}. Cette exécution de $\mathcal{T}_j$ n'apparaît pas sur le diagramme et nous supposons que y_j^{p-1} est enregistrée à l'adresse 1 du double-buffer. La donnée que $\mathcal{T}_i^k$ doit lire est donc y_j^{p-2} en tenant compte du délai de communication nécessaire pour pouvoir obtenir une implantation qui préserve le flôt de données. Cette donnée est enregistrée à l'autre adresse du double-buffer, 0 en l'occurrence. À l'activation de $\mathcal{T}_i^k$, arh pointe sur l'adresse d'écriture précédente : $arh = 0$ et $awl = 1$. L'entrée x_i^k est donc bien éagle à la sortie x_j^{p-2}.

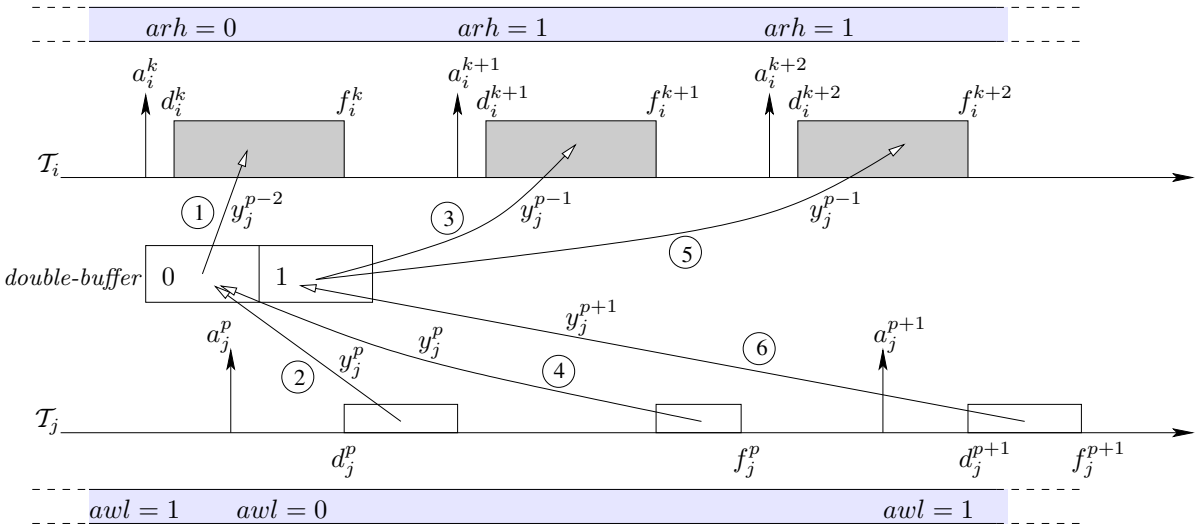


Fig. 5.14 – Exemple de communication où la tâche lectrice est prioritaire

2. Lorsque $\mathcal{T}_j$ est activée pour la p-ième fois, $\mathcal{T}_i^k$ n'a pas fini de s'exécuter et le pointeur d'adresse d'écriture bascule ($awl = 0$). Cependant la tâche $\mathcal{T}_j$ n'écrase pas la donnée y_j^{p-2} que $\mathcal{T}_i^k$ est en train de lire car la priorité de $\mathcal{T}_j$ est inférieure. $\mathcal{T}_j$ écrit y_j^p à l'adresse mémoire où $\mathcal{T}_i$ a lu ($arh = 0$ et $awl = 0$) mais une fois son exécution terminée. La donnée y_j^p est donc enregistrée à l'adresse 0 du double-buffer.
3. L'activation de $\mathcal{T}_i^{k+1}$ à l'instant a_i^{k+1} interrompt et suspend l'exécution courante de $\mathcal{T}_j$. Le pointeur de lecture bascule ($arh = 1$) et lit la donnée produite lors de la précédente exécution de $\mathcal{T}_j$. Cette donnée, y_j^{p-1}, est enregistré à l'adresse 1 du double-buffer. En conséquence, $\mathcal{T}_i^{k+1}$ lit l'avant dernière donnée produite par $\mathcal{T}_j$ conformément à la spécification de l'exécution dans le modéle idéal.
4. $\mathcal{T}_j^p$ reprend puis finit son exécution et enfin écrit ses résultats à l'adresse 1 du double-buffer car awl n'a pas changé.
5. La tâche $\mathcal{T}_i$ est de nouveau activée à l'instant a_i^{k+2}. L'avant dernière donnée produite par $\mathcal{T}_j$ à cet instant est la même qu'à l'instant a_i^{k+1}, y_j^{p-1} en l'occurrence, malgré le fait que $\mathcal{T}_j^p$ ait fini de s'exécuter. $\mathcal{T}_i^{k+2}$ lit la même valeur que $\mathcal{T}_i^{k+1}$.
6. L'événement a_j^{p+1} fait de nouveau basculer le pointeur d'adresse ($awl = 1$ désormais). $\mathcal{T}_j^{p+1}$ s'exécute sans interruption à l'instant où $\mathcal{T}_i^{k+2}$ a terminé (f_i^{k+2}) et écrit ses résultats à l'adresse 1, écrasant ainsi y_j^{p-1}. L'avant dernière donnée produite par $\mathcal{T}_j$ à l'instant f_j^{p+1} est y_j^p, enregistrée à l'adresse 0 du double-buffer.

Nous donnons en figure 5.16 le code qui implante l'algorithme qui permet aux tâches de communiquer entre elles au moyen d'un double-buffer. Ces double-buffers sont modélisés par deux tableaux à trois dimensions, B_{h2l} pour les communications de type *high to low* et B_{l2h} pour les communications de type *low to high*. La procédure `task()` permet à chaque tâche de communiquer avec l'ensemble des autres tâches du système en utilisant ces deux tableaux, selon son ordre de priorité avec chacune des autres tâches.

```
/* prio est un tableau qui associe un entier indiquant le niveau de
priorite a chaque tache de l'ensemble T */
int prio[T] ;
/* adresse ou une tache ecrit pour une tache de priorite superieure */
bool awl[T][T] ;
/* adresse ou une tache ecrit pour une tache de priorite inferieure */
bool awh[T][T] ;
/* adresse ou une tache lit en provenance d'une tache de priorite superieure */
bool arl[T][T] ;
/* adresse ou une tache lit en provenance d'une tache de priorite inferieure */
bool arh[T][T] ;
```

```
void os(taskid i){
   taskid w,r ;
   for(w = 0 ; w < t ; w++){
    arh[i][w] =!awl[w][i] ;
    arl[i][w] =  awh[w][i] ;
   }
   for(r = 0 ; r < t ; r++){
    awl[i][r] =!awl[i][r] ;
    if (arl[r][i] == awh[i][r])
      awh[i][r] =!awh[i][r] ;
   }
}
```

I_1: mise à jour des adresses de lecture pour la tâche i

I_2: mise à jour des adresses d'écriture pour la tâche i

Fig. 5.15 – Déclaration des variables d'adresse globales et le code C de la procédure os associée à l'événement a_i

La case de tableau `out[i]` implante la donnée de sortie y_i et la case `inp[j][i]` implante la donnée x_j prise en entrée par $\mathcal{T}_j$ en provenance de $\mathcal{T}_i$.

La figure 5.17 donne une représentation graphique des tableaux qui servent à implanter les double-buffers.

```
/* Bl2h[w][r] = double-buffer pour une communication entre une tache
ecrivain w et une tache lectrice r de priorite superieure */
data Bl2h[T][T][2] ;
/* Bh2l[w][r] = double-buffer pour une communication entre une tache
ecrivain w et une tache lectrice r de priorite inferieure */
data Bh2l[T][T][2] ;

data inp[T][T] ; // inp[r][w] = entree d'une tache r en provenance d'une tache w
data out[T] ; // out[i] = sortie de la tache i
                    void task(taskid i){
                       taskid w,r ;
                     ⎧ for(w = 0 ; w < t ; w++){
      I3: lecture    ⎪  if(prio[i] > prio[w])
   dans le buffer    ⎨    inp[i][w] = Bl2h[w][i][ lrh[i][w] ] ;
     des entrées     ⎪  else
  pour la tâche i    ⎪    inp[i][w] = Bh2l[w][i][ lrl[i][w] ] ;
                     ⎩  }
   I4: calcul des    {  out[i] = computation(i,inp[i]) ;
        résultats
     I5: écriture    ⎧ for(r = 0 ; r < t ; r++){
   dans le buffer    ⎨  Bl2h[i][r][ lwl[i][r] ] = out[i] ;
    des résultats    ⎪  Bh2l[i][r][ lwh[i][r] ] = out[i] ;
  pour la tâche i    ⎩  }
                      }
```

Fig. 5.16 – Déclaration des double-buffers et des variables globales d'entrée et de sortie et le code C de la procédure associée à l'événement d_i permettant les échanges des données

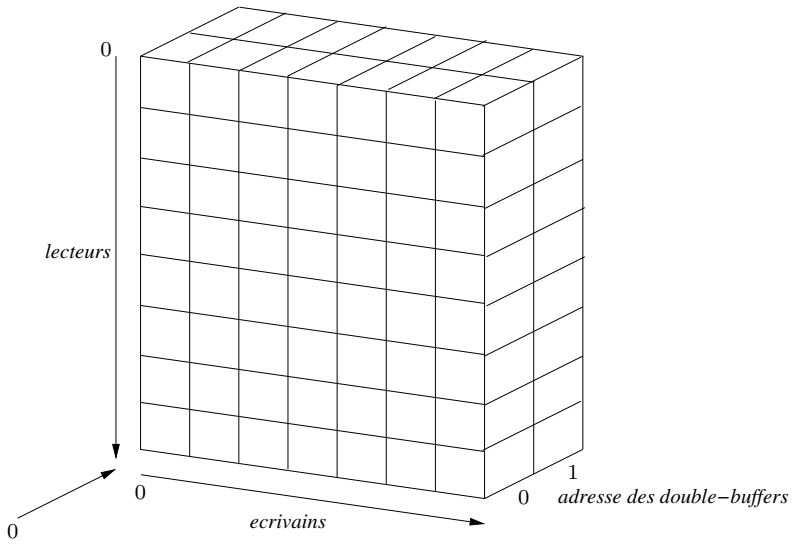


Fig. 5.17 – Représentation graphique des tableaux à trois dimensions de la structure de donnée des double-buffers B_{h2l} et B_{l2h}

5.4 Preuve de correction du DBP protocole

Les auteurs de ce protocole, en ont établi la validité au moyen du model-checker Lesar [Ray08]. Pour ce faire, les auteurs expliquent qu'il est correct de se ramener à un système comprenant seulement deux tâches, comme nous l'avons fait pour illustrer le mécanisme du protocole dans les exemples des sections précédentes, puis d'explorer toutes les traces d'exécution possibles grâce à un outil spécialisé dans ce genre d'exploration exhaustive [SC04]. Cette preuve de la validité du protocole repose donc sur la validité du model-checker Lesar, dont nous n'avons aucune garantie de la correction de son implantation. Valider le protocole par model-checking [SQ82] se heurte donc aux problèmes de confiances mentionnés dans le chapitre 1 (*cf.* section 1.4).

Pour atteindre le niveau de confiance requis par les autorités de certification, une preuve formelle vérifiable par machine du protocole pour un nombre variable de tâches, s'appuyant sur le code c proposé en figure 5.16 est exigée. La propriété de correction du protocole, assurant qu'il préserve le flôt de données de la modélisation idéale, se formule de la manière suivante, pour un ensemble de tâches T :

$$\begin{array}{l} \forall \sigma \in \Sigma(T), \forall w, r \in [1, |T|], \forall k, p \in \mathbb{N}, \\ \quad f_r^p \in \sigma \ \wedge \ a_w^k \prec_\sigma a_r^p \prec_\sigma a_w^{k+1} \Rightarrow \\ \qquad \bigwedge \left(\begin{array}{l} (p(\mathcal{T}_w) < p(\mathcal{T}_r) \ \wedge \ \mathcal{T}_w \xrightarrow{-1} \mathcal{T}_r) \Rightarrow (\texttt{inp[r][w]}^p = \texttt{out[w]}^{k-1}) \\ (p(\mathcal{T}_w) > p(\mathcal{T}_r) \ \wedge \ \mathcal{T}_w \rightarrow \mathcal{T}_r) \Rightarrow (\texttt{inp[r][w]}^p = \texttt{out[w]}^{k}) \end{array} \right) \end{array}$$

Prouver la correction du protocole via cette formule implique d'être en mesure de faire le lien entre les événements et le code du protocole dans le langage du vérificateur de preuve, de modéliser la concurrence entre les tâches dans ce langage, puis de raisonner sur des traces d'exécution de longueur arbitraire. Ce n'est pas ce que nous visons ici[1]. Ce que nous souhaitons, c'est étudier dans quelle mesure des outils instrumentés (comme

[1]Nous avons construit manuellement une preuve de ce protocole, vérifiable par un vérificateur de

Enkidu) peuvent être employés pour faciliter la certification d'un tel protocole developpé pour exécuter un système complexe. Nous reprenons donc la démarche proposée dans [SC04] pour justifier la correction du protocole : considérer les flôts de données entre deux tâches en faisant abstraction des autres tâches du système. Pour augmenter le niveau de confiance que l'on peut avoir dans cette justification, nous en avons produit une preuve vérifiable par Coq. Grâce à l'utilisation d'un outil instrumenté générant des certificats vérifiables par machine, nous sommes parvenus à certifier le protocole dans un système simplifié à deux tâches. De plus nous formalisons dans cette section une partie des raisonnements de [SC04], permettant de se ramener à deux tâches, plutôt que de se limiter à en donner l'intuition. Pour cela, nous utilisons le calcul de plus faible pré-condition pour l'affectation de tableaux, présenté en section 2.1.2.2.

5.4.1 Simplification du système pour raisonner sur deux tâches

Comme proposé dans [SC04], pour une communication entre deux tâches $\mathcal{T}_w$ (l'écrivain) et $\mathcal{T}_r$ (le lecteur), il est possible de simplifier le système et le protocole en ne considérant que ces deux tâches. En effet, les autres tâches du système, même de prioriété supérieure, ne peuvent perturber l'exécution de ces deux tâches au point de modifier à l'exécution les données qu'elles doivent s'échanger selon le flôt de données spécifié par la modélisation idéale du système. Une tâche $\mathcal{T}_z$ de priorité supérieure aux deux autres, peut certes interrompre et retarder les exécutions de $\mathcal{T}_w$ et $\mathcal{T}_r$, mais ne peut en aucun cas (1) ni écrire dans le double-buffer partagé par $\mathcal{T}_w$ et $\mathcal{T}_r$, (2) ni modifier le flôt de données de ces deux tâches en retardant l'exécution de l'une des deux. Le premier argument se prouve grâce au *calcul de plus faible pré-condition explicitant les alias de tableaux* (voir section 2.1.2.2) : pour tout autre tâche du système ($\mathcal{T}_z$ par exemple), il est possible de montrer formellement que toutes les affectations qu'elle effectuera lors de ses exécutions ne pourront altérer les données échangées entre $\mathcal{T}_i$ et $\mathcal{T}_j$. Le second argument vient directement de l'hypothèse de schédulabilité du système (*cf.* définition 5.2.2), que les auteurs du protocole font. Si le système est schédulable alors aucune interruption ne peut retarder une tâche au point qu'une de ses exécutions soit déclenchée sans que la précédente ait terminé. D'après cette hypothèse, la dernière (ou avant dernière) activation d'une tâche (permettant de déterminer les valeurs à échanger) lors de l'exécution d'une autre tâche ne peut différer du modèle idéal à l'exécution du fait des interruptions. Une preuve de cet argument s'obtient à partir de l'axiome de schédulabilité (*cf.* définition 5.2.2). Cette preuve nécessite de développer un système d'inférence pour raisonner sur le prédicat de précédence et ce n'est pas le propos de cette section ; nous ne détaillerons pas cette preuve ici.

Ces arguments nous permettent de diviser le système multi-tâches, afin de raisonner pour chaque communication sur un unique couple de tâches. Nous donnons pour une tâche $\mathcal{T}_z$, une preuve que toutes ses modifications du tableau $B_{h2l}[\,][\,][\,]$ n'affectent en rien les cases partagées par $\mathcal{T}_w$ et $\mathcal{T}_r$ ($B_{h2l}[w][r][0]$ et $B_{h2l}[w][r][1]$) :

preuves *ad hoc*, en utilisant une certaine axiomatisation du système. Cette preuve est décrite dans [GP08].

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\textit{par l'absurdité de l'hypothèse}\quad z = w}{\vdash_{\text{NJ}} e = x}}{\vdash_{\text{NJ}} \begin{array}{l}(z = w \wedge r = r \wedge awh[z][r] = 0)\\ \Rightarrow e = x\end{array}} \Rightarrow_i (H_2) \qquad \dfrac{\overbrace{\vdash_{\text{NJ}} B_{h2l}[w][r][0] = x}^{H_1}}{\vdash_{\text{NJ}} \begin{array}{l}(z \neq w \vee r \neq r \vee awh[z][r] \neq 0)\\ \Rightarrow B_{h2l}[w][r][0] = x\end{array}} \Rightarrow_i (H_3)}{\vdash_{\text{NJ}} \wedge \left(\begin{array}{l}(z = w \wedge r = r \wedge awh[z][r] = 0) \Rightarrow e = x\\ (z \neq w \vee r \neq r \vee awh[z][r] \neq 0) \Rightarrow B_{h2l}[w][r][0] = x\end{array}\right)} \wedge_e}{\vdash_{\text{NJ}} B_{h2l}[w][r][0] = x \Rightarrow \wedge \left(\begin{array}{l}(z = w \wedge r = r \wedge awh[z][r] = 0) \Rightarrow e = x\\ (z \neq w \vee r \neq r \vee awh[z][r] \neq 0) \Rightarrow B_{h2l}[w][r][0] = x\end{array}\right)} \Rightarrow_i (H_1)}{\vdash_{\text{H}} \{B_{h2l}[w][r][0] = x\}\texttt{Bh2l[z][r][awh[z][r]] := e}\{B_{h2l}[w][r][0] = x\}} \textit{sem}$$

La branche gauche de l'arbre de preuve est prouvée en exploitant l'absurdité de l'égalité $z = w$ puisque $\mathcal{T}_z$ désigne une troisième tâche de priorité supérieure à $\mathcal{T}_w$ et $\mathcal{T}_r$. Cette égalité a été introduite par le calcul de plus faible pré-condition, qui effectue la substitution de $B_{h2l}[w][r][0]$ par e dans la post-condition seulement si les trois conditions $z = w \wedge r = r \wedge awh[z][r] = 0$ sont vérifiées. Les tâches ayant toutes un numéro d'identifiant différent, z ne peut donc être égale à w. Ce cas est par conséquent absurde et se prouve trivialement. La branche droite représente le cas où il n'y pas d'alias entre la case du double-buffer affectée et celle de la post-condition. La case $B_{h2l}[w][r][0]$ reste donc inchangée, et cela se prouve en utilisant la pré-condition du triplet.

Pour être complet, il faudrait donner des preuves similaires pour la case $B_{h2l}[w][r][1]$, puis pour les autres affectations réalisées par la tâche $\mathcal{T}_z$. Ces preuves sont très semblables à celle présentée ci-dessus et nous ne les donnons pas ici. Ces preuves légitiment la réduction du système à deux tâches et la simplification du code du protocole. La suite de ce chapitre portera ainsi sur un couple de tâches $\mathcal{T}_w$ (l'écrivain) et $\mathcal{T}_r$ (le lecteur) représentatif de toutes les communications du système multi-tâches global.

5.4.1.1 Code du protocole simplifié

La réduction du système à deux tâches nous permet de supprimer d'une part les boucles dans le code du protocole puis de réduire les tableaux B_{h2l} et B_{l2h} à une dimension : les deux cases du double-buffer. Cela présente l'avantage que les tableaux à une dimension rentrent dans le champs d'application de l'analyseur Enkidu. Par contre, Enkidu ne sachant pas traiter les booléens, nous remplaçons les pointeurs d'adresse par des variables d'indice et la négation booléene $b :=!b$ doit être remplacée par l'instruction conditionnelle $\textit{if}(b = 0)\quad \textit{then}\quad b := 1\quad \textit{else}\quad b := 0$. La structure de donnée du système à deux tâches est déclarée en figure 5.18.

Un autre avantage, celui recherché initialement dans [SC04], est qu'il est possible désormais d'énumérer pour une longueur fixée de séquences, toutes les traces d'exécution possibles entre deux occurrences d'événements données. Cette énumération n'est pas applicable dans le cas général car beaucoup trop grande si nous considérons l'ensemble des tâches du système. À chaque trace d'exécution de cette énumération on associe le code correspondant aux événements des deux tâches. L'étude des programmes ainsi générés permettra de prouver que le protocole préserve le flôt de données attendu du système pour toutes les traces d'exécution possibles.

En ignorant les calculs effectués par une tâche pour produire ses données, nous pouvons associer à chaque événement une instruction du protocole. Les programmes correspondants aux traces d'exécution s'obtiennent alors en remplaçant chaque événement par l'instruction qui lui est associée.

```
/* adresse ou une la tache w ecrit pour une tache r de priorite superieure */
int awl ;
/* adresse ou une tache w ecrit pour une tache r de priorite inferieure */
int awh ;
/* adresse ou une tache r lit en provenance d'une tache w de priorite superieure */
int arl ;
/* adresse ou une tache r lit en provenance d'une tache w de priorite inferieure */
int arh ;
/* double-buffer pour une communication entre une tache
ecrivain w et une tache lectrice r de priorite superieure */
int Bl2h[2] ;
/* double-buffer pour une communication entre une tache
ecrivain w et une tache lectrice r de priorite inferieure */
data Bh2l[2] ;
data inp ; // entree de la tache r
data out ; // sortie de la tache w
```

Fig. 5.18 – Structure de donnée du protocole pour un système à deux tâches

Nous détaillons maintenant le code du protocole à deux tâches en faisant abstraction des calculs effectués. En effet, les calculs des données de sortie ne sont pas pertinents pour la correction du système qui concerne seulement les flôts de données. Notre objectif est de prouver automatiquement la correction du protocole, pour chaque trace d'exécution possible vérifiant les hypothèses de la propriété de correction, en utilisant la version instrumentée d'Enkidu.

Code du protocole à deux tâches dans le cas *high to low*. Dans le cas d'une communication *high to low* $\mathcal{T}_w \rightarrow \mathcal{T}_r$, lorsque la tâche $\mathcal{T}_w$ est déclenchée par l'événement a_w, alors nous exécutons la permutation du pointeur d'adresse à laquelle $\mathcal{T}_w$ doit écrire si arl est égale à awh :

$a_w \longrightarrow$
```
if(arl == awh)
  then if(awh == 0)
         then awh = 1
         else awh = 0
```

Le lecteur de plus faible priorité, $\mathcal{T}_r$, est quant à lui déclenché par l'événement a_r. Nous associons à cet événement le code qui permet à cette tâche de lire la dernière donnée écrite par $\mathcal{T}_w$ en affectant au pointeur d'adresse de lecture arl la valeur courante du pointeur d'adresse d'écriture awh :

$a_r \longrightarrow$ `arl = awh`

L'exécution de la tâche $\mathcal{T}_r$ consiste à lire les données présentes dans le double-buffer à l'adresse arl :

$d_r \longrightarrow$ `inp = Bh2l[arl]`

L'exécution de la tâche $\mathcal{T}_w$ consiste à écrire dans le double-buffer les données calculées. Nous faisons abstraction du calcul de la valeur de `out`.

d_w	$\longrightarrow$	`Bh2l[awh] = out`

Les événements f dans les deux cas, ne font qu'exprimer la fin de l'exécution de la tâche et n'ont pas d'instruction associée.

5.4.1.2 Propriété de correction dans le cas à deux tâches

Nous formalisons maintenant la propriété de correction du protocole, assurant que le flôt de données du modèle idéal est préservé à l'exécution, en ne considérant que deux tâches. Une chose importante est de distinguer les occurrences des variables d'entrée et de sortie de chaque tâche pour exprimer la préservation du flôt de données. Les occurences des variables d'entrée et de sortie sont spécifiées explicitement dans le nom des variables scalaires qui sont créées dymamiquement lors de l'énumération des traces d'exécution possibles. Pour la k-ième exécution d'une tâche, son résultat de sortie est implanté par la variable scalaire `out_k` (de même la p-ième entrée d'une tâche est implantée par la variable scalaire `inp_p`). La propriété de correction, assurant la préservation du flôt de données se simplifie donc de la manière suivante :

Dans le cas *high to low*, $\mathcal{T}_w \rightarrow \mathcal{T}_r$:

$$\forall \sigma \in \Sigma(T), \forall k, p \in \mathbb{N},$$
$$p(\mathcal{T}_w) > p(\mathcal{T}_r) \ \wedge \ f_r^p \in \sigma \ \wedge \ a_w^k \prec_\sigma a_r^p \prec_\sigma a_w^{k+1} \Rightarrow \texttt{inp\_p} = \texttt{out\_k}$$

Dans le cas *low to high*, $\mathcal{T}_w \xrightarrow{-1} \mathcal{T}_r$:

$$\forall \sigma \in \Sigma(T), \forall k, p \in \mathbb{N},$$
$$p(\mathcal{T}_w) < p(\mathcal{T}_r) \ \wedge \ f_r^p \in \sigma \ \wedge \ a_w^{k+1} \prec_\sigma a_r^p \prec_\sigma a_w^{k+2} \Rightarrow \texttt{inp\_p} = \texttt{out\_k}$$

Nous avons donné ci-dessus la propriété de correction pour les deux types de communication (selon l'ordre de priorité entre le lecteur et l'écrivain). Les raisonnements sont simillaires dans les deux cas et la suite se concentrera sur les communications de type *high to low*.

5.4.2 Calcul de la sémantique du protocole à deux tâches

Les traces d'exécution σ qui vérifient les hypothèses de la propriété de correction, à savoir $f_r^p \in \sigma$ signifiant que l'exécution de la tâche lectrice termine et $a_w^k \prec_\sigma a_r^p \prec_\sigma a_w^{k+1}$ signifiant que la dernière donnée produite par $\mathcal{T}_w$ lors de l'exécution de $\mathcal{T}_r^p$ est y_w^k (qui est implantée par `out_k`), sont énumérables. Nous utilisons l'automate donné en figure 5.5 pour générer l'ensemble de ces traces d'exécution. À chacune des ces séquences d'événements correspond un programme qui doit vérifier la propriété de correction. Pour chaque exécution d'une tâche, le code correspondant utilise des variables scalaires dont le nom porte le numéro de l'occurrence de cette tâche. De plus, pour chaque trace d'exécution, Enkidu nous donne, par interprétation abstraite, automatiquement la propriété souhaitée `inp_p` = `out_k`, associée au point de contrôle final du code qui correspond

à la trace. Pour obtenir une preuve formelle que cette propriété est garantie, pour chaque trace d'exécution, il suffit de lancer la version d'Enkidu instrumentée, sur chacun des programmes obtenus par l'énumération des séquences d'exécution possibles entre deux tâches.

Nous détaillons maintenant comment nous énumérons l'ensemble des trace d'exécution possible selon les contraintes posées par la propriété de correction que nous venons de donner, puis les propriétés de tableaux calculées par Enkidu desquelles nous pouvons déduire l'égalité qui nous intéresse.

5.4.2.1 Calcul des séquences d'exécution possibles

Pour calculer la sémantique du protocole dans un système comprenant une unique tâche en lecture et une unique tâche en écriture, il nous faut distinguer les différentes configurations du système provenant des différents ordonnancements des événements. À un ordonnancement correspond un programme qui comporte des instructions en provenance des deux tâches. L'automate proposé en figure 5.5 nous permet dans un premier temps de générer les traces d'exécution, puis les programmes correspondants s'obtiennent en remplaçant chaque événement par le code qui lui est associé (*cf.* section 5.4.1.1). En partant de l'événement a_w^k, nous donnons en figure 5.19 l'arbre des traces d'exécution possibles dans le cas *high to low*. Sur cette figure nous avons coloré en rouge une trace d'exécution particulière où l'événement a_r^{p+1} est déclenché sans que a_w^{k+1} ne l'ait été. Il est possible de poursuivre cette exécution par une séquence aussi longue que l'on veut des événements $a_r; d_r; f_r$. Qu'importe, ce qui nous intéresse, c'est ce qu'il se passe entre a_w^k et f_r^p, car ce sont ces deux événements qui déterminent le flôt de données spécifié dans la propriété de correction. Si a_r^{p+1} est déclenché avant a_w^{k+1} le flôt de donné pour les exécutions de $\mathcal{T}_r^p$ et $\mathcal{T}_w^k$ est le même. La propriété de correction précise que $a_r^p \prec_\sigma a_w^{k+1}$ pour spécifier que la dernière donnée produite par $\mathcal{T}_w$ à l'instant a_r^p est y_w^k. Ce qu'il se passe entre a_r^p et a_w^{k+1} n'a pas d'importance. De plus, du fait que $\mathcal{T}_r$ soit en lecture, les exécutions de $\mathcal{T}_r$ postérieures à $\mathcal{T}_r^p$ n'écrivent pas dans le double-buffer utilisé pour les communications de $\mathcal{T}_w$ vers $\mathcal{T}_r$. Nous en faisons donc abstraction et la séquence colorée en rouge en figure 5.19 est ainsi confondue avec les autres traces d'exécution qui passent par le noeud coloré. Les programmes que nous générons ne correspondent donc qu'aux événements compris entre a_w^k et f_r^p. Le point de contrôle final des systèmes de transitions correspondants à ces programmes est celui qui suit l'instruction associée à f_r^p.

5.4.2.2 Interprétation abstraite du code protocole

Nous allons ici utiliser Enkidu (*cf.* chapitre 4) pour interpréter le code du protocole, afin d'en dégager une approximation de sa sémantique (*cf.* section 3.2.1). Afin de pouvoir exécuter Enkidu sur les programmes obtenus il nous faut donner la partition de tableau qui permet de définir le domaine abstrait utilisé par Enkidu. Nous rappelons qu'Enkidu n'admet pas un unique domaine abstrait, mais une famille de domaines paramétrés par la partition des tableaux utilisés par les programmes. Dans le cas du protocole pour un système à deux tâches, le programme n'utilise dans chaque schéma de communication (*high to low* ou *low to high*) qu'un seul tableau à deux cases mémoire et deux pointeurs d'adresse *awh* et *arl* (dans le cas *high to low*) implantés par deux variables d'indice (*cf.* figure 5.18). Nous avons donc défini la partition selon les deux pointeurs d'adresse *awh* et *arl* en cinq tranches de la façon suivante :

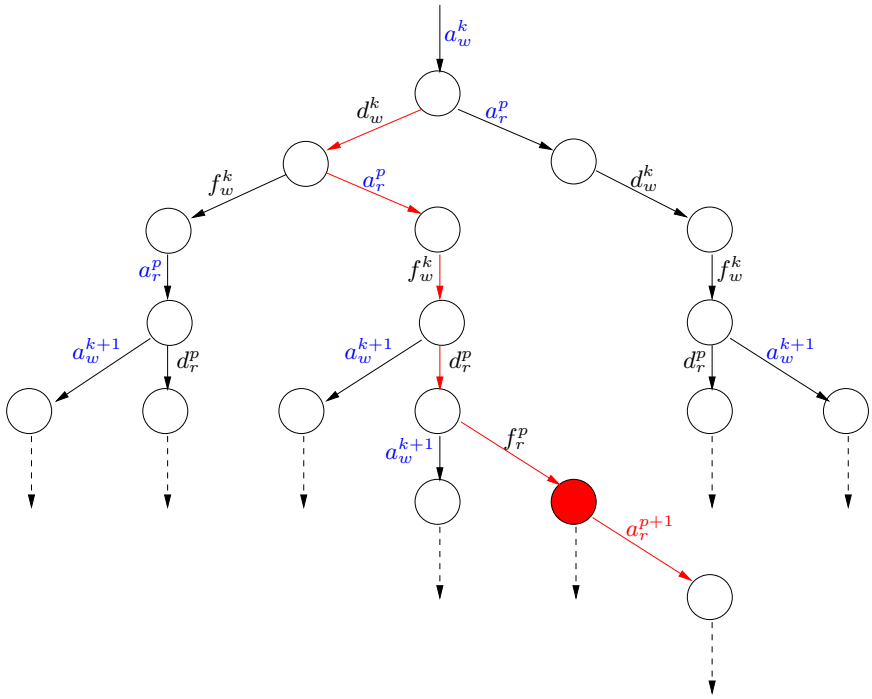


Fig. 5.19 – Sequence d'exécution possible à deux tâches dans le cas $h2l$

$$P \stackrel{def}{=} \begin{cases} \varphi_1 \stackrel{def}{=} (\ell = awh \wedge \ell = arl), \\ \varphi_2 \stackrel{def}{=} (\ell = awh \wedge \ell > arl), \\ \varphi_3 \stackrel{def}{=} (\ell = awh \wedge \ell < arl), \\ \varphi_4 \stackrel{def}{=} (\ell = arl \wedge \ell > awh), \\ \varphi_5 \stackrel{def}{=} (\ell = arl \wedge \ell < awh), \end{cases}$$

Remarque. *Parmi ces cinq tranches certaines peuvent être vides selon le contexte. Par exemple, dans le contexte $awh = arl$, les tranches $\varphi_2, \varphi_3, \varphi_4$ et φ_5 sont vides, dans le sens où elles sont insatisfiables. Nous avons choisi cependant cette partition car elle couvre de façon exhaustive toutes les configurations possibles selon les valeurs des deux variables d'indice.*

À partir de cette partition Enkidu calcule pour chaque programme une propriété de tableaux à chaque point de contrôle. Selon les différentes traces d'exécution, les propriétés de tableaux ne sont pas les mêmes car les variables du système ne sont pas modifiées dans le même ordre. Cependant, pour toutes les traces d'exécution, le point de contrôle final admet l'égalité `inp_p` = `out_k`.

Exemple 5.4.1. *Soit la trace d'exécution $a_w^k; a_r^p; d_w^k; f_w^k; d_r^p; f_r^p$ à laquelle est associé le programme suivant, que nous avons décoré d'une partie des invariants calculés par* Enkidu *:*

`if(arl == awh)`	$\{arl = awh \wedge \forall \ell, \bigwedge_{k=1}^{5} \varphi_k \Rightarrow \top_{zone}\}$
`then if(awh == 0)`	$\{awh = 0 \wedge arl = 0 \wedge \ldots\}$
`then awh = 1`	$\{awh = 1 \wedge arl = 0 \wedge \ldots\}$
`else awh = 0`	$\{awh = 0 \wedge arl = 1 \wedge \ldots\}$
	$\{\top_{\mathscr{A}(P)}\}$
`arl = awh ;`	$\{arl = awh \wedge \forall \ell, \bigwedge_{k=1}^{5} \varphi_k \Rightarrow \top_{zone}\}$
`Bh2l[awh] = out_k ;`	$\{arl = awh \wedge \forall \ell, \bigwedge \begin{pmatrix} \varphi_1 \Rightarrow B_{h2l}[\ell] = out\_k \\ \bigwedge_{k=2}^{5} \varphi_k \Rightarrow \top_{zone} \end{pmatrix}\}$
`inp_p = Bh2l[arl]`	$\{\Psi\}$

avec $\Psi \stackrel{def}{=} \bigwedge \begin{pmatrix} arl = awh \\ inp\_p = out\_k \end{pmatrix} \wedge \forall \ell, \bigwedge \begin{pmatrix} \varphi_1 \Rightarrow B_{h2l}[\ell] = out\_k \wedge inp\_p = B_{h2l}[\ell] \\ \bigwedge_{k=2}^{5} \varphi_k \Rightarrow \top_{zone} \end{pmatrix}$

En observant ce programme, il est facile de se rendre compte que l'égalité à démontrer, `inp_p = out_k`*, est vérifiée : les deux dernières affectations et l'égalité entre arl et awh, nous donne par transitivité l'égalité entre* `inp_p` *et* `out_k`*. Cette égalité est calculée par* Enkidu *et apparaît dans le contexte de la propriété de tableaux* Ψ *en sortie de programme.* Enkidu *calcule des informations supplémentatires qui ne sont pas pertinentes pour la propriété de correction du protocole. Ceci vient du fait qu'*Enkidu *est un analyseur statique dont le but est de découvrir le plus d'informations possibles, contrairement aux outils de vérification qui se focalisent spécifiquement sur la propriété à valider. Ces informations superflus viennent de l'accumulation des propriétés découvertes aux autres points de contrôle et qui sont nécessaires pour prouver automatiquement qu'une fois ce code exécuté, l'égalité entre* `inp_p` *et* `out_k` *est vérifiée. Ces informations inutiles aux point de sortie peuvent être éliminées lors de la génération de la preuve comme proposé dans [SYYH07].*

Enkidu associe donc pour toutes les traces d'exécution, l'égalité `inp_p = out_k` au point de contrôle final du programme qui lui correspond. La version instrumentée d'Enkidu produit pour chaque programme la preuve de cette égalité – parmi de nombreuses autres preuves d'invariants qui ne nous intéressent pas. Dans la section suivante, nous donnons l'arbre de dérivation construit par notre version instrumentée d'Enkidu qui permet d'établir formellement pour une trace d'exécution la correction du protocole.

5.4.3 Certification de la correction du protocole

La sémantique du protocole calculée par Enkidu est prouvée formellement grâce à l'instrumentation que nous avons développée aux chapitre 4. Cela nous permet dans le cas du système restreint à deux tâches de prouver que le flôt de données est préservé pour tout ordonnancement possible du système. Nous donnons ci-dessous la preuve générée par Enkidu pour la trace d'exécution donnée à l'exemple 5.4.1 ($a_w^k; a_r^p; d_w^k; f_w^k; d_r^p; f_r^p$ en l'occurrence). Cette preuve est donnée transition par transition, en partant du point de sortie du programme. Dans les arbres de preuves ci-dessous, nous omettons une partie des propriétés calculées par Enkidu pour faciliter la lecture. Le triplet de Hoare $\{arl = awh \wedge$

$\forall \ell, \bigwedge \left(\begin{array}{l} \varphi_1 \Rightarrow B_{h2l}[\ell] = out_k \\ \bigwedge_{k=2}^{5} \varphi_k \Rightarrow \top_{zone} \end{array} \right)\}$ `inp_p = Bh2l[arl]`$\{\Psi\}$ est simplifié par $\{\forall \ell, \varphi_1 \Rightarrow B_{h2l}[\ell] = out_k\}$`inp_p := Bh2l[arl]`$\{inp_p = out_k\}$, qui permet de se focaliser sur la partie de la propriété de tableaux qui nous intéresse :

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\text{NJ}} arl = arl}\,{=_{refl}} \qquad \dfrac{\dfrac{par\ \ (\forall \ell, \varphi_1 \Rightarrow B_{h2l}[\ell] = out_k) \sqsubseteq^{\pi}_{\mathscr{A}_{gen}} (\forall \ell, \ell = arl \Rightarrow B_{h2l}[\ell] = out_k)}{\vdash_{\text{NJ}} \forall \ell, \ell = arl \Rightarrow B_{h2l}[\ell] = out_k}}{\vdash_{\text{NJ}} arl = arl \Rightarrow B_{h2l}[arl] = out_k}\,\forall_e[\ell/arl]}{\vdash_{\text{NJ}} B_{h2l}[arl] = out_k}\,\Rightarrow_e}{\vdash_{\text{NJ}} \forall \ell, \varphi_1 \Rightarrow B_{h2l}[\ell] = out_k \Rightarrow B_{h2l}[arl] = out_k}\,\Rightarrow_i}{\vdash_{\text{H}} \{\forall \ell, \varphi_1 \Rightarrow B_{h2l}[\ell] = out_k\}\texttt{inp_p := Bh2l[arl]}\{inp_p = out_k\}}\,sem$$

La transition qui précède celle ci-dessus est l'affectation `Bh2l[awh] = out_k`. Nous simplifions également pour cette transition les propriétés de tableaux découvertes par Enkidu :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash_{\text{NJ}} out_k = out_k}\,{=_{refl}}}{\vdash_{\text{NJ}} \varphi_1 \Rightarrow out_k = out_k}\,\Rightarrow_i}{\vdash_{\text{NJ}} \ell = awh \Rightarrow (\varphi_1 \Rightarrow out_k = out_k)}\,\Rightarrow_i \qquad \dfrac{\dfrac{\dfrac{\dfrac{\text{par la contradiction de } \ell \neq awh \text{ et } \varphi_1}{\vdash_{\text{NJ}} \bot}}{\vdash_{\text{NJ}} B_{h2l}[\ell] = out_k}\,\bot_e}{\vdash_{\text{NJ}} \varphi \Rightarrow B_{h2l}[\ell] = out_k}\,\Rightarrow_i}{\vdash_{\text{NJ}} \ell \neq awh \Rightarrow (\varphi \Rightarrow B_{h2l}[\ell] = out_k)}\,\Rightarrow_i}{\vdash_{\text{NJ}} \begin{array}{l} \ell = awh \Rightarrow (\varphi_1 \Rightarrow out_k = out_k) \\ \wedge \\ \ell \neq awh \Rightarrow (\varphi_1 \Rightarrow B_{h2l}[\ell] = out_k) \end{array}}\,\wedge_i}{\vdash_{\text{NJ}} \forall \ell, \left(\begin{array}{l} \ell = awh \Rightarrow (\varphi_1 \Rightarrow out_k = out_k) \\ \wedge \\ \ell \neq awh \Rightarrow (\varphi_1 \Rightarrow B_{h2l}[\ell] = out_k) \end{array} \right)}\,\forall_i}{\vdash_{\text{NJ}} \top \Rightarrow \forall \ell, \left(\begin{array}{l} \ell = awh \Rightarrow (\varphi_1 \Rightarrow out_k = out_k) \\ \wedge \\ \ell \neq awh \Rightarrow (\varphi_1 \Rightarrow B_{h2l}[\ell] = out_k) \end{array} \right)}\,\Rightarrow_i}{\vdash_{\text{H}} \{\top\}\texttt{Bh2l[awh] = out_k}\{\forall \ell, \varphi_1 \Rightarrow B_{h2l}[\ell] = out_k\}}\,sem$$

Les scripts de tactiques produits par Enkidu sur l'ensemble des traces d'exécution possibles du système avec les propriétés de tableaux complètes sont donnés en annexe.

5.5 Conclusion

L'objectif de ce chapitre était d'illustrer l'usage d'un outil de certification obtenu par instrumentation d'un analyseur statique pour la certification formelle d'un système complexe destiné à l'avionique. Nous avons montré comment exploiter la version instrumentée d'Enkidu pour automatiser une partie de la construction de la preuve de correction. Cette preuve porte sur un système simplifié comprenant un unique couple de tâches communicant entre elles. Nous avons justifié cette simplification du système en utilisant le calcul de plus faible pré-condition pour l'affectation des tableaux afin de

prouver qu'il est correcte de ne considérer que deux tâches pour prouver la correction du protocole. Nous avons montré formellement que pour un couple de tâches communicant, toute autre tâche, de priorité inférieure, intermédiaire ou supérieure, ne peut perturber cette communication au point de changer le flôt de données du système entre le modèle idéal et son implantation. La certification du protocole a donc été effectuée sur un système à deux tâches dont l'implantation a également été simplifiée (au niveau de sa structure de données notamment). Pour ce système simplifié, il est possible d'énumérer les traces d'exécution possibles conformes aux hypothèses de la propriété de correction. Pour chacune de ces traces d'exécution nous avons généré un code permettant d'exécuter le système selon cet ordonnancement particulier. Puis nous avons utilisé Enkidu pour calculer une approximation de la sémantique du système dans chaque cas. Pour chacune de ces traces d'exécution, Enkidu a calculé une propriété de tableaux qui contient (entre autres) l'égalité formulée dans la propriété de correction du protocole. Nous avons ensuite utilisé la version instrumentée d'Enkidu pour certifier la correction du protocole dans chacune des configurations possibles selon l'ordonnancement des événements du système. Nous avons ainsi certifié formellement par un ensemble de preuves, toutes vérifiées par Coq, que le protocole permet d'implanter correctement le système multi-tâches en préservant le flôt de données de la modélisation idéale du système. Au delà de cette étude de cas, certes simplifiée pour s'adapter aux capacités d'analyse de l'outil Enkidu, il nous paraît important de mettre en avant la faisabilité de l'utilisation d'outils de certification automatiques pour assurer le bon fonctionnement de programmes embarqués. Bien sûr, pour s'attaquer à des protocoles plus réalistes et encore plus complexes il faudrait étendre la démarche à des analyseurs avec un champs d'application plus large qu'Enkidu (capable de traiter des tableaux à plusieurs dimension notamment). Automatiser la construction des preuves de correction de l'implantation de tel systèmes embarqués est un enjeu économique majeur et notre approche permet de développer des outils de certification à partir d'analyseurs statiques existants à un coût raisonnable.

Chapitre 6

Conclusions et Perspectives

> « *Tout ce qui proprement peut être dit, peut être dit clairement. Et sur ce dont on ne peut parler, il faut garder le silence* »
>
> Ludwig Wittgenstein

6.1 Bilan sur les contributions de nos travaux

L'objectif principal de ces travaux de thèse était de certifier automatiquement des programmes impératifs dans un système digne de confiance tel que le système Coq [BC04]. Le système Coq est à la fois un logiciel intéractif de construction de preuves formelles, et un vérificateur automatique de preuves formelles. Construire des preuves formelles intéractivement à l'aide du système est extrêmement fastidieux. Nous avons donc cherché un moyen de construire ces preuves automatiquement dans le cadre qui nous intéresse : certifier la correction des programmes. Dans le cadre de la certification de programmes, il est nécessaire de faire le lien entre la sémantique des programmes, et la logique du vérificateur de preuves [CH88]. Pour faire ce lien, nous avons adopté la technique de Floyd-Hoare [Flo67, Hoa69], qui consiste à spécifier chaque instruction du programme par une pré et une post-condition. Pour transformer ces spécifications des instructions de programmes en termes purement logiques, nous avons utilisé un calcul de plus faible pré-condition [Dij75].

Afin de générer automatiquement ces certificats de programmes, nous avons opté pour une approche par instrumentation de méta-programmes, initié par les travaux de [PSS98, Nam01]. L'instrumentation consiste à étendre le code des programmes de façon à ce qu'ils génèrent des justifications sous forme de preuves formelles de leurs résultats. La technique d'instrumentation que nous avons proposée s'appuie sur des patrons de preuve que nous associons à certaines fonctions du méta-programme considéré. Les méta-programmes auxquels nous nous sommes interessés, sont des analyseurs statiques par interprétation abstraite. Les interpréteurs abstraits représentent un large classe d'outils dans le domaine de la validation de programmes. Ces analyseurs calculent des invariants inductifs de programmes et sont de bons candidats pour être instrumentés car ils présentent deux phases bien distinctes : la première consiste à calculer des invariants par un calcul de point-fixe ; la seconde vérifier que ces invariants sont stable. Nous avons pu montrer que toute la partie concernant la recherche des invariants n'est pas utile pour produire des certificats de correction de ces invariants. Concernant la phase de

vérification de la stabilité des invariants calculés, il s'est avéré que tout n'est pas non plus à instrumenter. Nous avons étudier la structures des interpréteurs abstraits afin de dégager, de manière générale, quelles parties de leur code nécessitent d'être instrumentées pour qu'ils puissent générer des certificats de leurs analyses. Selon les cas, nous avons montré qu'il est possible de se limiter à l'instrumentation des opérateurs ensemblistes du domaine abstrait utilisé par l'analyse. En effet, nous avons mis en évidence le fait que dès lors que le calcul de plus faible pré-condition préserve les invariants en post-conditions dans le domaine abstrait de l'analyse, le nombre de fonctions à instrumenter est très raisonnable. Seuls les opérateurs ensembliste et la fonction permettant d'abstraire les gardes vers le domaine abstrait nécessitent d'être instrumentés pour assurer que chaque exécution correcte d'un analyseur retournera un certificat de son résultat. Contrairement aux approche proposées par [SYY03, Cha06] nous avons montré qu'il n'est donc pas utile de systèmatiquement instrumenter les fonctions de transfert des analyseurs statiques. Limiter l'instrumentation aux opérateurs ensemblistes du domaine abstrait présente l'avantage d'être portable (d'une analyse à une autre utilisant le même domaine abstrait), et de ne pas avoir à se pencher sur les fonctions de transfert de l'analyse. Ce dernier point, concernant les fonctions de transfert, peut parraître anecdotique, mais présente une contribution importante du point de vue de la mise en application de cette approche de certification par instrumentation.

Dans nos travaux, les certificats que les interpréteurs abstraits instrumentés produisent sont des preuves formelles données dans un format vérifiable par le système Coq. L'un des objectifs de cette thèse, était d'automatiser la construction des preuves de correction de programmes dans le système Coq, afin de bénéficier de la fiabilité de son vérificateur de preuves, sans souffrir de son manque d'efficacité pour construire les preuves. La première étape a été de prendre en main ce logiciel afin de comprendre la structure des preuves vérifiables par Coq, des termes du cic en l'occurrence. Si générer directement des termes du cic semble difficile, l'utilisation de patrons de preuve, traduits à l'exécution par des scripts de tactiques, s'est avéré beaucoup plus simple. En effet, l'environnement dédié à la preuve assistée du système Coq traduit des séquences de tactiques en des termes du cic. Une tactique formalise un (ou plusieurs) pas de raisonnement formelle (dans des systèmes d'inférences comme ceux de G.Gentzen [Pra65]). Pour chaque raisonnement utilisé dans nos patrons de preuve, il suffit de lui associer la tactique qui lui correspond. Si pour un raisonnement donné une telle tactique n'existe pas déjà dans le système Coq, il suffit d'en développer une avec le langage Ltac, que fourni Coq.

Pour évaluer la faisabilité de notre approche, nous avons mené une instrumentation sur un prototype d'analyse statique de programmes manipulant des tableaux, Enkidu [HP08], développé au laboratoire Verimag. Le domaine abstrait de cette analyse permet d'exprimer des informations sur le contenu des tableaux de programmes et est paramétré par une partition des cases de ces tableaux. Pour de nombreuses instructions analysables par Enkidu, le calcul de plus faible pré-condition ne préserve pas les invariants dans le domaine abstrait. Nous avons donc entrepris dans un premier temps une instrumentation des fonctions de transfert des instructions en question, en suivant l'approche proposé dans [SYY03]. Ce travail fut long et difficile car il a fallu, avant d'instrumenter, comprendre le fonctionnement et les comportements de nombreuses heuristiques utilisées par ces fonctions de transfert. Cela nous a mené à des performances médiocres aux niveau du temps de vérification des certificats et nous avons fait le constat que cette approche est

difficilement applicable en pratique.

Dans un second temps, nous avons réalisé que si le calcul de plus faible pré-condition ne préservait pas les invariants dans le domaine abstrait utilisé par Enkidu, il était possible de les normaliser afin de pouvoir utiliser l'instrumentation des opérateurs ensemblistes pour achever les preuves. Nous avons identifié trois cas concernant les formules retournées par le calcul de plus faible pré-condition : (1) la formule appartient au domaine abstrait et la preuve est obtenue par une instanciation des patrons de preuve associés aux opérateurs ensemblistes du domaine ; (2) la formule n'appartient pas au domaine abstrait mais il est possible de la normaliser vers le domaine abstrait et de finir la preuve comme dans le premier cas ; (3) la formule n'appartient pas au domaine abstrait et il n'est pas possible de la normaliser vers le domaine abstrait. Il n'est donc pas possible dans le troisième cas de se passer des informations apportées par les fonctions de transfert, et celles-ci doivent être instrumentées. Nous n'avons pas été confronté au troisième cas lors de l'instrumentation d'Enkidu. Le travail d'instrumentation s'est donc limité aux opérateurs ensemblistes du domaine. Cette expérience confirme donc deux choses : instrumenter des interpréteurs abstraits avec des patrons de preuve est faisable en pratique à condition de pouvoir éviter d'instrumenter les fonctions de transfert ; instrumenter les opérateurs ensemblistes limite le coût de cette instrumentation et la rend d'autant plus portable. Nous voyons en cela un moyen intéressant et efficace pour permettre à des outils à l'état de prototype d'atteindre des niveaux de confiance dès plus élevés (celui du vérificateur de preuves du système Coq).

La dernière partie de ces travaux de thèse, montre comment exploiter un analyseur statique certifiant ses résultats, tel Enkidu, pour certifier des programmes utilisés dans des systèmes critiques. Notre objectif était de certifier un protocole de communication pour système multi-tâches, pour lequel il n'existe pas de méta-programme permettant de l'analyser, de vérifier sa correction et donc encore moins de la certifier. Les programmes utilisés dans l'industrie sont des systèmes complexes mélangeant plusieurs paradigmes de programmation (parallélisme, synchronisation, *etc*) et diverses structures de données (pointeurs, tableaux à plusieurs dimension, tableaux de tableaux, *etc*). Pour ce genre de programme, il n'existe presque jamais d'analyseurs statiques capables de les traiter entièrement. Les analyseurs statiques, pour être efficaces, ont des domaines d'entrées très spécifiques, et comprennent de nombreuses heuristiques qui limite la forme des programmes qu'ils sont en mesure de traiter. Néanmoins, s'il n'est pas possible à l'heure actuelle de certifier automatiquement un tel protocole, nous avons souhaité éviter de le certifier interactivement à l'aide de l'assistant à la preuve Coq, car cette tâche aurait été extrêmement fastidieuse. Nous avons donc étudier ce protocole de manière à en dégager les parties que nous pouvons déleguer à Enkidu. En reprenant les simplifications proposer par les auteurs de protocole [SC04] afin de justifier (non par des preuves formelles) la correction de leur protocole, nous avons pu automatiser certaines parties de la construction de la preuves de correction. Nous avons réalisé cette expérience dans le but de montrer que même si les outils intrumentés ne peuvent traiter l'intégralité d'un problème d'une certaine complexité, ils peuvent tout de même être mis à contribution pour facilité la résolution de ce problème. Cette étude montre, en plus des diverses architectures de *proof-carrying code*, que l'utilisation des méta-programmes certifiant leurs résultats peut être appliquée dans le cadre de la certification de systèmes critiques.

6.2 Perspectives

Une première perspective que nous avons déjà mentionnée en section 4.5.2 concerne l'implantation de l'instrumentation d'Enkidu et les différentes optimisations que nous pourrions lui apporter. En effet, un certain nombre de sous-buts triviaux sont prouvés par des arbres de dérivations inutilement grands dans ces cas, du fait de la dimension statique des patrons de preuve que nous avons développés. La structure de ces patrons de preuve s'avère optimale dans de nombreuses situations, mais pas pour certains cas triviaux. Il serait envisageable d'ajouter des patrons de preuve de taille inférieure pour prouver ces sous-buts triviaux lorsque nous les reconnaissons. Ainsi pour une fonction instrumentée donnée, il serait interressant de développer un ensemble de patrons de preuve permettant de prouver la correction de chaque résultat de cette fonction avec une taille minimale. La version actuelle de notre instrumentation d'Enkidu associe un seul patron de preuve à chaque fonction instrumentée (ces fonctions sont l'implantation des opérateurs ensemblistes du domaine abstrait). Concernant la taille des preuves, de nombreuses coupures ou redondances sont présentes du fait qu'il nous est impossible de déterminer statiquement la structure d'une preuve issue de plusieurs patrons de preuve (ou d'assemblages récursifs d'un même patron de preuve). Une solution envisageable, est d'intégrer à la traduction des preuves en scripts de tactiques Coq des simplifications afin d'éliminer une partie de ces coupures. Pour les parties redondantes nous proposons de générer des lemmes accompagnés de leur preuve vérifiés une fois préalable par Coq. Pour chaque dérivation redondante, l'appel au lemme correspondant permettrait de réduire la taille des preuves. Cette réduction de la taille est essentiel et présente un interêt majeur si l'analyseur instrumenté est employé dans des protocoles de proof-carrying code. En effet, dans ce domaine les preuves sont transmise du développeur à l'utilisateur via un réseau et la tailles des preuves ne peut excéder une certaine limite. Dans le cadre de la certification de systèmes embarqués l'interêt de réduire la taille des preuves réside simplement dans le temps de vérification de ces preuves. Pour réduire le temps de vérification des preuves nous envisageons de nous abstenir d'utiliser la tactique `omega` et de développer soit des tactiques Coq spécialisées pour les formules rencontrées lors de la justification des analyses d'Enkidu, soit de développer un ensemble plus grand de patrons de preuve accompagné de jeu de tests permettant à l'exécution d'utiliser le patron de preuve le plus efficace pour prouver une formule donnée. Cela est utile lorsque l'analyseur ne fourni aucune fonction (à instrumenter) pour calculer des données utilisées par l'analyseur (comme les partitions dans la version d'Enkidu que nous avons instrumentée).

Une autre perspective ouverte par ces travaux de thèse concerne l'applicabilité de notre méthodologie d'instrumentation à des outils de *model-checking* [SQ82]. Les analyseurs statiques sont de bons candidats à l'instrumentation en raison de la dichotomie entre la phase de recherche des invariants et la phase de vérification de la stabilité de ces invariants. Les model-checkers calculent un invariant qui sur-approxime les états atteignables du programme, où les états sont généralement modélisés par des ensembles. De nombreuses méthodes de représentation d'ensembles d'états ont vu le jour dont l'une des plus connues utilise comme modèle les diagrammes de décision binaire (bdd) [CMCHG96]. La suite consiste à exprimer également la propriété que l'on souhaite vérifier par un bdd puis à vérifier que les états atteignables du programme sont inclus dans les états qui vérifient la propriété. Les propriétés à vérifier sont souvent exprimés en

logique temporelle [Pnu77] et cela implique qu'il faut se doter d'un vérificateur capable d'aborder de telle logique. Pour passer du model-checking à la certification l'instrumentation doit porter sur deux points : (1) justifier formellement que l'invariant calculé est un sur-ensemble des états atteignable ; (2) puis justifier l'inclusion de cette approximation des états atteignables dans la modélisation de la propriété.

Concernant le chapitre 5, pour pouvoir prouver la correction du protocole sans simplifications ni axiomatisation, il serait utile de formaliser le parallélisme et la concurrence en Coq. Quelques travaux récents ont été menés à ce sujet [AK08], mais beaucoup reste à faire. Cette problèmatique constitue plus qu'une perspective ouverte par nos travaux, et attire ma curiosité pour de futurs recherches.

Bibliographie

[AB07] Andrew W. Appel and Sandrine Blazy. Separation logic for small-step c-minor. In *TPHOLs*, pages 5–21, 2007. (cité page 42)

[AK08] Reynald Affeldt and Naoki Kobayashi. A Coq library for verification of concurrent programs. *Electronic Notes in Theoretical Computer Science*, 199 :17–32, 2008. (cité page 183)

[APH04] Elvira Albert, Germán Puebla, and Manuel V. Hermenegildo. Abstraction-carrying code. In *LPAR*, pages 380–397, 2004. (cité page 52)

[App01] Andrew. W. Appel. Foundational Proof-Carrying Code. In *IEEE Symposium on Logic in Computer Science*, pages 247–258, 2001. (cité page 52)

[Bar81] Henk Barendregt. *The Lambda Calculus : its Syntax and Semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. North-Holland, 1981. Second edition, 1984. (cité pages 6 et 25)

[BBM95] Nikolaj Bjørner, Anca Browne, and Zohar Manna. Automatic generation of invariants and assertions. In *CP*, pages 589–623, 1995. (cité page 22)

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development (Coq'Art : The Calculus of Inductive Constructions)*. Texts in Theoretical Computer Science. Springer, 2004. (cité pages 8, 25, 26, 37 et 179)

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *STTT*, 9(5-6) :505–525, 2007. (cité page 7)

[BJP06] Frédéric Besson, Thomas P. Jensen, and David Pichardie. Proof-carrying code from certified abstract interpretation and fixpoint compression. *Theor. Comput. Sci.*, 364(3) :273–291, 2006. (cité page 52)

[BLP03] Liana Bozga, Yassine Lakhnech, and Michaël Périn. Hermes : An automatic tool for verification of secrecy in security protocols. In *CAV*, pages 219–222, 2003. (cité page 7)

[BM08] Aaron R. Bradley and Zohar Manna. Property-directed incremental invariant generation. *Formal Asp. Comput.*, 20(4-5) :379–405, 2008. (cité page 22)

[BMS06] Aaron.R Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays ? In *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *LNCS*, pages 427–442, 2006. (VMCAI'06). (cité pages 99 et 107)

[Bor00] Richard Bornat. Proving pointer programs in hoare logic. In *Conference on Mathematics of Program Construction*, pages 102–126, 2000. (MPC'00). (cité page 37)

[BRMT05] Gogul Balakrishnan, Thomas W. Reps, David Melski, and Tim Teitelbaum. WYSINWYX : What you see is not what you execute. In *VSTTE*, pages 202–213, 2005. (cité page 46)

[Bro76] Felix Browder, editor. *Mathematical Developments Arising from Hilbert Problems*, number 28 in Proceedings of Symposia in Pure Mathematics. American Mathematical Society, 1976. (cité page 1)

[Bro81] Jan Brouwer. *Brouwer's Cambridge Lectures on Intuitionism*. Cambridge University Press, 1981. Edited by Dirk van Dalen. (cité page 2)

[CC76] Patrick Cousot and Radhia Cousot. Static Determination of Dynamic Properties of Programs. In *ISOP'76 : 2nd International Symposium on Programming*, pages 106–130. Dunod, avril 1976. (cité pages 5, 47, 58, 74, 82 et 97)

[CC77] Patrick. Cousot and Radhia. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77 : 4th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM, janvier 1977. (cité pages 5, 47, 71, 73 et 76)

[CC92] Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4) :511–547, aout 1992. (cité pages 54 et 78)

[CC04] Robert Clarisó and Jordi Cortadella. The octahedron abstract domain. In *SAS'04 : 11th International Symposium on Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer, aout 2004. (cité page 82)

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The Astrée analyzer. In *ESOP*, pages 21–30, 2005. (cité pages 7 et 149)

[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 *ASIAN*, pages 272–300, 2006. (cité page 149)

[CCF+07] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, Laurent Mauborgne, David Monniaux, and Xavier Rival. Varieties of static analyzers : A comparison with Astrée. In *TASE*, pages 3–20, 2007. (cité page 149)

[CCM09] Géraud Canet, Pascal Cuoq, and Benjamin Monate. A value analysis for c programs. In *SCAM*, pages 123–124, 2009. (cité page 44)

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *POPL'78 : 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 84–96. ACM, janvier 1978. (cité pages 22 et 82)

[CH88] Thierry Coquand and Gérard P. Huet. The calculus of constructions. *Inf. Comput.*, 76(2/3) :95–120, 1988. (cité pages 25 et 179)

[Cha06] Amine Chaieb. Proof-producing program analysis. In *International Colloquium on Theoretical Aspects of Computing*, volume 4281 of *LNCS*, pages 287–301, 2006. (ICTAC'06). (cité pages 50, 52, 53, 54, 55, 58, 83, 84, 86, 87, 88, 95, 96, 117, 149 et 180)

[Chl08] Adam Chlipala. *Certified Programming with Dependent Types*. Creative Commons, 2008. (cité page 26)

[Cho59] Noam Chomsky. On certain formal properties of grammars. *Information and Control 2*, pages 137–167, 1959. (cité page 154)

[CIO00] Cristiano Calcagno, Samin S. Ishtiaq, and Peter W. O'Hearn. Semantic analysis of pointer aliasing, allocation and disposal in hoare logic351292. In *PPDP*, pages 190–201, 2000. (cité page 42)

[CJPR05] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342(1) :56–78, September 2005. (cité pages 7 et 61)

[CMCHG96] Edmund M. Clarke, Kenneth L. McMillan, Sérgio Vale Aguiar Campos, and Vassili Hartonas-Garmhausen. Symbolic model checking. In *Computer Aided Verification*, pages 419–427, 1996. (cité page 182)

[CMM+09] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09 : Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 79–90, New York, NY, USA, 2009. ACM. (cité page 42)

[Col89] Alain Colmerauer. Une introduction à prolog iii. In *FODO*, pages 264–288, 1989. (cité page 4)

[Cor05] Pierre Corbineau. *Démonstration automatique en Théorie des Types*. PhD thesis, Université Paris 11, 2005. (cité page 35)

[CP10] David Cachera and David Pichardie. Programmation d'un interpréteur abstrait certifié en logique constructive. *Technique et Science Informatiques (TSI)*, 2010. To appear. (cité page 7)

[CSB+09] Pascal Cuoq, Julien Signoles, Patrick Baudin, Richard Bonichon, Géraud Canet, Loïc Correnson, Benjamin Monate, Virgile Prevosto, and Armand Puccetti. Experience report : Ocaml for an industrial-strength static analysis framework. In *ICFP*, pages 281–286, 2009. (cité page 44)

[Cur63] Haskell Curry. *Foundations of Mathematical Logic*. McGraw–Hill, 1963. Republished by Dover, 1977. (cité page 6)

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, pages 81–94, 2006. (cité page 81)

[Det86] Michael Detlefsen. *Hilbert's Program : an Essay on Mathematical Instrumentalism*. Number 182 in Synthese Library. Reidel, 1986. (cité page 1)

[Dij75] Edsger Wybe Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975. (cité pages 5, 21, 37 et 179)

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. (cité page 21)

[Dil89] David L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989. (cité pages 99 et 101)

[dMDS07] Leonardo Mendonça de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *CAV*, pages 20–36, 2007. (cité page 81)

[FGK+96] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Laurent Mounier, Radu Mateescu, and Mihaela Sighireanu. CADP - a protocol validation and verification toolbox. In *CAV*, pages 437–440, 1996. (cité page 7)

[Fil99] Jean-Christophe Filliâtre. *Preuve de programmes impératifs en théorie des types*. Thèse de doctorat, Université Paris-Sud, July 1999. (cité pages 43 et 54)

[Fil03] Jean-Christophe Filliâtre. Why : a multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003. (cité pages 43 et 54)

[Flo67] Robert. W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics / Mathematical Aspects of Computer Science*, pages 19–32. AMS, 1967. (cité pages 5, 35, 76 et 179)

[FM04] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of c programs. In *ICFEM*, pages 15–29, 2004. (cité pages 43 et 54)

[FM07] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *19th International Conference on Computer Aided Verification*, Lecture Notes in Computer Science, Berlin, Germany, juillet 2007. Springer-Verlag. (cité pages 43, 44 et 54)

[FR74] Michael J. Fischer and Michael O. Rabin. Super-exponential complexity of Presburger arithmetic. In *Proceedings of the SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, 1974. (cité page 70)

[Fre60] Gottlob Frege. *Translations from the Philosophical Writings of Gottlob Frege*. Blackwell, 1960. Edited by Peter Geach and Max Black ; third edition, 1980. (cité page 2)

[Fre84] Gottlob Frege. *Collected Papers on Mathematics, Logic and Philosophy*. Blackwell, 1984. Edited by Brian McGinness. (cité page 2)

[Gen35] Gerhard Gentzen. Untersuchungen über das Logische Schliessen. *Mathematische Zeitschrift*, 39 :176–210 and 405–431, 1935. (cité page 15)

[Gen69] Gerhard Gentzen. *The Collected Papers of Gerhard Gentzen*. Studies in Logic and the Foundations of Mathematics. North-Holland, 1969. Edited by M. E. Szabo. (cité page 3)

[Gir87] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50 :1–102, 1987. (cité page 81)

[Gir06] Jean-Yves Girard. *Le point aveugle, cours de logique, tome 1 : vers la perfection*. collection "Visions des Sciences". Editions Hermann, 2006. (cité page 2)

[GLT89] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*. Number 7 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989. (cité page 6)

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish. Tiwari. Lifting abstract interpreters to quantified logical domains. In *Principles of Programming Languages*, pages 235–246. ACM Press, janvier 2008. (cité pages 82, 98, 110, 117, 126, 127, 130 et 133)

[Göd80] Kurt Gödel. *Kurt Gödel : Collected Works*. Oxford University Press, 1980. Edited by Solomon Feferman and others. (cité page 3)

[GP08] Manuel Garnacho and Michaël Périn. Convincing proofs for program certification. In *International Workshop on Certification of Safety-Critical Software Controlled Systems*. Elsevier Science Publishers, 2008. (cité page 170)

[GRS05] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In *Principles of Programming Languages*, pages 338–350, 2005. (cité pages 97, 108 et 109)

[Hal93a] Nicolas Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993. (cité page 152)

[Hal93b] Nicolas Halbwachs. A tutorial of lustre. 1993. (cité page 152)

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with Blast. In *SPIN Workshop on Model Checking Software*, pages 235–239, 2003. (cité page 7)

[Hoa69] Charles Anthony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969. (cité pages 5, 35, 37, 76 et 179)

[HOL] The HOL interactive proof assistant for higher order logic. http ://hol.sourceforge.net/. (cité page 8)

[Hol05] Gerard J. Holzmann. Software model checking with Spin. *Advances in Computers*, 65 :78–109, 2005. (cité page 7)

[HP08] Nicolas Halbwachs and Mathias. Péron. Discovering properties about arrays in simple programs. In *ACM Conference on Programming Language Design and Implementation*, pages 339–348. ACM Press, 2008. (PLDI'08). (cité pages 58, 82, 96, 97, 98, 109, 126, 132 et 180)

[HW73] Charles Anthony Richard Hoare and Niklaus Wirth. An axiomatic definition of the programming language pascal. *Acta Informatica*, 2 :335–355, 1973. (cité page 37)

[ISA] The isabelle system. http ://isabelle.in.tum.de/. (cité page 8)

[Kle52] Stephen C. Kleene. *Introduction to Metamathematics*, volume 1 of *Bibliotheca Mathematica*. North-Holland, 1952. (cité page 77)

[KN06] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4) :619–695, 2006. (cité page 52)

[KOSS04] Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, and Ofer Strichman. Abstraction-based satisfiability solving of presburger arithmetic. In *CAV*, pages 308–320, 2004. (cité page 81)

[KS09] Daniel Kroening and Ofer Strichman. A framework for satisfiability modulo theories. *Formal Asp. Comput.*, 21(5) :485–494, 2009. (cité page 81)

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006. (cité pages 7, 42, 47, 48, 54 et 61)

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009. (cité pages 7, 47, 48, 54 et 61)

[Min01] Antoine Miné. The octagon abstract domain. In *AST'01 : Workshop on Analysis, Slicing, and Transformation*, pages 310–319. IEEE, octobre 2001. (cité pages 82 et 89)

[Min02] Antoine Miné. A Few Graph-Based Relational Numerical Abstract Domains. In *SAS'02 : 9th International Symposium on Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 117–132. Springer, septembre 2002. (cité page 82)

[Min04] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, décembre 2004. (cité pages 98, 99, 104 et 107)

[Min06] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1) :31–100, march 2006. (cité page 82)

[Mog89] Eugenio Moggi. Computational lambda-calculus and monads. In *LICS*, pages 14–23, 1989. (cité page 44)

[Mon98] David Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998. French. (cité pages 7, 47 et 54)

[Mon09] David Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *Computer-aided verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 570–583. Springer Verlag, 2009. (cité page 81)

[Mor82] J. M. Morris. A general axiom of assignment. assignment and linked data structures. a proof of the schorr-waite algorithm. In *Theoretical Foundations of Programming Methodology (Lecture Notes of the 1981 International Marktoberdorf Summer School)*, pages 25–51, 1982. (cité page 37)

[MP67] John. McCarthy and James.A. Painter. Correctness of a compiler for arithmetic expressions. In *Symposium in Applied Mathematics*, volume 19 of *Mathematical Aspect of Computer Science*, pages 33–41. AMS, 1967. (cité page 37)

[Nam01] Kedar. S. Namjoshi. Certifying model checkers. In *Computer Aided Verification*, volume 2102 of *LNCS*, pages 2–13, 2001. (cité pages 50, 54, 57 et 179)

[Nec97] George. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119. ACM Press, 1997. (cité pages 51 et 60)

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, pages 83–95, June 2000. (cité pages 50 et 54)

[NMB06] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *ICFP*, pages 62–73, 2006. (cité page 42)

[NMB08] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18(5-6) :865–911, 2008. (cité page 42)

[NMS+08] Aleksandar Nanevski, Greg Morrisett, Avraham Shinnar, Paul Govereau, and Lars Birkedal. Ynot : dependent types for imperative programs. In *ICFP*, pages 229–240, 2008. (cité page 42)

[O'H04] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, pages 49–67, 2004. (cité page 42)

[OY02] Peter W. O'Hearn and Hongseok Yang. A semantic basis for local reasoning. In *FoSSaCS*, pages 402–416, 2002. (cité page 42)

[Pau87] Lawrence Paulson. *Logic and Computation : Interactive proof with Cambridge LCF*. Number 2 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987. (cité page 6)

[Pea89] Giuseppe Peano. *Arithmetices Principia, Nova Methodo Exposita*. Fratres Bocca, Turin, 1889. (cité page 3)

[Pic05] David Pichardie. *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1, 2005. In french. (cité pages 7, 26, 47, 54 et 150)

[PM89a] Christine Paulin-Mohring. Extracting F(omega)'s programs from proofs in the calculus. In *Principles of Programming Languages*, pages 89–104, 1989. (cité page 25)

[PM89b] Christine Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, janvier 1989. (cité pages 48 et 49)

[Pnu77] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977. (cité pages 81 et 183)

[Pop06] Sebastian Pop. *La représentation SSA : sémantique, analyses et implémentation dans GCC*. PhD thesis, Ecole des Mines de Paris, 2006. (cité page 154)

[Pra65] Dag Prawitz. *Natural Deduction : a Proof-Theoretical Study*. Number 3 in Stockholm Studies in Philosophy. Almquist and Wiskell, 1965. (cité pages 3, 15 et 180)

[Pra77a] Vaughan R. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, Cambridge University, 1977. (cité page 101)

[Pra77b] Dag Prawitz. Meanings and proofs : on the conflict between classical and intutitionistic logic. *Theoria*, 43 :2–40, 1977. (cité page 3)

[PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, pages 151–166, 1998. (cité pages 50, 54, 57, 60 et 179)

[Ray08] Pascal Raymond. Synchronous program verification with LUSTRE/LESAR. In *Modeling and Verification of Real-Time Systems*, chapter 6. ISTE/Wiley, 2008. (cité pages 7 et 169)

[Rey02] John C. Reynolds. Separation logic : A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002. (cité page 42)

[Ric53] Henry G. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2) :358–366, march 1953. (cité pages 6, 11 et 71)

[Rob65] John Alan Robinson. *A machine-oriented logic based on the resolution principle*. J. Assoc. Comput.Mach., 1965. (cité page 4)

[Rus03] Bertrand Russell. *The Principles of Mathematics*. Cambridge University Press, 1903. (cité page 2)

[Rus08] Bertrand Russell. Mathematical logic based on the theory of types. *American Journal of Mathematics*, 30 :222–262, 1908. (cité page 2)

[RV01] John Alan Robinson and Andrei Voronkov. *Handbook of Automated Reasoning*. Elsevier and MIT Press, 2001. (cité page 4)

[RW13] Bertrand Russell and Alfred North Whitehead. *Principia Mathematica*. Cambridge University Press, 1910–13. (cité page 2)

[SC04] Norman Scaife and Paul Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *ECRTS*, pages 119–126, 2004. (cité pages 153, 157, 159, 161, 169, 170, 171 et 181)

[Sif82] Joseph Sifakis. Global and local invariants in transition systems. In *ICALP*, pages 510–522, 1982. (cité page 22)

[SQ82] Joseph Sifakis and Jean-Pierre Queille. Specification and verification of concurrent systems in CESAR. In *Proceedings of the International Symposium on Programming*, volume 137 of *LNCS*, 1982. (cité pages 6, 169 et 182)

[STC06] Christos Sofronis, Stavros Tripakis, and Paul Caspi. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *EMSOFT*, pages 21–33, 2006. (cité page 153)

[SYY03] Sunae Seo, Hongseok Yang, and Kwangkeun Yi. Automatic construction of hoare proofs from abstract interpretation results. In *Asian Programming Languages and Systems Symposium*, volume 2895 of *LNCS*, pages 230–245, 2003. (APLAS'03). (cité pages 50, 53, 54, 55, 87, 95, 96, 117, 149 et 180)

[SYYH07] Sunae Seo, Hongseok Yang, Kwangkeun Yi, and Taisook Han. Goal-directed weakening of abstract interpretation results. *ACM Trans. Program. Lang. Syst.*, 29(6), 2007. (cité page 176)

[Ten91] Robert Tennent. *Semantics of Programming Languages*. International Series in Computer Science. Prentice-Hall, 1991. (cité page 5)

[Tro69] Anne Sjerp Troelstra. *Principles of Intuitionism*. Number 95 in Lecture Notes in Mathematics. Springer-Verlag, 1969. (cité page 2)

[TSSC05] Stavros Tripakis, Christos Sofronis, Norman Scaife, and Paul Caspi. Semantics-preserving and memory-efficient implementation of inter-task

communication on static-priority or edf schedulers. In *EMSOFT*, pages 353–360, 2005. (cité pages 153 et 161)

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society. Second Series*, 42 :230–265, 1936. This is the paper that introduced what is now called the Universal Turing Machine. (cité pages 3, 10, 71 et 81)