N° d'ordre: 3262

THÈSE

présentée

devant l'Université de Rennes 1

pour obtenir

le grade de : Docteur de l'Université de Rennes 1
Mention Informatique

par

David Pichardie

Équipe d'accueil : Lande (Irisa,Rennes)
École Doctorale : Matisse
Composante universitaire : IFSIC

Titre de la thèse :

Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés

Soutenue le 6 décembre 2005 devant la commission d'examen

M. :	Jean-Pierre	Banâtre	Président
M. :	Patrick	Cousot	Rapporteurs
M. :	Xavier	Leroy	
Mme. :	Christine	Paulin-Mohring	Examinateurs
M. :	David	Schmidt	
M. :	Thomas	Jensen	Directeurs
M. :	David	Cachera	

Remerciements

Je remercie tout d'abord M. Jean-Pierre Banâtre, Professeur à l'Université Rennes 1, d'avoir accepté de présider mon jury. Ma carrière d'étudiant rennais en informatique se termine comme elle avait commencé : dans un amphithéâtre et en sa présence. Pour cette fois, c'est moi qui aurait tenu l'estrade.

Je remercie M. Patrick Cousot, Professeur à l'École Normale Supérieure, de m'avoir fait l'honneur d'être rapporteur. Assimiler la théorie de l'interprétation abstraite dont Radhia Cousot et lui sont les créateurs fut pour moi un des grands défis intellectuels de ce travail. Son intérêt pour mes travaux est déjà une immense récompense. Et puis une soutenance de thèse sans Patrick Cousot aurait été tellement fade...

Je remercie M. Xavier Leroy, Directeur de recherche à l'INRIA Rocquencourt, qui a accepté d'être rapporteur de ce travail. Le hasard a voulu que nous nous penchions au même moment sur un sujet de recherche similaire. Son intérêt pour mon travail est la cerise sur le gâteau de ces trois années de thèse.

Je remercie Mme. Christine Paulin-Mohring, Professeur à l'Université Paris Sud pour avoir participé à mon jury et pour avoir en grande partie contribué à l'outil formidable (si si...) sur lequel repose cette thèse : Coq.

Je remercie M. David A. Schmidt, Professeur à l'Université de l'état du Kansas, qui a traversé l'atlantique pour participer à mon jury. Je garde un excellent souvenir de nos échanges lors de ses visites à l'IRISA.

Je remercie mes deux directeurs de thèse David Cachera et Thomas Jensen. J'ai été particulièrement gâté en ce qui concerne l'encadrement de ma thèse. David et Thomas ont toujours été présents lorsque j'avais besoin d'eux, m'ont laissé une grande liberté d'action et continuent, après cette thèse, à m'accorder de leurs temps. David m'a beaucoup apporté pour la rigueur nécessaire à la rédaction scientifique. Sa rapidité de compréhension m'a souvent bluffé. Thomas m'a proposé une formation accélérée au métier de chercheur : conférences, articles, projets européens. Ses conseils auront une grande influence sur mes responsabilités futures.

Je remercie l'équipe Interprétation Abstraite et Sémantique de l'École Normale Supérieure pour m'avoir accueilli durant deux mois, en particulier David Monniaux dont les discussions m'ont énormément appris, en pourtant peu de temps.

Je remercie une nouvelle fois l'équipe en charge du développement de Coq pour son formidable dévouement. En particulier, je tiens à remercier Pierre Letouzey pour sa disponibilité lors de mes problèmes d'extraction.

Merci à Gilles Barthe et à toute son équipe pour m'avoir accueilli durant la dernière ligne droite de ces travaux de thèse. Gilles a su m'apporter son soutien au moment opportun. Je lui suis redevable de son aide.

Merci à Yves Bertot et à toute l'équipe Lemme (devenu Marelle) pour m'avoir fait plonger dans le bain de la recherche et plus spécifiquement de la preuve formelle.

Merci à Thierry Coquand d'avoir si simplement discuter avec moi lors de ce repas en septembre 2003, à quelques kilomètres de Rome. Son aide précieuse pour ma recherche de postdoc me laisse penser que le hasard propose effectivement parfois des rencontres inestimables.

Je ne peux, bien entendu, oublier le formidable laboratoire qu'est l'IRISA. Merci en particulier à tous les membres de l'équipe Lande et de sa petite cousine Vertecs : Yoann Padioleau, Frédéric Besson, Thomas Genet, Pascal Fradet, Vlad Rusu , Florimont Ployette, Vlad Rusu, Gerardo Schneider, Pascal Fradet, Olivier Ridoux, Mireille Ducassé, Jacques Noyer, Bertrand Jeannet, Arnaud Gotlieb, plus tous ceux que j'oublie.

Merci aussi à mes directeurs successifs au sein de l'antenne de Bretagne de l'ENS Cachan : Michel Pierre, Luc Bougé et Patrice Quinton. Ils m'ont chacun apporté leur soutien au moment où j'en avais besoin.

Un grand merci également à tous les enseignants qui ont croisé mon (long...) parcours scolaire et ont chacun contribué à construire l'enseignant que je suis aujourd'hui : Satar Djebali, Michel Pierre, Jean-Pierre Conze, Hubert Hennion, David Cachera (encore lui...), André Couvert, Charles Consel resteront parmi mes références pédagogiques.

Les derniers remerciements s'adressent à mes proches. Merci à Benoît et à Nolwenn pour nous avoir accompagnés dans nos derniers jours à St-Malo. Merci à ma famille pour son soutien durant ces longues études.

Merci enfin à Alex pour son aide et sa patience durant ma rédaction de thèse, pour son soutien et sa tendresse durant les bons et les mauvais moments de ma vie et pour ce formidable cadeau : Gauvain.

Table des matières

Chapitre 1

Introduction

1.1 Motivations

Les sociétés modernes accordent une confiance grandissante aux systèmes informatiques pour ce qui est de la gestion du quotidien. Que nous empruntions une rame de métro automatique, que nous voyagions avec le dernier Airbus ou que nous passions à moins d'une centaine de kilomètres d'une centrale nucléaire, nous faisons nécessairement confiance aux programmes informatiques placés au cœur de tous ces systèmes. Le rôle de plus en plus important des programmes informatiques dans des systèmes critiques va pourtant de pair avec leur complexité. Faire confiance à de tels programmes n'est donc pas anodin. Certaines défaillances de programme, les *bogues*, sont célèbres, notamment dans le domaine de la recherche spatiale, avec les disparitions de la sonde Mariner 1 en 1962 et de la fusée européenne Ariane 5 en 1996. Les répercussions économiques de tels accidents sont considérables (de l'ordre du milliard d'euros). Et dans les premiers exemples que nous avons donnés, les répercussions d'éventuels bogues pourraient se chiffrer en vies humaines.

Afin d'augmenter la fiabilité des programmes, plusieurs méthodes de vérification sont désormais utilisées. La méthode la plus employée à l'heure actuelle reste le *test* : le comportement du programme est étudié sur plusieurs cas particuliers d'utilisation. Si le test permet en pratique de prouver la présence d'erreurs de programmation, il ne permet généralement pas d'en garantir l'absence (le problème du test exhaustif étant indécidable). Les *méthodes formelles* ont cette ambition : obtenir l'assurance du bon fonctionnement d'un programme. La preuve de programme permet ainsi d'obtenir une preuve (au sens mathématique) de ce bon fonctionnement. Sans être du niveau technique de la démonstration du théorème de Fermat, la preuve de programme est un exercice difficile car les démonstrations nécessaires apparaissent vite longues et fastidieuses (en particulier sur les programmes de grande taille) pour être réalisées « à la main ». Il est ainsi devenu courant de vérifier les programmes au moyen d'autres programmes. Le théorème de Rice assure néanmoins qu'aucun programme n'est capable de vérifier une propriété non triviale pour tous les programmes d'un langage de programmation Turing-complet. En clair, l'outil automatique qui dira pour n'importe quel programme

s'il est correct ou non, n'existe pas.

Face à ce résultat théorique fondamental, deux approches distinctes sont proposées. La première consiste à demander l'interaction de l'homme durant le calcul de l'outil de vérification. C'est par exemple le cas de *la preuve assistée par ordinateur* : une propriété sur le comportement d'un programme est alors prouvée de façon interactive entre l'ordinateur et l'utilisateur. L'ordinateur se charge de vérifier la correction des étapes de raisonnement, et éventuellement d'effectuer certaines parties de la preuve automatiquement. La deuxième approche consiste à utiliser des outils automatiques, mais qui n'apportent que des réponses partielles. *L'analyse statique* propose ainsi d'analyser un programme, sans interaction de l'utilisateur, mais sans être capable de donner forcément une réponse informative (l'outil déclare parfois : « je ne sais pas ! »).

Chacune des ces approches possède des avantages et des inconvénients. Le point fort des analyses statiques réside dans leur caractère automatique qui les rend capables d'analyser des programmes de grande taille. L'analyseur Astrée [CCF+05] est par exemple capable de vérifier le programme de commande de vol des Airbus A380, long de plus de 1.000.000 lignes. Pour sa part, la preuve assistée a actuellement beaucoup de mal à passer à une telle échelle. Chaque preuve interactive a en effet un coût de développement élevé en homme/mois. L'analyse statique reste quant à elle dédiée aux *spécifications faibles* de programmes. Seule une partie de la spécification fonctionnelle d'un programme est traitée. Dans le cas d'Astrée, il s'agit essentiellement d'une vérification sur les calculs numériques d'un programme. Avec la preuve assistée, il est possible de prouver la correction totale d'un programme, c'est à dire la conformité de son comportement à notre attente. Mais encore faut-il disposer d'une spécification formelle complète, ce qui est, encore une fois, chose rare sur les gros logiciels. L'analyse statique apparaît ainsi être une méthode de choix pour vérifier les grands programmes sur des propriétés ciblées.

Un dernier point mérite cependant d'être étudié : la fiabilité. Pour pouvoir élever les réponses des outils de vérification au rang de « preuves », il faut en effet estimer que ce sont eux mêmes des programmes corrects. À ce jeu, les assistants de preuve semblent actuellement les mieux placés. Certains assistants de preuve comme Coq [Coq], Isabelle/HOL [NPW02] ou encore HOL Light [Har96] utilisent en effet la notion *d'objet de preuve*. La fiabilité d'un assistant repose alors uniquement sur le vérificateur de preuve, un petit programme chargé de vérifier si un objet de preuve est une preuve valide pour un théorème donné. Durant chaque preuve interactive, un objet est construit et vérifié à la fin de la preuve. Si un bogue dans le système de preuve interactive a permis d'utiliser un raisonnement invalide, l'objet de preuve sera refusé par le vérificateur. La confiance dans l'outil repose ainsi uniquement sur le code du vérificateur, *le noyau* de l'assistant, qui doit être assez simple pour convaincre « à l'oeil nu » de sa correction. Cette pierre de voûte est installée une fois pour toutes au cœur de l'assistant et sera utilisée pour toutes les propriétés logiques exprimables dans la logique de l'outil. À l'opposé, les analyseurs statiques sont souvent dédiés à un type bien précis de propriété à vérifier et doivent être notablement reprogrammés à chaque changement de propriété. La correction des analyses statiques repose cependant sur une théorie solide : *l'interprétation abstraite* [CC77]. Grâce à cette théorie, la correction d'une analyse peut être prouvée

vis-à-vis de la *sémantique* d'un programme, c'est à dire par rapport à une description formelle de son comportement dynamique. Nous pouvons cependant objecter que c'est, cette fois, l'analyse statique qui a du mal à passer à l'échelle, pour le problème précis de la preuve de correction. Les preuves des analyses statiques utilisées sur les langages réalistes sont en effet rarement présentées *in extenso*. De plus, même lorsque le fragment de preuve existant est convaincant, l'implémentation et la maintenance du programme associé ne sont pas triviales et peuvent être source de nombreuses erreurs difficiles à détecter[1].

En conclusion, les analyses statiques semblent être actuellement les outils les plus efficaces pour affronter la vérification réaliste des logiciels critiques, mais faire confiance à leurs calculs constitue généralement une hypothèse très forte. A l'opposé, l'utilisation des assistants de preuve demande actuellement trop d'interventions humaines pour être économiquement viable dans le domaine du développement logiciel, mais leur fiabilité paraît être une hypothèse raisonnable.

Dans nos travaux de thèse, nous proposons d'utiliser le point fort des assistants de preuve pour combler le point faible des analyses statiques. Nous proposons de prouver, à l'aide d'un assistant de preuve, que l'implémentation d'une analyse est correcte. Nous ne faisons ainsi plus reposer la correction de l'analyse que sur le noyau de l'assistant de preuve.

Nos travaux se basent sur l'assistant de preuve Coq. Cet assistant est capable de produire une implémentation Caml à partir d'une fonction manipulée dans l'assistant. Nous pourrons ainsi non seulement prouver la correction d'analyses statiques, mais aussi extraire une implémentation. La preuve formelle est ainsi directement liée à l'implémentation de l'analyseur[2] et non à la description haut-niveau d'un algorithme.

Il nous semble cependant nécessaire de bien cibler les défis scientifiques d'un tel travail. La formalisation d'**une** analyse statique particulière ne nous paraît pas scientifiquement suffisante car elle démontre juste qu'une personne a été suffisamment compétente pour le faire une fois. Comme nous l'avons vu, il y a au moins autant d'analyses statiques que de langages de programmation et de propriétés à vérifier. Il ne s'agit donc pas de savoir formaliser une analyse, mais plusieurs. Nous nous attachons donc dans ces travaux à proposer un cadre de développement pour les analyses statiques certifiées qui soit assez général pour englober de nombreuses analyses de natures différentes, et qui permette de réduire le temps de preuve interactive nécessaire pour la maintenance, l'évolution, voire la création d'autres analyses. Pour satisfaire le premier point, nous avons choisi de nous baser sur la théorie de l'interprétation abstraite qui est assez générale pour englober toutes sortes d'analyses statiques. Pour le second point nous tâcherons premièrement de proposer une méthode de preuve qui pourra être suivie pour chaque nouvelle analyse, et également de formuler et prouver des théorèmes généraux réutilisables pour plusieurs analyses.

Nous appliquerons ces différentes techniques à l'analyse de programme en bytecode

[1] Le passage à l'échelle souhaité pour les analyseurs statiques impose notamment d'utiliser des structures de données efficaces qui sont rarement formalisées au niveau de la preuve de correction de l'analyse, et pourtant non triviales à implémenter.

[2] La correction de cette implémentation dépend alors aussi de la fiabilité du mécanisme d'extraction.

Java. Ce langage a l'avantage de posséder une sémantique formelle déjà étudiée. La portabilité des programmes Java accroît le besoin d'analyses statiques pour ce langage. Il est important pour l'utilisateur qui télécharge un programme en bytecode, de pouvoir s'assurer avant l'exécution du programme s'il vérifie certaines règles de sécurité de son environnement.

1.2 Contexte

La preuve de correction d'analyses statiques dans un assistant de preuve est une activité de recherche assez récente (la plupart des travaux datent de moins d'un dizaine d'années). Les assistants de preuve semblent désormais avoir acquis la maturité nécessaire pour permettre ce type de développement.

Le thème de cette thèse fait suite à une première exploration du sujet par David Monniaux[Mon98], durant son stage de DEA. Le cadre théorique choisi était alors celui des connexions de Galois. Ces travaux témoignent des nombreuses difficultés inhérentes à la construction de connexions de Galois en Coq. L'assistant de preuve n'était à l'époque pas doté du même mécanisme d'extraction et n'a alors pas été en mesure d'extraire des analyseurs à partir des développements réalisés.

Une autre approche se base elle aussi sur l'interprétation abstraite, il s'agit du langage de programmation Rhodium [LMRC05, LMC03] dédié à la spécification d'analyses statiques et de transformation de programme. L'environnement proposé est capable de générer des optimisateurs écrits en C, accompagné d'une preuve de correction générée automatiquement. Les cas d'études présentés dans ces travaux sont impressionnants (analyses inter-procédurales sensibles au contexte, invariants arithmétiques, analyses d'alias), mais la correction de l'environnement Rhodium ne repose pas sur des fondations aussi solides que Coq. Les preuves de correction sont donc elles aussi réalisées dans une logique *dédiée*.

Les autres approches que nous pouvons citer utilisent toutes un assistant de preuve pour prouver la correction d'analyses statiques. Aucune de ces approches ne repose explicitement sur la théorie de l'interprétation abstraite. Le vérificateur de bytecode Java a particulièrement été étudié durant ces dernières années. Les premiers travaux sur ce sujet furent menés par Cornelia Pusch [Pus99]. Par la suite, Yves Bertot a proposé une formalisation de l'analyse d'initialisation des objets [Ber01] dans le vérificateur Java.

Gilles Barthe *et al.* [BDJ+01] ont ensuite montré comment formaliser la vérification de bytecode Java, en raisonnant sur une sémantique exécutable du langage. Ils ont ensuite proposé [BDHdS01] d'automatiser la dérivation d'un vérifieur certifié. L'environnement Jakarta permet ainsi de spécifier le comportement de la machine virtuelle Java défensive (qui agit sur des valeurs typées en vérifiant dynamiquement si les opérations effectuées sont bien typées) à l'aide de règles de réécriture, puis de proposer des règles pour abstraire la machine défensive. Deux machines sont générées à partir de la description de la machine défensive et des abstractions données. La première machine est une machine offensive qui agit sur des valeurs non typées et qui ne réalise pas de vérification de type dynamique. La deuxième est une machine abstraite non détermi-

niste qui ne manipule que des types. Cette machine permet de réaliser un vérificateur de bytecode. Une correspondance entre ces trois machines est ensuite prouvée (une partie des preuves est générée automatiquement).

D'autres approches ont été suivies pour formaliser le vérificateur de bytecode. Ludovic Casset *et al.* [CBR02] ont extrait un vérificateur *sur carte* par raffinement de machine B. Un tel développement a demandé la preuve de plus de 1000 obligations de preuve, dont plusieurs centaines ont demandé une preuve interactive. Gerwin Klein et Tobias Nipkow ont utilisé Isabelle/HOL pour formaliser le vérificateur de bytecode [KN02, Kle03]. Leurs travaux autour de Java sont considérables puisqu'ils proposent [KN04] une formalisation complète de la sémantique du langage source et du langage de bytecode, d'un compilateur non-optimisant et d'un vérificateur de bytecode, le tout pour un langage à objet similaire à Java.

Ils proposent un librairie de semi-treillis pour construire leur analyse. Les preuves ne sont néanmoins pas faites au niveau de l'implémentation de l'analyse. L'analyseur extrait de ce type de développement n'atteint donc pas le même niveau de certification qu'en Coq. Tous ces travaux sont orientés vers la vérification de types. Encore récemment, la vérification de bytecode a donné lieu à des développements formels. Solange Coupet-Grimal et William Delobel [CGD04] proposent une approche utilisant intensivement les types dépendants de Coq.

D'autre travaux de certification d'analyse peuvent être cités. Alexandru Salcianu et Konstantine Arkoudas [SA05] utilisent l'assistant de preuve Athena [Ath] pour formaliser une analyse de flot de données sur du code 3 adresses. Gilles Barthe et Leonor Prensa Nieto [BN04] proposent une analyse de flux d'information pour un langage de programmation concurrente formalisée en Isabelle. David Naumann [Nau05] se base quant à lui sur PVS [ORS92] pour formaliser une analyse de non-interférence pour un fragment de Java.

Parallèlement à nos travaux de thèse, Leroy *et al.* [Ler06, GBL04] proposent un compilateur certifié partant d'un langage impératif similaire à C (Cminor) et produisant du code assembleur PowerPc. La plus grande nouveauté dans ce travail est le caractère optimisant du compilateur. Ce travail se rapproche donc du nôtre pour ce qui concerne les analyses statiques utilisées durant la phase d'optimisation. Il s'agit d'analyses réalisées sur un langage intermédiaire de type *3 adresses*. Les analyses de flot de données formalisées sont actuellement la propagation de constante et la recherche de sous-expressions communes. À notre connaissance, il s'agit du seul travail, avec le notre, où une implémentation efficace a pu être extraite.

1.3 Structure de ce manuscrit

Ce document est organisé de la façon suivante. Le chapitre 2 présente l'assistant à la preuve Coq de façon didactique. L'objectif de ce chapitre est de permettre à un programmeur fonctionnel de comprendre les premières bases de l'outil. Le chapitre 3 présente la partie de la théorie de l'interprétation abstraite dédiée à la spécification d'analyses statiques (alors vues comme des sémantiques abstraites). Nous finissons ce chapitre par

une discussion sur le cadre théorique à adopter dans nos développements Coq, en présentant le choix « de raison » que nous avons finalement retenu. Nous basons en effet nos travaux sur la notion de fonction de concrétisation plutôt que sur celle de connexion de Galois. Le chapitre 4 présente les méthodes de calcul (et d'approximation) de points fixes issues de la théorie des treillis et de l'interprétation abstraite. Nous appliquerons ces techniques en Coq pour construire des outils génériques utiles pour implémenter les calculs itératifs des analyses statiques. Le chapitre 5 propose le premier exemple d'interpréteur abstrait certifié de cette thèse, en se basant sur un langage WHILE jouet. Les deux chapitres suivants proposent des techniques pour faciliter le développement d'analyses statiques certifiées. Le chapitre 6 propose une librairie permettant de construire des treillis par composition de foncteurs. Ces structures algébriques constituent un élément important dans l'architecture d'un analyseur et il est donc important de savoir les construire facilement dans un assistant de preuve. Afin de proposer un développement formel ouvert au changement de spécification d'une analyse, il est nécessaire de proposer une technique de preuve modulaire. Le chapitre 7 propose l'une de ces techniques, dans le contexte des analyse de références pour langages à objet. Le chapitre 8 présente enfin différents cas d'études basés sur les techniques précédentes. Quatre analyses sur divers fragments du bytecode Java sont proposées : une analyse d'intervalles, une analyse de classe, une analyse de référence modulaire et une analyse d'usage mémoire.

1.4 Nos contributions

Nous présentons maintenant les différentes contributions de ce travail de thèse.

Détermination d'un cadre théorique Notre première contribution a été d'isoler, parmi la vaste théorie de l'interprétation abstraite, un cadre théorique adapté à la preuve formelle dans un assistant de preuve comme Coq. Le cadre retenu, et présenté dans la section 3.3, repose sur la notion de fonction de concrétisation, vérifiant une propriété de morphisme d'intersection.

Développement modulaire de treillis Notre deuxième contribution concerne la notion de foncteur de treillis. Les structures de treillis que nous considérons comprennent une propriété assurant la terminaison de calculs itératifs généraux. Une telle propriété est très difficile à prouver sur des grosses structures. Nous proposons une librairie de foncteurs permettant de construire des structures complexes par simple composition fonctorielle. La construction de ces foncteurs repose notamment sur un résultat nouveau (le théorème 6.3.1 de la page 138) concernant le produit lexicographique de relations *partiellement accessibles*.

Un technique de preuve modulaire La preuve formelle d'analyse statique fait apparaître un besoin accru de modularisation. Sans ce type de méthodologie, les développements deviennent des véritables châteaux de cartes qui s'écroulent au moindre petit changement dans l'analyse. Nous avons ainsi proposé une technique de preuve

adaptée à l'utilisation des concrétisations paramétrées. Cette technique s'est révélée intéressante pour la formalisation d'analyses de références sur les langages à objets.

Différents cas d'études originaux Plusieurs cas d'études ont été proposés durant nos travaux.

- L'analyse du langage WHILE est inspirée des travaux de Patrick Cousot [Cou99]. Il s'agit de la première analyse certifiée utilisant une abstraction numérique (les intervalles) non triviale, ainsi que les techniques d'élargissement et de rétrécissement.
- Cette analyse a été ensuite été adaptée par nos soins dans le contexte du bytecode Java. Nous avons alors ajouté une manipulation des tableaux ainsi qu'un domaine abstrait permettant la décompilation des expressions.
- Un autre cas d'étude concerne une analyse de classe, inter-procédurale, pour le bytecode Java. Cette analyse est inspirée de l'analyse proposée par René Rydhof Hansen [Han02]. La preuve de correction de cette analyse a été réalisée en collaboration avec Vlad Rusu.
- La preuve de correction de cette dernière analyse a ensuite été complètement reconditionnée pour la rendre plus modulaire. Nous avons ainsi obtenu une analyse plus précise permettant une abstraction des références par points de création. Ce type d'abstraction [RMR01] n'avait jamais été formalisé dans un assistant de preuve, voire sur une preuve papier.
- Notre dernière étude de cas concerne l'usage mémoire des programmes bytecode Java. L'algorithme de cette analyse a été proposé par Gerardo Schneider. Nous nous sommes chargé de la conception et de la formalisation de la preuve de correction et de l'implémentation de cette analyse.

Ces travaux ont donné lieu à plusieurs publications. Un première proposition de cadre théorique (alors moins orienté vers l'interprétation abstraite) ainsi que le cas d'étude de l'analyse de classe ont été présentés dans la conférence ESOP [CJPR04] et dans la revue internationale TCS [CJPR05]. L'analyse d'usage mémoire a été présentée dans la conférence internationale FM [CJPS05]. La technique de preuve modulaire du chapitre 7 fait l'objet d'un article pour le workshop CASSIS [Pic05].

Chapitre 2

Programmation dans le Calcul des Constructions Inductives

Coq est un assistant de preuve basé sur le Calcul des Constructions, un λ-calcul richement typé. Cette base lui procure le statut de langage fonctionnel d'un genre particulier : le système de type est assez riche pour exprimer complètement le comportement attendu d'un programme. Dans ce chapitre, nous proposons une introduction à Coq sous l'angle des langages de programmation. Le discours se veut accessible à un programmeur ML standard. Nous baserons nos exemples sur le langage Caml [Cam].

Plusieurs présentations de l'outil Coq existent déjà [BC04, The04]. Nous proposons ici une approche nouvelle : présenter l'outil sans commencer par ses fondements théoriques. L'exercice a déjà été tenté et réussi pour les langages fonctionnels classiques : certains ouvrages d'introduction à Caml [WL93, CM95] ne dévoilent pas son noyau théorique. Dans le cas de Coq, il paraît intéressant de pouvoir présenter l'outil sans un chapitre préliminaire sur la théorie des types. Une telle approche est suivie dans le tutoriel [HKPM04] de l'outil, mais sous l'angle de l'assistant de preuve. Nous proposons ici une ascension par la voie « programmation ». La théorie des types sous-jacentes reste cependant indispensable pour s'assurer une compréhension profonde de toutes les possibilités de l'outil Coq. Nous finirons donc ce chapitre par une présentation synthétique de cette base théorique, à la lumière des exemples concrets proposés dans la première partie.

Nous allons tout d'abord aborder les deux visages de Coq : un langage de programmation avec lequel on peut écrire des programmes et les évaluer dans un interpréteur, mais aussi un assistant de preuve dans lequel on peut écrire des énoncés de théorèmes et les prouver de manière interactive, avec approbation de l'outil pour chacune des étapes du raisonnement. Ces deux visages seront respectivement abordés dans les sections 2.1 et 2.2. La présentation de ces deux fonctionnalités fera apparaître deux « mondes » a priori distincts : le monde informatif (celui des programmes) et le monde logique (celui des preuves). Nous verrons dans la section 2.3 que ces deux mondes peuvent être mélangés pour écrire des programmes avec des types beaucoup plus riches que dans les langages de programmation plus classiques. La section 2.4 lèvera le voile sur les liens

profonds qui sont tissés entre ces mondes. Nous finirons ce chapitre par une présentation du paradigme de programmation par preuve dans la section 2.5, puis du mécanisme d'extraction de programme Caml à partir de programme Coq dans la section 2.6.

2.1 Programmation en Coq

Dans cette section nous allons faire une présentation de Coq sous l'angle des langages de programmation. Nous proposerons pour cela des exemples de programme en fournissant à chaque fois leurs équivalents Caml. Les programmes Coq seront placés à gauche et les programmes Caml à droite.

Nous allons tout d'abord donner deux exemples de définition de type. Le premier type permet de coder les entiers binaires. Tout entier binaire non nul de la forme $\overline{1 a_n \cdots a_0}^2$ sera codé par le terme `(`$\overline{a_0}$ `(... (`$\overline{a_n}$ `xH)))` avec $\overline{0}$ = `xO` et $\overline{1}$ = `xI`.

```
Inductive positive : Set :=
    xI : positive -> positive
  | xO : positive -> positive
  | xH : positive
```

```
type positive =
  | XI of positive
  | XO of positive
  | XH
```

Cet exemple montre que le type inductif `positive` est défini grâce à la liste de ses constructeurs et de leurs types respectifs. Le mot clé **`Set`** indique que le type inductif proposé représente une structure de données. La signification théorique de **`Set`** prendra tout son sens dans la section 2.4.

Il est aussi possible de coder les arbres binaires sur un type de base quelconque.

```
Inductive tree : Set :=
  | leaf : tree
  | node : A -> tree -> tree -> tree.
```

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

En Coq, `A` est définie dans le contexte courant comme un type quelconque. En Caml, le type `tree` est donc polymorphe.

Nous pouvons ensuite définir notre première fonction

```
Definition mult2 p := xO p.
```

```
let mult2 p = XO p
```

Le système de types de Coq étant plus complexe que celui de Caml, il n'est pas toujours possible d'inférer automatiquement le type des arguments d'une fonction. Pour cette raison, il nous faudra souvent préciser ce type lors de la définition des fonctions. Dans le cas de la fonction `mult2` cela donne :

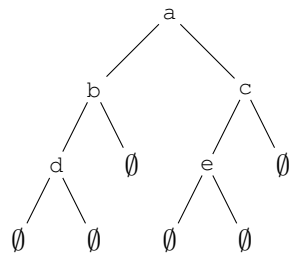
```
Definition mult2 (p:positive): positive := xO p.
```

Les deux types `positive` et `tree` formeront la base de notre exemple conducteur durant ce chapitre. Le but est de donner une représentation efficace des tableaux d'éléments de type `A` [OG98]. Les indices de ces tableaux sont de type `positive` et les tableaux sont codés sous forme d'arbres. Les éléments du tableau sont placés sur les nœuds de

l'arbre. Pour trouver l'élément associé à un indice binaire `p`, on suit le chemin tracé par la séquence de bits qui composent `p` (en partant du bit de poids faible), en partant de la racine et en appliquant la convention

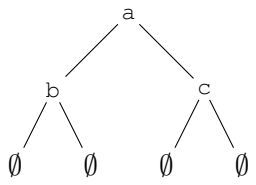
`xH`	$\rightarrow$	arrêt sur le nœud courant
`xI`	$\rightarrow$	on suit la branche droite
`xO`	$\rightarrow$	on suit la branche gauche

Lorsque l'on rencontre une feuille avant d'avoir fini le chemin imposé par `p` on renvoie une valeur par défaut `bot`. Ainsi, le tableau `[a;b;c;d;e]` sera représenté par l'arbre



- `a` est à la position $1 = \overline{1}^2$, donc à la racine de l'arbre
- `b` est à la position $2 = \overline{10}^2$, donc on suit le chemin racine $\rightarrow$ fils gauche
- `c` est à la position $3 = \overline{11}^2$, donc on suit le chemin racine $\rightarrow$ fils droit
- `d` est à la position $4 = \overline{100}^2$, donc on suit le chemin racine $\rightarrow$ fils gauche $\rightarrow$ fils gauche
- `e` est à la position $5 = \overline{101}^2$, donc on suit le chemin racine $\rightarrow$ fils droit $\rightarrow$ fils gauche

Dans le cas où `d=e=bot`, on peut aussi représenter le tableau avec l'arbre



Cette recherche est effectuée par la fonction récursive `get`.

```
Fixpoint get (bot:A) (t : tree)
  (p : positive) {struct t} : A :=
  match t with
  | leaf => bot
  | node a tO tI =>
      match p with
      | xH => a
      | xO p => get bot tO p
      | xI p => get bot tI p
      end
  end.
```

```
let rec get bot t p =
  match t with
     | Leaf -> bot
     | Node (a, tO, tI) ->
         (match p with
            | XH -> a
            | XO p0 -> get bot tO p0
            | XI p0 -> get bot tI p0)
```

Le mot clé **Fixpoint** joue ici le rôle du **let rec** de Caml (construction de point fixe). L'annotation {**struct** t} indique à Coq que la fonction est récursive structurelle vis à vis de son argument t. Cette indication permet d'assurer la terminaison de la fonction dès sa construction.

Une autre fonction importante doit permettre de modifier un tableau à une position donnée. Il est pour cela nécessaire de savoir créer l'arbre ne contenant qu'une valeur v en position p.

```
Fixpoint subst_leaf (bot : A) (p : positive) (v : A) {struct p} : tree :=
  match p with
  | xH => node v leaf leaf
  | xO p' => node bot (subst_leaf bot p' v) leaf
  | xI p' => node bot leaf (subst_leaf bot p' v)
  end.
```

La fonction subst_leaf réalise cette tâche. Finalement, nous proposons une fonction de modification. Au lieu de faire une simple substitution en un indice p d'un tableau t, nous remplaçons l'ancienne valeur (get t p) par (f (get t p)). Cela permet de faire un seul accès au lieu de deux lorsque la nouvelle valeur dépend de l'ancienne.

```
Fixpoint modify (bot : A)
                (t : tree) (p : positive) (f : A -> A) {struct t} : tree :=
  match t with
  | leaf => subst_leaf bot p (f bot)
  | node a tO tI =>
      match p with
      | xH => node (f a) tO tI
      | xO p' => node a (modify bot tO p' f) tI
      | xI p' => node a tO (modify bot tI p' f)
      end
  end.
```

Les fonctions précédentes peuvent être testées grâce à l'interpréteur. Nous utilisons pour cela deux valeurs a et bot de type A dont on suppose l'existence grâce à la commande **Variable** a bot : A.

```
Eval compute in
    (subst_leaf bot 4 a).

= node bot
        (node bot
              (node a leaf leaf)
              leaf)
        leaf : tree
```

```
# let _=
    subst_leaf "bot" (int2pos 4) "a";;

- : string tree =
Node ("bot",
      Node ("bot",
            Node ("a", Leaf, Leaf),
            Leaf),
      Leaf)
```

En Caml, on ne peut pas faire une telle exécution symbolique de fonction polymorphe, on spécialise donc la fonction pour un type précis (ici le type string).

Ces exemples nous montrent ainsi l'inclusion du noyau purement fonctionnel de Caml dans le langage de programmation de Coq. Cette inclusion n'est qu'apparente car certaines fonctions Caml ne sont pas programmables en Coq : toutes les fonctions Coq

terminent alors que l'on peut écrire des fonctions qui ne terminent pas en Caml. De plus, nous n'avons exhibé que des fonctions récursives structurelles. Nous verrons dans le chapitre 4 comment définir des fonctions récursives plus générales, mais il s'agit d'une technique plus avancée.

Il nous reste une dernière analogie à faire entre Coq et Caml, à travers leurs systèmes de module. Dans les deux langages on peut encapsuler des types et des fonctions dans des modules.

```
Module I.
  Definition t : Set := nat.
  Definition v : t := 0.
End I.
```

```
module I = struct
  type t = nat
  let v = O
end
```

Il faut remarquer que le mot clé **Definition** sert à la fois à définir des programmes, mais aussi des types. Nous utilisons ici le type des entiers `nat` de Péano.

```
Inductive nat : Set :=
   O : nat
 | S : nat -> nat.
```

```
type nat =
  | O
  | S of nat
```

Les modules possèdent un type : une signature.

```
Module Type T.
  Parameter t : Set.
  Parameter v : t.
End T.
```

```
module type T = sig
  type t
  val v : t
end
```

Nous pouvons ainsi vérifier que le module `I` vérifie la signature `T` en créant un nouveau module `I'` de type `T` à partir de `I`. Les seuls élément du nouveau module `I'` sont ceux déclarés dans l'interface `T`.

```
Module I' : T := I.
```

```
module I' : T = I
```

Nous verrons dans le chapitre 6, une utilisation plus poussée des modules, à travers les foncteurs de module.

2.2 Preuve formelle

Coq permet « aussi » de réaliser des preuves formelles : l'utilisateur peut énoncer un lemme ou un théorème puis réaliser une preuve interactive à l'aide du langage de *tactiques*. Le langage de référence n'est plus Caml mais plutôt celui des textes mathématiques. Nous ne mettrons donc plus en face de nos exemples du code Caml, mais du « code LaTeX ».

```
Lemma get_subst_leaf1 :
  forall (p : positive) (b v : A),
  get b
    (subst_leaf b p v) p = v.
Proof.
  induction p; auto.
Qed.
```

Lemme 2.1.
$\forall p \in \texttt{positive},\ b, v \in \texttt{A}$,
$\texttt{get}(b, \texttt{subst\_leaf}(b, p, v), p) = v$

Preuve : «Preuve par induction sur `p`, puis preuve automatique ». □

```
Lemma get_subst_leaf2 :
 forall (p1 p2 : positive) (b v : A),
 p2 <> p1 ->
 get b
   (subst_leaf b p1 v) p2 = b.
Proof.
  induction p1; destruct p2;
  simpl; auto || intuition.
Qed.
```

Lemme 2.2.
$\forall p_1, p_2 \in \texttt{positive},\ b, v \in \texttt{A}$,
$p_1 \neq p_2 \Rightarrow$
$\texttt{get}(b, \texttt{subst\_leaf}(b, p_1, v), p_2) = b$

Preuve : «Preuve par induction sur `p1`, une étude de cas sur `p2`, puis preuve automatique ». □

À partir de maintenant, nous noterons ∀ le mot clé **`forall`**.

Les commandes placées entre les mots clés **`Proof`** et **`Qed`** sont des tactiques qui indiquent à Coq les étapes de raisonnement à suivre pour prouver le résultat attendu. Suivons en détail la preuve du premier lemme. La première commande indique une preuve par induction sur la variable `p1`. Nous utilisons pour cela le principe d'induction structurelle associé aux objets de type `positive` : pour prouver une propriété de la forme `∀ p:positive, P p`, il suffit de prouver le cas de base `P xH` puis les deux cas inductifs `∀ p, P p → P (xO p)` et `∀ p, P p → P (xI p)`.

Le système génère alors 3 sous buts à prouver, en indiquant le contexte du premier.

```
3 subgoals

  A : Set
  p : positive
  IHp : ∀ b v : A, get b (subst_leaf b p v) p = v
  ============================
   ∀ b v : A, get b (subst_leaf b (xI p) v) (xI p) = v

subgoal 2 is:
 ∀ b v : A, get b (subst_leaf b (xO p) v) (xO p) = v
subgoal 3 is:
 ∀ b v : A, get b (subst_leaf b xH v) xH = v
```

Le premier sous but correspond au cas inductif `∀ p, P p → P (xI p)`. Ce but contient des expressions simplifiables. Nous demandons à Coq de faire cette simplification avec la tactique **`simpl`**. Le but courant devient alors

```
  A : Set
  p : positive
  IHp : ∀ b v : A, get b (subst_leaf b p v) p = v
  ============================
   ∀ b v : A, get b (subst_leaf b p v) p = v
```

Nous pouvons donc utiliser l'hypothèse d'induction nommée `IHp` dans le contexte pour conclure, avec la tactique **`apply`** `IHp`. Coq passe alors au sous-but suivant. Ce sous-but est similaire au précédent, on remplace juste `xI` par `xO`. Les même tactiques permettent donc de conclure. Nous pouvons les enchaîner avec un point virgule : **`simpl; apply`** `IHp`. Le sous-but restant est

```
A : Set
============================
 ∀ b v : A, get b (subst_leaf b xH v) xH = v
```

Cette fois, nous n'avons pas d'hypothèse d'induction à utiliser. Nous pouvons commencer par éliminer les quantifications avec la commande **`intros`**. Cette tactique applique autant de fois que possible la règle intro$_\forall$ de la déduction naturelle.

```
A : Set
b : A
v : A
============================
 get b (subst_leaf b xH v) xH = v
```

Le but obtenu peut être simplifié par la commande **`simpl`**.

```
A : Set
b : A
v : A
============================
 v = v
```

Ce dernier but est trivial. La tactique **`reflexivity`** est dédiée à ce type de sous but. La preuve est ainsi terminée. Le script de preuve final est

```
Proof.
  induction p.

  simpl.
  apply IHp.

  simpl; apply IHp.

  intros.
  simpl.
  reflexivity.
Qed.
```

Le script donné initialement est plus court car il utilise les possibilités d'automatisation de Coq. Cet exemple nous permet déjà de faire une première remarque sur le langage de preuve : un même énoncé peut admettre des preuves de longueurs radicalement différentes. Pour cette raison, il souvent difficile d'évaluer la difficulté d'une preuve Coq en raisonnant sur sa taille. Ce qu'un utilisateur débutant prouvera en 1000 lignes, sera peut-être prouvé en 200 lignes par un utilisateur plus expérimenté maîtrisant davantage le langage de commandes de preuves.

Nous ne donnerons pas plus de détails sur le langage de preuve de Coq. Nous aurons principalement besoin dans ce manuscrit du langage de spécification de Coq pour expliquer ce qui a été prouvé en Coq, plutôt que de quelle manière cela a été prouvé. Le langage de spécification d'un assistant de preuve constitue une couche sensible avec l'utilisateur. Si chaque preuve de théorème est certes vérifiée par la machine, c'est à l'homme de s'assurer que l'énoncé final du résultat prouvé est bien celui attendu.

Le langage de spécification de Coq reprend la syntaxe classique de la logique des prédicats enrichie avec la notion de définition inductive. Une définition inductive définit une propriété comme un plus petit ensemble d'objets clos par certaines règles de construction. Nous allons illustrer cela sur l'exemple de la relation `inf_log` telle que `(inf_log p n)` est vérifiée si et seulement si le binaire `p` comporte au plus `n` symboles `xO` ou `xI`, c'est à dire au plus $n+1$ bits.

```
Inductive inf_log :
  positive → nat → Prop :=
   inf_log_xH :
   ∀ (n:nat), (inf_log xH n)
 | inf_log_xO :
   ∀ (p:positive) (n:nat),
     (inf_log p n) →
     (inf_log (xO p) (S n))
 | inf_log_xI :
   ∀ (p:positive) (n:nat),
     (inf_log p n) →
     (inf_log (xI p) (S n)).
```

$$\frac{n \in \mathbb{N}}{\texttt{inf\_log}(1, n)}$$

$$\frac{p \in \texttt{positive},\ n \in \mathbb{N},\ \texttt{inf\_log}(p, n)}{\texttt{inf\_log}(\overline{p\,0}^2, n+1)}$$

$$\frac{p \in \texttt{positive},\ n \in \mathbb{N},\ \texttt{inf\_log}(p, n)}{\texttt{inf\_log}(\overline{p\,1}^2, n+1)}$$

Chaque définition inductive se voit associer un principe d'induction. Pour prouver une propriété du type `∀ p n, inf_log p n → P p n`, il suffit par exemple de prouver

- `∀ n, P xH n`,
- `∀ p n, inf_log p n → P p n → P (xO p) (S n)`,
- `∀ p n, inf_log p n → P p n → P (xI p) (S n)`.

Ce principe d'induction découle de la caractérisation du prédicat inductif comme plus petit post-point fixe. Nous reviendrons sur ce point dans le chapitre 3.

2.3 Mélanger programmes et preuves

Nous revenons maintenant à l'aspect programmation, en présentant un point particulièrement original de Coq, l'utilisation de preuves dans les programmes.

Nous pouvons ainsi mettre des preuves dans les structures de données

```
Inductive word : Set :=
  | make_word : ∀ p:positive, inf_log p 32 → word.
```

Le mot clé `∀` est utilisé ici dans un contexte a priori différent de celui des énoncés de théorèmes. Il s'agit ici d'un constructeur de type. Un objet de type `word` est de la forme `(make_word p h)` avec `p` un élément de type `positive` et `h` une preuve de `(inf_log p 32)`. Le type de `make_world` est donc de la forme `positive` → *propriété* →

`word` mais puisque la propriété dépend du premier argument de type `positive` de `make_world`, on nomme cet argument à l'aide du quantificateur ∀. La notation → devient alors superflue puisqu'un type `A→B` peut se noter `∀ _:A, B`. Cette construction de type se nomme *produit dépendant*. Nous reviendrons sur cette construction importante dans la prochaine section.

Cet exemple nous montre que Coq manipule les preuves comme des objets informatiques à part entière. On peut ainsi écrire précisément le type de certaines structures de données avec des invariants internes (listes triées, arbres équilibrés...). La manipulation des valeurs possédant de tels types est cependant plus complexe qu'avec des types simples car il faut constamment s'assurer que les invariants sont vérifiés. Prenons l'exemple du type `bin n` des binaires d'au plus `n` + 1 bits.

```
Inductive bin (n:nat) : Set :=
  | make_bin : ∀ p, inf_log p n → bin n.
```

Ce type est paramétré par un entier `n`. Programmons maintenant la fonction successeur sur ce type. Nous dispose pour cela de la fonction `Psucc : positive → positive` définie par

```
Fixpoint Psucc (x : positive) : positive :=
  match x with
  | xI x' ⇒ xO (Psucc x')
  | xO x' ⇒ xI x'
  | xH ⇒ xO xH
  end.
```

La fonction `succ_bin` que nous allons programmer prendra un binaire sur `n` bits et renverra son successeur sur `(S n)` bits. Il nous faudra prouver que ce successeur est bien codé sur `(S n)` bits au plus. Nous prouvons pour cela le lemme suivant[1]

```
  Lemma Psucc_bin : ∀ n p, inf_log p n → inf_log (Psucc p) (S n).
  Proof. ... Defined.
```

Le mot clé **`Defined`** est équivalent au mot **`Qed`** déjà rencontré, à la seule différence que les preuves terminée par **`Defined`** peuvent être utilisées dans les calculs.

Nous pouvons alors programmer la fonction `succ_bin`

```
Definition succ_bin (n:nat) (w:bin n) : bin (S n) :=
  let (p,h) := w in
    make_bin (Psucc p) (Psucc_bin n p h).
```

La notation **`let`** `(p,h) := w` **`in`** permet d'accéder aux paramètres `p` et `h` du constructeur `make_bin` formant `w`. Elle est équivalente au filtrage

```
match w with make_bin p h ⇒ ... end
```

L'expression `(Psucc_bin n p h)` qui apparaît dans ce programme représente la construction d'une preuve de `inf_log (Psucc p) (S n)` par *application* du lemme `Psucc_bin`. `Psucc_bin` est donc utilisée comme une fonction à trois arguments : un entier `n`, un

[1] Ici, comme dans le reste du document, nous omettons les scripts de preuve. Ces scripts peuvent être consultés et rejoués avec les fichiers Coq associés au manuscrit.

binaire `p` et une preuve `h` de `inf_log p n`. Avec ces arguments, elle calcule alors une preuve de `inf_log (Psucc p) (S n)`.

Nous avons ainsi deux versions différentes du calcul du successeur d'un binaire : `Psucc` qui opère sur les entiers binaires de base, et `succ_bin` qui opère sur les entiers binaires d'au plus `n` bits, en promettant de renvoyer un entier sur au plus $n+1$ bits. Il est intéressant de comparer l'exécution de ces deux fonctions. Nous définissons pour cela un grand entier `max`.

```
Definition max : positive :=
   xI (xI (xI (xI (xI (xI (xI (xI (xI (xI (xI
  (xI (xI (xI (xI (xI (xI (xI (xI (xI xH))))))))))))))))))).

Time Eval compute in (Psucc max).
     = 4194304%positive
     : positive
Finished transaction in 0. secs (0.u,0.s)
```

La commande **`Time`** permet d'afficher le temps de calcul, qui est ici négligeable.

Que se passe-t-il avec `succ_bin` ? Pour le voir, il nous faut d'abord encapsuler `max` dans un constructeur `make_bin` avec une preuve de `inf_log max 21`.

```
Lemma inf_log_max : inf_log max 21.
Proof. ... Defined.

Time Eval compute in (succ_bin 21 (make_word max inf_log_max)).
     = make_bin 4194304
         (inf_log_xO 2097152 21
            (inf_log_xO 1048576 20
               (inf_log_xO 524288 19
                  (inf_log_xO 262144 18
                     (inf_log_xO 131072 17
                        (inf_log_xO 65536 16
                           (inf_log_xO 32768 15
                              (inf_log_xO 16384 14 (..)))))))))
     : bin 22
Finished transaction in 262. secs (191.8u,2.09s)
```

Le temps de calcul est beaucoup plus long[2] ! Cette différence provient du calcul effectué par `Psucc_bin` pour construire une preuve de `inf_log (Psucc max) 22`. Un tel calcul est pourtant inutile puisqu'il ne fait que fabriquer une preuve dont on connaît statiquement l'existence : le type de `succ_bin` nous assurait déjà que le binaire

```
succ_bin 21 (make_word max inf_log_max)
```

aurait au plus 23 bits.

[2] Nous avons en fait pris soin de prouver de manière très inefficace le lemme `Psucc_bin` pour bien faire apparaître les calculs supplémentaires effectués. La preuve utilisée possède une complexité de calcul exponentielle par rapport à n alors qu'une complexité linéaire était possible. Il convient aussi de mentionner, pour cet exemple, la possibilité d'indiquer à Coq de ne pas déplier la définition de `Psucc_bin` lors de ses calculs.

Coq peut donc certes être considéré comme un langage de programmation richement typé mais l'exécution de ses programmes peut parfois occasionner des calculs inutilement longs. Pour remédier à ce problème, le mécanisme d'extraction de Coq permet de transformer un programme Coq en un programme Caml où les calculs sur les preuves sont évités. L'extraction de `succ_bin` produit ainsi un simple appel à `Psucc`.

```
Extraction succ_bin.
    let succ_bin n w = psucc w
```

Le rôle du mécanisme d'extraction est donc de supprimer les parties logiques d'un programme Coq. Si les termes extraits sont bien typés en Coq, cela assure que les preuves ignorées dans le calcul Caml existaient bien et donc que les propriétés vérifiées par les fonctions Coq le seront encore par les fonctions Caml extraites. Nous reviendrons sur l'extraction dans la section 2.6, à la lumière des éléments théoriques que nous allons maintenant dévoiler.

2.4 Les preuves sont des programmes

Les exemples précédents nous montrent que Coq manipule les preuves comme des programmes fonctionnels :

- une preuve peut être passée en argument (comme `h` dans `(make_world p h)`),
- une preuve peut être utilisée comme une fonction (voir l'exemple précédent de `(Psucc_bin n p h)`)

Ce mélange des genres est possible car programmes et preuves sont identifiés en Coq, plus précisément les preuves sont codées par des programmes (des termes de λ-calcul). Le mécanisme de preuve interactive est donc en théorie superflu puisque toutes les preuves peuvent être directement « programmées ». Il s'agit juste d'une facilité offerte à l'utilisateur pour construire des programmes de manière interactive. En pratique, les preuves interactives restent fort utiles pour construire progressivement des preuves complexes et automatiser certaines parties. La construction d'un terme de preuve demande de plus une très bonne compréhension du codage des preuves par les λ-termes alors que la construction interactive est plus accessible pour l'utilisateur débutant. À titre d'exemple, voici une construction directe de la preuve `get_subst_leaf1` précédemment proposée :

```
Fixpoint get_subst_leaf1'
  (p : positive) (b v : A) {struct p} : (get b (subst_leaf b p v) p = v) :=
  match p return (get b (subst_leaf b p v) p = v) with
  | xH ⇒ refl_equal _
  | xO p' ⇒ get_subst_leaf1' p' b v
  | xI p' ⇒ get_subst_leaf1' p' b v
  end.
```

Cet exemple est donné à titre informatif, nous ne détaillerons pas sa construction.

L'idée de représenter les preuves par des λ-termes est liée à l'isomorphisme de Curry-Howard [How80, GLT89, Bar91] qui relie λ-calcul typé et système de déduction selon les trois correspondances suivantes :

type	$\leftrightarrow$	énoncé de théorème
terme	$\leftrightarrow$	preuve
réduction	$\leftrightarrow$	élimination des coupures

Le lambda calcul typé utilisé en Coq se nomme *Calcul des Constructions Inductives* (CCI). Ce calcul a d'abord été proposé par Gérard Huet et Thierry Coquand dans une version sans types inductifs [CH88]. Les types inductifs ont été rajoutés par la suite par Thierry Coquand et Christine Paulin [CPM90]. L'originalité de ce calcul réside en partie dans sa grande uniformité : types et termes appartiennent au même langage. Les types sont donc des termes à part entière et ont donc eux aussi un type. Le type d'un type est appelé *sorte* (et le type d'une sorte est une sorte). Comme nous l'avons vu dans les exemples précédents un type t peut représenter un propriété logique, un type informatif ou une sorte. Pour simplifier un peu[3], on peut affirmer que ces trois cas sont distingués en Coq, en donnant trois sortes possibles à t. Si t a le type `Set`, c'est un type informatif. Si t a le type `Prop`, c'est une propriété logique. Sinon t a le type `Type` : par exemple `Prop : Type` et `Set : Type`, en utilisant la notation standard « : » pour la relation « est de type ». Nous ne donnerons pas en détail la syntaxe et les règles de typage du CCI. Nous nous contenterons de montrer son expressivité à travers l'étude de deux de ces constructions fondamentales : le produit dépendant et l'abstraction. Le produit dépendant $\forall x : t_1,\ t_3$ est un type qui ne peut être habité[4] (dans un monde clos) que par une abstraction de la forme $\lambda x : t_1,\ t_2$[5] avec t_2 un terme de type t_3 et x une variable qui peut à la fois apparaître dans t_2 et t_3. Le tableau de la figure 2.1 présente différentes possibilités pour les types de t_1, t_2 et t_3 avec un exemple associé.

	type de t_1	type de t_3	exemple
1	`Set`	`Type`	$\lambda x : \texttt{nat},\ \texttt{vect}\ x\ :\ \forall x : \texttt{nat},\ \texttt{Set}$
2	`Set`	`Type`	$\lambda x : \texttt{nat},\ x \geq 0\ :\ \forall x : \texttt{nat},\ \texttt{Prop}$
3	`Prop`	`Type`	$\lambda x : \texttt{Prop},\ \neg x\ :\ \forall x : \texttt{Prop},\ \texttt{Prop}$
4	`Prop`	`Set`	$\lambda x : n \neq 0,\ \texttt{div}\ m\ n\ x\ :\ \forall x : n \neq 0,\ \texttt{nat}$
5	`Type`	`Type`	$\lambda x : \texttt{Set},\ \texttt{list}\ x\ :\ \forall x : \texttt{Set},\ \texttt{Set}$
6	`Set`	`Set`	$\lambda x : \texttt{nat},\ x + 1\ :\ \forall x : \texttt{nat},\ \texttt{nat}$
7	`Type`	`Type`	$\lambda x : \texttt{Set},\ (\lambda a : x,\ a)\ :\ \forall x : \texttt{Set},\ x \to x$
8	`Set`	`Set`	$\lambda x : \texttt{nat},\ \texttt{init}\ x\ :\ \forall x : \texttt{nat},\ \texttt{vect}\ x$
9	`Set`	`Prop`	$\lambda x : \texttt{nat},\ \texttt{inf\_log\_xH}\ x\ :\ \forall x : \texttt{nat},\ \texttt{inf\_log}\ \texttt{xH}\ x$

FIG. 2.1 – Différentes versions du produit dépendant $\forall x : t_1,\ t_3$, habité par $\lambda x : t_1,\ t_2$.

Pour les six premières lignes de ce tableau, la notation $\forall x : t_1,\ t_3$ peut être remplacée par $t_1 \to t_3$ car x n'apparaît pas dans t_3. Dans la ligne 1 `vect` x est le type des vecteurs de taille x. La ligne 2 présente le prédicat des entiers positifs. En notation ensembliste, il se note $\{\ x : \texttt{nat} \mid x \geq 0\ \}$. La ligne 3 présente la négation logique. Dans la ligne 4,

[3] La sorte `Type` est en effet suffisamment grande pour englober tout ces cas.

[4] un terme t_1 *habite* un type t_2 s'il admet t_2 comme type

[5] Le CCI autorise en fait de prendre $\lambda x : t_1',\ t_2$ avec t_1 et t_1' équivalents pour les règles de réductions définies dans le CCI.

la division entière `div` de type $\forall m\ n : \texttt{nat}, n \neq 0 \rightarrow \texttt{nat}$ est partiellement instanciée sur des arguments symboliques n et m. Son troisième argument est une preuve que son diviseur est bien différent de zéro. C'est ainsi que l'on décrit le caractère partiel de cette fonction. Le type des listes polymorphes est donné ligne 5. La simple fonction successeur des entiers occupe la ligne 6, tandis que la fonction identité polymorphe est présentée ligne 7. Ligne 8, on trouve la fonction `init` qui construit un vecteur de taille donnée. Enfin un exemple de terme de preuve est donné dans la ligne 9 : `inf_log_xH` x est une preuve de `inf_log xH` x.

Dans [Bar91] Henk Barendregt, classifie différents λ-calculs typés en fonction des dépendances autorisées lors de la construction des abstractions. La construction de base consiste à faire dépendre un terme d'un autre terme (λ-calcul simplement typé, exemple de la ligne 6). Trois pouvoirs d'expressivité distincts sont ensuite identifiés :

1. un terme peut dépendre d'un type (exemple de la ligne 7),
2. un type peut dépendre d'un terme (exemple de la ligne 1),
3. un type peut dépendre d'un type (exemple de la ligne 5).

Ces trois formes de dépendances sont traditionnellement représentées comme des directions orthogonales dans un espace à trois dimensions. Différents λ-calculs typés sont placés aux extrémités d'un cube en fonction des abstractions qui y sont autorisées. Le Calcul des Constructions se place comme le plus expressif de ces calculs, il se trouve à l'opposé du λ-calcul simplement typé, en possédant les 3 formes de dépendances précédemment énoncées.

Le CCI est donc un calcul très expressif. À travers l'isomorphisme de Curry-Howard, c'est le langage de spécification de Coq qui y gagne en pouvoir d'expression. Le système des types inductifs permet quant à lui de dériver automatiquement des principes d'itération et de preuve par induction puissants. Un tel principe d'induction a déjà été présenté sur l'exemple de `inf_log`.

2.5 Programmation par preuve

Nous décrivons maintenant une technique de programmation un peu particulière : la programmation par preuve. Nous avons vu dans la section précédente que les preuves étaient codés par des programmes (λ-termes). Nous pouvons en fait tout aussi bien retourner la situation : Coq manipule les programmes comme des preuves. La fonction `succ_bin` peut en effet tout aussi bien être programmée à l'aide d'une preuve interactive.

```
Definition succ_bin' (n:nat) (w:bin n) : bin (S n).
Proof.
  intros n w.
  destruct w as [p h].
  apply make_bin with (Psucc p).
  apply Psucc_bin; assumption.
Defined.
```

Avant d'expliquer la preuve (ou le programme) réalisé, nous pouvons vérifier que la fonction obtenue est équivalente à la précédente, avec le lemme suivant

```
Lemma succ_bin_eq_succ_bin' : ∀ n w, succ_bin n w = succ_bin' n w.
Proof.
  reflexivity.
Qed.
```

La commande **reflexivity** s'assure en effet que les deux termes sont égaux modulo les règles de conversion du calcul.

Revenons maintenant sur ce petit moment de magie... En clôturant la première ligne **Definition** succ_bin' ... par un point au lieu de :=, nous avons indiqué à Coq que la définition de succ_bin' serait interactive. Coq nous demande alors une preuve interactive du résultat mathématique suivant « pour tout n de type nat, pour tout w de type bin n, le type bin (S n) est habité ».

```
1 subgoal

  ============================
   ∀ n : nat, bin n → bin (S n)
```

Nous allons maintenant suivre pas à pas la construction interactive de ce programme, en donnant à chaque étape le λ-terme sous-jacent. La première étape consiste à introduire les deux argument n et w de la fonction avec la commande **intros** n w.

n : nat w : bin n ============================ bin (S n)	**Definition** succ_bin (n:nat) (w:bin n) : bin (S n) := ?

La preuve consiste ensuite à extraire les valeurs (p:positive) et (h:inf_log p n) de w (par **destruct** w as [p h]).

n : nat p : positive h : inf_log p n ============================ bin (S n)	**Definition** succ_bin (n:nat) (w:bin n) : bin (S n) := **let** (p,h) := w **in** ?

Nous construisons ensuite un élément de type bin (S n) grâce au constructeur make_bin appliqué avec la valeur (Psucc p) (commande **apply** make_bin **with** (Psucc p)). Coq génère alors un dernier sous but qui demande de donner une preuve de l'invariant inf_log (Psucc p) (S n) : le deuxième argument manquant de make_bin.

n : nat p : positive h : inf_log p n ============================ inf_log (Psucc p) (S n)	**Definition** succ_bin (n:nat) (w:bin n) : bin (S n) := **let** (p,h) := w **in** make_bin (Psucc p) ?

Cette preuve est construite grâce au lemme `Psucc_bin` (**apply** `Psucc_bin`).

```
n : nat
p : positive
h : inf_log p n
============================
 inf_log  p n
```

```
Definition succ_bin
  (n:nat) (w:bin n) : bin (S n) :=
    let (p,h) := w in
      make_bin (Psucc p)
               (Psucc_bin n p ?)
```

Le système demande alors une dernière preuve pour l'hypothèse `inf_log p n` du lemme `Psucc_bin`. Cette hypothèse se trouve dans l'environnement courant (`h`), la tactique **assumption** laisse Coq la trouver automatiquement.

```
Proof completed.
```

```
Definition succ_bin
  (n:nat) (w:bin n) : bin (S n) :=
    let (p,h) := w in
      make_bin (Psucc p)
               (Psucc_bin n p h)
```

Ainsi se termine notre exemple de programmation par preuve.

La manipulation de types dépendants devient parfois trop complexe pour pouvoir construire certains programmes en une seule définition. La programmation par preuve permet alors de simplifier un peu la tâche de l'utilisateur. Nous aurons l'occasion de revoir un exemple de programmation par preuve dans le chapitre 4.

2.6 Extraction de programme

Nous terminons ce chapitre par une description synthétique du mécanisme d'extraction de Coq. Nous avons déjà parlé d'extraction dans la section 2.3. Le but est de supprimer les parties logiques d'un programme pour obtenir une version plus efficace mais vérifiant la propriété de correction assurée par son type Coq. Ce travail peut paraître facile puisqu'il suffit d'enlever les preuves apparaissant dans un programme. Nous avons vu dans la section 2.4 que les preuves Coq ne sont rien d'autres que des programme. La tâche est donc plus difficile que prévue, car programmes et preuves sont a priori indifférentiables en Coq. Une première solution consiste à faire une analyse de code mort pour détecter les termes dont la réduction n'est pas nécessaire pour le calcul global. La solution proposée dans la thèse de Christine Paulin, et retenue depuis dans le système est de demander à l'utilisateur d'annoter lui même les parties logiques de ses programmes. Les deux sortes « jumelles » **Set** et **Prop** précédemment présentées sont alors mises à contribution. Le mécanisme d'extraction utilise donc cette information pour supprimer les calculs occasionnés par la réduction des termes logiques dont le type est de sorte **Prop**.

La solution naïve qui consiste à supprimer les termes dont le type est de sorte **Prop** n'est cependant pas satisfaisante car elle peut mener à des programmes qui s'éva-

luent différemment en Coq et en Caml. Ainsi un programme de division[6] Coq de type `div : ∀ n:nat, n<>0 → nat`, prenant deux arguments (un entier `n` et une preuve que `n` est différent de zéro) deviendrait une fonction Caml `div : nat → nat` prenant un seul argument. Une expression comme `(div 0)` ne devrait jamais apparaître dans le code Caml extrait car aucune preuve de `0<>0` n'existe, et pourtant `(div 0)` est bien typé en Coq (de type `0<>0 → nat`), et peut donc être extrait ! L'extraction ne supprime donc pas toujours les arguments logiques des fonctions, elle les remplace par des termes sous forme normale pour éviter les calculs. Le mécanisme d'extraction a encore récemment fait l'objet d'une thèse [Let04], il s'agit toujours d'un sujet de recherche actif.

Pour l'utilisateur, l'extraction est un mécanisme « presse-bouton » moyennant le fait qu'il doit lui même indiquer les types logiques et les types informatifs. La différence de nature entre **`Prop`** et **`Set`** impose certaines restrictions sur les manipulations possibles. Ainsi lors de la construction d'un terme informatif, on ne peut pas faire de preuve par cas sur un objet logique car l'objet ne sera plus présent dans le code extrait : le calcul d'un objet informatif ne doit jamais dépendre du contenu des objets logiques. Cette limitation a motivé notre décision de ne pas utiliser de connexions de Galois dans formalisations Coq. Nous reviendrons sur ce point dans la chapitre suivant.

2.7 Conclusion

Nous avons présenté dans ce chapitre l'assistant à la preuve Coq sous l'angle des langages de programmation, que sa nature λ-calcul typé lui confère tout naturellement. Depuis son récent changement de syntaxe (depuis la version 8.0 [The04]) sa ressemblance avec Caml est encore plus évidente. Dans cette thèse, Coq a été utilisé pour programmer. C'est pour cette raison qu'il nous paraissait intéressant de le présenter comme un langage de programmation à part entière. Depuis que Coq s'est vu doté d'un compilateur [GL02], ce statut n'en est que plus légitime. Nous n'avons cependant pas utilisé les possibilités de calcul de Coq car ils sont en général inutilement coûteux. Notre protocole de programmation est donc le suivant :

- les programmes sont écrits dans le langage de Coq,
- le type (riche) des programmes est vérifié par Coq,
- les programmes sont extraits vers le langage Caml, compilés par le compilateur Caml puis exécutés.

[6] Cet exemple est donné dans la thèse de Pierre Letouzey [Let04].

Chapitre 3

Spécification d'analyse statique par interprétation abstraite

La preuve de programme est fondée sur la modélisation mathématique du comportement des programmes. Cette interprétation logique est donnée par la sémantique du langage de programmation considéré. En général la sémantique d'un programme n'est pas calculable pour un programme quelconque. L'interprétation abstraite est une méthode pour concevoir des sémantiques approchées de programme. L'abstraction employée permet de concentrer la preuve sur les propriétés pertinentes pour le problème à résoudre. Si on utilise une abstraction assez forte, on peut obtenir une sémantique abstraite calculable. Les informations ainsi calculées permettent alors de prouver statiquement (sans exécuter le programme) des propriétés sur le comportement dynamique du programme (quel que soit son environnement d'exécution). Une telle sémantique abstraite constitue alors un outil de vérification automatique qui permet de calculer des propriétés non triviales sur les programmes. Cependant, le théorème de Rice « plane » : aucune propriété non-triviale d'un langage Turing-complet n'est décidable. La sémantique abstraite est donc nécessairement incomplète : pour certains programmes, la propriété calculée n'est pas toujours assez fine. Les outils d'analyse statique construits par interprétation abstraite restent cependant des outils de vérification de choix :

- contrairement aux méthodes déductives, ils ne demandent pas d'intervention de l'utilisateur, ce qui les rend plus adaptés pour la preuve de programmes de plusieurs milliers de lignes,
- il n'est pas nécessaire de disposer d'une spécification formelle détaillée du programme à analyser, l'analyse s'effectue directement sur le texte du programme (contrairement aux techniques de *model-checking* qui travaillent sur un modèle abstrait de programme, souvent construit à la main, et une spécification des propriétés attendues pour le comportement du programme, en général à l'aide d'une logique temporelle)
- en ce qui concerne la correction de la méthode de vérification, la théorie de l'interprétation abstraite propose des outils mathématiques pour assurer qu'une sémantique abstraite calcule des approximations correctes de la sémantique d'un

programme.

Dans ce chapitre, nous allons présenter la méthode proposée par l'interprétation abstraite pour spécifier une analyse statique. Les trois ingrédients de base d'une analyse statique correcte sont la sémantique du langage de programmation analysé, l'abstraction choisie sur le domaine de la sémantique et enfin l'analyseur statique lui-même. Un des messages clés de l'interprétation abstraite, est qu'une fois que les deux premiers ingrédients ont été choisis, la spécification du troisième ingrédient peut être déduite de ces deux premiers choix. De plus, la spécification finalement dérivée peut être facilement implémentée à l'aide d'algorithmes génériques de résolution ou d'approximation de point fixe. De tels algorithmes seront présentés dans le chapitre suivant.

Puisque que le cœur du sujet est la notion d'abstraction, nous allons tout d'abord présenter les structures mathématiques employées pour modéliser cette notion. Le but de cette section est d'amener de la manière la plus naturelle possible à la notion de connexion de Galois, une « clé de voûte » dans la théorie de l'interprétation abstraite. Nous présenterons ensuite des techniques pour spécifier une abstraction correcte d'une sémantique concrète dans la section 3.2. La section 3.3 présentera enfin notre première contribution : le cadre d'interprétation abstraite que nous avons retenu pour la programmation d'interpréteurs abstraits en Coq. Nous expliquerons pourquoi le cadre des connexions de Galois n'a pas été choisi et sur quels résultats théoriques nous pourrons néanmoins encore nous appuyer dans le cadre retenu.

3.1 La « bonne » abstraction

Le postulat de base de l'interprétation abstraite est que toute sémantique $[\![P]\!]$ d'un programme P peut être exprimée comme un ensemble de valeurs prises dans un domaine $\mathcal{D}$ (ensemble d'états, ensemble de traces). De manière générale le domaine sémantique est ainsi muni d'une structure de treillis complet.

Définition 3.1.1. Ensemble partiellement ordonné (poset).
Un ensemble partiellement ordonné (poset) *est un doublet* $(A, \sqsubseteq)$ *avec* A *un ensemble, et* $\sqsubseteq$ *une relation d'ordre partielle, c'est à dire :*

$$\begin{array}{ll} \forall x \in A,\ x \sqsubseteq x & \textit{(réflexivité)} \\ \forall x, y \in A,\ x \sqsubseteq y \ \wedge\ y \sqsubseteq x \ \Rightarrow\ x = y & \textit{(antisymétrie)} \\ \forall x, y, z \in A,\ x \sqsubseteq y \ \wedge\ y \sqsubseteq z \ \Rightarrow\ x \sqsubseteq z & \textit{(transitivité)} \end{array}$$

Définition 3.1.2. Treillis.
Un treillis *est un quadruplet* $(A, \sqsubseteq, \sqcup, \sqcap)$ *avec*

- $(A, \sqsubseteq)$ *un poset,*
- $\sqcup$ *est une borne supérieure binaire :*

$$\begin{array}{l} \forall x, y \in A,\ x \sqsubseteq x \sqcup y \ \wedge\ y \sqsubseteq x \sqcup y \\ \forall x, y, z \in A,\ x \sqsubseteq z \ \wedge\ y \sqsubseteq z \ \Rightarrow\ x \sqcup y \sqsubseteq z \end{array}$$

– $\sqcap$ *est une borne inférieure binaire :*

$$\forall x, y \in A,\ x \sqcap y \sqsubseteq x \ \wedge\ x \sqcap y \sqsubseteq y$$
$$\forall x, y, z \in A,\ z \sqsubseteq x \ \wedge\ z \sqsubseteq y \ \Rightarrow\ z \sqsubseteq x \sqcap y$$

Définition 3.1.3. Treillis complet.
Un treillis complet est un triplet $(A, \sqsubseteq, \bigsqcup)$ *avec*
- A *un ensemble,*
- $\sqsubseteq$ *une relation d'ordre partielle (réflexive, antisymétrique et transitive)*
- $\bigsqcup$ *est une borne supérieure : pour toute partie* S *de* A*,*
  - $\forall a \in S,\ a \sqsubseteq \bigsqcup S$
  - $\forall b \in A, (\forall a \in S,\ a \sqsubseteq b) \ \Rightarrow\ \bigsqcup S \sqsubseteq b$

Remarque 3.1. *Un treillis complet possède nécessairement une borne inférieure* $\bigsqcap$ *(pour toute partie* S *de* A*,* $\forall a \in S,\ \bigsqcap S \sqsubseteq a$ *et* $\forall b \in A, (\forall a \in S,\ b \sqsubseteq a) \ \Rightarrow\ b \sqsubseteq \bigsqcap S$*).*

$$\bigsqcap S = \bigsqcup \{\ y \mid \forall x \in S, y \sqsubseteq x\ \}$$

Remarque 3.2. *Un treillis complet possède toujours un plus grand élément* $\top = \bigsqcap \emptyset = \bigsqcup A$*, et un plus petit élément* $\bot = \bigsqcup \emptyset = \bigsqcap A$*.*

Si le domaine sémantique est de la forme $\mathcal{P}(\mathcal{D})$ (lus les parties de $\mathcal{D}$), le treillis complet associé est $(\mathcal{P}(\mathcal{D}), \subseteq, \cup, \cap)$. Les éléments de $\mathcal{P}(\mathcal{D})$ sont vus comme des propriétés sur les objets de $\mathcal{D}$. L'ordre partiel $\subseteq$ modélise ainsi la précision de la propriété calculée : si $P_1 \subseteq P_2$, P_1 est une propriété plus précise que P_2 car elle décrit moins de comportements possibles que P_2. Le fait de présenter la sémantique d'un langage de programmation sous cette forme n'est pas toujours naturel (notamment pour le cas des sémantiques dénotationnelles déterministes) mais cela permet de placer tous les langages de programmation dans un même cadre théorique. Dorénavant, nous considérerons que le domaine sémantique $\mathcal{A}$ est muni d'une structure de treillis complet $(A, \sqsubseteq, \bigsqcup, \bigsqcap)$. Les éléments de $\mathcal{A}$ seront souvent appelés des *propriétés*.

Abstraire la sémantique d'un programme, c'est restreindre l'ensemble des propriétés utilisées pour exprimer le comportement de ce programme. La notion d'abstraction est donc naturellement représentée par une partie $\overline{\mathcal{A}} \subseteq \mathcal{A}$ des propriétés possibles. Une propriété quelconque $P \in \mathcal{A}$ sera ainsi *approchée* par un élément $\overline{P}$ de $\overline{\mathcal{A}}$. $\overline{P}$ sera une *approximation correcte* si elle est moins précise que P : $P \sqsubseteq \overline{P}$ (nous nous restreignons au cas des sur-approximations).

La question qu'il convient de se poser à ce stade est « qu'est ce qu'une bonne abstraction ? ». En d'autres termes, parmi tous les choix possibles pour définir $\overline{\mathcal{A}}$, quels sont les meilleurs ? L'adjectif *bonne* employé ici ne doit pas être confondu avec correcte. Le critère de correction d'une sémantique abstraite vis à vis d'une sémantique concrète sera abordé dans la prochaine section. Nous étudions ici les qualités qui devront nous faire préférer certaines abstractions plutôt que d'autres.

Pour toute propriété P, notons $\mathcal{Q}_P$ l'ensemble $\{\overline{P} \in \overline{A} \mid P \sqsubseteq \overline{P}\}$ des approximations correctes de P. La moindre des choses, c'est bien sûr qu'il existe toujours une approximation ($\forall P, \mathcal{Q}_P \neq \emptyset$), mais nous pouvons aller plus loin : si $\overline{P}_1$ et $\overline{P}_2$ sont deux

approximations correctes d'une même propriété P, laquelle choisir ? Si $\overline{P}_1$ et $\overline{P}_2$, sont comparables pour $\sqsubseteq$ on prendra bien sûr la plus petite (plus précise), mais dans le cas contraire nous n'aurons pas de meilleur choix que d'essayer les deux propriétés et voir laquelle donne, au final (dans le reste des calculs) la réponse la plus fine. Une telle situation n'est pas très confortable, car elle demande de faire des choix non déterministes durant les calculs pour assurer une précision maximum. Une « bonne abstraction » devrait donc permettre d'éviter ce genre de déconvenue. Ces remarques nous amènent ainsi tout naturellement à « l'hypothèse raisonnable » proposée par Patrick et Radhia Cousot dans leur article fondateur [CC79] : pour tout propriété P, l'ensemble $\mathcal{Q}_P$ des approximations correctes de P doit posséder un plus petit élément[1], noté $\rho(P)$[2]. Notre premier critère de bonne abstraction peut donc être résumé par l'affirmation suivante.

> $\overline{\mathcal{A}} \subseteq \mathcal{A}$ est « une bonne abstraction » si pour toute propriété $P \in \mathcal{A}$, $\mathcal{Q}_P = \{\overline{P} \in \overline{A} \mid P \sqsubseteq \overline{P}\}$ admet un plus petit élément.

Nous allons maintenant présenter diverses re-formulation de ce critère, en suivant toujours l'article fondateur de Patrick et Radhia Cousot [CC79]. Une première reformulation peut être proposée au moyen de la notion de *Famille de Moore*.

Définition 3.1.4. Famille de Moore.
Soit $(A, \sqsubseteq, \sqcup, \sqcap)$ *un treillis complet et* $\mathcal{M} \subseteq A$ *une partie de* A, $\mathcal{M}$ *est une* famille de Moore de A *si elle close par plus grande borne inférieure : pour toute partie* S *de* $\mathcal{M}$, $\bigsqcap S$ *appartient à* $\mathcal{M}$.

Remarque 3.3. *Si* $\mathcal{M} \subseteq A$ *est une famille de Moore de* A, *alors nécessairement* $\top \in \mathcal{M}$. *En effet,* $\emptyset \subseteq \mathcal{M}$, *donc* $\top = \bigsqcap \emptyset \in \mathcal{M}$. *Notre définition diffère quelque peu de celle prise dans la thèse d'état de Patrick Cousot [Cou78][3]. C'est pourquoi nous donnerons parfois les preuves de certains résultats de cette section.*

Théorème 3.1.1. *Soit* $(A, \sqsubseteq, \bigsqcup, \bigsqcap)$ *un treillis complet et* $\overline{A} \subseteq A$. *Si pour tout* $P \in A$, $\left\{\overline{P} \in \overline{A} \mid P \sqsubseteq \overline{P}\right\}$ *admet un plus petit élément alors* $\overline{A}$ *est une famille de Moore de* A, *et réciproquement.*

Preuve : Supposons tout d'abord que $\bigsqcap \mathcal{Q}_P \in \mathcal{Q}_P, \forall P \in A$ (si le plus petit élément d'une partie B existe, c'est nécessairement $\bigsqcap B$), considérons une partie S de $\overline{A}$ et montrons que $\bigsqcap S$ appartient à $\overline{A}$. Par hypothèse sur la famille des $(\mathcal{Q}_P)_{P \in A}$, il nous suffit de montrer que $\bigsqcap \mathcal{Q}_{\bigsqcap S} = \bigsqcap S$. Or, d'une part $\forall x \in \mathcal{Q}_{\bigsqcap S}$, $\bigsqcap S \sqsubseteq x$, par définition de $\mathcal{Q}_{\bigsqcap S}$, donc on peut déjà affirmer que $\bigsqcap S \sqsubseteq \bigsqcap \mathcal{Q}_{\bigsqcap S}$. D'autre part, pour tout $x \in S$, $x \in \overline{A}$ et $\bigsqcap S \sqsubseteq x$ donc $x \in \mathcal{Q}_{\bigsqcap S}$. On peut ainsi affirmer que $\mathcal{S} \subseteq \mathcal{Q}_{\bigsqcap S}$ et par suite que $\bigsqcap \mathcal{Q}_{\bigsqcap S} \sqsubseteq \bigsqcap S$. Ceci termine la preuve du premier sens de notre énoncé.

Si nous supposons maintenant que $\overline{A}$ est une famille de Moore de A, il nous faut montrer que pour tout $P \in A$, $\bigsqcap \mathcal{Q}_P$ appartient à $\mathcal{Q}_P$. Remarquons tout d'abord que

[1] Ce qui implique $\mathcal{Q}_P \neq \emptyset$

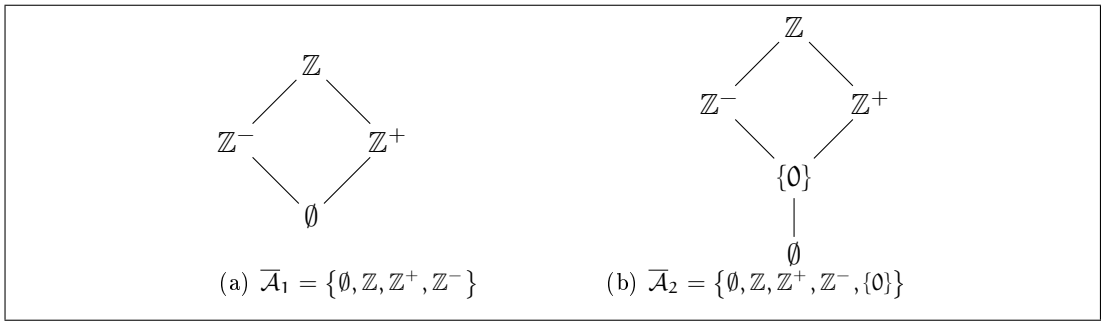
[2] Un plus petit élément étant unique, lorsqu'il existe, la notation fonctionelle $\rho(P)$ est légitime

[3] $\top \in \mathcal{M}$ n'était pas une conséquence de la définition de famille de Moore car $\bigsqcap \emptyset$ n'était pas considéré comme égal à $\top$.

$P \sqsubseteq \bigsqcap \mathcal{Q}_P$ car $\forall \overline{P} \in \mathcal{Q}_P,\ P \sqsubseteq \overline{P}$. Il suffit ensuite d'utiliser l'hypothèse de famille de Moore avec $\mathcal{Q}_P$: on a bien $\mathcal{Q}_P \subseteq \overline{A}$, donc $\bigsqcap \mathcal{Q}_P \in \overline{A}$. Puisque $\bigsqcap \mathcal{Q}_P \in \overline{A}$ et $P \sqsubseteq \bigsqcap \mathcal{Q}_P$, on peut affirmer que $\bigsqcap \mathcal{Q}_P \in \mathcal{Q}_P$ et ainsi conclure. □

$\overline{\mathcal{A}} \subseteq \mathcal{A}$ est « une bonne abstraction » si c'est une famille de Moore de $\mathcal{A}$.

Exemple 1. *Considérons l'exemple classique d'une abstraction par les signes sur les entiers relatifs. Nous avons alors* $\mathcal{D} = \mathbb{Z}$ *et* $\mathcal{A} = \mathcal{P}(\mathbb{Z})$. $\overline{\mathcal{A}_1} = \{\emptyset, \mathbb{Z}, \mathbb{Z}^+, \mathbb{Z}^-\}$ *est une partie de* $\mathcal{A}$*, donc un candidat pour réaliser une abstraction. En considérant l'ordre partiel* $\subseteq$, $\overline{\mathcal{A}_1}$ *est un treillis dont le diagramme de Hasse est donné dans la figure 3.1(a).* $\overline{\mathcal{A}_2} = \{\emptyset, \mathbb{Z}, \mathbb{Z}^+, \mathbb{Z}^-, \{0\}\}$ *est un autre exemple de partie de* $\mathcal{A}$ *dont le diagramme de Hasse est donné dans la figure 3.1(b). Cependant, seule* $\overline{\mathcal{A}_2}$ *représente une bonne abstraction pour le critère que nous avons choisi :* $\overline{\mathcal{A}_2}$ *est une famille de Moore alors que* $\overline{\mathcal{A}_1}$ *ne l'est pas. En effet,* $\mathbb{Z}^- \cap \mathbb{Z}^+ = \{0\} \notin \overline{\mathcal{A}_1}$. *Le problème de* $\overline{\mathcal{A}_1}$ *vient du fait que* $\{0\}$ *n'a pas de meilleure approximation.* $\mathbb{Z}^-$ *et* $\mathbb{Z}^+$ *en sont deux approximations correctes, mais sont incomparables.*



(a) $\overline{\mathcal{A}_1} = \{\emptyset, \mathbb{Z}, \mathbb{Z}^+, \mathbb{Z}^-\}$ (b) $\overline{\mathcal{A}_2} = \{\emptyset, \mathbb{Z}, \mathbb{Z}^+, \mathbb{Z}^-, \{0\}\}$

FIG. 3.1 – Deux exemples d'abstraction par les signes

La fonction $\rho \in \mathcal{A} \to \overline{\mathcal{A}}$ définie précédemment, envoie ainsi chaque propriété $\mathcal{P} \in \mathcal{A}$ sur sa meilleure approximation $\rho(P) \in \overline{\mathcal{A}}$. Elle vérifie les hypothèses d'une fermeture supérieure.

Définition 3.1.5. Fermeture supérieure.
Soit $(A, \sqsubseteq, \bigsqcup, \bigsqcap)$ *un treillis complet. Une fonction* $\rho \in A \to A$ *est une* fermeture supérieure de A *si elle vérifie les trois conditions suivantes :*

- ρ *est monotone* $(\forall x, y \in A,\ x \sqsubseteq y \ \Rightarrow\ \rho(x) \sqsubseteq \rho(y))$
- ρ *est extensive* $(\forall x \in A,\ x \sqsubseteq \rho(x))$
- ρ *est idempotente* $(\forall x \in A,\ \rho(\rho(x)) = \rho(x))$

Les trois conditions de cette définition sont assez naturelles. La monotonie assure que si P_1 est une propriété plus précise que P_2 ($P_1 \sqsubseteq P_2$), alors sa meilleure approximation $\rho(P_1)$ est plus précise que celle de P_2 ($\rho(P_1) \sqsubseteq \rho(P_2)$). L'extensivité assure qu'une

propriété est toujours plus précise que sa meilleure approximation. L'idempotence affirme que si une propriété est une meilleure approximation d'une autre (donc qu'elle est de la forme $\rho(P)$), elle est elle-même sa meilleure approximation.

Le théorème suivant nous montre qu'on peut définir de manière équivalente une bonne abstraction grâce à la notion de fermeture supérieure.

Théorème 3.1.2. *Soit* $(A, \sqsubseteq, \bigsqcup, \bigsqcap)$ *un treillis complet et* $\overline{A} \subseteq A$. $\overline{A}$ *est une famille de Moore de* A *si et seulement si il existe une fermeture supérieure de* A, $\rho \in A \to A$ *telle que* $\overline{A} = \rho(A)$.

Preuve : Supposons dans un premier temps que $\overline{A}$ est une famille de Moore de A. Montrons maintenant que la fonction ρ définie par $\forall P \in A, \rho(P) = \bigsqcap \mathcal{Q}_P$ est une fermeture supérieure de A. Monotonie : si $P_1, P_2 \in A$ vérifient $P_1 \sqsubseteq P_2$, alors par définition des $(\bigsqcap \mathcal{Q}_P)_P$ et transitivité de $\sqsubseteq$, $\bigsqcap \mathcal{Q}_{P_2} \subseteq \bigsqcap \mathcal{Q}_{P_1}$ et par conséquent $\rho(P_1) = \bigsqcap \mathcal{Q}_{P_1} \sqsubseteq \bigsqcap \mathcal{Q}_{P_2} = \rho(P_2)$. Extensivité : pour tout P dans A et pour tout $\overline{P}$ dans $\mathcal{Q}_P$, $P \sqsubseteq \overline{P}$ donc $P \sqsubseteq \bigsqcap \mathcal{Q}_P = \rho(P)$. Idempotence : soit P un élément de A, montrons que $\rho(\rho(P)) = \rho(P)$. $\mathcal{Q}_P$ est une partie de $\overline{A}$ donc puisque $\overline{A}$ est une famille de Moore de A, $\rho(P) = \bigsqcap \mathcal{Q}_P \in \overline{A}$. Or pour tout élément $\overline{P}$ de $\overline{A}$, $\overline{P} \in \mathcal{Q}_{\overline{P}}$ et $\forall \overline{P'} \in \mathcal{Q}_{\overline{P}}$, $\overline{P} \sqsubseteq \overline{P'}$ donc $\overline{P} = \bigsqcap \mathcal{Q}_{\overline{P}} = \rho(\overline{P})$. On en déduit en particulier que $\rho(\rho(P)) = \rho(P)$.

Il nous reste à démontrer que $\rho(A) = \overline{A}$. $\rho(A) \subseteq \overline{A}$ découle du fait que $\overline{A}$ est une famille de Moore de A ($\bigsqcap \mathcal{Q}_P \in \overline{A}\ \forall P \in A$) et $\rho(A) \supseteq \overline{A}$ du point précédent : $\overline{P} = \rho(\overline{P})$ pour tout $\overline{P} \in \overline{A}$.

Inversement, si $\rho \in A \to A$ est une fermeture supérieure de A, montrons que pour tout $P \in A$, $\left\{\overline{P} \in \rho(A) \mid P \sqsubseteq \overline{P}\right\}$ admet un plus petit élément $\rho(P)$, ce qui d'après le théorème 3.1.1, démontrera que $\rho(A)$ est une famille de Moore de A. Soit P un élément de A. Par extensivité de ρ, $\rho(P) \in \left\{\overline{P} \in \rho(A) \mid P \sqsubseteq \overline{P}\right\}$. De plus, pour tout élément $\overline{P}$ de $\left\{\overline{P} \in \rho(A) \mid P \sqsubseteq \overline{P}\right\}$, $\rho(P) \sqsubseteq \rho(\overline{P})$ par monotonie de ρ et $\rho(\overline{P}) = \overline{P}$ par idempotence de ρ. Ainsi $\rho(P)$ est le plus petit élément de $\left\{\overline{P} \in \rho(A) \mid P \sqsubseteq \overline{P}\right\}$, ce qui conclut notre démonstration. □

> $\overline{\mathcal{A}} \subset \mathcal{A}$ est « une bonne abstraction » si elle est de la forme $\overline{A} = \rho(A)$ avec ρ une fermeture supérieure.

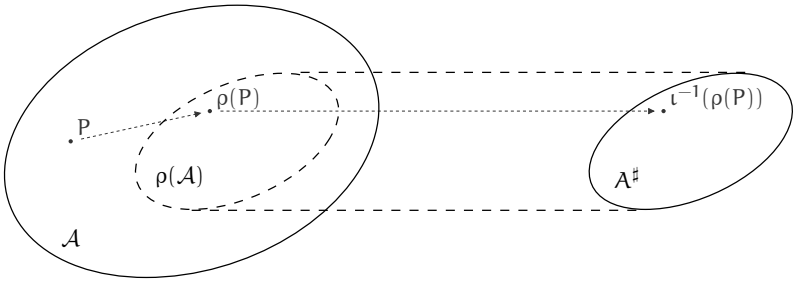
Le domaine d'abstraction est donc de la forme $\rho(A)$. Si l'on restreint l'ordre logique à cet ensemble on peut lui donner une structure de treillis complet.

Théorème 3.1.3. *Soit* $(A, \sqsubseteq, \bigsqcup, \bigsqcap)$ *un treillis complet et* $\rho \in A \to A$ *une fermeture supérieure. Le quadruplet* $(\rho(A), \sqsubseteq, \lambda S.\rho(\bigsqcup S), \bigsqcap)$ *est un treillis complet.*

Preuve : *c.f.* [War42] □

Le « monde abstrait » se voit donc tout naturellement muni d'une structure de treillis complet. Les raisonnements effectués sur ce monde abstrait peuvent se faire à travers un isomorphisme d'ordre. On choisit ainsi un nouveau treillis complet $\left(A^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$ isomorphe à $(\rho(\mathcal{A}), \subseteq, \lambda S.\rho(\cup S), \cap)$ par une fonction $\iota : A^\sharp \to \rho(\mathcal{A})$. Si on regarde les liens ainsi tissés entre $\mathcal{A}$ et $A^\sharp$, nous avons alors obtenu une fonction $\iota \in A^\sharp \to \mathcal{A}$ qui

exprime les propriétés représentées par les éléments de $A^\sharp$, et une fonction $\iota^{-1} \circ \rho \in \mathcal{A} \to A^\sharp$ qui donne la meilleure abstraction d'une propriété, élément de $\mathcal{A}$. Dans ce contexte, un objet abstrait $p^\sharp \in A^\sharp$ sera une approximation correcte d'une propriété concrète $P \in \mathcal{A}$ si $P \subseteq \iota(p^\sharp)$.



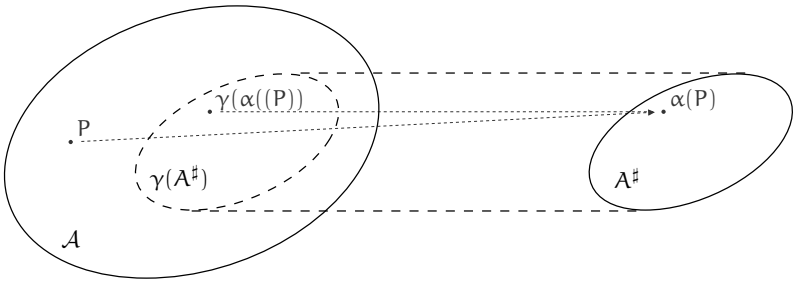
Un tel couple de fonctions $\left(\iota^{-1} \circ \rho \in A \to A^\sharp, \iota \in A^\sharp \to A\right)$ forme une insertion de Galois.

Définition 3.1.6. Insertion de Galois.
Soit $(L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1)$ *et* $(L_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2)$ *deux treillis complets. Une paire de fonctions* $\alpha \in L_1 \to L_2$ *et* $\gamma \in L_2 \to L_1$ *est une* insertion de Galois *si elle vérifie les conditions suivantes :*

- $\forall x_1 \in L_1, \forall x_2 \in L_2,\ \alpha(x_1) \sqsubseteq_1 x_2 \iff x_1 \sqsubseteq_2 \gamma(x_2)$
- γ *est injective*

Réciproquement, si (α, γ) est une insertion de Galois entre deux treillis complets $(L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1)$ et $(L_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2)$, $\gamma \circ \alpha \in L_1 \to L_1$ constitue une fermeture supérieure sur L_1 telle que $\gamma \circ \alpha(L_1)$ est isomorphe à L_2.

Un treillis complet $\left(A^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$ est « une bonne abstraction » s'il existe une insertion de Galois entre $(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap)$ et $\left(A^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$.



Si l'abstraction choisie est assez forte (c'est à dire $\overline{\mathcal{A}}$ assez petit par rapport à $\mathcal{A}$), nous pourrons espérer calculer une approximation de la sémantique d'un programme

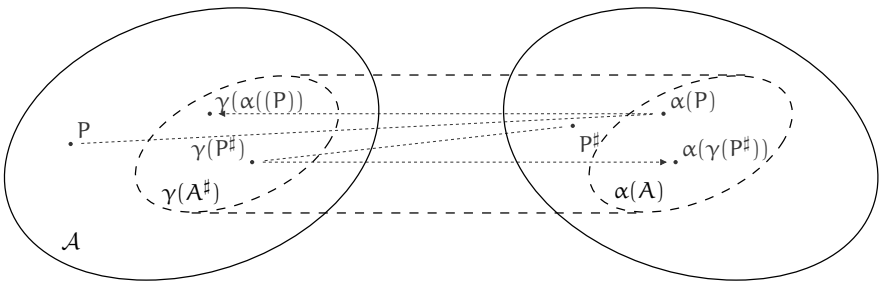
dans le domaine abstrait. L'utilisation d'insertions de Galois permettra alors d'implémenter les abstractions par des objets informatiques dans le domaine $A^\sharp$. La nécessité d'avoir une fonction γ injective est parfois trop contraignante : elle oblige a avoir un unique représentant abstrait $P^\sharp \in A^\sharp$ pour chaque propriété abstraite $\overline{P} \in \overline{\mathcal{A}}$. On peut relâcher cette contrainte, en utilisant seulement une connexion de Galois.

Définition 3.1.7. Connexion de Galois.
Soit $(L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1)$ *et* $(L_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2)$ *deux treillis complets. Une paire de fonctions* $\alpha \in L_1 \to L_2$ *et* $\gamma \in L_2 \to L_1$ *est une* connexion de Galois *si elle vérifie la condition*

$$\forall x_1 \in L_1, \forall x_2 \in L_2,\ \alpha(x_1) \sqsubseteq_1 x_2 \iff x_1 \sqsubseteq_2 \gamma(x_2)$$

Le passage d'une connexion de Galois à une insertion de Galois est facile en théorie, il suffit de considérer un quotient du domaine abstrait par la relation d'équivalence $a_1^\sharp \equiv a_2^\sharp \iff \gamma(a_1^\sharp) = \gamma(a_2^\sharp)$.

Un treillis complet $\left(A^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$ est « une bonne abstraction » s'il existe une connexion de Galois entre $(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap)$ et $\left(A^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$.



Les connexions de Galois sont certes définies par une propriétés très simple mais vérifient de nombreuses propriétés. Nous présentons maintenant quelques unes de ces propriétés que nous serons amenés à utiliser par la suite.

Théorème 3.1.4. *Soit* $(L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1)$ *et* $(L_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2)$ *deux treillis complets, et* (α, γ) *une connexion de Galois entre* L_1 *et* L_2*. Nous notons cette situation*

$$\left(L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1\right) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(L_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2\right)$$

Les propriétés suivantes sont alors vérifiées :

- α *est une fonction monotone*
- γ *est une fonction monotone*
- $\gamma \circ \alpha \dot{\sqsubseteq}_1 \mathrm{id}_{L_1}$ *(avec* $\dot{\sqsubseteq}_1$ *l'ordre point à point sur les fonctions)*
- $\mathrm{id}_{L_2} \dot{\sqsubseteq}_2 \alpha \circ \gamma$

- $\alpha \circ \gamma \circ \alpha = \mathrm{id}_{L_2}$
- $\gamma \circ \alpha \circ \gamma = \mathrm{id}_{L_1}$
- α *est un morphisme d'union :* $\forall S \subseteq L_1,\ \gamma\left(\bigsqcup_1 S\right) = \bigsqcup_2 \alpha(S)$
- γ *est un morphisme d'intersection :* $\forall S \subseteq L_2,\ \gamma\left(\bigsqcap_2 S\right) = \bigsqcap_1 \gamma(S)$

Inversement, si $(L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1)$ *et* $(L_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2)$ *sont deux treillis complets, si* $\gamma \in L_2 \to L_1$ *est un morphisme d'intersection, alors il existe une unique fonction* $\alpha \in L_1 \to L_2$ *telle* (α, γ) *soit une connexion de Galois entre* L_1 *et* L_2. α *est définie par* $\alpha(x) = \bigsqcap_2 \{\, y \mid x \sqsubseteq_2 \gamma(y) \,\}$

Preuve : *c.f.* [CC92a] □

Au final nous avons donc présenté quatre définitions équivalentes de la notion d'abstraction. Seules les deux dernières parlent explicitement d'un monde abstrait distinct du monde concret : ce sont les définitions de choix pour raisonner sur les implémentations des abstractions. Si l'on s'attache uniquement à l'existence d'une bonne abstraction, insertions et connexions de Galois sont équivalentes car une connexion peut toujours être raffinée en une insertion. En pratique, un domaine abstrait sera considéré comme meilleur s'il est relié au monde concret par une insertion. Dans le cas contraire, l'espace de recherche dans le domaine abstrait est inutilement gros et peut donc occasionner des calculs inutiles. L'implémentation du quotient théorique évoqué précédemment est toujours possible, mais peut être coûteuse. L'opérateur de réduction que nous présenterons dans la section 3.2 présentera un moyen de s'accommoder d'une fonction de concrétisation γ non injective, sans perdre trop de précision dans les calculs.

3.2 Spécification d'une sémantique abstraite

Les différentes notions d'abstraction que nous venons de voir font apparaître deux mondes distincts : le monde concret et le monde abstrait. Nous supposerons dans cette section que les liens entre ces deux mondes sont exprimés à l'aide d'une connexion de Galois

$$\left(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap\right) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(\mathcal{A}^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$$

Nous allons maintenant présenter les résultats généraux de l'interprétation abstraite qui permettent de spécifier une sémantique abstraite, ayant fixé l'abstraction choisie à l'aide d'une connexion de Galois. Étant donné une sémantique concrète $[\![P]\!] \in \mathcal{A}$ d'un programme P, le comportement approché de P pourra être calculé par une sémantique abstraite $[\![P]\!]^\sharp \in \mathcal{A}^\sharp$. Les comportements ainsi calculés seront corrects si et seulement si $[\![P]\!] \sqsubseteq \gamma([\![P]\!]^\sharp)$, ou de manière équivalente si $\alpha([\![P]\!]) \sqsubseteq^\sharp [\![P]\!]^\sharp$.

Nous avons ainsi entre les mains une première spécification d'une sémantique abstraite correcte. Dans ce contexte, $\alpha([\![P]\!])$ représente la meilleure sémantique abstraite que l'on puisse espérer avec le choix d'abstraction (α, γ). L'interprétation abstraite fournit heureusement plus de détails sur la méthode à suivre pour construire $[\![P]\!]^\sharp$. L'idée de base est de suivre la structure de $[\![P]\!]$. Ainsi, si $[\![P]\!]$ s'exprime comme une composition de fonctions, $[\![P]\!]^\sharp$ pourra être spécifiée comme une composition de fonctions abstraites. Si $[\![P]\!]$ s'exprime comme un plus petit point fixe, $[\![P]\!]^\sharp$ pourra aussi être spécifié comme un

plus petit point fixe. Nous allons maintenant présenter ces deux constructions de base. Nous présenterons ensuite d'autres résultats de construction de sémantique abstraite.

3.2.1 Évaluation abstraite

Lorsque certains calculs dans le monde concret sont indécidables ou trop coûteux, le monde abstrait peut être utilisé pour effectuer une version simplifiée de ces calculs. Le résultat de ces calculs devra néanmoins toujours donner une réponse conservatrice vis à vis du calcul concret. Pour formaliser cette notion « d'évaluation abstraite », considérons une fonction $\mathsf{f} \in \mathcal{A} \to \mathcal{A}$ dans le monde concret. Pour approcher les calculs de f dans le monde abstrait, il faut définir une fonction $\mathsf{f}^\sharp \in \mathcal{A}^\sharp \to \mathcal{A}^\sharp$ qui approche correctement chaque calcul concret. Formellement, un calcul concret est représenté par un couple $(\mathsf{a}, \mathsf{f}(\mathsf{a}))$ avec $\mathsf{a} \in \mathcal{A}$. Lorsque le calcul $\mathsf{f}^\sharp$ est effectué sur une approximation correcte de a, donc une valeur abstraite $\mathsf{a}^\sharp$ vérifiant $\alpha(\mathsf{a}) \sqsubseteq^\sharp \mathsf{a}^\sharp$, le résultat du calcul doit correctement approcher $\mathsf{f}(\mathsf{a})$: $\alpha(\mathsf{f}(\mathsf{a})) \sqsubseteq^\sharp \mathsf{f}^\sharp(\mathsf{a}^\sharp)$. La situation peut être résumée par le diagramme suivant

$$\begin{array}{ccc} \mathcal{A} & \xrightarrow{\mathsf{f}} & \mathcal{A} \\ \downarrow & & \downarrow \\ \mathcal{A}^\sharp & \xrightarrow{\mathsf{f}^\sharp} & \mathcal{A}^\sharp \end{array}$$

et formalisée par la définition suivante

Définition 3.2.1. Approximation correcte de fonctions.
Étant donnée une connexion de Galois $(\mathcal{A}, \sqsubseteq, \cup, \cap) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(\mathcal{A}^\sharp, \sqsubseteq^\sharp, \cup^\sharp, \cap^\sharp\right)$*, une fonction* $\mathsf{f}^\sharp \in \mathcal{A}^\sharp \to \mathcal{A}^\sharp$ *est appelée* approximation correcte *de* $\mathsf{f} \in \mathcal{A} \to \mathcal{A}$ *si*

$$\forall \mathsf{a} \in \mathcal{A}, \mathsf{a}^\sharp \in \mathcal{A}^\sharp, \alpha(\mathsf{a}) \sqsubseteq^\sharp \mathsf{a}^\sharp \Rightarrow \alpha(\mathsf{f}(\mathsf{a})) \sqsubseteq^\sharp \mathsf{f}^\sharp(\mathsf{a}^\sharp)$$

Pour les fonctions abstraites monotones, on peut énoncer plusieurs critères de correction équivalents.

Théorème 3.2.1. *Étant donné une connexion de Galois*

$$(\mathcal{A}, \sqsubseteq, \cup, \cap) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(\mathcal{A}^\sharp, \sqsubseteq^\sharp, \cup^\sharp, \cap^\sharp\right)$$

une fonction $\mathsf{f}^\sharp \in \mathcal{A}^\sharp \to \mathcal{A}^\sharp$ *monotone et une fonction* $\mathsf{f} \in \mathcal{A} \to \mathcal{A}$*, les quatre assertions suivantes sont équivalentes :*

(i) $\mathsf{f}^\sharp$ *est une approximation correcte de* f*,*

(ii) $\alpha \circ \mathsf{f} \mathrel{\dot{\sqsubseteq}^\sharp} \mathsf{f}^\sharp \circ \alpha$

(ii) $\alpha \circ \mathsf{f} \circ \gamma \mathrel{\dot{\sqsubseteq}^\sharp} \mathsf{f}^\sharp$

(iv) $\mathsf{f} \circ \gamma \mathrel{\dot{\sqsubseteq}^\sharp} \gamma \circ \mathsf{f}^\sharp$

Si l'on désire obtenir une précision optimale, il est naturel de s'intéresser aux cas limites des assertions précédentes. Deux cas limites sont particulièrement intéressants. Le cas $f^\sharp = \alpha \circ f \circ \gamma$ fournit une spécification optimale pour $f^\sharp$, puisque si $f_1^\sharp$ et $f_2^\sharp$ sont deux approximations correctes d'une fonction concrète f, si $f_1^\sharp$ est plus petite que $f_2^\sharp$ (pour l'ordre point à point $\dot{\sqsubseteq}^\sharp$), les calculs de $f_1^\sharp$ seront toujours plus précis que ceux de $f_2^\sharp$. $\alpha \circ f \circ \gamma$ ne doit cependant pas être confondu avec une implémentation de $f^\sharp$. En pratique, $\alpha \circ f \circ \gamma$ représente seulement la spécification de la meilleure abstraction et non un algorithme, car la fonction α est par exemple rarement calculable. Le cas $\alpha \circ f = f^\sharp \circ \alpha$ fournit quant à lui un cadre idéal pour approcher les calculs de points fixes. Nous y reviendrons dans la section suivante.

Exemple 2. *Prenons l'exemple d'une abstraction numérique à base de congruences. Le domaine concret est* $(\mathcal{P}(\mathbb{Z}), \subseteq, \bigcup, \bigcap)$. *Si l'on s'intéresse à la congruence modulo 2, nous allons sélectionner parmi* $\mathcal{P}(\mathbb{Z})$ *deux parties disjointes* $\mathbb{Z}_0 = \{\ x \in \mathbb{Z} \mid x \equiv 0\ [2]\ \}$ *et* $\mathbb{Z}_1 = \{\ x \in \mathbb{Z} \mid x \equiv 1\ [2]\ \}$. *Pour obtenir une famille de Moore, il nous faut aussi considérer* $\emptyset$ *(car* $\emptyset = \bigcap\{\mathbb{Z}_0, \mathbb{Z}_1\}$*) et* $\mathbb{Z}$ *(car* $\mathbb{Z} = \bigcap \emptyset$*). Choisissons maintenant un monde abstrait pour cette abstraction. Construisons maintenant une insertion de Galois. Nous considérons donc un ensemble* $\{\bot, \overline{0}, \overline{1}, \top\}$, *copie abstraite de* $\{\emptyset, \mathbb{Z}_0, \mathbb{Z}_1, \mathbb{Z}\}$. *La précision relative de nos quatre abstractions (*$\emptyset \subseteq \mathbb{Z}_0 \subseteq \mathbb{Z}$ *et* $\emptyset \subseteq \mathbb{Z}_1 \subseteq \mathbb{Z}$*) nous impose alors le diagramme de Hasse de la figure 3.2. L'insertion de Galois correspondante est donnée par*

$$\begin{array}{rcl} \gamma(\bot) & = & \emptyset \\ \gamma(\overline{0}) & = & \mathbb{Z}_0 \\ \gamma(\overline{1}) & = & \mathbb{Z}_1 \\ \gamma(\top) & = & \mathbb{Z} \end{array} \qquad \alpha(P) = \begin{cases} \bot & \textit{si} \quad P = \emptyset \\ \overline{0} & \textit{si} \quad P \neq \emptyset \ \textit{et}\ P \subseteq \mathbb{Z}_0 \\ \overline{1} & \textit{si} \quad P \neq \emptyset \ \textit{et}\ P \subseteq \mathbb{Z}_1 \\ \top & \textit{si} \quad P \not\subseteq \mathbb{Z}_0 \ \textit{et}\ P \not\subseteq \mathbb{Z}_1 \end{cases}$$

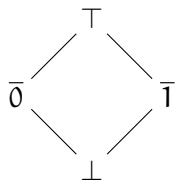


FIG. 3.2 – Diagramme de Hasse de l'abstraction par congruence modulo 2

Supposons maintenant que nous voulions abstraire la multiplication par 2. *Nous cherchons alors une abstraction de la fonction* $f : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ *définie par* $\forall P \subseteq \mathbb{Z},\ f(P) = \{\ 2x \mid x \in P\ \}$. *Nous pouvons dans ce cas précis partir de la spécification* $\alpha \circ f \circ \gamma$ *et la raffiner jusqu'à obtenir une implémentation abstraite.*

$$\alpha \circ f \circ \gamma(\bot) = \alpha \circ f(\emptyset) = \alpha(\{\ 2x \mid x \in \emptyset\ \}) = \bot \tag{3.1}$$
$$\alpha \circ f \circ \gamma(\overline{0}) = \alpha \circ f(\mathbb{Z}_0) = \alpha(\{\ 2x \mid x \in \mathbb{Z}_0\ \}) = \overline{0} \tag{3.2}$$
$$\alpha \circ f \circ \gamma(\overline{1}) = \alpha \circ f(\mathbb{Z}_1) = \alpha(\{\ 2x \mid x \in \mathbb{Z}_1\ \}) = \overline{0} \tag{3.3}$$
$$\alpha \circ f \circ \gamma(\top) = \alpha \circ f(\mathbb{Z}) = \alpha(\{\ 2x \mid x \in \mathbb{Z}\ \}) = \overline{0} \tag{3.4}$$

Ainsi f *peut être correctement abstraite par la fonction* $f^\sharp = \lambda x.\texttt{if } x = \bot \texttt{ then } \bot \texttt{ else } \overline{0}$. *Plusieurs remarques peuvent être faites sur ce simple exemple. Tout d'abord, nous pouvons remarquer que la fonction abstraite* $\lambda x.\overline{0}$ *est aussi une abstraction correcte de* f *mais moins précise que* $f^\sharp$. $f^\sharp$ *est de toute façon la meilleure abstraction monotone d'après le théorème 3.2.1.* $\lambda x.\overline{0}$ *présente par contre l'avantage d'être plus efficace que* $f^\sharp$ *puisqu'elle économise un test. On peut donc tout à fait se contenter de* $\lambda x.\overline{0}$ *par souci d'efficacité, ou si le calcul abstrait a peu de chance d'être appliqué sur* $\bot$.

Notre dernière remarque porte sur le raffinement effectué ici. Cette remarque va en partie motiver les choix que nous ferons dans la section 3.3. Nous avons, pour chaque valeur abstraite, procédé par réductions successives de γ, f *puis* α. *Certaines de ces réductions cachent des preuves, d'autres sont des simples dépliages de définition. Pour* γ *et* f, *il s'agissait d'un simple dépliage. Pour* α, *deux preuves sont généralement nécessaires. Prenons le cas de la ligne 3.2. Pour affirmer que* $\alpha(\{\ x \in \mathbb{Z} \mid x \equiv 0\ [4]\ \})$ *est égal à* $\overline{0}$ *il faut en effet prouver que* $\{\ x \in \mathbb{Z} \mid x \equiv 0\ [4]\ \} \neq \emptyset$ *(*i.e. $\exists x \in \mathbb{Z},\ x \equiv 0\ [4]$*) et* $\{\ x \in \mathbb{Z} \mid x \equiv 0\ [4]\ \} \subseteq \mathbb{Z}_0$. *Seule la dernière propriété est indispensable pour la correction finale. On peut en effet se passer de la preuve de* $\{\ x \in \mathbb{Z} \mid x \equiv 0\ [4]\ \} \neq \emptyset$ *en prouvant la correction de* $f^\sharp(\overline{0}$ *avec le critère* $f \circ \gamma \subseteq \gamma \circ f^\sharp$.

Nous pouvons donc observer à travers cet exemple que la manipulation d'une fonction d'abstraction α *cache des preuves. Même si les preuves paraissent triviales ici, elles ne le sont pas toujours en pratique, surtout pour le niveau de détail demandé par un assistant de preuve comme Coq. Lorsque que l'on désire seulement prouver la correction d'une fonction abstraite (et non son éventuelle optimalité) , la manipulation de* γ *apparaît plus simple que celle de* α. *Si l'on reprend le contexte des fermetures supérieures de la section 3.1,* α *était de la forme* $\iota^{-1} \circ \rho$ *alors que* γ *était égal à* ι. *Ainsi* γ *contient un simple décodage de chaque élément abstrait en une propriété logique, alors que* α *contient une opération de fermeture qui est équivalente à une preuve.*

Le théorème 3.2.1 apporte d'autres commentaires. Tout d'abord, il est facile de généraliser la notion d'approximation correcte à des fonctions à plusieurs variables : une fonction abstraite $f^\sharp \in \mathcal{A}^{\sharp n} \to \mathcal{A}^\sharp$ à n variables sera une *approximation correcte* de $f \in \mathcal{A}^n \to \mathcal{A}$ si $\forall a_1^\sharp, \ldots, a_n^\sharp \in A^\sharp,\ f(\gamma(a_1^\sharp), \ldots, \gamma(a_n^\sharp)) \sqsubseteq f^\sharp(a_1^\sharp, \ldots, a_n^\sharp)$. En particulier si $o \in A^2 \to A$ est un opérateur binaire concret en notation infixe, une approximation correcte $o^\sharp \in A^{\sharp 2} \to A^\sharp$ de o devra vérifier $\forall a_1^\sharp, a_2^\sharp \in A^\sharp,\ \gamma(a_1^\sharp)\ o\ \gamma(a_1^\sharp) \sqsubseteq \gamma(a_1^\sharp\ o^\sharp\ a_2^\sharp)$. Notre deuxième remarque concerne la propriété de composition de cette définition d'approximation correcte.

Lemme 3.1. *Si* $f_1^\sharp, f_2^\sharp \in A^\sharp \to A^\sharp$ *sont des approximations correctes des fonctions concrètes* f_1 *et* f_2 *dans* $A \to A$, *si* f_1 *est monotone, alors* $f_1^\sharp \circ f_2^\sharp$ *est une approximation correcte de* $f_1 \circ f_2$.

Preuve : $\alpha \circ f_1 \circ f_2 \circ \gamma \mathrel{\dot{\sqsubseteq}^\sharp} \alpha \circ f_1 \circ \gamma \circ \alpha \circ f_2 \circ \gamma \mathrel{\dot{\sqsubseteq}^\sharp} f_1^\sharp \circ f_2^\sharp$ par monotonie de α, $f_1^\sharp$ et extensivité de $\alpha \circ \gamma$. □

La notion d'abstraction optimale est par contre rarement compositionnelle (la condition suffisante $\gamma \circ \alpha = \mathrm{id}$ est trop forte pour être vérifiée en pratique : le domaine abstrait serait au moins aussi précis que le domaine concret puisque (γ, α) (et non (α, γ)) serait

une insertion de Galois). Par conséquent, si l'abstraction d'une composition de fonction concrète $f_1 \circ \cdots \circ f_n$ peut être méthodiquement réduite à l'abstraction de chacune des f_i, pour $i = 1 \ldots n$, cela sera généralement au prix d'une perte de précision. Cette remarque montre la capacité des connexions de Galois à raisonner non seulement sur la correction des calculs abstraits, mais aussi sur leur précision.

3.2.2 Points fixes concrets et points fixes abstraits

La section précédente fournit une méthode pour décomposer la spécification de $[\![P]\!]^\sharp$ en fonction des différentes compositions d'opérateurs concrets qui apparaissent dans $[\![P]\!]$. Une autre construction apparaît fréquemment : le point fixe.

Exemple 3. *La sémantique d'un programme peut par exemple être exprimée comme l'ensemble* $[\![P]\!] \in \mathcal{P}(Etat)$ *des états accessibles à partir d'un ensemble d'états initiaux* $\mathcal{S}_0$ *et pour une relation de transition* $\rightarrow_P \subseteq Etat \times Etat$.

$$[\![P]\!] = \{\ s \mid \exists s_0 \in \mathcal{S}_0,\ s_0 \rightarrow_P^* s\ \} = \mathrm{post}[\rightarrow_P^*](\mathcal{S}_0)$$

avec $\mathrm{post}[\cdot] \in \mathcal{P}(Etat \times Etat) \rightarrow \mathcal{P}(Etat) \rightarrow \mathcal{P}(Etat)$ *l'opérateur défini par*

$$\mathrm{post}[\rightarrow](S) = \{\ s_2 \mid \exists s_1 \in S, s_1 \rightarrow s_2\ \}, \forall \rightarrow \subseteq Etat \times Etat,\ \forall S \subseteq Etat$$

$[\![P]\!]$ *peut alors être caractérisé comme le plus petit point fixe de l'opérateur monotone* $\lambda S.\mathcal{S}_0 \cup \mathrm{post}[\rightarrow_P](S)$

$$[\![P]\!] = \mathrm{lfp}\,(\lambda S.\mathcal{S}_0 \cup \mathrm{post}[\rightarrow_P](S))$$

Une fois encore, la théorie de l'interprétation abstraite propose de mimer le concret. C'est donc tout naturellement par un point fixe abstrait que nous chercherons à approcher un point fixe concret. Le cadre des treillis complets et des connexions de Galois fournit alors deux résultats importants.

Théorème 3.2.2. ***(Knaster-Tarski)*** *Dans un treillis complet* $(A, \sqsubseteq, \sqcup, \sqcap)$*, pour toute fonction monotone* $f \in A \rightarrow A$*, le plus petit point fixe* $\mathrm{lfp}(f)$ *de* f *existe et vaut* $\sqcap\{\ x \in A \mid f(x) \sqsubseteq x\ \}$.

Preuve : *c.f.* [Tar55] □

Ainsi, puisque nous nous intéressons à des domaines abstraits munis d'une structure de treillis complet, tout opérateur abstrait monotone est assurée d'avoir un plus petit point fixe.

Théorème 3.2.3. *Étant données une connexion de Galois*

$$\left(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap\right) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(\mathcal{A}^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$$

une fonction $f^\sharp \in \mathcal{A}^\sharp \rightarrow \mathcal{A}^\sharp$ *monotone et une fonction* $f \in \mathcal{A} \rightarrow \mathcal{A}$ *monotone, si* $f^\sharp$ *est une approximation correcte de* f *alors*

$$\alpha(\mathrm{lfp}(f)) \sqsubseteq \mathrm{lfp}(f^\sharp)$$

Si $\alpha \circ f = f^\sharp \circ \alpha$*, on a alors*

$$\alpha(\mathrm{lfp}(f)) = \mathrm{lfp}(f^\sharp)$$

Preuve : *c.f.* [Cou78] □

Par conséquent, le critère de correction précédemment énoncé pour abstraire des fonctions concrètes permet aussi d'abstraire des points fixes. Le cas optimal n'est par contre plus $\alpha \circ f \circ \gamma = f^\sharp$ mais désormais $\alpha \circ f = f^\sharp \circ \alpha$.

Exemple 4. *Si nous reprenons l'exemple 3 à la lumière des nouveaux théorèmes de cette section, nous pouvons spécifier une sémantique abstraite comme un plus petit point fixe* $[\![P]\!]^\sharp = \mathrm{lfp}(F^\sharp)$ *sous la condition que* $F^\sharp$ *soit une approximation correcte de* $\lambda S.\mathcal{S}_0 \cup \mathrm{post}[\rightarrow_P](S)$.

Il reste alors à spécifier une telle approximation. Les résultats de la section 3.2.1 nous incitent alors à décomposer ce nouveau problème en trois sous-problèmes :

1. *trouver une approximation correcte de* $\mathcal{S}_0$,
2. *trouver une approximation correcte de la fonction* $\mathrm{post}[\rightarrow_P] \in \mathcal{P}(Etat) \rightarrow \mathcal{P}(Etat)$,
3. *trouver une approximation correcte de l'opérateur binaire* $\cup$.

Les deux premiers points dépendent de la structure des états et de la définition de $\rightarrow_P$. *Le dernier point est gratuit dans le cadre des connections de Galois :* $\sqcup^\sharp$ *est une approximation correcte de* $\cup$ *car* α *est un morphisme d'union.*

3.2.3 Techniques d'abstraction

Nous complétons maintenant notre présentation avec différentes techniques génériques pour construire des spécifications de sémantique abstraite.

3.2.3.1 Réduction

Comme nous l'avons vu dans la section 3.1, le domaine abstrait est parfois redondant. Certains éléments abstraits distincts possèdent la même concrétisation. Une telle situation peut mener à une imprécision des fonctions abstraites comme le montre l'exemple suivant.

Exemple 5. *Supposons que l'on dispose de deux connexions de Galois*

$$\left(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap\right) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \left(\mathcal{A}_1^\sharp, \sqsubseteq_1^\sharp, \bigsqcup_1^\sharp, \bigsqcap_1^\sharp\right) \text{ et } \left(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap\right) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \left(\mathcal{A}_2^\sharp, \sqsubseteq_2^\sharp, \bigsqcup_2^\sharp, \bigsqcap_2^\sharp\right)$$

pour abstraire un même domaine concret $\mathcal{A}$. *Nous pouvons combiner ces deux abstractions avec une nouvelle connexion*

$$\left(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap\right) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(\mathcal{A}_1^\sharp \times \mathcal{A}_2^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$$

où

$$(a_1^\sharp, a_2^\sharp) \sqsubseteq^\sharp (b_1^\sharp, b_2^\sharp) \iff a_1^\sharp \sqsubseteq_1^\sharp b_1^\sharp \wedge a_2^\sharp \sqsubseteq_2^\sharp b_2^\sharp, \quad \forall (a_1^\sharp, a_2^\sharp), (b_1^\sharp, b_2^\sharp) \in \mathcal{A}_1^\sharp \times \mathcal{A}_2^\sharp$$

pour tout $S \subseteq \mathcal{A}_1^\sharp \times \mathcal{A}_2^\sharp$,

$$\begin{aligned}\bigsqcup^\sharp(S) &= \left(\bigsqcup_1^\sharp(\{a_1^\sharp | \exists a_2^\sharp, (b_1^\sharp, b_2^\sharp) \in S\}), \bigsqcup_2^\sharp(\{a_2^\sharp | \exists a_1^\sharp, (b_1^\sharp, b_2^\sharp) \in S\})\right)\\ \bigsqcap^\sharp(S) &= \left(\bigsqcap_1^\sharp(\{a_1^\sharp | \exists a_2^\sharp, (b_1^\sharp, b_2^\sharp) \in S\}), \bigsqcap_2^\sharp(\{a_2^\sharp | \exists a_1^\sharp, (b_1^\sharp, b_2^\sharp) \in S\})\right)\end{aligned}$$

et

$$\begin{aligned}\alpha(a) &= (\alpha_1(a), \alpha_2(a)) \ \forall a \in \mathcal{A}\\ \gamma(a_1^\sharp, a_2^\sharp) &= \gamma(a_1^\sharp) \sqcap \gamma(a_2^\sharp), \ \forall (a_1^\sharp, a_2^\sharp) \in \mathcal{A}_1^\sharp \times \mathcal{A}_2^\sharp\end{aligned}$$

La nouvelle connexion ainsi obtenue est cependant rarement une insertion, même si (α_1, γ_1) *et* (α_2, γ_2) *le sont. Ainsi, si on combine l'insertion de Galois de l'exemple 1 où*

$$(\mathcal{P}(\mathbb{Z}), \subseteq, \bigcup, \bigcap) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \left(\{\bot_1, 0_1, -_1, +_1, \top_1\}, \sqsubseteq_1^\sharp, \bigsqcup_1^\sharp, \bigsqcap_1^\sharp\right)$$

$$\begin{aligned}\gamma_1(\bot_1) &= \emptyset\\ \gamma_1(0_1) &= \{0\}\\ \gamma_1(-_1) &= \mathbb{Z}^-\\ \gamma_1(+_1) &= \mathbb{Z}^+\\ \gamma_1(\top_1) &= \mathbb{Z}\end{aligned} \qquad \alpha_1(P) = \begin{cases} \bot_1 & si \ P = \emptyset\\ 0_1 & si \ P = \{0\}\\ -_1 & si \ P \not\subseteq \{0\} \ et \ P \subseteq \mathbb{Z}^-\\ +_1 & si \ P \not\subseteq \{0\} \ et \ P \subseteq \mathbb{Z}^+\\ \top_1 & si \ P \not\subseteq \mathbb{Z}^- \ et \ P \not\subseteq \mathbb{Z}^+\end{cases}$$

avec l'insertion de l'exemple 2

$$(\mathcal{P}(\mathbb{Z}), \subseteq, \bigcup, \bigcap) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \left(\{\bot_2, 0_2, 1_2, \top_2\}, \sqsubseteq_2^\sharp, \bigsqcup_2^\sharp, \bigsqcap_2^\sharp\right)$$

$$\begin{aligned}\gamma_2(\bot_2) &= \emptyset\\ \gamma_2(0_2) &= \mathbb{Z}_0\\ \gamma_2(1_2) &= \mathbb{Z}_1\\ \gamma_2(\top_2) &= \mathbb{Z}\end{aligned} \qquad \alpha_2(P) = \begin{cases} \bot_2 & si \ P = \emptyset\\ 0_2 & si \ P \neq \emptyset \ et \ P \subseteq \mathbb{Z}_0\\ 1_2 & si \ P \neq \emptyset \ et \ P \subseteq \mathbb{Z}_1\\ \top_2 & si \ P \not\subseteq \mathbb{Z}_0 \ et \ P \not\subseteq \mathbb{Z}_1\end{cases}$$

tous les couples dans $\{(\bot_1, \bot_2), (\bot_1, 0_2), (\bot_1, 1_2), (\bot_1, \top_2), (0_1, \bot_2)(-_1, \bot_2), (+_1, \bot_2), (\top_1, \bot_2), (0_1, 1_2)\}$ *ont la même concrétisation* $\emptyset$.

Les méfaits d'une telle situation apparaissent lors de l'abstraction d'une fonction comme $\mathrm{succ} : \mathcal{P}(\mathbb{Z}) \to \mathcal{P}(\mathbb{Z})$ *définie par* $\mathrm{succ}(S) = \{x + 1 | x \in S\}$. *Il est tentant de vouloir combiner une approximation correcte de* succ *par les signes avec une approximation correcte par les congruence modulo 2. Ainsi on construit la fonction* $\mathrm{succ}^\sharp : \mathcal{A}_1^\sharp \times \mathcal{A}_2^\sharp \to \mathcal{A}_1^\sharp \times \mathcal{A}_2^\sharp$ *définie par* $\mathrm{succ}^\sharp(a_1^\sharp, a_2^\sharp) = (\mathrm{succ}_1^\sharp(a_1^\sharp), \mathrm{succ}_2^\sharp(a_2^\sharp))$ *avec* $\mathrm{succ}_1^\sharp$ *et* $\mathrm{succ}_2^\sharp$ *définie par*

$$\begin{aligned}\mathrm{succ}_1^\sharp(\bot_1) &= \bot_1\\ \mathrm{succ}_1^\sharp(0_1) &= +_1\\ \mathrm{succ}_1^\sharp(-_1) &= \top_1\\ \mathrm{succ}_1^\sharp(+_1) &= +_1\\ \mathrm{succ}_1^\sharp(\top_1) &= \top_1\end{aligned} \qquad \begin{aligned}\mathrm{succ}_2^\sharp(\bot_2) &= \bot_2\\ \mathrm{succ}_2^\sharp(0_2) &= 1_2\\ \mathrm{succ}_2^\sharp(1_2) &= 0_2\\ \mathrm{succ}_2^\sharp(\top_2) &= \top_2\end{aligned}$$

$\text{succ}^\sharp$ *est en effet une approximation correcte mais elle n'est pas du tout optimale. On a par exemple*

$$(\bot_1, \bot_2) = \alpha \circ \text{succ} \circ \gamma(0_1, 1_2) \sqsubset \text{succ}^\sharp(0_1, 1_2) = (+_1, 0_2)$$

Pourtant $\text{succ}^\sharp_1$ *et* $\text{succ}^\sharp_2$ *sont des approximations optimales.*

Si l'on dispose d'une connexion de Galois $(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(\mathcal{A}^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp\right)$ avec une fonction γ non injective, l'utilisation de constructions génériques pour définir de nouveaux opérateurs abstraits (dans l'exemple précédent, le fait de combiner une approximation correcte du premier domaine avec une approximation correcte du second) peut occasionner une grande perte de précision. Les constructions génériques sont pourtant fort utiles pour économiser la recherche de nouvelles approximations.

Ce problème peut être, en partie, résolu en utilisant un opérateur $\rho \in \mathcal{A}^\sharp \to \mathcal{A}^\sharp$ qui conserve les approximation correctes : $\gamma \dot{\subseteq} \gamma \circ \rho$ (si $a^\sharp \in \mathcal{A}^\sharp$ est une approximation correcte de $a \in \mathcal{A}$, $\rho(a^\sharp)$ est aussi une approximation correcte). En terme de précision un tel opérateur devient intéressant lorsqu'il est réductif :$\rho \dot{\sqsubseteq}$ id ($\rho(a^\sharp)$ est alors une approximation correcte plus petite que $a^\sharp$). Avec un tel opérateur, tout approximation correcte $f^\sharp$ d'un opérateur concret f peut être améliorée en considérant $\rho \circ f^\sharp \circ \rho$.

Le cadre des connexions de Galois fourni une nouvelle fois un choix optimal pour ρ. En effet si on remarque que $\gamma \dot{\subseteq} \gamma \circ \rho$ est équivalent à $\alpha \circ \gamma \dot{\sqsubseteq} \rho$, $\alpha \circ \gamma$ apparaît comme un opérateur de réduction optimal. Il possède de plus toutes les propriétés d'une réduction précise car il est monotone, réductif et idempotent (*fermeture inférieure*).

Exemple 6. *Dans l'exemple précédent l'opérateur de réduction optimal est définie, après calcul, par la table suivante*

ρ	$\bot_1$	0_1	$-_1$	$+_1$	$\top_1$
$\bot_2$	$(\bot_1, \bot_2)$	$(\bot_1, \bot_2)$	$(\bot_1, \bot_2)$	$(\bot_1, \bot_2)$	$(\bot_1, \bot_2)$
0_2	$(\bot_1, \bot_2)$	$(0_1, 0_2)$	$(-_1, 0_2)$	$(+_1, 0_2)$	$(\top_1, 0_2)$
1_2	$(\bot_1, \bot_2)$	$(\bot_1, \bot_2)$	$(-_1, 1_2)$	$(+_1, 1_2)$	$(\top_1, 1_2)$
$\top_2$	$(\bot_1, \bot_2)$	$(0_1, 0_2)$	$(-_1, \top_2)$	$(+_1, \top_2)$	$(\top_1, \top_2)$

$\rho \circ \text{succ}^\sharp \circ \rho$ *est alors une approximation correcte de* succ *plus précise que* $\text{succ}^\sharp$. *On a par exemple cette fois* $\rho \circ \text{succ}^\sharp \circ \rho(0_1, 1_2) = (\bot_1, \bot_2) = \alpha \circ \text{succ} \circ \gamma(0_1, 1_2)$. *L'opérateur de réduction permet ainsi de faire « communiquer » les abstractions par signes et par congruence. La situation n'est toute fois toujours pas optimale puisque*

$$(-_1, 0_2) = \alpha \circ \text{succ} \circ \gamma(-_1, 1_2) \sqsubset \rho \circ \text{succ}^\sharp \circ \rho(-_1, 1_2) = (\top_1, 0_2)$$

Ainsi un opérateur de réduction permet d'améliorer facilement des approximations correctes d'un opérateur f, *mais pour obtenir une approximation optimale il faut revenir à la définition de* $\alpha \circ f \circ \gamma$.

3.2.3.2 Composition

Les connexions de Galois se prêtent bien à une composition d'abstraction. Si

$$\left(\mathcal{A}_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1\right) \underset{\alpha_{1,2}}{\overset{\gamma_{1,2}}{\leftrightarrows}} \left(\mathcal{A}_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2\right)$$

et

$$\left(\mathcal{A}_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2\right) \underset{\alpha_{2,3}}{\overset{\gamma_{2,3}}{\leftrightarrows}} \left(\mathcal{A}_3, \sqsubseteq_3, \bigsqcup_3, \bigsqcap_3\right)$$

sont des connexions de Galois alors

$$\left(\mathcal{A}_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1\right) \underset{\alpha_{2,3} \circ \alpha_{1,2}}{\overset{\gamma_{1,2} \circ \gamma_{2,3}}{\leftrightarrows}} \left(\mathcal{A}_3, \sqsubseteq_3, \bigsqcup_3, \bigsqcap_3\right)$$

est une connexion de Galois. On peut ainsi définir une chaîne de sémantique abstraite.

3.2.3.3 Partitionnement

Pour abstraire un domaine concret de la forme $(\mathcal{P}(A \times B), \subseteq, \bigcup, \bigcap)$ on peut utiliser une technique générique de *partitionnement*. Pour n'importe quelle connexion de Galois

$$\left(\mathcal{P}(B), \subseteq, \bigcup, \bigcap\right) \underset{\alpha_B}{\overset{\gamma_B}{\leftrightarrows}} \left(B^\sharp, \sqsubseteq^\sharp_B, \bigsqcup^\sharp_B, \bigsqcap^\sharp_B\right)$$

permettant d'abstraire le domaine $\mathcal{P}(B)$, on peut construire la connexion

$$\left(\mathcal{P}(A \times B), \subseteq, \bigcup, \bigcap\right) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \left(A \to B^\sharp, \dot{\sqsubseteq}^\sharp_B, \dot{\bigsqcup}^\sharp_B, \dot{\bigsqcap}^\sharp_B\right)$$

avec

$$\begin{aligned} \alpha(S) &= \lambda a.\alpha_B(\{\ b \mid (a,b) \in S\ \})\ \forall S \subseteq A \times B \\ \gamma(F) &= \{\ (a,b) \mid b \in \gamma_B(F(a))\ \}\ \forall F \in A \to B^\sharp \end{aligned}$$

Il s'agit d'une technique standard lorsque A représente un ensemble de points de contrôle d'un programme. On attache ainsi une information abstraite de $B^\sharp$ à chaque point de contrôle d'un programme.

3.3 Une théorie minimale

La théorie de l'interprétation abstraite que nous avons présentée était essentiellement basée sur la notion de treillis complet et de connexions de Galois. Comme nous l'avons observé, un tel cadre est particulièrement bien adapté pour énoncer des résultats généraux sur différentes techniques de construction d'abstraction, en mettant à chaque fois en évidence les cas optimaux. Dans le cadre de cette thèse, notre but est de spécifier des sémantiques abstraites et de prouver, dans un assistant de preuve, leur

correction vis-à-vis d'une sémantique concrète. Ainsi les informations calculées par une telle sémantique sont sûres. Nous n'établirons cependant que très peu de résultats sur la précision de ces informations. Les outils d'analyse statique certifiée que nous proposerons seront assurés de renvoyer uniquement des réponses correctes aux questions qui leur seront posées, mais aucun théorème Coq ne les empêchera de répondre systématiquement « je ne sais pas ». La capacité d'une analyse à prouver automatiquement des propriétés non-triviales pour certains programmes restera évaluée de manière classique : par preuve « papier » ou tout simplement par test. Seule la propriété critique de correction sémantique sera donc certifiée.

Avec un tel objectif, certaines structures mathématiques trop riches ne sont plus rigoureusement indispensables. Nous allons maintenant expliquer pourquoi l'utilisation des treillis complets et des connexions de Galois n'est pas idéale en Coq. Nous présenterons ensuite le cadre alternatif retenu en présentant les résultats généraux qui y restent valable.

3.3.1 Formalisation des treillis complets en Coq

La figure 3.3 présente la définition du type des treillis complets en Coq. Nous utilisons ici un type enregistrement. La syntaxe est similaire à la déclaration des enregistrements en Caml, si ce n'est que le type du $n^{\text{ième}}$ champ peut dépendre de la valeur du $m^{\text{ième}}$ champ, si $m < n$. Contrairement aux enregistrements Caml, un champ peut être un type. C'est ici le cas du champ `set` (le domaine de base du treillis) qui est déclaré de type **`Type`**[4]. Les quatre autres champs principaux sont `eq`, `order`, `join` et `meet`. Les autres champs correspondent aux preuves des différentes propriétés que les champs principaux doivent vérifier. `order` exprime un ordre partiel vis-à-vis de la relation d'équivalence `eq`[5]. `join` et `meet` sont respectivement la plus petite borne supérieure et la plus grande borne inférieure. Ces deux fonctions prennent en argument une partie du domaine de base, modélisée ici par un prédicat[6] sur les éléments de `set` (donc de type `set` → **`Prop`**). Il s'agit d'un codage standard de la théorie des ensembles en théorie des types. L'appartenance d'un élément `a` de type `set` à une partie `S` (de type `set` → **`Prop`**) s'exprime par la propriété `S a` (`a` vérifie `S`).

S'il est assez facile de prouver des résultats généraux sur les treillis complets en Coq (le fichier `complete.v` regroupe la définition des treillis complets et la preuve des théorèmes 3.1.1 et 3.2.2), il est beaucoup plus difficile d'instancier une telle structure. Si `set` est un type informatif, la fonction `join`, de type `(set→`**`Prop`**`)→set`, devra pour tout prédicat `P` sur `set` calculer l'élément de `set` qui est la plus petite borne supérieure des éléments vérifiant `P`. Or comme nous l'avons vu dans le chapitre 2, le système de types de Coq s'assure que le calcul d'un objet informatif (comme `(join S)`) ne dépend jamais

[4] Nous utilisons ici **`Type`** (et non **`Set`**) pour pouvoir à la fois définir des treillis complets avec un domaine de base informatif (pour le treillis des entiers par exemples), mais aussi de plus haut niveau dans la hiérarchie des types (le treillis des prédicats par exemple).

[5] La formalisation des quotients étant problématique en théorie des types [CPS02], il est généralement plus aisée de raisonner modulo une relation d'équivalence que par rapport à l'égalité standard de Coq

[6] Nous faison une petite simplification ici. Il faudrait en fait considérer les prédicats compatibles avec la relation d'équivalence `eq`. *c.f.* fichier `complete.v` (annexe A)

```
Record CompleteLattice : Type :=
 { set : Type;
   eq : set → set → Prop;
   eq_refl : ∀ x, eq x x;
   eq_sym : ∀ x y, eq x y → eq y x;
   eq_trans : ∀ x y z, eq x y → eq y z → eq x z;

   order : set → set → Prop;
   order_refl : ∀ x y, eq x y → order x y;
   order_antisym : ∀ x y, order x y → order y x → eq x y;
   order_trans : ∀ x y z, order x y → order y z → order x z;

   join : (set → Prop) → set;
   join_bound : ∀ x (f:set → Prop), f x → order x (join f);
   join_lub : ∀ (f:set → Prop) z,
     (∀ x, f x → order x z) → order (join f) z;

   meet : (set → Prop) → set;
   meet_bound : ∀ x (f:set → Prop), f x → order (meet f) x;
   meet_glb : ∀ (f:set → Prop) z,
     (∀ x, f x → order z x) →  order z (meet f)
 }.
```

FIG. 3.3 – Type des treillis complets en Coq

du contenu des objets logiques (comme `S`). Par conséquent si une telle fonction `join` était programmable en Coq, elle serait constante ! La définition Coq des treillis complets avec un domaine de base informatif est donc tout simplement impossible. Pourtant le domaine abstrait doit être informatif si l'on veut pouvoir extraire un analyseur certifié, il faut donc utiliser une structure de treillis plus pauvre pour le domaine abstrait.

Pour le domaine concret la situation est meilleure car son type sera rarement informatif.

Exemple 7. *Le domaine concret est généralement de forme* `A → Prop` *(ensemble d'états par exemple). Sur un tel domaine, la définition de treillis complet en Coq est tout à fait possible. Le lecteur interessé pour se réferer au fichier* `complete.v` *(voir annexe A) qui contient la construction d'un treillis complet de la forme*

```
{ set = A →  Prop,
  eq = fun P Q ⇒ ∀ x, P x ↔ Q x,
  order = fun P Q ⇒ ∀ x, P x → Q x,
  join = fun Q ⇒ fun x ⇒ ∃ P, Q P ∧ P x,
  meet = fun Q ⇒ fun x ⇒ ∀ P, Q P → P x }
```

Un tel exemple permet de bien comprendre le codage ensembliste à l'aide de prédicats.

3.3.2 Formalisation des connexions de Galois en Coq

Les connexions de Galois peuvent être formalisées en Coq à l'aide d'un type enregistrement comme celui présenté dans la figure 3.4. La définition est paramétrée par deux

treillis complets `A` et `B`.

```
Record Galois (A B : CompleteLattice) : Type :=
  { alpha : set A → set B;
    gamma : set B → set A;
    gal_prop1 : ∀ a b, order B (alpha a) b → order A a (gamma b);
    gal_prop2 : ∀ a b, order A a (gamma b) → order B (alpha a) b
  }.
```

FIG. 3.4 – Type des connexions de Galois en Coq

Là encore, il est très facile de prouver les résultats généraux classiques sur les connexions de Galois (*c.f.* fichier `galois.v`). Il est par contre nettement moins aisé de construire une connexion de Galois.

Pour illustrer ce fait, nous allons reprendre l'exemple 2 et présenter la construction de la connexion de Galois associée en Coq (cette construction se trouve dans le fichier `mod2.v`).

Le domaine de base du treillis abstrait est un type énuméré à quatre valeurs.

```
Inductive t : Set:= Top | M0 | M1 | Bot.
```

Le domaine de base du treillis concret est $\mathcal{P}(\mathbb{Z})$, donc `Z → Prop` en Coq. Les quatre abstractions choisies dans le domaine concret sont ainsi définies :

```
Definition emptyZ : Z → Prop := fun _ ⇒ False.
Definition allZ : Z → Prop := fun _ ⇒ True.
Definition Z0 : Z → Prop := fun x ⇒ x mod 2 = 0.
Definition Z1 : Z → Prop := fun x ⇒ x mod 2 = 1.
```

La définition de `gamma : t → (Z → Prop)` est particulièrement simple.

```
Definition gamma (a:t) : Z → Prop :=
  match a with
    Top ⇒ allZ
  | M0 ⇒ Z0
  | M1 ⇒ Z1
  | Bot ⇒ emptyZ
  end.
```

Puisque le domaine concret est celui des prédicats sur `Z`, `gamma` peut tout aussi bien être défini comme une relation inductive sur `t` et `Z`.

```
Inductive gamma' : t → Z → Prop :=
   gamma_top : ∀ x, gamma' Top x
 | gamma_0 : ∀ x, x mod 2 = 0 → gamma' M0 x
 | gamma_1 : ∀ x, x mod 2 = 1 → gamma' M1 x.
```

Cette définition est préférable pour pouvoir profiter des tactiques puissantes qui sont associées aux types inductifs Coq.

Pour la fonction `alpha`, la tâche est beaucoup plus difficile. En effet, son type `(Z → Prop) → t` laisse présager la même difficulté que pour la construction de `join` dans les treillis complets. Pour contourner les restrictions du système de types de Coq il

faut se résoudre à introduire des axiomes. L'axiome dont nous avons besoin ici est celui de la *description classique*.

```
Axiom description : ∀ (A B:Type) (R:A → B → Prop),
   (∀ x:A, ∃ y : B, R x y) →
   (∀ x:A, (∀ y1 y2:B, R x y1 → R x y2 → y1 = y2)) →
    A → B.
Axiom description_prop : ∀ (A B:Type) (R:A → B → Prop)
   (defined:∀ x:A, ∃ y : B, R x y)
   (functional:∀ x:A, (∀ y1 y2:B, R x y1 → R x y2 → y1 = y2)),
 ∀ x:A, R x (description A B R defined functional x).
```

Si une relation R sur des ensembles A et B est définie pour tout élément de A et est fonctionnelle alors il existe une fonction $f \in A \to B$ qui vérifie $(a, f(a)) \in R$ pour tout $a \in A$. Avec un tel axiome `alpha` peut être définie. On introduit pour cela la relation `alpha_rel` associée à son graphe.

```
Inductive alpha_rel (P:Z→Prop) : t → Prop :=
 | alpha_bot : incl P emptyZ → alpha_rel P Bot
 | alpha_M0 :
    ¬ incl P (Empty Z) → incl P Z0 → alpha_rel P M0
 | alpha_M1 :
    ¬ incl P (Empty Z) → incl P Z1 → alpha_rel P M1
 | alpha_top :
    ¬ incl P Z0 → ¬ incl P Z1 → alpha_rel P Top.
```

`incl` désigne ici l'inclusion logique entre deux prédicats. Pour pouvoir utiliser l'axiome de description il faut prouver les deux lemmes suivants.

```
Lemma alpha_defined : ∀ P, ∃ a:t, alpha_rel P a.
Proof. ...(16 tactiques)... Qed.

Lemma alpha_functional : ∀ P a1 a2,
  alpha_rel P a1 → alpha_rel P a2 → a1=a2.
Proof. ...(42 tactiques)... Qed.
```

Les preuves de ces deux lemmes demandent d'utiliser l'axiome du tiers exclu, par exemple pour pouvoir affirmer qu'on se trouve dans l'un des quatre cas $P = \emptyset$ ou $P \neq \emptyset \wedge P \subseteq \mathbb{Z}_0$ ou $P \neq \emptyset \wedge P \subseteq \mathbb{Z}_1$ ou $P \not\subseteq \mathbb{Z}_0 \wedge P \not\subseteq \mathbb{Z}_1$. La fonction `alpha` peut ainsi être finalement définie.

```
Definition alpha : (Z → Prop) → t :=
  description (Z → Prop) t alpha_rel alpha_defined alpha_functional.
```

La différence de difficulté entre la définition de `alpha` et `gamma` est ici flagrante. De plus, l'axiome de description classique que nous avons utilisé ici est extrêmement fort pour le Calcul des Constructions. Dans [CPS02] Laurent Chicli *et al.* démontrent qu'une version plus faible de cet axiome rend le CCI incohérent. Ce résultat est cependant relatif à une version du CCI différente de celle actuellement utilisée dans Coq[7]. Un tel axiome aurait pu être évité en se passant d'une vision fonctionnelle de `alpha`, mais il aurait de toute

[7] Le système de types sous-jacent à Coq a en effet été récemment modifié : les définitions *impredicatives* d'éléments de type **Set** (définitions d'objets dans **Set** qui utilisent une quantification sur tous les éléments de **Set**) sont désormais interdites.

façon fallu prouver les lemmes `alpha_defined` et `alpha_functional` (et donc utiliser le tiers exclu) pour pouvoir raisonner par la suite sur `alpha`.

Ce type de difficulté nous a poussé à chercher un cadre pour l'interprétation abstraite plus adapté à la logique intuitionniste de Coq (donc sans avoir recours au tiers exclu) et aux restrictions du système de types de Coq sur la construction des objets informatifs à partir d'objets logiques.

3.3.3 Le cadre retenu

Notre motivation pour appauvrir le cadre d'interprétation abstraite présenté au début de chapitre, n'est pas de pouvoir y faire rentrer plus de choses, mais d'économiser l'écriture de certaines preuves, voire l'usage d'axiomes dangereux pour la cohérence de Coq.

Au final, le cadre que nous avons retenu ce caractérise par

- une simple structure de treillis sur le domaine abstrait $A^\sharp$
- un domaine concret de la forme $A = \mathcal{P}(\mathcal{D})$ (`D → Prop` en Coq)
- une fonction de concrétisation $\gamma \in A^\sharp \to A$, morphisme d'intersection binaire : $\forall a_1^\sharp, a_2^\sharp \in A^\sharp,\ \gamma(a_1^\sharp \sqcap^\sharp a_2^\sharp) = \gamma(a_1^\sharp) \sqcap \gamma(a_2^\sharp)$
- l'utilisation de la notion de post-point fixe au lieu de celle de point fixe

Nous allons maintenant commenter chacun de ces points.

Une simple structure de treillis sur le domaine abstrait Dans ce cadre l'existence de point fixe n'est plus assurée pour les fonctions monotones du domaine abstrait. En pratique, il nous faudra de toutes façon proposer des algorithmes pour calculer des points fixes (ou les approcher), ce qui n'était pas assuré dans le cadre des treillis complets (le calcul des points fixes y est indécidable, *c.f.* [Cou78]).

Un domaine concret de la forme $A = \mathcal{P}(\mathcal{D})$ La restriction prise sur la forme des domaines concrets nous permettra de définir les fonctions de concrétisation à l'aide de relations inductives (de type `A♯ → D → Prop`) et ainsi de profiter de l'habilitée du système à manipuler ces constructions. Un tel domaine concret ne paraît pas limiter l'expressivité du cadre, en pratique.

Une fonction de concrétisation $\gamma \in A^\sharp \to A$, morphisme d'intersection binaire Ne disposant plus de fonction d'abstraction, le seul critère dont nous disposons pour abstraire les fonctions devient :

Définition 3.3.1. Approximation correcte de fonctions (nouvelle définition). *Soit* $(A, \sqsubseteq)$ *et* $(A^\sharp, \sqsubseteq^\sharp)$ *deux ensembles partiellement ordonnés,* γ *une fonction de* $(A^\sharp, \sqsubseteq^\sharp)$ *dans* $(A, \sqsubseteq)$, $f^\sharp \in A^\sharp \to A^\sharp$ *est* une approximation correcte *de* $f \in A \to A$ *si et seulement si* $\forall a^\sharp \in A^\sharp, f \circ \gamma(a^\sharp) \sqsubseteq \gamma \circ f(a^\sharp)$.

Sans même faire d'hypothèse sur γ, nous conservons un principe de composition entre les abstractions correctes de fonctions.

Lemme 3.2. *Si* $f_1^\sharp, f_2^\sharp \in A^\sharp \to A^\sharp$ *sont des approximations correctes des fonctions concrètes* f_1 *et* f_2 *dans* $A \to A$*, si* f_1 *est monotone, alors* $f_1^\sharp \circ f_2^\sharp$ *est une approximation correcte de* $f_1 \circ f_2$*.*

Preuve :

$$\begin{aligned} f_1 \circ f_2 \circ \gamma &\dot{\sqsubseteq}^\sharp f_1 \circ \gamma \circ f_2^\sharp \\ &\dot{\sqsubseteq}^\sharp \gamma \circ f_1^\sharp \circ f_2^\sharp \end{aligned}$$

□

La propriété minimale à requérir sur γ est sa monotonie. Elle permet en effet de conserver un résultat de transfert de point fixe similaire à celui du théorème 3.2.3. Si l'on se restreint à la notion de post-point fixe, on peut énoncer le résultat suivant :

Théorème 3.3.1. *Soit* $(A, \sqsubseteq)$ *et* $(A^\sharp, \sqsubseteq^\sharp)$ *deux ensembles partiellement ordonnés,* γ *une fonction monotone entre* $(A^\sharp, \sqsubseteq^\sharp)$ *et* $(A, \sqsubseteq)$*,* $f \in A \to A$ *une fonction admettant un plus petit post-point fixe* $\mathrm{lpfp}(f)$ *et* $f^\sharp \in A^\sharp \to A^\sharp$ *une abstraction correcte de* f*. Tous les post-points fixes de* $f^\sharp$ *sont des approximations correctes de* $\mathrm{lpfp}(f)$ *:*

$$\forall a^\sharp \in A^\sharp,\ f^\sharp(a^\sharp) \sqsubseteq^\sharp a^\sharp \ \Rightarrow\ \mathrm{lpfp}(f) \sqsubseteq \gamma(a^\sharp)$$

Preuve : Soit $a^\sharp$ un post-point fixe de $f^\sharp$: $f^\sharp(a^\sharp) \sqsubseteq^\sharp a^\sharp$. Puisque γ est monotone, nous avons : $\gamma \circ f^\sharp(a^\sharp) = \gamma(f^\sharp(a^\sharp)) \sqsubseteq \gamma(a^\sharp)$. Or $f^\sharp$ est une approximation correcte de f, donc par transitivité $f \circ \gamma(a^\sharp) = f(\gamma(a^\sharp)) \sqsubseteq \gamma(a^\sharp)$. $\gamma(a^\sharp)$ est donc un post-point fixe de f. $\mathrm{lpfp}(f)$ étant le plus petit, nous pouvons en conclure $\mathrm{lpfp}(f) \sqsubseteq \gamma(a^\sharp)$. □

La monotonie de γ n'est cependant pas toujours suffisante : elle n'assure pas un bon comportement de γ vis-à-vis de l'intersection abstraite. Le théorème suivant précise la situation.

Théorème 3.3.2. *Soit* $(A, \sqsubseteq, \sqcup, \sqcap)$ *et* $(A^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp)$ *deux treillis et* γ *une fonction de* $A^\sharp \to A$*. Les 3 assertions suivantes sont équivalentes*

(i) γ *est monotone*

(ii) $\forall a_1^\sharp, a_2^\sharp \in A^\sharp,\ \gamma(a_1^\sharp) \sqcup \gamma(a_2^\sharp) \sqsubseteq \gamma(a_1^\sharp \sqcup^\sharp a_2^\sharp)$

(iii) $\forall a_1^\sharp, a_2^\sharp \in A^\sharp,\ \gamma(a_1^\sharp \sqcap^\sharp a_2^\sharp) \sqsubseteq \gamma(a_1^\sharp) \sqcap \gamma(a_2^\sharp)$

Preuve : Supposons que *(i)* est vrai. $a_1^\sharp \sqsubseteq^\sharp a_1^\sharp \sqcup^\sharp a_2^\sharp$ et $a_2^\sharp \sqsubseteq^\sharp a_1^\sharp \sqcup^\sharp a_2^\sharp$ donc, par monotonie de γ, on peut affirmer que $\gamma(a_1^\sharp) \sqsubseteq \gamma(a_1^\sharp \sqcup^\sharp a_2^\sharp)$ et $\gamma(a_2^\sharp) \sqsubseteq \gamma(a_1^\sharp \sqcup^\sharp a_2^\sharp)$. On en déduit ainsi *(ii)*.

Inversement, si *(ii)* est vérifié, pour tout $a_1^\sharp, a_2^\sharp$ dans $A^\sharp$ tels que $a_1^\sharp \sqsubseteq^\sharp a_2^\sharp$, $a_1^\sharp \sqcup^\sharp a_2^\sharp = a_2^\sharp$ donc $\gamma(a_2^\sharp) \sqcup \gamma(a_2^\sharp) = \gamma(a_1^\sharp \sqcup^\sharp a_2^\sharp) = \gamma(a_2^\sharp)$. On en déduit que $\gamma(a_1^\sharp) \sqsubseteq \gamma(a_2^\sharp)$.

La preuve de *(i)*⇔*(iii)* est similaire. □

Ce résultat nous indique une bonne nouvelle pour $\sqcup^\sharp$ et une mauvaise pour $\sqcap^\sharp$, lorsque la fonction de concrétisation est monotone. $\sqcup^\sharp$ est en effet une approximation correcte de $\sqcup$, mais aucun résultat similaire n'est cependant valable pour l'intersection (l'inégalité *(iii)* est dans le mauvais sens).

Le fait que $\sqcap^\sharp$ soit une approximation correcte de $\sqcap$ est pourtant une propriété fort utile en pratique, puisqu'elle permet de combiner deux approximations correctes d'une même valeur concrète. C'est pour cette raison que nous nous intéressons aux fonctions de concrétisation morphisme d'intersection. Ce qui revient à demander, de manière équivalente que γ soit monotone et que $\sqcap^\sharp$ soit une approximation correcte de $\sqcap$.

Une telle propriété assure aussi une certaine qualité (au sens énoncé dans la section 3.1) : si un élément concret admet un ensemble fini d'approximations correctes, alors il existe une meilleure approximation pour cet ensemble.

Théorème 3.3.3. *Soit* $(A, \sqsubseteq, \sqcup, \sqcap)$ *et* $\left(A^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp\right)$ *deux treillis,* γ *un morphisme d'intersection entre* $\left(A^\sharp, \sqcap^\sharp\right)$ *et* $(A, \sqcap)$*,* a *un élément de* A *et* $a_1^\sharp, \ldots, a_n^\sharp \in A^\sharp$ *des approximations correctes de* a*, alors* $a_1^\sharp \sqcap^\sharp \cdots \sqcap^\sharp a_n^\sharp$ *est une approximation correcte de* a*, meilleure que celles fournies par chacun des* $a_1^\sharp, \ldots, a_n^\sharp$*.*

Preuve : On montre par récurrence sur n que $\gamma(a_1^\sharp \sqcap^\sharp \cdots \sqcap^\sharp a_n^\sharp) = \gamma(a_1^\sharp) \sqcap \cdots \sqcap \gamma(a_n^\sharp)$. □

Utilisation de la notion de post-point fixe Enfin, nous utiliserons la notion de post-point fixe plutôt que de point fixe car en Coq, les sémantiques se caractérisent plus facilement sous forme de plus petit post-point fixe que sous forme de plus petit point fixe. Les deux notions sont équivalentes mais le principe d'induction associé à une définition inductive permet de prouver la première caractérisation sans effort. Prenons l'exemple 3 des états accessibles à partir d'un ensemble d'états initiaux $\mathcal{S}_0$ et pour une relation de transition R donnée. Nous définissons donc l'environnement Coq un prédicat `S0` et une relation `R` sur un type de base `A`.

```
Variable A : Set.
Variable R : A → A → Prop.
Variable S0 : A → Prop.
```

L'ensemble des états accessibles se définit ensuite à l'aide d'une définition inductive avec deux règles : les états initiaux sont accessibles et les successeurs des états accessibles sont accessibles.

```
Inductive reach : A → Prop :=
  reach_init : ∀ a, S0 a → reach a
| reach_next : ∀ a1 a2, R a1 a2 → reach a1 → reach a2.
```

Comme pour toute définition inductive, Coq génère un principe d'induction associé à `reach`, exprimant le fait que ce prédicat est le plut petit clos par ces constructeurs `reach_init` et `reach_next`.

```
reach_ind
     : ∀ P : A → Prop,
       (∀ a : A, S0 a → P a) →
       (∀ a1 a2 : A, R a1 a2 → reach a1 → P a1 → P a2) →
       ∀ a : A, reach a → P a
```

En termes de post-point fixe, les deux constructeurs de `reach` affirment que le prédicat `reach` est un post-point fixe de l'opérateur $\lambda S.\mathcal{S}_0 \cup \text{post}[R](S)$ et le principe d'induction

`reach_ind` démontre que c'est le plus petit. `reach` est ainsi caractérisé sous forme de plus petit post-point fixe sans aucune preuve Coq !

3.4 Conclusions et références bibliographiques

La section 3.1 est directement inspirée de [CC79] et [Cou78]. La section 3.2 s'inspire plus particulièrement de [CC92a] et [Cou78]. Le but de ces deux premières sections était de présenter l'interprétation abstraite de manière introductive en motivant chacune des définitions proposées. Il nous apparaissait en effet nécessaire d'éviter une succession de définitions mathématiques sans motiver leur introduction. Cette synthèse constitue, de plus, un témoignage de la réflexion que nous avons menée pour sélectionner, en connaissance de cause, « notre » cadre.

Le cadre traditionnel présenté dans ces deux sections est parfois apparu comme trop restrictif : certaines abstractions utiles ne satisfont pas l'hypothèse d'existence d'une meilleure abstraction. C'est par exemple le cas de l'abstraction d'un ensemble de variables numériques par leur enveloppe convexe [CH78]. Une large couverture des différents cadres possibles est présentée dans [CC92b]. En ce qui concerne nos travaux, nous avons modifié le cadre standard pour une tout autre raison : leur facilité de manipulation dans la logique intuitionniste de Coq et dans son système de types.

La qualité de ce cadre apparaîtra lorsque nous présenterons les différentes analyses statiques qu'il nous a permis de certifier. La théorie de l'interprétation abstraite avait déjà été formalisée en Coq durant les travaux de DEA de David Monniaux [Mon98]. Le cadre alors choisi était celui des connexions de Galois. Plusieurs théorèmes sur les combinaisons de connexions y étaient formalisés. Cependant, le choix des connexions de Galois avait nécessité l'introduction de nombreux axiomes et n'avait pas permis de proposer au final, des analyses statiques certifiées conséquentes. C'est pour cette raison qu'il nous est apparu important de bien sélectionner dans la théorie standard les ingrédients suffisants pour aboutir dans le temps imparti à cette thèse à des analyses intéressantes.

Le cadre que nous avons retenu perd néanmoins une propriété importante du cadre standard : le fait de pouvoir dériver une approximation correcte $f^\sharp$ à partir de la spécification $\alpha \circ f \circ \gamma$. Plusieurs exemples de telles dérivations sont donnés dans [Cou99]. Il nous paraît intéressant de trouver un cadre qui permette ce genre de manipulation symbolique, tout en restant facilement formalisable en Coq. Nous y reviendrons dans notre chapitre de conclusion.

Chapitre 4

Calcul et approximation de points fixes

Dans le chapitre précédant, nous avons présenté différentes techniques pour spécifier une analyse statique. Afin de pouvoir implémenter de telles spécifications, il nous faut maintenant proposer des algorithmes permettant de calculer des post-points fixes de fonctions. La théorie de l'interprétation abstraite, et plus généralement celle des treillis, propose de tels algorithmes. Nous allons nous intéresser dans ce chapitre à certains d'entre eux. Tous seront génériques : ils pourront être utilisés tels quels pour une grande palette d'analyses statiques. Ainsi, la majeure partie du travail nécessaire à la programmation d'analyse statique en Coq se situera au niveau de la spécification, la partie résolution de (post-)point fixe étant pour sa part réutilisable dans de nombreux contextes.

Nous allons dans une première section récapituler les résultats d'existence de plus petit point fixe dans les treillis complets. Nous aborderons ensuite trois critères permettant de calculer ou d'approcher un plus petit point fixe : la condition de chaîne ascendante dans la section 4.2, les techniques d'approximation par élargissement/rétrécissement dans la section 4.3 puis un critère un peu moins standard, adapté du précédent, lors de la section 4.4. Pour chacun de ces critères nous proposerons des programmes certifiés de calcul itératif.

4.1 Les points fixes d'une fonction monotone dans un treillis complet

Le théorème 3.2.2 de Knaster-Tarski présenté dans le chapitre précédent prouve l'existence, dans un treillis complet, d'un plus petit point fixe pour toute fonction monotone. Un résultat dual peut être énoncé pour les plus grands points fixes.

Théorème 4.1.1. ***(Knaster-Tarski pour les plus grands points fixes)*** *Dans un treillis complet* $(\mathsf{A}, \sqsubseteq, \bigsqcup)$*, pour toute fonction monotone* $\mathsf{f} \in \mathsf{A} \to \mathsf{A}$*, le plus grand point fixe* $\mathrm{gfp}(\mathsf{f})$ *de* f *existe et vaut* $\bigsqcup\{\mathsf{x} \in \mathsf{A} \mid \mathsf{x} \sqsubseteq \mathsf{f}(\mathsf{x})\}$.

Ces deux résultats font ainsi apparaître trois zones importantes : l'ensemble des points fixes

$$\mathrm{FP}(f) \stackrel{\text{def}}{=} \{ x \in A \mid f(x) = x \}$$

l'ensemble des post-points fixes

$$\mathrm{PostFP}(f) \stackrel{\text{def}}{=} \{ x \in A \mid f(x) \sqsubseteq x \}$$

et l'ensemble des pré-points fixes

$$\mathrm{PréFP}(f) \stackrel{\text{def}}{=} \{ x \in A \mid x \sqsubseteq f(x) \}$$

La situation énoncée peut être résumée par le schéma de la figure 4.1.

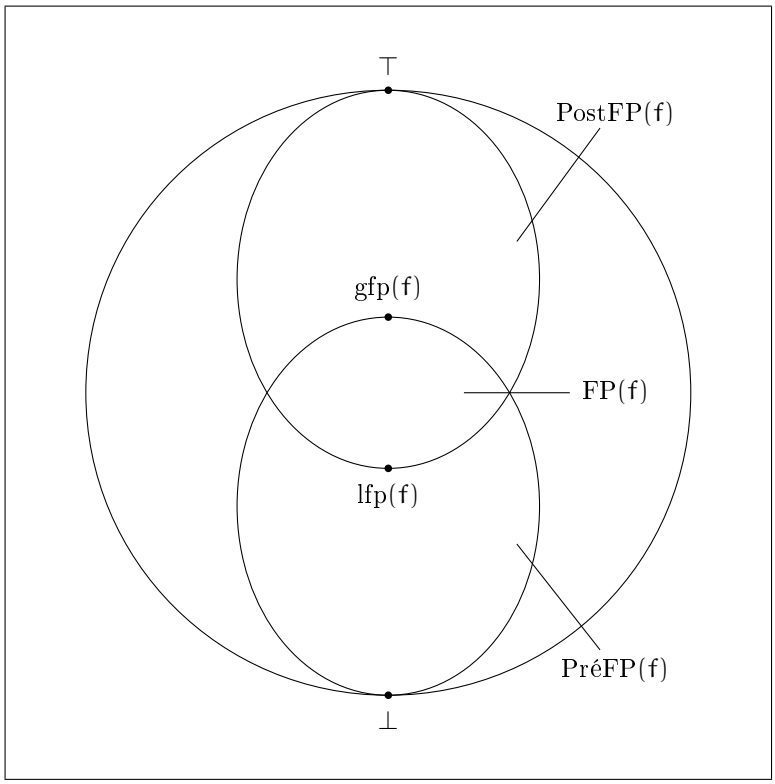


FIG. 4.1 – Positions relatives des points fixes, post-points fixes et pré-points fixes dans un treillis complet

Le plus petit élément du treillis $\bot$ est aussi le plus petit élément de PostFP(f) et le plus grand élément $\top$ est le plus grand élément de PréFP(f). Ainsi PostFP(f) (respectivement PréFP(f)) n'est pas vide si $\bot$ (respectivement $\top$) existe. D'après la remarque 3.2, ces deux conditions sont respectées dans un treillis complet.

Le théorème de Knaster-Tarski a l'inconvénient de ne pas proposer de méthode de calcul pour le plus petit point fixe d'une fonction, car l'opérateur $\bigsqcap$ est rarement

calculable. Lorsque la fonction f est continue[1] ($\forall s \subseteq \mathcal{A},\ \sqcup f(S) = f(\sqcup S)$), le plus petit point fixe de f peut être caractérisé par une formule « plus constructive », au sens où elle donne une méthode de calcul.

Théorème 4.1.2. *Dans un treillis complet* $(A, \sqsubseteq, \bigsqcup)$*, pour toute fonction continue* $f \in A \to A$*, le plus petit point fixe* $\mathrm{lfp}(f)$ *de* f *est égal à* $\bigsqcup\{\ f^n(\bot) \mid n \in \mathbb{N}\ \}$.

Il faut remarquer que, puisque la suite $\bot, f(\bot), \ldots, f^n(\bot), \ldots$ est croissante (par monotonie de f, toute fonction continue étant monotone), nous avons pour tout entier k

$$\bigsqcup\{\ f^n(\bot) \mid n \in \mathbb{N},\ n \leq k\ \} = f^k(\bot)$$

Le résultat précédent peut donc se noter sous la forme

$$\mathrm{lfp}(f) = \lim_{n \to +\infty} f^n(\bot)$$

avec $\lim_{n\to+\infty} x_n$ définie pour toute chaîne croissante $(x_n)_n$ par

$$\lim_{n \to +\infty} x_n = \bigsqcup\{\ x_n \mid n \in \mathbb{N}\ \}$$

La méthode de calcul suggérée ici consiste donc à itérer f sur $\bot$ jusqu'à obtenir le plus petit point fixe de f. Il s'agit cependant d'une méthode semi-complète puisqu'un tel calcul peut ne pas terminer.

Exemple 8. $\overline{\mathbb{N}} = \mathbb{N} \cup \{\infty\}$ *est un treillis complet pour l'ordre* $\leq$ *étendu (avec* ∞ *comme plus grand élément et* 0 *comme plus petit). L'opérateur* succ *défini par* $\mathrm{succ}(n) = n + 1$ *(*$\infty + 1 = \infty$*) est continu. L'ensemble des post-points fixes de* f *est réduit à* $\{\infty\}$*. Son plus petit point fixe est donc* ∞*. Si l'on considère la suite des itérés* $(\mathrm{succ}^n(0))_n$*, pour tout* n*,* $\mathrm{succ}^n(0) = n < \infty$*. Cette chaîne est donc infinie et n'atteint jamais le point fixe de la fonction* succ[2].

Pour calculer le plus petit point fixe d'une fonction monotone (ou continue) avec la méthode d'itération précédente il faut donc disposer d'hypothèses supplémentaires assurant la finitude des calculs.

4.2 Condition de chaîne ascendante

Le critère le plus simple que l'on puisse proposer consiste à interdire l'existence de chaînes croissantes infinies. On obtient ainsi un critère indépendant d'une fonction particulière, résultat primordial pour proposer un outil de calcul générique : la preuve de terminaison de l'outil ne dépend pas de l'analyse statique étudiée, seulement du treillis abstrait utilisé.

[1] L'hypothèse de continuité peut être réduite à une simple monotonie en considérant des itérations transfinies [Cou78].

[2] Le point fixe est atteint en itérant jusqu'au premier ordinal ω :

$$\mathrm{succ}^\omega(\bot) = \bigsqcup_{n<\omega} \mathrm{succ}^n(\bot) = \bigsqcup\{\ \mathrm{succ}^n(0) \mid n \in \mathbb{N}\ \} = \bigsqcup\{\ n \mid n \in \mathbb{N}\ \} = \infty$$

4.2.1 Condition de chaîne ascendante classique

Définition 4.2.1. Condition de chaîne ascendante classique.
Un ensemble partiellement ordonné $(A, \sqsubseteq)$ *vérifie* la condition de chaîne ascendante classique *si pour toute suite croissante* $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ *il existe un indice* k *à partir duquel la suite est stationnaire* $(\forall n \geq k,\ x_k = x_n)$.

Théorème 4.2.1. *Soit* $(A, \sqsubseteq)$ *un ensemble partiellement ordonné vérifiant la condition de chaîne ascendante classique,* f *une fonction monotone et* a *un pré-point fixe de* f. *La suite* $a, f(a), \ldots, f^n(a), \ldots$ *est stationnaire à partir d'un certain rang. Sa limite est le plus petit point fixe de* f *plus grand que* a.

Preuve : Puisque a est un pré-point fixe de f et que f est monotone, on peut montrer par récurrence que la suite $a, f(a), \ldots, f^n(a), \ldots$ est croissante. La condition de chaîne ascendante assure alors qu'il existe un rang k tel que $f^k(a) = f^{k+1}(a)$. $f^k(a)$ est ainsi un point fixe de f. Si x est un point fixe de f plus grand que a, on montre par récurrence que $f^n(a) \sqsubseteq x$ pour tout entier n. Cela démontre en particulier que $f^k(a) \sqsubseteq x$. □

Si $(A, \sqsubseteq)$ possède un plus petit élément $\bot$, c'est un pré-point fixe de f. La série $(f^n(\bot))_n$ converge alors vers le plus petit point fixe de f.

Nous énonçons ce résultat pour un simple ensemble partiellement ordonné. La structure de treillis n'est plus nécessaire puisque l'existence d'un plus petit point fixe est maintenant assurée de manière constructive. La condition de chaîne ascendante est cependant une hypothèse forte : les treillis vérifiant cette condition sont généralement des treillis complets. Il leur suffit de posséder un plus petit élément.

Théorème 4.2.2. *Tout treillis muni d'un plus petit élément et vérifiant la condition de chaîne ascendante admet une structure de treillis complet relative au même ordre partiel.*

Preuve : *cf.* [DP90] page 39 □

4.2.2 Condition de chaîne ascendante constructive

Pour pouvoir implémenter cet algorithme de calcul en Coq, il nous faut proposer une définition de la propriété de chaîne ascendante qui se prête mieux aux preuves constructives[3]. Nous utiliserons pour cela les notions d'*accessibilité* [Acz77, Nor88] et de *relation bien fondée*.

Définition 4.2.2. Accessibilité.
Étant donnée une relation $\prec$ *sur un ensemble* A, *l'ensemble* $\mathrm{Acc}_\prec$ *des éléments* accessibles *pour* $\prec$ *est défini inductivement par*

$$\frac{\forall y \in A,\ y \prec x \ \Rightarrow\ y \in \mathrm{Acc}_\prec}{x \in \mathrm{Acc}_\prec}$$

[3] Nous verrons en effet que le critère classique est difficilement manipulable en logique intuitionniste.

Définition 4.2.3. Relation bien fondée.
Une relation $\prec$ *sur un ensemble* A *est* bien fondée *si tous les éléments de* A *sont accessibles pour* $\prec$.

Intuitivement un élément est accessible pour une relation $\prec$ s'il n'est le point de départ d'aucune chaîne infinie décroissante pour $\prec$. Un exemple trivial d'élément accessible est un élément sans prédécesseur.

Définition 4.2.4. Condition de chaîne ascendante constructive.
Un poset $(A, \sqsubseteq)$ *vérifie* la condition de chaîne ascendante *si l'ordre strict inverse* $\sqsupset$ *associé est bien fondé.*

En logique classique les deux critères de condition de chaîne ascendante sont équivalents. Le sens "Classique $\Rightarrow$ Constructif" demande même l'utilisation de l'axiome du choix pour exhiber une chaîne croissante. En logique intuitionniste, il n'y a plus d'équivalence. Pour étayer un peu cette affirmation, nous proposons maintenant une démonstration pour établir qu'il n'existe pas de preuve constructive du fait que le poset $(\mathbb{B}, =, \sqsubseteq)$ vérifie la condition de chaîne ascendante classique avec $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$ et $\mathsf{F} \sqsubseteq_{\mathbb{B}} \mathsf{T}$. Le diagramme de Hasse de ce poset très simple est présenté dans la figure 4.2.

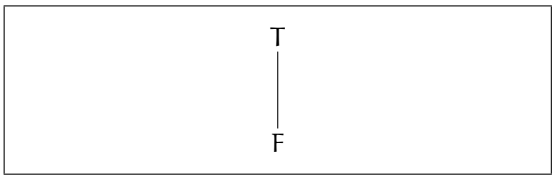


FIG. 4.2 – Poset bi-valué

Théorème 4.2.3. *Il n'existe pas de preuve constructive du fait que le poset* $(\mathbb{B}, =, \sqsubseteq_{\mathbb{B}})$ *vérifie la condition de chaîne ascendante classique.*

Preuve : Si une telle preuve constructive existait, il existerait une procédure effective (sous forme d'une machine de Turing $M_{\mathbb{B}}$) qui, pour tout codage de fonction $f : \mathbb{N} \to \mathbb{B}$ monotone, construit un entier k vérifiant $\forall n, n \geq k \Rightarrow f(k) = f(n)$. Montrons que l'on pourrait alors décider le problème de l'arrêt d'une machine de Turing sur le mot vide, problème pourtant notoirement indécidable.

Notons $L_\emptyset$ le langage des machines de Turing dont l'exécution sur le mot vide termine et considérons $M_\emptyset$ une machine de Turing qui, étant donné le codage d'une machine, réalise les opérations suivantes :

1. $M_\emptyset$ produit le codage de la fonction $\text{arret}_M \in \mathbb{N} \to \mathbb{B}$ définie par

$$\forall n \in \mathbb{N},\ \text{arret}(n) = \begin{cases} \mathsf{F} & \text{si } \varepsilon \to_n \cdot \text{ est définie} \\ \mathsf{T} & \text{si } \varepsilon \to_n \cdot \text{ n' est pas définie} \end{cases}$$

avec $m \to_n (d, c)$ si et seulement si la machine M va dans la direction d en remplaçant le caractère courant par c, après n transition(s) sur le mot m.

2. $\mathsf{M}_\emptyset$ exécute ensuite la machine $\mathsf{M}_\mathbb{B}$ sur le codage de arret_M (entrée licite puisque arret_M est monotone)
3. $\mathsf{M}_\emptyset$ utilise enfin le résultat k du calcul précédent pour tester la valeur de $\text{arret}_\mathsf{M}(\mathsf{k})$: si $\text{arret}_\mathsf{M}(\mathsf{k}) = \mathsf{T}$ alors $\mathsf{M}_\emptyset$ accepte son entrée M, sinon elle la refuse.

Si $\mathsf{M}_\emptyset$ accepte une machine M, alors il existe un entier k tel que $\varepsilon \rightarrow_\mathfrak{n} \cdot$ n' est pas défini pour tout $\mathfrak{n}$ supérieur ou égal à k : M s'arrête sur le mot vide.

Inversement, si M s'arrête sur le mot vide, alors il existe un entier k (le nombre de pas nécessaire pour stopper la machine) tel que pour tout entier $\mathfrak{n}$ supérieur ou égal à k, $\text{arret}_\mathsf{M}(\mathsf{k})$ soit égal à T. Le calcul de $\mathsf{M}_\mathbb{B}$ sur arret_M va donc retourner un entier k' tel que $\text{arret}_\mathsf{M}(\mathsf{k}')$ est égal à T. $\mathsf{M}_\emptyset$ va donc accepter M.

La machine $\mathsf{M}_\emptyset$ déciderait donc $\mathsf{L}_\emptyset$, ce qui est impossible. □

Le même poset vérifie pourtant la condition de chaîne ascendante constructive, et cela peut être prouvé en logique intuitionniste.

Théorème 4.2.4. *Il existe une preuve constructive du fait que le poset* $(\mathbb{B}, =, \sqsubseteq_\mathbb{B})$ *vérifie la condition de chaîne ascendante constructive.*

Preuve : L'ordre strict inverse de $\sqsubseteq_\mathbb{B}$ étant réduit à un seul couple (T, F), la preuve est aisée.

- $\mathsf{T} \in \text{Acc}_{\sqsupset_\mathbb{B}}$ puisque T n'a pas de prédécesseur pour $\sqsupset_\mathbb{B}$.
- $\mathsf{F} \in \text{Acc}_{\sqsupset_\mathbb{B}}$ puisque le seul prédécesseur de F est T et qu'il est lui même accessible.

□

Nous ferons désormais uniquement référence au critère constructif. Nous allons maintenant présenter la formalisation Coq d'un calcul de plus petit point fixe sur les treillis vérifiant cette condition de chaîne ascendante.

4.2.3 Calcul de plus petit point fixe en Coq

La structure de base sur laquelle nous effectuons nos calculs est le treillis.

4.2.3.1 Définition des treillis en Coq

La figure 4.3 présente la signature des treillis en Coq. Cette signature nécessite plusieurs commentaires.

Module/enregistrements Nous utilisons ici les modules de Coq. Le type donné est donc celui d'un module. Nous aurions aussi bien pu utiliser un enregistrement Coq, comme nous l'avions fait pour la définition des treillis complets dans la section 3.3.1. Le mécanisme d'extraction éprouve cependant des difficultés à extraire certains enregistrements lorsque ceux-ci contiennent des définitions de type. En effet, les enregistrements Caml n'en contiennent pas. Une telle difficulté n'existe pas avec les modules. La figure 4.4 présente la signature des treillis extraite à partir de la signature `Coq`. Tous les objets logiques assurant la correction des différents opérateurs sont alors supprimés.

```
Module Type Lattice.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  Parameter eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter order : t → t → Prop.
  Parameter order_refl : ∀ x y : t, eq x y → order x y.
  Parameter order_antisym :
    ∀ x y : t, order x y → order y x → eq x y.
  Parameter order_trans :
    ∀ x y z : t, order x y → order y z → order x z.
  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

  Parameter join  : t → t → t.
  Parameter join_bound1 : ∀ x y : t, order x (join x y).
  Parameter join_bound2 : ∀ x y : t, order y (join x y).
  Parameter join_least_upper_bound :
    ∀ x y z : t, order x z → order y z → order (join x y) z.

  Parameter meet  : t → t → t.
  Parameter meet_bound1 : ∀ x y : t, order (meet x y) x.
  Parameter meet_bound2 : ∀ x y : t, order (meet x y) y.
  Parameter meet_greatest_lower_bound :
    ∀ x y z : t, order z x → order z y → order z (meet x y).

  Parameter bottom : t.
  Parameter bottom_is_bottom : ∀ x : t, order bottom x.

End Lattice.
```

FIG. 4.3 – Signature des treillis en Coq

```
module type Lattice =
 sig
  type t

  val eq_dec : t → t → sumbool

  val order_dec : t → t → sumbool

  val join : t → t → t

  val meet : t → t → t

  val bottom : t
 end
```

FIG. 4.4 – Signature (extraite) des treillis en Caml

Égalité standard/raisonnement modulo L'égalité utilisée sur le treillis n'est pas forcément l'égalité standard car celle-ci est souvent trop forte. Par exemple, lorsque nous voudrons définir le treillis des intervalles en Coq, nous prendrons un type de base de la forme

```
Inductive interval : Set :=
 | ITV : ∀ a b : Z', a <=' b → interval.
```

Un intervalle est donc de la forme `(ITV a b h)` avec `h` une preuve que l'entier `a` est inférieur (pour l'ordre étendu sur $\mathbb{Z} \cup \{-\infty, +\infty\}$) à l'entier `b`. Si nous définissons maintenant l'égalité des intervalles avec l'égalité standard de Coq, deux intervalles `(ITV a1 b1 h1)` et `(ITV a2 b2 h2)` seront égaux si et seulement si les bornes des deux intervalles sont égales, mais aussi si les preuves `h1` et `h2` sont identiques (à réduction près). Cette dernière condition est beaucoup trop forte : une propriété peut admettre plusieurs preuves, de la même manière qu'une spécification peut admettre plusieurs implémentations.

Une égalité ad hoc permet aussi de modéliser les ensembles quotients en Coq. $\mathbb{Z}/n\mathbb{Z}$ sera ainsi représenté par $\mathbb{Z}$, muni de la relation d'équivalence $x \equiv y \iff n|x-y$.

C'est pour cette raison que nous raisonnons modulo une relation d'équivalence `eq`. Dans le cas des intervalles, deux intervalles seront déclarés égaux si et seulement si leurs bornes sont deux à deux égales, mais pas forcément leurs preuves :

```
Inductive eq : interval → interval → Prop :=
  eq_def : ∀ a b h1 h2, eq (ITV a b h1) (ITV a b h2).
```

Les booléens richement typés de Coq Les signatures de `eq_dec` et `order_dec` nécessitent quelques explications. Le type Coq `{P}+{Q}` est un type inductif informatif[4] défini à l'aide de deux constructeurs `left : P → {P}+{Q}` et `right : Q → {P}+{Q}`, avec `P` et `Q` des propriétés logiques (donc de type `Prop`). Un terme de type `{P}+{Q}` est donc soit de la forme `(left h)` avec `h` une preuve de `P`, soit de la forme `(right h)` avec `h`

[4] de type `Set`.

une preuve de `Q`. Dans le cas particulier où `Q` est égal à `¬P`, le type `{P}+{¬P}` correspond ainsi à un type booléen enrichi avec une preuve de la propriété `P` ou de sa négation. Une fois extrait, le type `{P}+{Q}` devient non dépendant et complètement équivalent au type booléen classique. Il devient ainsi le type `sumbool` Caml suivant

```
type sumbool =
  | Left
  | Right
```

La fonction `eq_dec` est ainsi un test d'égalité et `order_dec` un test d'ordre.

Une structure de treillis enrichie Cette notion de treillis diffère un peu de la définition standard (définition 3.1.2) puisque nous demandons l'existence d'un plus petit élément. `bot` sera utilisé comme pré-point fixe de départ pour les calculs itératifs de point fixe (ou de post-point fixe).

`join` et `meet` compatibles avec `eq` L'utilisation d'une relation d'équivalence nous obligera fréquemment à prouver que les fonctions manipulant des éléments de type `t` sont compatibles avec l'équivalence `eq`. Dans le cas des deux fonctions `join` et `meet` déclarées ici, cette compatibilité peut être déduite des propriétés vérifiées par les deux opérateurs. Aucune propriété de compatibilité n'est donc requise dans la signature `Lattice`. Au lieu de cela, nous prouvons que pour tout treillis ces fonctions sont compatibles. Ces preuves sont réalisées dans un foncteur de module.

```
Module Lattice_prop (L:Lattice).

Import L.

Lemma join_eq1 :
  ∀ x y z, eq x y → eq (join x z) (join y z).
Proof. ... Qed.

Lemma join_eq2 :
  ∀ x y z : t, eq y z → eq (join x z) (join x y).
Proof. ... Qed.

...

End Lattice_prop.
```

Un foncteur de module est une fonction prenant un (ou plusieurs) module en argument et retournant un autre module. Le module `Lattice_prop` peut être trouvé dans le fichier `lattice_def.v`. Plusieurs résultats généraux y sont prouvés.

4.2.3.2 Calcul de plus petit point fixe par récursion bien fondée

Nous présentons maintenant la programmation du solveur de plus petit point fixe. Ce solveur est défini à l'intérieur d'un module `SolveLeastFixPoint` (voir fichier `solve_wf.v` dans l'annexe A) paramétré par un module `L` de type `Lattice`.

Dans la suite de cette section nous allons supposer que le treillis `L` vérifie la condition de chaîne ascendante. Nous ouvrons pour cela une section Coq et nous ajoutons l'hypothèse de chaîne ascendante dans le contexte courant. Lorsque nous fermerons la section, toutes les hypothèses ajoutées seront déchargées.

```
Section ascending_chain_condition.

Variable ascending_chain_condition :
  well_founded (fun x y ⇒ ¬ eq y x ∧ order y x).
```

La notion de relation bien fondée est définie dans la librairie standard de Coq. La relation `(fun x y ⇒ ¬ eq y x ∧ order y x)` représente ici l'ordre strict inverse $\sqsupset$ du treillis.

Pour programmer notre solveur nous ouvrons une nouvelle section et posons dans l'environnement une fonction `f` monotone.

```
Section Fix.

Variable f :  t →  t.
Variable f_monotone : monotone f.
```

La fonction récursive correspondant au calcul de plus petit point fixe est donnée par le pseudo-code suivant

```
iter_fix x = if (f x)=x then x else iter_fix (f x)
```

Pour un pré-point fixe `x`, `(iter_fix x)` calcule le plus petit point fixe de `f` plus grand que `x`. `iter_fix` n'est pas récursive structurelle, nous allons donc réaliser une récursion bien fondée. C'est un exercice un peu délicat en théorie des types. La récursion générale fait encore aujourd'hui l'objet de recherche [Bov02, Bal02]. Nous choisissons de présenter ce point difficile sous l'angle de la programmation par preuve.

La fonction `iter_fix` peut a priori être construite à partir d'une preuve de `t → t`. Il est cependant préférable d'enrichir le type de `iter_fix` afin de ne pas avoir à utiliser le corps de la fonction dans le reste des preuves[5]. Le premier raffinement que nous pouvons proposer sur le type de `iter_fix` permet de restreindre son domaine de définition aux pré-points fixes de `f`.

```
∀ x, order x (f x) → t
```

`iter_fix` prend ainsi deux arguments : un élément `x` de type `t` et une preuve que `x` est un pré-point fixe. Nous pouvons ensuite enrichir le codomaine de `f`

```
∀ x, order x (f x) → { a : t | eq a (f a) }
```

Le type Coq `{ a : t | P }` est habité par une paire d'éléments constituée d'un terme informatif `a` de type `t` et d'une preuve de la propriété `P` (qui peut dépendre de `a`). `{ a : t | P }` est un type informatif (de type **`Set`**). Son équivalent logique est le type `∃ a : t, P` qui est lui aussi habité par une paire (*terme*,*preuve*) mais qui n'est, en revanche, pas extrait, car non informatif. Nous exprimons ainsi que `iter_fix` calcule

[5] La thèse d'Antonia Balaa [Bal02] propose un exemple simple qui illustre la difficulté de manipulation d'une fonction définie par récursion bien fondée

bien un point fixe de `f`. Afin de pouvoir prouver que ce point fixe est le plus petit, plus grand que `x`, nous ajoutons enfin une dernière propriété à la spécification de `iter_fix`.

```
∀ x, order x (f x) → { a : t | eq a (f a) ∧ ∃ n : nat, a = iter f n x }
```

`(iter f n)` représente ici la `n`-ième itérée de `f`. La construction de `iter_fix` peut ainsi se faire en prouvant que pour tout pré-point fixe `x`, il existe un élément `a` du treillis point fixe de `f` qui s'exprime comme le résultat d'un n-ième itérée de `f` sur `x`.

```
Lemma iter_fix : ∀ x, order x (f x) →
  { a :  t | eq a (f a) ∧ (∃ n : nat, a = iter f n x) }.
Proof.
```

Nous allons réaliser une preuve par induction bien fondée. Ce principe est donné par la règle de déduction suivante[6]

$$\frac{\forall x \in \mathrm{Acc}_{\prec}(x), (\forall y, y \prec x \Rightarrow P(y)) \Rightarrow P(x)}{\forall a \in \mathrm{Acc}_{\prec}(a), P(a)}$$

Pour prouver un propriété $P(a)$ pour tous les éléments accessibles a d'une relation $\prec$, il suffit de prouver que cette propriété est vérifiée par les éléments accessibles x dont tous les prédécesseurs y vérifient eux-mêmes la propriété.

Par hypothèse, tous les éléments du treillis sont accessibles (pour la relation $\sqsupset$), nous pouvons donc prouver la propriété attendue par induction.

```
  intros x.
  induction (ascending_chain_condition x) as [x _ rec].
```

L'hypothèse d'induction nous autorise ainsi à faire un appel récursif (avec `rec`) sur tous les prédécesseurs `y` de `x` pour la relation $\sqsupset$:

```
x : t
rec : ∀ y : t,
      ¬ eq x y ∧ order x y →
      order y (f y) →
      {a : t | eq a (f a) ∧ (∃ n : nat, a = iter f n y)}
============================
 order x (f x) → {a : t | eq a (f a) ∧ (∃ n : nat, a = iter f n x)}
```

Nous testons ensuite la propriété `(eq (f x) x)` grâce à la fonction `eq_dec`. Nous prenons soin de placer le résultat de `(f x)` dans une variable intermédiaire pour ne pas faire de calculs inutiles par la suite (commande `set (fx := f x)`).

```
  intros H; set (fx := f x).
  destruct (eq_dec fx x).
```

Deux cas sont alors générés selon que le résultat de `(eq_dec (f x) x)` est `(left e)` ou `(right n)` avec `e` une preuve de `(eq (f x) x)` et `n` une preuve de `(¬ eq (f x) x)`. Dans le premier cas, `x` est un résultat valide : c'est un point fixe et il est égal à `(iter f 0 x)`.

```
  exists x; intuition.
  exists 0%nat; intuition.
```

[6] Pour simplifier, nous notons $\forall a \in \mathrm{Acc}_{\prec}(a), P(a)$ au lieu de $\forall a, \mathrm{Acc}_{\prec}(a) \Rightarrow P(a)$.

Dans le deuxième cas, nous faisons un appel récursif sur `(f x)`.

```
destruct (rec fx) as [a' Ha']; unfold fx in *; auto.
```

Le système génère à cette occasion deux obligations de preuve : la première demande de prouver que `(f x)` est un pré-point fixe, la deuxième que `(f x)` est un prédécesseur de `x` pour $\sqsupset$. Les deux obligations sont déchargées automatiquement avec **`auto`**. Le système présente ensuite le résultat `(a',Ha')` de l'appel réalisé sur `(f x)`.

```
...
a' : t
Ha' : eq a' (f a') ∧ (∃ n : nat, a' = iter f n (f x))
============================
 {a : t | eq a (f a) ∧ (∃ n0 : nat, a = iter f n0 x)}
```

`a'` est alors un parfait candidat pour construire le résultat final. C'est un point fixe de `f` et il peut être écrit sous la forme `iter f n (f x) = iter f (S n) x`.

```
 exists a'; intuition.
 destruct H1 as [n1].
 exists (S n1); simpl; auto.
 rewrite iter_assoc; auto.
Qed.
```

Ainsi se termine la construction de `iter_fix`. Il nous reste à prouver que `(iter_fix x)` calcule le plus petit des points fixes plus grands que `x`. Il suffit de montrer que `(iter_fix x)` est plus petit que tous les post-points fixes de `f` plus grands que `x`. Ce résultat découle des lemmes suivants.

```
Lemma postfix_implies_postfix_iter :
  ∀ x :  t, order (f x) x → ∀ n : nat, order (iter f n x) x.
```

Preuve : Par induction sur `n` et grâce à la monotonie de `f`. □

```
Lemma iter_lower_bound_pfp :
  ∀ a y, order a y → order (f y) y →
  ∀ n, order (iter f n a) y.
```

Preuve : `order a y` implique `order (iter f n a) (iter f n y)` car `(iter f n)` est monotone. `order (f y) y` implique `order (iter f n y) y` d'après le lemme précédent. On peut ensuite conclure par transitivité. □

Ce dernier résultat montre que pour tout élément a du treillis, toutes les itérées $f^n(a)$ sont des minorants de l'ensemble des post-points fixes de f plus grands que a. On en déduit que si une itérée $f^n(a)$ est un post-point fixe de f, c'est nécessairement le plus petit des post-points fixes de f plus grands que a.

Ces arguments permettent de facilement définir les solveurs suivants :

– calcul du plus petit des points fixes plus grands qu'un pré-point fixe donné

```
Lemma lfp_a0 : ∀ a0, order a0 (f a0) →
  { a :  t | eq a (f a) ∧
             ∀ y, order a0 y → eq y (f y) → order a y }.
```

– calcul du plus petit point fixe

```
Lemma lfp :
  { a :  t | eq a (f a) ∧
             ∀ y, eq y (f y) → order a y }.
```

– calcul du plus petit des post-points fixes plus grands qu'un pré-point fixe donné

```
Lemma lpfp_a0 : ∀ a0, order a0 (f a0) →
  { a :  t | order (f a) a ∧
             ∀ y, order a0 y → order (f y) y → order a y }.
```

– calcul du plus petit post-point fixe

```
Lemma lpfp :
  { a :  t | order (f a) a ∧
             ∀ y, order (f y) y → order a y }.
```

Au final `lpfp` et `pfp` sont implémentés avec la même partie calculatoire `iter_fix bottom`. Nous avons ainsi démontré que le plus petit point fixe d'une fonction monotone est aussi son plus petit post-point fixe, sans s'être placé dans un treillis complet, mais en utilisant la condition de chaîne ascendante.

4.2.3.3 Résolution d'un système d'inéquations

La technique de partitionnement présenté dans la section 3.2.3.3 utilise un domaine abstrait de la forme $A \to B^\sharp$. Lorsque A est fini, ce domaine est isomorphe à $B^{\sharp n}$. Les post-points fixes d'un opérateur abstrait $f^\sharp \in B^{\sharp n} \to B^{\sharp n}$ peuvent alors être caractérisés par un système d'inéquations en considérant les projections $f^\sharp_i$ de $f^\sharp$ sur chaque composante de $B^{\sharp n}$.

$$f^\sharp\left(b^\sharp_1,\ldots,b^\sharp_n\right) \sqsubseteq_{B^{\sharp n}} \left(b^\sharp_1,\ldots,b^\sharp_n\right) \iff \begin{cases} f^\sharp_1\left(b^\sharp_1,\ldots,b^\sharp_n\right) & \sqsubseteq_{B^\sharp} & b^\sharp_1 \\ & & \cdots \\ f^\sharp_n\left(b^\sharp_1,\ldots,b^\sharp_n\right) & \sqsubseteq_{B^\sharp} & b^\sharp_n \end{cases}$$

Les n-upplets solutions de ce système peuvent être aussi caractérisés comme les post-points fixes d'un ensemble de fonctions $F^\sharp_1,\ldots,F^\sharp_n \in B^{\sharp n} \to B^{\sharp n}$

$$\begin{cases} F^\sharp_1\left(b^\sharp_1,\ldots,b^\sharp_n\right) & \sqsubseteq_{B^{\sharp n}} & \left(b^\sharp_1,\ldots,b^\sharp_n\right) \\ & \cdots & \\ F^\sharp_n\left(b^\sharp_1,\ldots,b^\sharp_n\right) & \sqsubseteq_{B^{\sharp n}} & \left(b^\sharp_1,\ldots,b^\sharp_n\right) \end{cases}$$

avec, pour tout i dans $\{1,\ldots,n\}$

$$F^\sharp_i\left(b^\sharp_1,\ldots,b^\sharp_n\right)_j = \begin{cases} f^\sharp_i\left(b^\sharp_1,\ldots,b^\sharp_n\right) & \text{si } i = j \\ \bot_{B^\sharp} & \text{sinon} \end{cases}$$

Ceci nous ramène ainsi au problème de déterminer le plus petit post-point fixe d'un ensemble d'opérateurs monotones $F^\sharp_1,\ldots,F^\sharp_n$. Nous pouvons certes résoudre ce problème

en itérant la fonction $\lambda x.\bigsqcup_i^\sharp F_i^\sharp(x)$ mais la convergence peut être accélérée en « composant » les fonctions[7].

Théorème 4.2.5. *Soit* $(A, \sqsubseteq, \sqcup, \sqcap)$ *un treillis et* $F_1, \ldots, F_n$ *un ensemble fini de fonctions monotones de* $A \to A$. *Un élément* $a \in A$ *est un post-point fixe des* $F_1, \ldots, F_n$ *si et seulement si c'est un post-point fixe de l'opérateur monotone* $F = \overline{F}_1 \circ \cdots \overline{F}_n$ *avec* $\overline{F}_i(x) = x \sqcup F_i(x)$

$$\begin{cases} F_1(a) & \sqsubseteq & a \\ & \cdots & \\ F_n(a) & \sqsubseteq & a \end{cases} \iff \overline{F}_n \circ \cdots \overline{F}_1(a) \sqsubseteq a$$

Preuve : Nous prenons une définition de F récursive terminale :

$$F_{(F_n,\ldots,F_1)}(x) = \begin{cases} x & \text{si } n = 0 \\ F_{(F_{n-1},\ldots,F_1)}(x \sqcup F_n(x)) & \text{sinon} \end{cases}$$

Nous prouvons ensuite une série de résultats intermédiaires par récurrence sur n.

1. pour tout n, $F_{(F_n,\ldots,F_1)}$ est monotone :
   - $n = 0$, l'identité id est bien monotone
   - pour $n > 0$, $\lambda x.F_{(F_{n-1},\ldots,F_1)}(x \sqcup F_n(x))$ est monotone par composition de $F_{(F_{n-1},\ldots,F_1)}$ (monotone par hypothèse de récurrence) et de $\lambda x.x \sqcup F_n(x)$ ($\sqcup$ et F_n sont monotones).
2. pour tout n, $F_{(F_n,\ldots,F_1)}$ est extensif :
   - $n = 0$, l'identité id est bien extensive
   - pour $n > 0$ et $x \in A$, par hypothèse de récurrence on a

     $$x \sqcup F_n(x) \sqsubseteq F_{(F_{n-1},\ldots,F_1)}(x \sqcup F_n(x))$$

     Nous concluons alors par transitivité et $x \sqsubseteq x \sqcup F_n(x)$
3. pour tout n, pour tout $i \in \{1, \ldots, n\}$, pour tout $x \in A$, $F_i(x) \sqsubseteq F_{(F_n,\ldots,F_1)}(x)$:
   - pour $n = 0$, aucun i n'appartient à $\{1, \ldots, n\}$
   - pour $n > 0$, deux cas se présentent $i = n$ et $i < n$
     - pour $i = n$, on obtient par extensivité

       $$F_n(x) \sqsubseteq x \sqcup F_n(x) \sqsubseteq F_{(F_{n-1},\ldots,F_1)}(x \sqcup F_n(x))$$

     - pour $i < n$, par hypothèse de récurrence

       $$F_i(x \sqcup F_n(x)) \sqsubseteq F_{(F_{n-1},\ldots,F_1)}(x \sqcup F_n(x))$$

       or $F_i(x) \sqsubseteq F_i(x \sqcup F_n(x))$, par monotonie de F_i, donc nous pouvons conclure
4. pour tout n, si $x \in A$ est un post-point fixe des $F_1, \ldots, F_n$, c'est un post-point fixe de $F_{(F_n,\ldots,F_1)}$:
   - pour $n = 0$ tout $x \in A$ est un post-point fixe de id

[7] L'itération obtenue peut être vue comme un cas particulier d'itération chaotique [Cou78].

– si $n > 0$, x est un post-point fixe de F_n donc $x = x \sqcup F_n(x)$, nous pouvons alors en déduire

$$\begin{aligned} F_{(F_n,\ldots,F_1)}(x) &= F_{(F_{n-1},\ldots,F_1)}(x \sqcup F_n(x)) \\ &= F_{(F_{n-1},\ldots,F_1)}(x) \\ &\sqsubseteq x \end{aligned}$$

en utilisant l'hypothèse de récurrence ($F_{(F_{n-1},\ldots,F_1)}(x) \sqsubseteq x$).

Le dernier point démontre le premier sens $\Rightarrow$ du théorème. Pour l'autre sens, si a un est un post-point fixe de $F_{(F_n,\ldots,F_1)}$, nous pouvons utiliser le résultat du point 3 :

$$F_i(a) \sqsubseteq F_{(F_n,\ldots,F_1)} \sqsubseteq a$$

a est ainsi un post-point fixe de tous les F_i. □

En appliquant le solveur de plus petit post-point fixe sur $F_{(F_n,\ldots,F_1)}$ nous construisons ainsi un solveur de système d'inéquations.

```
Lemma lpfp_list : ∀ (l:list (t → t)), (∀ f, In f l → monotone f) →
  { a : t | (∀ f, In f l → order (f a) a) ∧
            (∀ y , (∀ f, In f l → order (f y) y) → order a y) }.
```

4.2.3.4 Extraction

Pour conclure cette partie, nous pouvons extraire le module `SolveLeastFixPoint` en Caml. Le résultat de l'extraction est présenté dans la figure 4.5. Le code produit est bien celui attendu.

4.3 Approximation par élargissement/rétrécissement

Plutôt que de calculer le plus petit point fixe (ou le plus petit post-point fixe) d'une fonction monotone, nous pouvons nous contenter d'une approximation. D'après le résultat du théorème 3.3.1 présenté dans notre cadre, tout post-point fixe est une approximation correcte. Obtenir le plus petit post-point fixe n'ajoute rien à la correction d'une analyse statique, seulement à sa précision.

La décision de ne pas forcément calculer un plus petit (post-) point fixe est généralement prise dans les cas suivants :

– Le treillis ne vérifie pas la condition de chaîne ascendante, l'itération

$$\bot, f(\bot), \ldots, f^n(\bot), \ldots$$

peut ne jamais terminer.

– La condition de chaîne ascendante est vérifiée mais la chaîne d'itération est trop longue pour permettre un calcul efficace.

– Enfin, certaines abstractions ne vérifient pas « l'hypothèse raisonnable » décrite dans le chapitre 3. Le treillis sous-jacent n'est alors pas complet et la limite des itérations croissantes n'appartient pas au domaine d'abstraction.

```
module SolveLeastFixPoint =
 functor (L:Lattice) →
 struct
  module PropL = Lattice_prop(L)

  let rec iter_fix f x =
    let fx = f x in
    (match L.eq_dec fx x with
       | Left → x
       | Right → iter_fix f fx)

  let lfp_a0 f a0 = iter_fix f a0

  let lfp f = lfp_a0 f L.bottom

  let lpfp_a0 f a0 = iter_fix f a0

  let lpfp f = lpfp_a0 f L.bottom

  let lpfp_list l = lpfp (fun x → PropL.iter_listf l x)
 end
```

FIG. 4.5 – Extraction du module `SolveLeastFixPoint`

4.3.1 Accélération de convergence par élargissement

La solution proposée par Patrick et Radia Cousot consiste à accélérer l'itération croissante, quitte à atteindre un point plus haut que $\mathrm{lfp}(f)$, mais dans la zone des post-points fixes. L'itération utilisée est alors de la forme $x_0 = \bot, x_{n+1} = x_n \nabla f(x_n)$ avec ∇ un opérateur binaire qui extrapole ses deux arguments. Intuitivement, à la n-ième itération, plutôt que de simplement itérer f (en calculant $f(x_n)$), $f(x_n)$ est comparé avec la valeur précédente x_n pour éventuellement détecter le début d'une chaîne infinie (ou très longue). Si c'est le cas, on extrapole directement vers un point plus avancé dans la chaîne d'itération.

Définition 4.3.1. Opérateur d'élargissement.
Étant donné un ensemble partiellement ordonné $(A, \sqsubseteq)$*, un opérateur binaire* $\nabla \in A \times A \to A$ *est appelé* opérateur d'élargissement *s'il vérifie les conditions suivantes :*

$$\forall x, y \in A,\ x \sqsubseteq x \nabla y \tag{4.1}$$

$$\forall x, y \in A,\ y \sqsubseteq x \nabla y \tag{4.2}$$

pour toute suite croissante $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ (4.3)
la chaîne $y_0 = x_0, y_{n+1} = y_n \nabla x_{n+1}$ *atteint nécessairement un rang* k *pour lequel* $y_k = y_{k+1}$

Théorème 4.3.1. *Si* ∇ *est un opérateur d'élargissement sur un ensemble partiellement ordonné* $(A, \sqsubseteq)$*, si* f *est un opérateur monotone sur* A *et* a *un pré-point fixe de* f *alors*

la chaîne $(x_n)_n$ *définie par*

$$\begin{cases} x_0 & = & a \\ x_{k+1} & = & x_k \nabla f(x_k) \end{cases}$$

atteint en un nombre fini de pas un post-point fixe de f *plus grand que* a.
Preuve : *cf.* [CC92a] □

4.3.2 Amélioration d'une approximation par rétrécissement

Lorsqu'un post-point fixe est atteint, une itération démarrant de ce point permet d'améliorer l'approximation de lfp(f). Le théorème suivant, version duale du théorème 4.1.2, mais plus détaillée, explique la situation.

Théorème 4.3.2. *Soit* $(A, \sqsubseteq, \bigsqcup)$ *un treillis complet,* f *un opérateur monotone sur* A *et* a *un post-point fixe de* f. *La chaîne* $(x_n)_n$ *définie par*

$$\begin{cases} x_0 & = & a \\ x_{k+1} & = & f(x_k) \end{cases}$$

admet pour limite $(\bigsqcap\{x_n\})$, *le plus grand point fixe de* f *plus petit que* a *(noté* $\mathrm{gfp}_a(f)$*). En particulier,* $\mathrm{lfp}(f) \sqsubseteq \bigsqcap\{x_n\}$. *Chaque étape intermédiaire de calcul est une approximation correcte :*

$$\forall k,\ \mathrm{lfp}(f) \sqsubseteq \mathrm{gfp}_a(f) \sqsubseteq x_k \sqsubseteq a$$

Si le treillis ne vérifie pas la condition de chaîne descendante (duale de la notion de chaîne ascendante), l'itération peut ne jamais stabiliser après un nombre fini d'étapes. Même en cas de stabilisation, le nombre d'itérations nécessaires peut être très important. Une technique d'accélération peut une nouvelle fois être utilisée. L'itération utilisée est cette fois de la forme $x_0 = a, x_{n+1} = x_n \Delta f(x_n)$ avec Δ un opérateur binaire. L'itération réalisée reste dans la zone des post-points fixes et stabilise en un temps fini. Son but est de rattraper une partie des approximations effectuées avec l'opérateur d'élargissement. Les deux opérateurs d'élargissement et de rétrécissement sont ainsi étroitement liés.

Définition 4.3.2. Opérateur de rétrécissement.
Étant donné un ensemble partiellement ordonné $(A, \sqsubseteq)$, *un opérateur binaire* $\Delta \in A \times A \to A$ *est appelé* opérateur de rétrécissement *s'il vérifie les conditions suivantes :*

$$\forall x, y \in A,\ y \sqsubseteq x \ \Rightarrow\ y \sqsubseteq x \Delta y \sqsubseteq x \tag{4.4}$$

pour toute suite décroissante $x_0 \sqsupseteq x_1 \sqsupseteq \cdots \sqsupseteq x_n \sqsupseteq \cdots$ (4.5)
la chaîne $y_0 = x_0, y_{n+1} = y_n \Delta x_{n+1}$ *atteint nécessairement un rang* k *pour lequel* $y_k = y_{k+1}$

Théorème 4.3.3. *Si* Δ *est un opérateur de rétrécissement sur un ensemble partiellement ordonné* $(A, \sqsubseteq)$, *si* f *est un opérateur monotone sur* A *et* a *un post-point fixe de* f *alors la chaîne* $(x_n)_n$ *définie par*

$$\begin{cases} x_0 & = & a \\ x_{k+1} & = & x_k \Delta f(x_k) \end{cases}$$

atteint en un nombre fini de pas un post-point fixe de f *plus petit que* a.

Preuve : *cf.* [CC92a] □

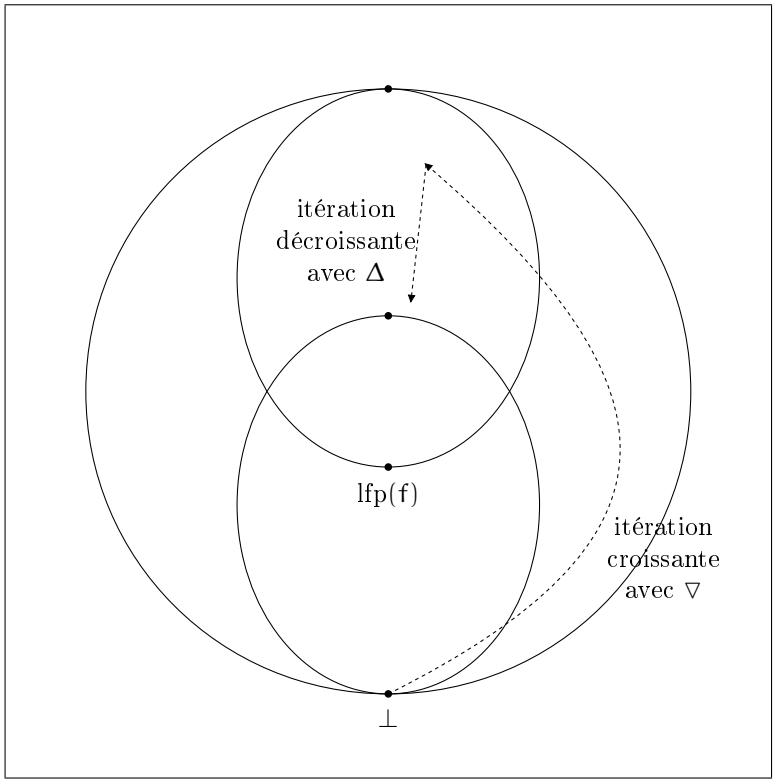


FIG. 4.6 – Itération croissante avec élargissement suivie d'une itération décroissante avec rétrécissement

L'enchaînement des deux itérations est schématisé dans la figure 4.6.

4.3.3 Élargissement/rétrécissement en logique constructive

Les critères de terminaison 4.3 et 4.5 des opérateurs d'élargissement et de rétrécissement ne sont pas adaptés aux preuves constructives car le rang de stabilisation n'est pas toujours calculable. Nous proposons maintenant de reformuler ces critères à l'aide de la notion d'accessibilité. Pour exprimer ces critères en terme d'accessibilité, il nous faut trouver les relations dont nous voulons proscrire certaines chaînes décroissantes infinies.

Définition 4.3.3. Opérateur d'élargissement (version constructive).
Étant donné un ensemble partiellement ordonné $(\mathsf{A}, \sqsubseteq)$*, un opérateur binaire* $\nabla \in \mathsf{A} \times$

$A \to A$ *est appelé* opérateur d'élargissement *s'il vérifie les conditions suivantes :*

$$\forall x, y \in A,\ x \sqsubseteq x \triangledown y \tag{4.6}$$

$$\forall x, y \in A,\ y \sqsubseteq x \triangledown y \tag{4.7}$$

$$\begin{array}{l} \textit{la relation } \prec_\triangledown \subseteq (A \times A) \times (A \times A) \textit{ définie par} \\ \forall x_1, x_2, y_1, y_2 \in A, \\ \quad (x_1, y_1) \prec_\triangledown (x_2, y_2) \iff x_2 \sqsubseteq x_1 \wedge y_1 = y_2 \triangledown x_1 \wedge y_1 \neq y_2 \\ \textit{vérifie } \forall x \in A,\ (x, x) \textit{ est accessible pour } \prec_\triangledown \end{array} \tag{4.8}$$

Théorème 4.3.4. *Les définitions 4.3.1 et 4.3.3 sont équivalentes.*

Preuve : Il suffit de constater l'équivalence

$$\left(\begin{array}{l} \text{il existe une chaîne } x_0 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots \\ \text{telle que la chaîne } y_0 = x_0, y_{n+1} = y_n \triangledown x_n \\ \text{vérifie pour tout } k,\ y_k \neq y_{k+1} \end{array}\right) \iff \left(\begin{array}{l} \text{il existe une suite } ((x_k, y_k))_{k \in \mathbb{N}} \\ \text{vérifiant } x_0 = y_0 \\ \text{et } \forall k,\ (x_{k+1}, y_{k+1}) \prec_\triangledown (x_k, y_k) \end{array}\right)$$

□

Définition 4.3.4. Opérateur de rétrécissement (version constructive). *Étant donné un ensemble partiellement ordonné* $(A, \sqsubseteq)$*, un opérateur binaire* $\Delta \in A \times A \to A$ *est appelé* opérateur de rétrécissement *s'il vérifie les conditions suivantes :*

$$\forall x, y \in A,\ y \sqsubseteq x \ \Rightarrow\ y \sqsubseteq x \Delta y \sqsubseteq x \tag{4.9}$$

$$\begin{array}{l} \textit{la relation } \prec_\Delta \subseteq (A \times A) \times (A \times A) \textit{ définie par} \\ \forall x_1, x_2, y_1, y_2 \in A, \\ \quad (x_1, y_1) \prec_\Delta (x_2, y_2) \iff x_1 \sqsubseteq x_2 \wedge y_1 = y_2 \Delta x_1 \wedge y_1 \neq y_2 \\ \textit{vérifie } \forall x \in A,\ (x, x) \textit{ est accessible pour } \prec_\Delta \end{array} \tag{4.10}$$

Théorème 4.3.5. *Les définitions 4.3.2 et 4.3.4 sont équivalentes.*

Preuve : Preuve similaire à la preuve du théorème 4.3.4. □

4.3.4 Calcul de post-point fixe par élargissement/rétrécissement en Coq

Nous allons maintenant utiliser les définitions constructives précédentes pour programmer un solveur de post-point fixe en Coq.

4.3.4.1 Le type Coq des opérateurs d'élargissement/rétrécissement

Les définitions précédentes donnent lieu à la définition du type Coq des opérateurs d'élargissement et de rétrécissement à l'aide d'enregistrements. Ces définitions sont énoncées dans le foncteur `Lattice_prop` du fichier `lattice_def.v`.

```
Definition widen_cl_rel (widen:t→t→t) : (t*t) → (t*t) → Prop :=
  fun x y ⇒ order (fst y) (fst x) ∧
            eq (snd x) (widen (snd y) (fst x)) ∧
            ¬ eq (snd y) (snd x).

Record widening_classic_operator : Set := WidOp_cl {
  widen_cl : t → t → t;
  widen_cl_bound1 : ∀ x y : t, order x (widen_cl x y);
  widen_cl_bound2 : ∀ x y : t, order y (widen_cl x y);
  widen_cl_eq1 : ∀ x y z: t, eq x y → eq (widen_cl x z) (widen_cl y z);
  widen_cl_eq2 : ∀ x y z: t, eq x y → eq (widen_cl z x) (widen_cl z y);
  widen_cl_bottom1 : ∀ x : t, eq x (widen_cl x bottom);
  widen_cl_bottom2 : ∀ x : t, eq x (widen_cl bottom x);
  widen_cl_acc_property : ∀ x:t, Acc (widen_cl_rel widen_cl) (x,x)
}.
```

Remarquons que cette fois les enregistrements ne contiennent pas de déclaration de type et sont donc facilement extraits vers Caml. Le mécanisme d'extraction ne conserve en fait que l'opérateur d'élargissement `widen_cl`, les autres champs étant purement logiques. Le type `widening_classic_operator` est alors directement identifié à `t → t → t`.

```
type widening_classic_operator =  t → t → t
```

Les champs `widen_cl_eq1` et `widen_cl_eq2` assurent que `widen_cl` est compatible avec l'équivalence `eq` du treillis. Les champs `widen_cl_bottom1` et `widen_cl_bottom2` n'appartiennent pas à la définition standard d'opérateur d'élargissement. Ils nous seront utiles par la suite et sont vérifiés en pratique par les opérateurs d'élargissement usuels : aucune accélération n'est généralement effectuée au niveau de $\bot$.

La définition du type des opérateurs de rétrécissement est similaire.

```
Definition narrow_cl_rel (narrow:t→t→t) : (t*t) → (t*t) → Prop :=
  fun x y ⇒ order (fst x) (fst y) ∧
            eq (snd x) (narrow (snd y) (fst x)) ∧
            ¬ eq (snd y) (snd x).

Record narrowing_classic_operator : Set := NarOp_cl {
  narrow_cl : t → t → t;
  narrow_cl_bound1 : ∀ x y : t, order y x →  order (narrow_cl x y) x;
  narrow_cl_bound2 : ∀ x y : t, order y x →  order y (narrow_cl x y);
  narrow_cl_eq1 : ∀ x y z: t, eq x y → eq (narrow_cl x z) (narrow_cl y z);
  narrow_cl_eq2 : ∀ x y z: t, eq x y → eq (narrow_cl z x) (narrow_cl z y);
  narrow_cl_acc_property : ∀ x:t, Acc (narrow_cl_rel narrow_cl) (x,x)
}.
```

4.3.4.2 Calcul de post-point fixe par élargissement/rétrécissement et récursion bien fondée

Nous présentons maintenant la programmation du calcul itératif de post-point fixe d'un opérateur monotone à l'aide d'opérateurs d'élargissement/rétrécissement. Les programmes Coq correspondants sont placés dans un foncteur `SolvePostFixPoint` paramétré par un treillis (voir fichier `solve_widen_classic.v`), à l'intérieur d'une section où

nous supposons l'existence d'opérateur d'élargissement `w` et de rétrécissement `n` sur le treillis, ainsi qu'une fonction monotone `f`.

Nous détaillons maintenant le cas du calcul ascendant utilisant l'opérateur d'élargissement. La programmation se fait une nouvelle fois par preuve, en utilisant une induction bien fondée. Le pseudo-code du programme souhaité est

```
pfp_widen_a0 y = if (f y) ⊑ y then y else pfp_widen_a0 (widen y (f y))
```

Cependant, notre critère d'accessibilité porte sur un couple de `t*t`. Nous allons donc programmer une version légèrement différente du calcul précédent.

```
pfp_widen_rec (x,y) = if (f y) ⊑ y then y else pfp_widen (f y,widen y (f y))
pfp_widen_a0 y = pfp_widen_rec (y,y)
pfp_widen = pfp_widen_a0 ⊥
```

Le premier élément du couple est utilisé ici pour justifier l'itération selon une chaîne croissante $x_0 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$.

Le coeur du calcul itératif est réalisé par la fonction `pfp_widen_rec`.

```
Lemma pfp_widen_rec :
  ∀ p:Pair_widen, Acc rel_pair p → {a : t | order (f a) a}.
```

Le type `Pair_widen` contient des triplets de la forme `(Pairc_widen x y h)` avec `x` et `y` des éléments du treillis et `h` une preuve de `(order x (f y))`. Il s'agit donc d'une restriction du type `t*t`. La relation `rel_pair` désigne la projection de $\prec_\nabla$ sur cette restriction. L'hypothèse `(Acc rel_pair p)` assure que l'itération s'effectue le long d'une chaîne finie. L'invariant `(order x (f y))` permet de démontrer que l'appel récursif `(f y,widen y (f y))` est bien un prédécesseur de `(x,y)` pour $\prec_\nabla$.

La fonction `pfp_widen_a0`, qui calcule un post-point fixe en partant d'un pré-point fixe, est ensuite définie.

```
Lemma pfp_widen_a0 : ∀ a0, order a0 (f a0) → {a : t | order (f a) a}.
```

Il suffit pour cela d'appeler `pfp_widen_rec` sur le couple `(a0,a0)` qui vérifie bien la contrainte imposée par le type `Pair_widen`, puisque `a0` est un pré-point fixe. Il est de plus accessible, puisque de la forme `(x,x)`. Le solveur de post-point fixe est finalement construit en partant de `bottom`.

```
Lemma pfp_widen : {a : t | order (f a) a}.
```

La construction de l'itération descendante est similaire. Le type du programme `pfp_narrow` est cependant plus riche :

```
Lemma pfp_narrow : ∀ x, order (f x) x →
 {a : t | order (f a) a ∧ (∃ n, order (iter f n x) a) ∧ order a x}.
```

En partant d'un post-point fixe `x` nous construisons un autre post-point fixe `a`, plus petit que `x` mais néanmoins plus grand que le résultat que nous aurions pu atteindre en itérant suffisamment `f`. Un tel type nous impose plusieurs commentaires.

- Un tel théorème est trivial à prouver car `x` est un parfait candidat pour construire le résultat `a`. Le type de `pfp_narrow` n'assure une nouvelle fois que la correction du calcul et non sa précision. C'est donc la manière dont la preuve sera construite qui déterminera la précision de l'opérateur.

– La preuve de ce résultat étant constructive, nous pourrions tout à fait calculer l'entier `n` apparaissant dans l'énoncé et ainsi proposer une meilleure approximation. L'opérateur de rétrécissement ne serait alors plus utilisé que pour calculer une borne sur le nombre d'itérations à effectuer pour améliorer un post-point fixe.

Après la clôture des sections ouvertes, nous pouvons programmer notre solveur de post-point fixe final, en enchaînant les calculs de `pfp_widen` et `pfp_narrow`[8]

```
Lemma pfp :
  widening_classic_operator → narrowing_classic_operator →
  ∀ f, monotone f → { a : t | order (f a) a }.
```

Enfin, comme dans la section précédente, nous pouvons réutiliser le résultat du théorème 4.2.5 pour programmer un solveur de système d'inéquations en combinant `pfp` et $\mathsf{F}_{(\mathsf{F}_n,\ldots,\mathsf{F}_1)}$

```
Lemma pfp_list:
   widening_classic_operator → narrowing_classic_operator →
   ∀ l, (∀ f, In f l → monotone f) →
     {a : t | ∀ f : t → t, In f l → order (f a) a}.
```

L'extraction du module `SolvePostFixPoint` est présenté dans la figure 4.7.

4.4 Une définition des opérateurs d'élargissement/rétrécissement plus souple

Dans la section précédente, nous avons scrupuleusement suivi les définitions d'élargissement et de rétrécissement telles qu'elles sont généralement proposées dans la littérature [CC77, CC92a, CC92c].

Ces définitions standards possèdent cependant des limites :

– Elles sont dédiées au calcul de post-points fixes de fonctions monotones (ou bien extensives) ; or certaines fonctions abstraites ne sont pas monotones. C'est par exemple le cas de certaines fonctions abstraites utilisées dans l'analyseur Astrée [CCF+05]. Dans notre contexte, même si un opérateur abstrait est monotone, il peut être intéressant de ne pas avoir à le prouver. Il faut alors disposer d'un solveur de post-points fixes pour fonctions quelconques.

– Une utilisation trop brutale de l'opérateur d'élargissement peut mener à une surapproximation trop grossière. Pour la résolution d'un système d'inéquations, il est conseillé [Cou78] de limiter le nombre d'inéquations sur lesquelles on utilise un élargissement : en fonction des dépendances entre les inéquations du système, certaines équations peuvent être itérées avec un opérateur $\sqcup$ au lieu de $\triangledown$

Nous proposons maintenant une nouvelle définition pour les opérateurs d'élargissement et de rétrécissement, qui permettra de réduire les limites précédentes. Nous étudierons ensuite un algorithme permettant de réduire les points d'élargissement et de rétrécissement dans un système d'inéquations.

[8] Pour améliorer la précision, nous entrecoupons ces deux calculs avec une itération de `f`.

```
module SolvePostFixPoint =
 functor (L:Lattice) →
 struct
  module PropL = Lattice_prop(L)

  type coq_Pair_widen =
    | Pairc_widen of L.t * L.t

  let rec pfp_widen_rec widop f = function
    | Pairc_widen (x, y) →
        let fy = f y in
        (match L.order_dec fy y with
           | Left → y
           | Right →
               pfp_widen_rec widop f (Pairc_widen (fy,
                 (PropL.widen_cl widop y fy))))

  let pfp_widen_a0 widop f a0 = pfp_widen_rec widop f (Pairc_widen (a0, a0))

  let pfp_widen widop f = pfp_widen_a0 widop f L.bottom

  type coq_Pair_narrow =
    | Pairc_narrow of L.t * L.t

  let rec pfp_narrow_rec narop f = function
    | Pairc_narrow (x, y) →
        let fy = f y in
        let y' = PropL.narrow_cl narop y fy in
        (match L.eq_dec y' y with
           | Left → y
           | Right → pfp_narrow_rec narop f (Pairc_narrow (fy, y')))

  let pfp_narrow narop f x =
    pfp_narrow_rec narop f (Pairc_narrow (x, x))

  let pfp w n f =
    pfp_narrow n f (f (pfp_widen w f))

  let pfp_list w n l =
    let f = fun x → PropL.iter_listf l x in
    pfp_narrow n f (f (pfp_widen w f))
 end
```

FIG. 4.7 – Extraction du module `SolvePostFixPoint`

4.4.1 Modification de la définition standard d'élargissement/rétrécissement

Suppression du critère de chaîne croissante/décroissante Afin de pouvoir utiliser les opérateurs d'élargissement/rétrécissement sur des opérateurs non-monotones, il suffit de supprimer l'hypothèse

$$\text{pour toute suite croissante } x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$$

et de la remplacer par

$$\text{pour toute suite } x_0, x_1, \ldots, x_n, \ldots$$

dans la définition 4.3.1 d'un opérateur d'élargissement. La même opération peut être effectuée pour l'opérateur de rétrécissement. On obtient alors un critère plus restrictif puisqu'un plus grand nombre de chaînes infinies est interdit. En pratique, les opérateurs usuels vérifient encore cette définition. Ce critère apparaît dans certains travaux récents utilisant l'interprétation abstraite [Fer05, Min04].

Nouvelles définitions retenues La nouvelle définition d'opérateur d'élargissement devient ainsi

$$\forall x, y \in A,\ x \sqsubseteq x \nabla y$$
$$\forall x, y \in A,\ y \sqsubseteq x \nabla y$$
$$\text{pour toute suite } x_0, x_1, \ldots, x_n, \cdots$$
$$\text{la chaîne } y_0 = x_0, y_{n+1} = y_n \nabla x_{n+1} \text{ atteint nécessairement}$$
$$\text{un rang } k \text{ pour lequel } y_k = y_{k+1}$$

La nouvelle définition d'opérateur de rétrécissement devient quant à elle

$$\forall x, y \in A,\ x \sqcap y \sqsubseteq x \Delta y \sqsubseteq x$$
$$\text{pour toute suite } x_0, x_1, \ldots, x_n, \ldots$$
$$\text{la chaîne } y_0 = x_0, y_{n+1} = y_n \Delta x_{n+1} \text{ atteint nécessairement}$$
$$\text{un rang } k \text{ pour lequel } y_k = y_{k+1}$$

Par rapport à la définition initiale 4.3.2, l'hypothèse $y \sqsubseteq x$ n'est plus utilisable car l'invariant $f(y_n) \sqsubseteq y_n$ ne sera plus assuré lors de l'itération $y_{n+1} = y_n \Delta f(y_n)$.

Ces nouvelles définitions ne représentent que deux possibilités parmi de nombreuses variantes possibles. Nous avons cependant pris soin de préserver une propriété importante que possédaient les définitions initiales : la possibilité de combiner les opérateurs. Nous verrons, en effet dans le chapitre 6 que les opérateurs standard possèdent de bonnes propriétés vis-à-vis des combinaisons de treillis (produit, somme, fonctions,...). Ces propriétés sont encore satisfaites avec nos nouvelles définitions.

Calcul de post-point fixe Les algorithmes itératifs utilisés avec ces nouvelles définitions sont les mêmes que précédemment,seules les propriétés (et par conséquent les types Coq) des solveurs subissent quelques changements[9]. L'itération ascendante `pfp_widen` accepte désormais n'importe quel opérateur `f` en argument.

```
Lemma pfp_widen : widening_operator → ∀ f : t → t, {a : t | order (f a) a}.
```

La situation se complique pour l'itération descendante `pfp_narrow`.

```
Lemma pfp_narrow : narrowing_operator →
  ∀ (f : t → t) (P : t → Prop),
    (∀ x y : t, P x → P y → P (meet x y)) →
    (∀ x y : t, order x y → P x → P y) →
    (∀ x : t, P x → P (f x)) →
      ∀ x : t, P x → {a : t | P a ∧ order a x}.
```

Sans hypothèse de monotonie sur `f`, l'itération descendante n'est plus assurée d'atteindre un post-point fixe. Ce type assure néanmoins que, si elle démarre sur un élément `x` vérifiant une propriété `P`, elle atteindra un élément `a` plus petit que `x` et vérifiant encore la propriété `P`. Cette dernière doit cependant vérifier certaines conditions :

- être stable par $\sqcap$ (`∀ x y : t, P x → P y → P (meet x y)`),
- être monotone (`∀ x y : t, order x y → P x → P y`),
- être stable par `f` (`∀ x : t, P x → P (f x)`).

Dans notre contexte d'utilisation, ces hypothèses seront ultérieurement vérifiées car `P` sera définie par

$$\mathtt{P}(x) \iff \mathrm{lfp}\, f^\flat \subseteq \gamma(x)$$

avec γ un morphisme d'intersection et $f^\flat$ une fonction concrète dont `f` sera une approximation correcte ($f^\flat \circ \gamma \dot{\subseteq} \gamma \circ \mathtt{f}$).

Le solveur final est obtenu en combinant `pfp_widen` et `pfp_narrow` et en ajoutant l'hypothèse que les post-points fixes de `f` vérifient tous la propriété `P`.

```
Lemma pfp : widening_operator → narrowing_operator →
  ∀ (f : t → t) (P : t → Prop),
    (∀ x y, P x → P y → P (meet x y)) →
    (∀ x y, order x y → P x → P y) →
    (∀ x, P x → P (f x)) →
    (∀ x, order (f x) x → P x) →
      { a : t | P a }.
```

4.4.2 Résolution d'un système d'inéquations

Dans cette section, nous avons entrepris de nous passer de l'hypothèse de monotonie sur les opérateurs. Dans ce contexte, le résultat du théorème 4.2.5 n'est plus utilisable.

Nous allons maintenant présenter la technique dite *d'itération chaotique avec élargissement/rétrécissement* proposée par Patrick Cousot [Cou78] pour résoudre les systèmes d'inéquations. Comme lors de la section 4.2.3.3, le but est de trouver une solution

[9] Tous ces programmes sont réunis dans le fichier `solve_widen.v`.

$(X_1, \ldots, X_n)$ vérifiant le système

$$\begin{cases} F_1(X_1, \ldots, X_n) & \sqsubseteq & X_1 \\ & \cdots & \\ F_n(X_1, \ldots, X_n) & \sqsubseteq & X_n \end{cases} \tag{4.11}$$

en se plaçant sur un treillis $(A, \sqsubseteq, \sqcup, \sqcap)$ avec un plus petit élément $\bot$. Nous supposons que ce treillis possède un opérateur d'élargissement $\triangledown$ et un opérateur de rétrécissement Δ. Nous cherchons une nouvelle fois une solution la plus petite possible afin d'approcher au plus près le plus petit point fixe du système.

Le calcul itératif standard avec élargissement consiste à construire la suite

$$\left(X_k^i\right)_{\substack{k \in \{1, \ldots, n\} \\ i \in \mathbb{N}}}$$

définie par

$$\begin{cases} X_k^0 & = & \bot & \forall k \in \{1, \ldots, n\} \\ X_k^i & = & X_k^{i-1} \triangledown F_k(X_1^{i-1}, \ldots, X_n^{i-1}) & \forall k \in \{1, \ldots, n\},\ i > 0 \end{cases}$$

en s'arrêtant dès qu'une solution du système (4.11) est atteinte. Nous pouvons néanmoins accélérer le calcul en injectant directement les valeurs $X_1^i, \ldots, X_{k-1}^i$ pour le calcul de X_k^i. Cette accélération se généralise en mettant à jour une seule variable à chaque étape et en n'oubliant jamais indéfiniment une variable. Nous considérons pour cela une suite $(k_i) \in \mathbb{N}^{\mathbb{N}}$ vérifiant $\forall k \in \mathbb{N}, \forall N \in \mathbb{N}, \exists i > N, k_i = k$. L'algorithme *d'itération chaotique* associé à (k_i) est alors

$(X_1, \ldots, X_n) \leftarrow (\bot, \ldots, \bot)$;
$i \leftarrow 0$;
repeat
  if $(F_{k_i}(X_1, \ldots, X_n) \not\sqsubseteq X_{k_i})$ **then** $X_{k_i} \leftarrow X_{k_i} \triangledown F_{k_i}(X_1, \ldots, X_n)$ **fi** ;
  $i \leftarrow i + 1$
until $(\forall k, F_k(X_1, \ldots, X_n) \sqsubseteq X_k)$

La suite (k_i) fixe un *ordre d'itération*. La précision du résultat obtenu par cet algorithme dépend fortement de l'ordre d'itération choisi, les opérateurs d'élargissement étant généralement non monotones. Dans l'algorithme que nous avons programmé en Coq, nous nous sommes restreints aux ordres d'itération cycliques, *i.e.* pour lesquels il existe un entier p tel que $\forall i, k_i = k_{i+p}$. Un tel ordre d'itération n'oublie jamais indéfiniment une variable si et seulement si $\{1, \ldots, n\} \subseteq \{k_1, \ldots, k_p\}$. L'algorithme précédent peut alors être spécialisé pour ce type d'ordre en utilisant une liste `l_iter` contenant au moins une fois **tous** les indices de $\{1, \ldots, n\}$.

$(X_1, \ldots, X_n) \leftarrow (\bot, \ldots, \bot)$;
repeat
  l $\leftarrow$ l_iter ;
  while l $\neq \emptyset$ **do**
    k $\leftarrow$head(l) ;
    if $(F_k(X_1, \ldots, X_n) \not\sqsubseteq X_k)$ **then** $X_k \leftarrow X_k \triangledown F_k(X_1, \ldots, X_n)$ **fi** ;
    l $\leftarrow$ tail(l)
  done
until $(\forall k, F_k(X_1, \ldots, X_n) \sqsubseteq X_k)$

Le test d'arrêt de la boucle principale peut être effectué à l'aide d'une variable booléenne `stable` affectée à `false` dès qu'une inéquation n'est pas vérifiée dans la boucle interne.

$(X_1, \ldots, X_n) \leftarrow (\bot, \ldots, \bot)$;
repeat
  l $\leftarrow$ l_iter ;
  stable $\leftarrow$ true ;
  while l $\neq \emptyset$ **do**
    k $\leftarrow$head(l) ;
    if $(F_k(X_1, \ldots, X_n) \not\sqsubseteq X_k)$ **then**
      stable $\leftarrow$ false ;
      $X_k \leftarrow X_k \triangledown F_k(X_1, \ldots, X_n)$
    fi ;
    l $\leftarrow$ tail(l)
  done
until stable

Afin d'améliorer la précision d'une itération chaotique avec élargissement, il est possible d'utiliser un élargissement sur une partie seulement des variables. Les autres variables sont alors modifiées avec $\sqcup$. Nous considérons donc une partie $\mathcal{L}$ de $\{1, \ldots, n\}$ qui sélectionne les points d'élargissement.

$(X_1, \ldots, X_n) \leftarrow (\bot, \ldots, \bot)$;
repeat
  l $\leftarrow$ l_iter ;
  stable $\leftarrow$ true ;
  while l $\neq \emptyset$ **do**
    k $\leftarrow$head(l) ;
    if $(F_k(X_1, \ldots, X_n) \not\sqsubseteq X_k)$ **then**
      stable $\leftarrow$ false ;
      if $k \in \mathcal{L}$ **then** $X_k \leftarrow X_k \triangledown F_k(X_1, \ldots, X_n)$
      else $X_k \leftarrow X_k \sqcup F_k(X_1, \ldots, X_n)$
      fi ;
    fi ;
    l $\leftarrow$ tail(l)
  done
until stable

Pour que cet algorithme termine, l'ensemble $\mathcal{L}$ doit vérifier certaines propriétés vis-à-vis des dépendances du système à résoudre. La fonction F_i *dépend* de sa j-ème composante si et seulement si

$$\exists x_1, \ldots, x_j, x'_j, \ldots, x_n \in A^{n+1}, F_j(x_1, \ldots, x_j, \ldots, x_n) \neq F_j(x_1, \ldots, x'_j, \ldots, x_n)$$

Le *graphe de dépendance* associé au système (4.11) admet n nœuds identifiés aux variables $X_1, \ldots, X_n$ du système. Un arc orienté est placé entre les nœuds X_j et X_i (noté $X_j \to X_i$) si et seulement si la fonction F_i dépend de sa j-ème composante. L'algorithme termine si tous les cycles du graphe de dépendance du système passent par un point d'élargissement (*i.e.* un noeud dont l'indice appartient à $\mathcal{L}$). Un tel résultat est prouvé dans [Cou78], mais n'est pas complètement trivial à démontrer en Coq. Le théorème correspondant serait de toutes façons assez lourd à utiliser par la suite puisqu'il demanderait, pour chaque système à résoudre, de prouver que les points d'élargissement choisis coupent les cycles de dépendance.

Nous avons opté pour une solution plus facile à prouver et plus souple à instancier par la suite. Nous modifions pour cela le test d'arrêt de la boucle principale en ne testant que la stabilisation au niveau des points d'élargissement.

$(X_1, \ldots, X_n) \leftarrow (\bot, \ldots, \bot)$;
repeat
  $l \leftarrow$ l_iter ;
  stable $\leftarrow$ true ;
  while $l \neq \emptyset$ **do**
    $k \leftarrow$head(l) ;
    if $(F_k(X_1, \ldots, X_n) \not\sqsubseteq X_k)$ **then**
      if $k \in \mathcal{L}$ **then**
        stable $\leftarrow$ false ;
        $X_k \leftarrow X_k \nabla F_k(X_1, \ldots, X_n)$
      else $X_k \leftarrow X_k \sqcup F_k(X_1, \ldots, X_n)$
      fi ;
    fi ;
    $l \leftarrow$ tail(l)
  done until stable

La terminaison de ce nouvel algorithme ne repose alors plus sur les dépendances du système.

Théorème 4.4.1. *Quelle que soit la valeur de* `l_iter` *et* $\mathcal{L}$*, l'algorithme précédent termine.*

Preuve : Appelons $X^{(i)}$ la valeur du vecteur X après i itération de la boucle principale. Considérons $p^{\mathcal{L}} : A^n \to A^{|\mathcal{L}|}$ la projection de X vers ces composantes dans $\mathcal{L}$. L'opérateur $\nabla \in A \to A \to A$ peut être généralisé à A^m ($m \in \mathbb{N}$) en posant $\nabla_{A^m}(X, Y)_i = X_i \nabla Y_i$ pour tout indice i. Nous verrons dans le chapitre 6 que cet opérateur vérifie les hypothèses d'un opérateur d'élargissement. En particulier, la relation $\prec_{\nabla_{A^m}}$ vérifie l'hypothèse selon laquelle tous les couples (X, X) sont accessibles. Il suffit alors de remarquer qu'entre la i-ème et la $i+1$-ème itération de la boucle interne, nous avons soit

$p^{\mathcal{L}}(X^{(i+1)}) = p^{\mathcal{L}}(X^{(i)})$ si `stable` vaut `true`, soit $p^{\mathcal{L}}(X^{(i+1)}) = p^{\mathcal{L}}(X^{(i)}) \triangledown_{\mathcal{A}^{|\mathcal{L}|}} p^{\mathcal{L}}(Y)$ avec $Y_i = F_k(X_1, \ldots, X_n)$ si $k \in \mathcal{L}$ et $F_k(X_1, \ldots, X_n) \not\sqsubseteq X_k$, et $Y_i = \bot$ sinon[10]. Cette remarque nous permet de conclure qu'après k itérations de la boucle principale, soit `stable` vaut `true` et l'algorithme se termine, soit il existe $Y^{(k)} \in \mathcal{A}^n$ tel que

$$\left(X^{(k)}, Y^{(k)}\right) \prec^{+}_{\triangledown_{\mathcal{A}^{|\mathcal{L}|}}} (\bot_{\mathcal{A}^{|\mathcal{L}|}}, \bot_{\mathcal{A}^{|\mathcal{L}|}})$$

avec $\prec^{+}_{\triangledown_{\mathcal{A}^{|\mathcal{L}|}}}$ la fermeture transitive de $\prec_{\triangledown_{\mathcal{A}^{|\mathcal{L}|}}}$.

Or $(\bot_{\mathcal{A}^{|\mathcal{L}|}}, \bot_{\mathcal{A}^{|\mathcal{L}|}})$ n'appartient à aucune chaîne infinie selon $\prec_{\triangledown_{\mathcal{A}^{|\mathcal{L}|}}}$. La boucle principale atteindra donc nécessairement une itération à la fin de laquelle `stable` vaudra `true`. □

Ce théorème a l'avantage d'être très facile à instancier en Coq, puisqu'aucune hypothèse n'est nécessaire sur `l_iter` et $\mathcal{L}$. Nous n'avons cependant aucune assurance que le résultat final sera une solution du système. En Coq, nous nous contentons de tester si le résultat est une solution. Nous prouvons maintenant que ce test réussira toujours si $\mathcal{L}$ et `l_iter` vérifient certaines hypothèses.

Théorème 4.4.2. *Soit $\mathcal{L}$ une partie de $\{1, \ldots, n\}$ telle que tous les cycles du graphe de dépendance de (4.11) passent par un noeud X_i, $i \in \mathcal{L}$. Soit `l_iter` une permutation de $\{1, \ldots, n\}$ telle que, pour tout arc $X_i \to X_j$ du graphe de dépendance, soit j appartient à $\mathcal{L}$, soit i apparaît avant j dans `l_iter`$= [\ldots, i, \ldots, j, \ldots]$ (*tri topologique faible*). Alors l'algorithme précédent termine sur une solution de (4.11).*

Preuve : Commençons par remarquer que lorsqu'une variable X_k, avec $k \notin \mathcal{L}$, est examinée durant la i-ème itération de la boucle principale, tous ses prédécesseurs dans le graphe de dépendance ont été mis à jour, grâce à l'hypothèse de tri topologique faible. Nous avons ainsi

$$X_k^{(i)} = X_k^{(i-1)} \sqcup F_k(X_1^{(i)}, \ldots, X_n^{(i)})$$

Notons i le nombre d'itérations nécessaires pour que l'algorithme termine. $i > 0$ puisqu'il s'agit d'une boucle **`repeat...until`**. Supposons qu'il existe une variable X_k qui ne soit pas stabilisée après cette dernière itération (*i.e.* $X_k^{(i)} \neq X_k^{(i-1)}$). Puisque la fin de l'algorithme a été atteint, nous avons nécessairement $k \notin \mathcal{L}$ (après une modification d'un X_k, $k \in \mathcal{L}$, `stable` vaut `false` durant toute la fin de la boucle interne). Considérons maintenant l'ensemble J des prédécesseurs de X_k dans le graphe de dépendance du système. Puisque $k \notin \mathcal{L}$, nous avons

$$X_k^{(i)} = X_k^{(i-1)} \sqcup F_k(X_1^{(i)}, \ldots, X_n^{(i)}) \tag{4.12}$$

$$X_k^{(i-1)} = X_k^{(i-2)} \sqcup F_k(X_1^{(i-1)}, \ldots, X_n^{(i-1)}) \tag{4.13}$$

L'une des variables de J est nécessairement non stabilisée. Dans le cas contraire, nous aurions $X_j^{(i)} = X_j^{(i-1)}$ pour tout j dans J. Par conséquent

$$F_k(X_1^{(i)}, \ldots, X_n^{(i)}) = F_k(X_1^{(i-1)}, \ldots, X_n^{(i-1)}) \tag{4.14}$$

[10] Nous utilisons ici l'hypothèse $x \triangledown \bot = x, \forall x$.

Or, d'après (4.13) nous avons $F_k(X_1^{(i-1)}, \ldots, X_n^{(i-1)}) \sqsubseteq X_k^{(i-1)}$. L'égalité (4.12) se réécrit alors

$$\begin{aligned} X_k^{(i)} &= X_k^{(i-1)} \sqcup F_k(X_1^{(i)}, \ldots, X_n^{(i)}) \\ &= X_k^{(i-1)} \sqcup F_k(X_1^{(i-1)}, \ldots, X_n^{(i-1)}) \\ &= X_k^{(i-1)} \end{aligned}$$

Ce qui contredit l'hypothèse $X_k^{(i)} \neq X_k^{(i-1)}$. Il existe donc une autre variable X'_k non stable avec $k' \in J$. En répétant le procédé nous aboutirons nécessairement à un cycle du graphe de dépendance qui ne passe que par des variables non-stabilisées. C'est impossible puisque les points d'élargissement sont tous stabilisés.

Nous pouvons ainsi en déduire que, pour tout $k \in \{1, \ldots, n\}$, $X_k^{(i)} = X_k^{(i-1)}$. Ceci implique que $X^{(i)}$ est une solution du système. □

La notion de tri topologique faible est présenté dans la thèse de François Bourdoncle [Bou92]. Plusieurs algorithmes des graphes permettant de sélectionner des points d'élargissement et de construire un tri topologique faible associé y sont présentés. En pratique on peut s'aider de la structure du programme analysé. Nous y reviendrons dans le chapitre 5. Le résultat du théorème 4.4.2 est rapidement démontré à la page 48 de [Bou92] durant une preuve de complexité.

Cet algorithme a été programmé en Coq, sous une version fonctionnelle[11]. Une technique similaire est utilisée avec l'opérateur de rétrécissement. Nous prouvons uniquement la terminaison. Le résultat final est ensuite testé. Si le test échoue, nous utilisons une itération avec élargissement/rétrécissement standard. Le test n'échouera pas, de toutes façons, si les hypothèses du théorème 4.4.2 sont respectées par $\mathcal{L}$ et `l_iter`.

Le chapitre suivant fournira des exemples d'utilisation de cet algorithme.

4.5 Regroupement des trois critères dans une même interface

Nous disposons maintenant de trois critères afin de calculer des post-points fixes dans un treillis. Nous allons maintenant finir ce chapitre en présentant l'interface commune qui regroupera ces critères en Coq[12].

Nous définissons trois drapeaux différents à l'aide d'un type énuméré,

```
Inductive termination_criteria_flag : Set :=
 | ASCENDING_CHAIN_CONDITION
 | CLASSIC_ACCELERATING
 | ACCELERATING_NOT_MONOTONE.
```

puis un type énuméré plus riche

```
Inductive termination_criteria : Set :=
 | ascending_chain_condition :
```

[11] Voir fichier `chaotic.v`.
[12] Voir fichier `solve.v`

```
      well_founded (fun x y ⇒ ¬ eq y x ∧ order y x) →
      termination_criteria
  | classic_accelerating :
      widening_classic_operator → narrowing_classic_operator →
      termination_criteria
  | accelerating_not_monotone :
      widening_operator → narrowing_operator →
      termination_criteria.
```

Un élément de type `termination_criteria` contient soit une preuve qu'un treillis vérifie la condition de chaîne ascendante, soit deux opérateurs d'élargissement/rétrécissement standards, soit deux opérateurs d'élargissement/rétrécissement dans leurs versions modifiées à la section 4.4. Un tel objet est donc une preuve constructive que l'un de nos trois critères est vérifié.

Nous utilisons une dernière relation reliant drapeaux et critères de terminaison.

```
Inductive termination_criteria_kind :
   termination_criteria → termination_criteria_flag → Prop :=
| termination_criteria_kind_ascending_chain_condition :
    ∀ h, termination_criteria_kind
                (ascending_chain_condition h)
                ASCENDING_CHAIN_CONDITION
| termination_criteria_kind_classic_accelerating :
    ∀ w n, termination_criteria_kind
                (classic_accelerating w n)
                CLASSIC_ACCELERATING
| termination_criteria_kind_accelerating_not_monotone :
    ∀ w n, termination_criteria_kind
                (accelerating_not_monotone w n)
                ACCELERATING_NOT_MONOTONE.
```

Grâce à ces déclarations de type, nous pouvons proposer une signature de module qui modélise les structures de treillis dans lesquelles nous savons calculer des post-points fixes.

```
Module Type LatticeSolve.

  Declare Module Lat : Lattice.

  Parameter termination_property :
   termination_criteria Lat.eq Lat.order Lat.bottom Lat.meet.

  Parameter termination_flag : termination_criteria_flag.

  Parameter termination_flag_correct :
    termination_criteria_kind termination_property termination_flag.

End LatticeSolve.
```

Lorsque nous voudrons parler d'un treillis vérifiant la condition de chaîne ascendante il nous suffira de considérer un module `S` de type

```
LatticeSolve with Definition termination_flag := ASCENDING_CHAIN_CONDITION.
```

La construction *signature* **with** *definition* permet ici de spécialiser le type d'un module.

Grâce à cette signature, nous pouvons écrire un foncteur de module `Solve` qui prend en argument un module `S` de type `LatticeSolve` et construit différents solveurs de post-points fixes. Certains solveurs nécessitent des hypothèses sur le type de critère apparaissant dans `S`.

```
Module Solve (S:LatticeSolve).

 Lemma solve_wf :
   S.termination_flag = ASCENDING_CHAIN_CONDITION →
   ∀ f, Def.monotone f →
   { a :  S.Lat.t | S.Lat.order (f a) a ∧
                    ∀ y, S.Lat.order (f y) y → S.Lat.order a y }.
 Proof. ... Qed.

 ...
End Solve.
```

4.6 Conclusion

Nous avons présenté trois critères permettant de calculer des post-points fixes dans un treillis. L'inclusion logique de ces critères est présentée dans la figure 4.8. Les opérateurs d'élargissement/rétrécissement proposent le cadre le plus large. Chaque critère propose un solveur de post-point fixe avec une spécification plus ou moins riche et contraignante à instancier. La situation est résumée dans le tableau de la figure 4.9.

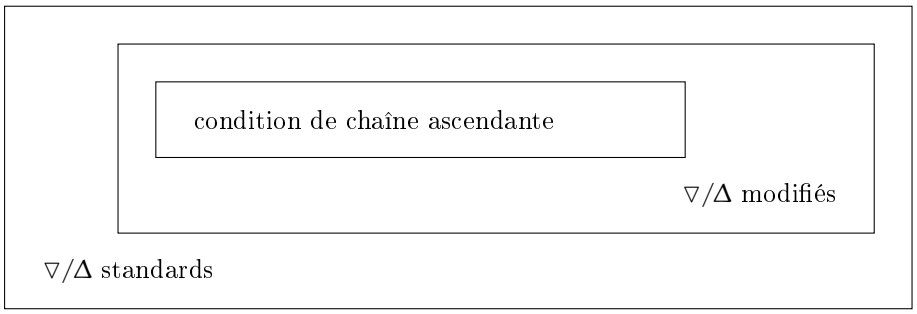


FIG. 4.8 – Inclusions des différents critères pour le calcul de post-point fixe.

Les opérateurs d'élargissement/rétrécissement classiques ont été principalement présentés pour montrer la possibilité d'utiliser les mêmes critères que ceux traditionnellement utilisé dans la littérature. La définition proposée dans la section 4.4 suit une autre démarche : nous prenons quelques libertés avec les critères et les algorithmes (pour l'itération chaotique) standards afin de simplifier les preuves et de proposer des théorèmes Coq plus facilement instantiables (avec moins d'hypothèses).

Remarquons pour conclure, que la voie choisie dans ce chapitre est relativement ambitieuse puisque nous programmons tous nos calculs itératifs en Coq. Cela nous permet-

	condition de chaîne ascendante	∇/Δ classiques	∇/Δ modifiés
calcul du plus petit (post-) point fixe	✓		
hypothèse f monotone pas nécessaire			✓
résolution de systèmes d'inéquations	✓	✓	✓ (itération chaotique)

FIG. 4.9 – Comparaison des différents solveurs associés à chaque critère de terminaison.

tra d'extraire, lors des chapitres 5 et 8 des analyseurs certifiés de type `program → verdict`. Une autre alternative consiste à seulement extraire un vérificateur de post-point fixe certifié : le calcul itératif peut alors être effectué par un programme non certifié, et son résultat postérieurement vérifié par le vérificateur extrait. L'analyseur extrait devient alors de type `program → indice → verdict`. Le deuxième argument est le résultat d'un calcul itératif non certifié. Le programme extrait se charge de vérifier la correction du résultat et donne un verdict en conséquence. Cette technique reste un cas particulier de nos travaux. Nous préférons démontrer dans ce chapitre qu'un assistant de preuve comme Coq est tout-à-fait capable de traiter des algorithmes itératifs non triviaux.

La première partie de cette démonstration a été faite durant ce chapitre : nous disposons maintenant d'un code extrait certifié, très proche de celui que nous aurions pu programmer directement en Caml. Chaque solveur générique repose néanmoins sur un critère de terminaison qu'il faudra être capable de prouver. Ces preuves feront l'objet du chapitre 6.

Chapitre 5

Un interpréteur abstrait modulaire pour un langage While

Dans ce chapitre, nous allons présenter un interpréteur abstrait pour un langage WHILE jouet. Cet analyseur sera entièrement programmé en Coq, avec un type assurant sa correction vis-à-vis de la sémantique du langage.

L'analyseur présenté ici s'inspire grandement des algorithmes présentés dans [Cou99]. Nous avons néanmoins entièrement repris les preuves de correction. En effet, Patrick Cousot s'attache dans [Cou99] à démontrer que les connexions de Galois permettent la dérivation systématique d'un analyseur à partir des spécifications de la forme $\alpha \circ f \circ \gamma$. Comme nous l'avons expliqué dans le chapitre 3, nous n'utilisons pas de connexions de Galois dans nos développements Coq. Par conséquent, ce genre de dérivations systématiques n'est pas à notre portée. Nous avons donc repris les preuves pour les adapter au cadre de l'interprétation abstraite que nous avions retenu.

Nous allons dans un premier temps présenter la sémantique du langage étudié dans la section 5.1. Nous présenterons ensuite les trois grands composants de notre analyseur : l'analyse des états mémoires en section 5.2, l'analyse des environnements de variable en section 5.3 et enfin l'abstraction des valeurs numériques dans la section 5.4. Nous conclurons par des exemples d'analyses effectuées grâce au programme extrait dans la section 5.5. La conception modulaire de notre analyseur nous permettra d'utiliser plusieurs techniques d'abstraction.

5.1 Présentation du langage

5.1.1 Syntaxe du langage

Le langage WHILE que nous considérons dans ce chapitre est un langage impératif rudimentaire. Sa syntaxe est résumée dans la figure 5.1. Un programme WHILE est une liste (appelée *bloc*) d'instructions. Une instruction est soit l'affectation d'une variable par une expression numérique, soit une conditionnelle **if**...**else**..., soit une boucle **while**. Une expression numérique comporte des constantes, des opérations arithmétiques (mul-

$$
\begin{array}{rll}
\mathit{expr} ::= & \mathtt{n} & \mathtt{n} \in \mathbb{Z} \\
\mid & \mathbf{?} & \\
\mid & \mathtt{x} & \mathtt{x} \in \mathbb{V} \\
\mid & - \ \mathit{expr} & \\
\mid & \mathit{expr} \ \odot \ \mathit{expr} & \odot \in \{+, -, \times\} \\
\\
\mathit{test} ::= & \mathit{expr} \ c \ \mathit{expr} & c \in \{<, \leq, >, \geq, =, \neq\} \\
\mid & \textbf{not} \ \mathit{test} & \\
\mid & \mathit{test} \ \textbf{and} \ \mathit{test} & \\
\mid & \mathit{test} \ \textbf{or} \ \mathit{test} & \\
\\
\mathit{instr} ::= & \mathtt{x} \ := \ \mathit{expr} & \\
\mid & \textbf{if} \ \mathit{test} \ \{ \ \mathit{block} \ \} \ \{ \ \mathit{block} \ \} & \\
\mid & \textbf{while} \ \mathit{test} \ \{ \ \mathit{block} \ \} & \\
\\
\mathit{block} ::= & \mathtt{p} & \mathtt{p} \in \mathbb{P} \\
\mid & \mathtt{p} \ \textbf{:} \ \mathit{instr} \ \textbf{;} \ \mathit{block} &
\end{array}
$$

FIG. 5.1 – Syntaxe du langage WHILE

tiplication, soustraction et addition), des variables et une constante particulière, notée ?, représentant une valeur numérique quelconque (pouvant par exemple provenir d'une entrée interactive). Les instructions de branchement conditionnel et de boucle sont gardées par des tests. Un test est une comparaison entre deux expressions arithmétiques. Une particularité du langage réside dans l'utilisation explicite de *points de programmes* (pris dans un ensemble $\mathbb{P}$). Ceux-ci nous seront utiles pour la définition de la sémantique du langage. Notre interpréteur abstrait attachera des invariants mémoires à chaque point de programme.

La figure 5.2 donne un exemple de programme WHILE. Dans la suite du document, nous omettrons les points de programme dans nos différents exemples, pour plus de lisibilité.

```
0: X = ?;
1: if X < 0 {
2:   while X < 0 {
3:     X = X + 1;
4:   };
5:   Y = X;
6: } else {
7:   Y = 0;
8: };
9:
```

FIG. 5.2 – Exemple de programme WHILE

Les notions de bloc et d'instruction dépendent l'une de l'autre. Il est donc nécessaire des les définir simultanément. En Coq, il nous faut ainsi recourir à une définition mutuellement inductive. La définition Coq de la syntaxe du langage WHILE est donnée dans la figure 5.3[1]. Un programme est constitué d'un bloc d'instructions et d'une liste de noms de variables. Cette liste représente les variables qui sont utilisées dans le programme. L'emploi de cette liste, à ce stade, correspond à un choix délibéré sur la sémantique du langage. Nous commenterons ce choix dans la section suivante.

Le type des points de programme et des noms de variables est le type `Word`. Nous aurions, a priori pu prendre n'importe quel type puisque qu'aucune opération ne sera effectuée sur ces éléments. Nous aurons néanmoins besoin de prouver que nous n'utilisons qu'un nombre fini de points de programmes et de noms de variables. Pour un programme donné, ces ensembles sont nécessairement finis. Une première solution serait donc d'utiliser des types dépendants d'un programme. Cette solution avait été choisie dans nos premiers travaux [CJPR04, CJPR05]. Le système de type de base (sans les modules) de Coq se prête tout-à-fait à ce genre de formalisation. Avec l'utilisation des modules, il devient beaucoup plus lourd de faire dépendre certains types d'autres éléments car un module ne peut pas dépendre d'un terme Coq, mais seulement d'un autre module. Nous avons ainsi opté dans ce travail pour une autre solution : l'usage du type `Word`. Ce type, défini par nos soins, contient par définition un nombre fini d'éléments. Il représente l'ensemble des entiers binaires représentables avec 32 bits. Nous utilisons pour cela le type `bin` présenté dans le chapitre 2.

5.1.2 Sémantique concrète du langage

Le but de notre analyseur sera de calculer des invariants vérifiés par les variables d'un programme, au cours de son exécution. Nous nous intéressons par conséquent à l'ensemble des états mémoires accessibles lors de l'exécution d'un programme. Un état mémoire est constitué d'un point de programme et d'un environnement de variables. Un environnement de variables est une fonction de l'ensemble des noms de variables vers les entiers relatifs. Cette définition du domaine sémantique est récapitulée dans la figure 5.4.

Pour pouvoir définir mathématiquement l'ensemble des états accessibles d'un programme, nous aurons besoin de définir au préalable la sémantique des expressions numériques et des tests.

Sémantique des expressions numériques La sémantique des expressions donnée en figure 5.5, associe à chaque expression e l'ensemble $[\![e]\!]\uparrow^{\rho}_{\text{expr}}$ des valeurs numériques qui peuvent résulter de l'évaluation de e dans l'environnement ρ. Certaines expressions (comme ?) ont une sémantique non déterministe, permettant ainsi de modéliser l'interaction avec l'environnement d'exécution. $[\![\odot]\!]\uparrow_{\text{op}}$ désigne ici la sémantique usuelle des opérateurs arithmétiques.

[1] Nous utilisons ici une version simplifiée des définitions inductives Coq, plus proche de la syntaxe des définitions de type en Caml

```
Definition VarName : Set := Word.
Definition ProgPoint : Set := Word.

Inductive op : Set := Add | Sub | Mult.

Inductive expr : Set :=
   Const (n:Z)
 | Unknown
 | Var (x:VarName)
 | Minus (e:expr)
 | Numop (o:op) (e1 e2:expr).

Inductive comp : Set := Eq | Neq | Lt | Le | Gt | Ge.

Inductive test : Set :=
 | Numcomp (c:comp) (e1 e2:expr)
 | Not (t:test)
 | And (t1 t2:test)
 | Or (t1 t2:test).

Inductive instr : Set :=
   Affect (x:VarName) (e:expr)
 | If (t:test) (b1 b2:block)
 | While (t:test) (b:block)
with block : Set :=
   Empty (p:ProgPoint)
 | Seq (p:ProgPoint) (i:instr) (b:block).

Record program : Set := prog {
 instrs : block;
 vars : list VarName
}.
```

FIG. 5.3 – Définition Coq de la syntaxe de WHILE

$$\begin{aligned} \text{Env} &\stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{Z} \\ \text{Etat} &\stackrel{\text{def}}{=} \mathbb{P} \times \text{Env} \end{aligned}$$

FIG. 5.4 – Domaines sémantiques

$$\begin{aligned} & [\![e]\!]\uparrow^{\rho}_{\text{expr}} \in \mathcal{P}(\mathbb{Z}), \quad e \in \mathit{expr},\ \rho \in \text{Env} \\ [\![n]\!]\uparrow^{\rho}_{\text{expr}} &= \{\, n \,\} \\ [\![?]\!]\uparrow^{\rho}_{\text{expr}} &= \mathbb{Z} \\ [\![x]\!]\uparrow^{\rho}_{\text{expr}} &= \{\, \rho(x) \,\},\ x \in \text{Var}_P \\ [\![-e]\!]\uparrow^{\rho}_{\text{expr}} &= \{\, [\![-]\!]\uparrow_{\text{op}} v \mid v \in [\![e]\!]\uparrow^{\rho}_{\text{expr}} \,\} \\ [\![e_1 \odot e_2]\!]\uparrow^{\rho}_{\text{expr}} &= \{\, v_1 [\![\odot]\!]\uparrow_{\text{op}} v_2 \mid v_1 \in [\![e_1]\!]\uparrow^{\rho}_{\text{expr}},\ v_2 \in [\![e_2]\!]\uparrow^{\rho}_{\text{expr}} \,\} \end{aligned}$$

FIG. 5.5 – Sémantique des expressions

En Coq (voir figure 5.6), cette sémantique est définie à l'aide d'une définition inductive, paramétrée par un programme et un environnement de variables. Le type de `(sem_expr P env)` peut être vu soit comme celui d'une relation entre expressions et valeurs numériques, soit comme celui d'une fonction des expressions vers un ensemble de valeurs numériques grâce à la correspondance entre $\mathcal{P}(\mathbb{Z})$ et `Z → Prop`.

```
Inductive sem_expr (P:program) (env:environment) :
                                        expr → Z → Prop :=
  sem_const : ∀ n, sem_expr P env (Const n) n
| sem_unknow : ∀ n, sem_expr P env Unknown n
| sem_var : ∀ x,
    In x (vars P) → sem_expr P env (Var x) (env x)
| sem_minus : ∀ e n,
    sem_expr P env e n →
    sem_expr P env (Minus e) (-n)
| sem_numop : ∀ o e1 e2 n1 n2 n,
    sem_expr P env e1 n1 →
    sem_expr P env e2 n2 →
    sem_op o n1 n2 n →
    sem_expr P env (Numop o e1 e2) n.
```

FIG. 5.6 – Sémantique des expressions en Coq

Notons que l'affectation d'une variable est seulement définie pour les variables déclarées dans la liste `(vars P)` attachée à un programme. Il s'agit ici d'une facilité que nous nous accordons. Une meilleure alternative serait de définir une sémantique sans restriction sur les variables utilisables dans les affectations. Nous pourrions alors démontrer l'équivalence de cette nouvelle sémantique avec l'ancienne. Cette équivalence pourrait en effet être établie pour les expressions arithmétiques qui apparaissent dans le texte d'un programme donné. Nous utilisons donc ici la sémantique la plus adaptée à la construction d'un analyseur certifié, même si un choix plus rigoureux aurait été possible, au prix de plusieurs preuves supplémentaires, notre définition courante faisant alors office de sémantique intermédiaire. Ce type de manipulation sémantique sera effectué lors du chapitre 8 sur l'analyse de bytecode Java.

Sémantique des tests La sémantique des tests, donnée en figure 5.7, associe à chaque test t l'ensemble $[\![\mathsf{e}]\!]\uparrow^{\rho}_{\text{test}}$ des booléens qui peuvent résulter de l'évaluation de t dans l'environnement ρ. Le non-déterminisme de la sémantique des expressions entraîne le non-déterminisme de $[\![\cdot]\!]\uparrow^{\rho}_{\text{test}}$. $[\![c]\!]$ désigne ici la sémantique usuelle des comparaisons arithmétiques. La traduction Coq de ces définitions est donnée dans la figure 5.8.

Sémantique opérationnelle Les définitions précédentes vont maintenant nous permettre de décrire l'évolution des variables d'un programme au cours de son exécution. Nous allons pour cela associer à chaque programme un système de transitions sur ces états. Ce système comportera un ensemble d'états initiaux $\mathcal{S}_0$ ainsi qu'une relation de

$$[\![t]\!]\uparrow^{\rho}_{\text{test}} \in \mathcal{P}(\mathbb{B}), \quad t \in \mathit{test},\ \rho \in \text{Env}$$

$$\frac{v_1 \in [\![e_1]\!]\uparrow^{\rho}_{\text{expr}} \quad v_2 \in [\![e_2]\!]\uparrow^{\rho}_{\text{expr}} \quad v_1[\![c]\!]\uparrow_{\text{comp}} v_2}{T \in [\![e_1\ c\ e_2]\!]\uparrow^{\rho}_{\text{test}}}$$

$$\frac{v_1 \in [\![e_1]\!]\uparrow^{\rho}_{\text{expr}} \quad v_2 \in [\![e_2]\!]\uparrow^{\rho}_{\text{expr}} \quad \neg\,(v_1[\![c]\!]\uparrow_{\text{comp}} v_2)}{F \in [\![e_1\ c\ e_2]\!]\uparrow^{\rho}_{\text{test}}}$$

$$\frac{b_1 \in [\![t_1]\!]\uparrow^{\rho}_{\text{test}} \quad b_2 \in [\![t_2]\!]\uparrow^{\rho}_{\text{test}}}{b_1 \wedge b_2 \in [\![t_1 \textbf{ and } t_2]\!]\uparrow^{\rho}_{\text{test}}} \qquad \frac{b_1 \in [\![t_1]\!]\uparrow^{\rho}_{\text{test}} \quad b_2 \in [\![t_2]\!]\uparrow^{\rho}_{\text{test}}}{b_1 \vee b_2 \in [\![t_1 \textbf{ or } t_2]\!]\uparrow^{\rho}_{\text{test}}}$$

FIG. 5.7 – Sémantique des tests

```
Inductive sem_comp : comp → Z → Z → Prop :=
 | sem_eq : ∀ n1 n2, n1=n2 → sem_comp Eq n1 n2
 | sem_neq : ∀ n1 n2, n1<>n2 → sem_comp Neq n1 n2
 | sem_lt : ∀ n1 n2, n1<n2 → sem_comp Lt n1 n2
 | sem_le : ∀ n1 n2, n1<=n2 → sem_comp Le n1 n2
 | sem_gt : ∀ n1 n2, n1>n2 → sem_comp Gt n1 n2
 | sem_ge : ∀ n1 n2, n1>=n2 → sem_comp Ge n1 n2.

Inductive sem_test (P:program) (env:environment) :
  test → bool → Prop :=
 | sem_numcomp_true : ∀ c e1 e2 n1 n2,
     sem_expr P env e1 n1 →
     sem_expr P env e2 n2 →
     sem_comp c n1 n2 →
     sem_test P env (Numcomp c e1 e2) true
 | sem_numcomp_false : ∀ c e1 e2 n1 n2,
     sem_expr P env e1 n1 →
     sem_expr P env e2 n2 →
     ¬ sem_comp c n1 n2 →
     sem_test P env (Numcomp c e1 e2) false
 | sem_not : ∀ t b,
     sem_test P env t b →
     sem_test P env (Not t) (negb b)
 | sem_and : ∀ t1 t2 b1 b2,
     sem_test P env t1 b1 →
     sem_test P env t2 b2 →
     sem_test P env (And t1 t2) (andb b1 b2)
 | sem_or : ∀ t1 t2 b1 b2,
     sem_test P env t1 b1 → sem_test P env t2 b2 →
     sem_test P env (Or t1 t2) (orb b1 b2).
```

FIG. 5.8 – Sémantique des tests en Coq

transition $\rightarrow_P \subseteq \wp(\text{Etat} \times \text{Etat})$ entre états mémoires. Afin de simplifier la preuve de correction de notre analyse vis à vis de la sémantique concrète, nous allons partitionner $\rightarrow_P$ en fonction de la nature de la transition. $\rightarrow_P$ sera donc de la forme

$$\rightarrow_P = \bigcup_{j \in I_P} \rightarrow_j$$

avec I_p un ensemble d'indices dépendant du programme P. Chaque indice sera de la forme (p_1, i, n, p_2) avec i une instruction, p_1 et p_2 deux points qui désignent la position de i dans le programme, et n un entier qui représente le numéro de la règle de transition associée à i. En effet, une même instruction peut se voir associer plusieurs règles de transition. C'est par exemple le cas de l'instruction conditionnelle qui, comme nous le verrons, sera associée à quatre transitions.

Pour définir plus précisément la relation $\rightarrow_j$ nous devons d'abord préciser la notion de *point d'entrée* et *point de sortie* d'un bloc. Nous introduisons pour cela la fonction *debut* qui désigne le premier point de programme d'un bloc

$$\begin{array}{rccc} \mathit{debut} : & \mathit{block} & \rightarrow & \mathbb{P} \\ & (p_1 : i_1 \ ; \ \ldots p_n : i_1 \ ; p_{n+1}) & \mapsto & p_1 \end{array}$$

et la fonction *fin* qui désigne le dernier point de programme.

$$\begin{array}{rccc} \mathit{fin} : & \mathit{block} & \rightarrow & \mathbb{P} \\ & (p_1 : i_1 \ ; \ \ldots p_n : i_1 \ ; p_{n+1}) & \mapsto & p_{n+1} \end{array}$$

La figure 5.9 présente les différentes règles de transition associées à chaque type d'indice (p_1, i, p_2, n). Une affectation de variable $x := e$ entre les points de programme p_1 et p_2 fait passer d'un état $\langle\!\langle p_1, \rho \rangle\!\rangle$ à un état $\langle\!\langle p_2, \rho[x \mapsto v] \rangle\!\rangle$ si v est une valeur possible pour l'évaluation de l'expression e dans l'environnement ρ. L'instruction conditionnelle se voit attacher quatre transitions. Les deux premières transfèrent l'environnement de variables courant vers les points de programmes situés au début des deux blocs b_1 et b_2 de l'instruction. Chacune de ces transitions dépend du succès ou de l'échec du test t dans l'environnement courant ρ. Les deux dernières transitions transfèrent les états atteints à la fin des blocs b_1 et b_2 vers le point de programme de sortie p_2 de l'instruction. L'instruction de boucle se voit, quant à elle, attacher trois transitions. En fonction du succès du test t dans l'environnement ρ de l'état $\langle\!\langle p_1, \rho \rangle\!\rangle$, le contrôle de l'exécution passe vers le début du bloc b (le corps de la boucle), ou saute vers le point de sortie p_2 de l'instruction. Une dernière transition assure le passage entre la fin du bloc b et le début de l'instruction, pour réévaluer le test t.

En Coq (voir figure 5.10), les indices (p_1, i, p_2, n) sont représentés par un triplet constitué de deux points de programmes et d'un élément de type `rule`. Le type `rule` consiste en une simple recopie du type instruction `instr` avec autant de duplication de constructeur qu'il y a de transitions possibles pour chaque instruction. Cette duplication est symbolisée par la fonction `instr_of_rule` qui convertit une règle en son instruction associée :

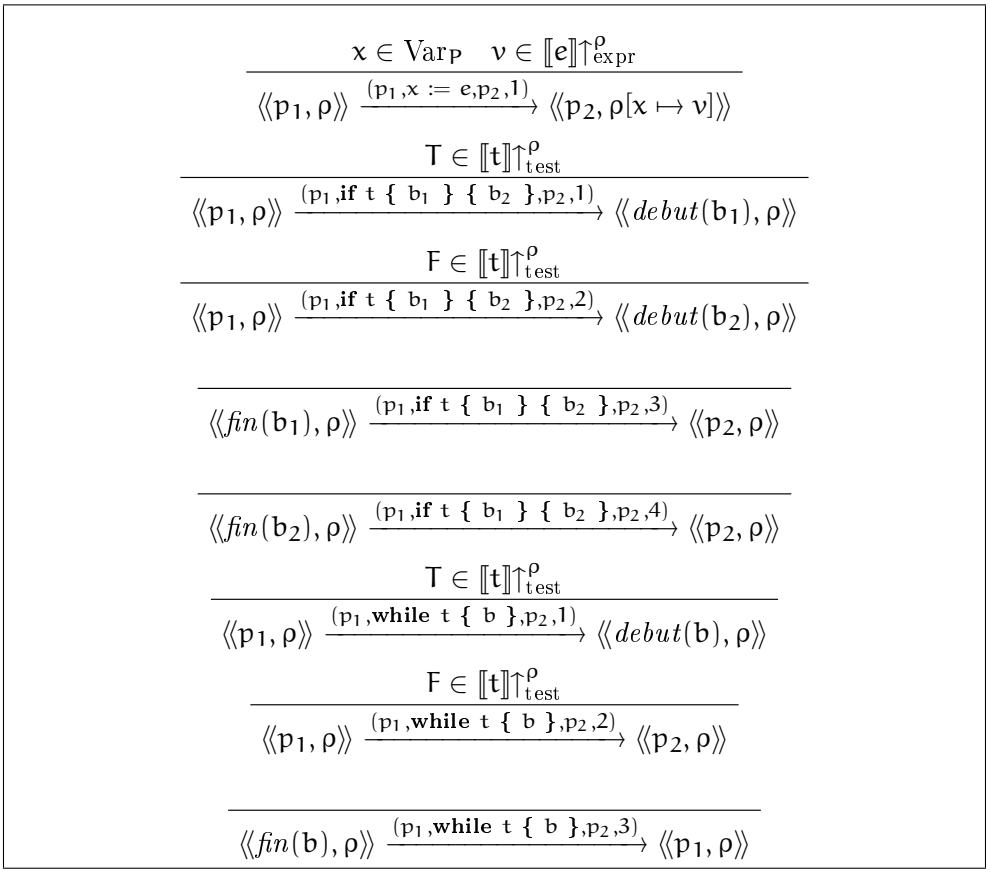


FIG. 5.9 – Sémantique opérationnelle

```
Definition instr_of_rule (r:rule) : instr :=
  match r with
    Affect' x e ⇒ Affect x e
  | If'_1 t b1 b2 ⇒ If t b1 b2
  | If'_2 t b1 b2 ⇒ If t b1 b2
  | If'_3 t b1 b2 ⇒ If t b1 b2
  | If'_4 t b1 b2 ⇒ If t b1 b2
  | While'_1 t b ⇒ While t b
  | While'_2 t b ⇒ While t b
  | While'_3 t b ⇒ While t b
  end.
```

Le mécanisme de notation de Coq nous permet de noter `<<p,env>>` un environnement $\langle\!\langle \mathrm{p}, \mathrm{env} \rangle\!\rangle$.

```
Inductive smallstepAt (P:program) :
     ProgPoint → rule → ProgPoint → state → state → Prop :=
 | smallstepAt_affect : ∀ p1 p2 x e n env1 env2,
     In x (vars P) →
     sem_expr P env1 e n →
     subst env1 x n env2 →
     smallstepAt P p1 (Affect' x e) p2 <<p1,env1>> <<p2,env2>>
 | smallstepAt_if_begin_true : ∀ p1 p2 t b1 b2 env,
     sem_test P env t true →
     smallstepAt P p1 (If'_1 t b1 b2) p2
                 <<p1,env>> <<begin_block b1,env>>
 | smallstepAt_if_begin_false : ∀ p1 p2 t b1 b2 env,
     sem_test P env t false →
     smallstepAt P p1 (If'_2 t b1 b2) p2
                 <<p1,env>> <<begin_block b2,env>>
 | smallstepAt_if_end_true : ∀ p1 p2 t b1 b2 env,
     smallstepAt P p1 (If'_3 t b1 b2) p2
                 <<end_block b1,env>> <<p2,env>>
 | smallstepAt_if_end_false : ∀ p1 p2 t b1 b2 env,
     smallstepAt P p1 (If'_4 t b1 b2) p2
                 <<end_block b2,env>> <<p2,env>>
 | smallstepAt_while_begin_true : ∀ p1 p2 t b env,
     sem_test P env t true →
     smallstepAt P p1 (While'_1 t b) p2
                 <<p1,env>> <<begin_block b,env>>
 | smallstepAt_while_end_true : ∀ p1 p2 t b env,
     smallstepAt P p1 (While'_2 t b) p2
                 <<end_block b,env>> <<p1,env>>
 | smallstepAt_while_false : ∀ p1 p2 t b env,
     sem_test P env t false →
     smallstepAt P p1 (While'_3 t b) p2 <<p1,env>> <<p2,env>>.
```

FIG. 5.10 – Sémantique opérationnelle en Coq

L'ensemble $\mathrm{I_P}$ des indices d'un programme doit maintenant être précisé. Il nous faut d'abord définir formellement le fait qu'une instruction i apparaisse entre deux points $\mathrm{p_1}$ et $\mathrm{p_2}$, dans un programme P. Nous définissons donc un prédicat $\mathrm{instrAt_P}$

qui désigne l'ensemble des triplets (p_1, i, p_2) valides. Nous utilisons pour cela deux relations mutuellement dépendantes $instrAt_{\text{block}}$ et $instrAt_{\text{instr}}$ qui définissent l'ensemble des triplets (p_1, i, p_2) qui apparaissent à l'intérieur d'un bloc (respectivement d'une instruction).

$$instrAt_{\text{block}} \in block \to \mathcal{P}(\mathbb{P} \times instr \times \mathbb{P}) \qquad instrAt_{\text{instr}} \in instr \to \mathcal{P}(\mathbb{P} \times instr \times \mathbb{P})$$

$$\frac{}{(p, i, debut(b)) \in instrAt_{\text{block}}(p\ :\ i\ ;\ b)} \qquad \frac{(p_1, i, p_2) \in instrAt_{\text{block}}(b_1)}{(p_1, i, p_2) \in instrAt_{\text{instr}}(\textbf{if}\ t\ \{\ b_1\ \}\ \{\ b_2\ \})}$$

$$\frac{(p_1, i', p_2) \in instrAt_{\text{block}}(b)}{(p_1, i', p_2) \in instrAt_{\text{block}}(p\ :\ i\ ;\ b)} \qquad \frac{(p_1, i, p_2) \in instrAt_{\text{block}}(b_2)}{(p_1, i, p_2) \in instrAt_{\text{instr}}(\textbf{if}\ t\ \{\ b_1\ \}\ \{\ b_2\ \})}$$

$$\frac{(p_1, i', p_2) \in instrAt_{\text{instr}}(i)}{(p_1, i', p_2) \in instrAt_{\text{block}}(p\ :\ i\ ;\ b)} \qquad \frac{(p_1, i, p_2) \in instrAt_{\text{block}}(b)}{(p_1, i, p_2) \in instrAt_{\text{instr}}(\textbf{while}\ t\ \{\ b\ \})}$$

La définition Coq est similaire (à l'aide d'une définition mutuellement inductive). Le prédicat instrAt_P est alors équivalent à $instrAt_{\text{block}}(P)$ (P est ici identifié à son bloc principal d'instructions).

La relation $\to_P$ peut finalement être définie en considérant l'union de toutes les transitions $\to_{(p_1,i,p_2,n)}$ où (p_1, i, p_2) est un triplet valide du programme P ($(p_1, i, p_2) \in \text{instrAt}_P$) et (i, n) un nom de règle valide. La définition Coq utilise une définition inductive avec une seule règle :

```
Inductive smallstep (P:program) : state → state → Prop :=
  smallstep_def : ∀ p1 r p2 st1 st2,
    instrAt (instrs P) p1 (instr_of_rule r) p2 →
    smallstepAt P p1 r p2 st1 st2 →
    smallstep P st1 st2.
```

Deux états s_1 et s_2 sont ainsi en relation si et seulement si

- il existe une règle r associée à une instruction i
- i apparaît entre des points p_1 et p_2 du programme P
- s_1 et s_2 sont en relation vis à vis de $\to_{(p_1,r,p_2)}$

L'ensemble des états initiaux est caractérisé par les états mémoires dont le point de programme est égal au premier point du programme P.

```
Inductive InitState (P:program) : state → Prop :=
  initState_def : ∀ env : environment,
    InitState P <<begin_block (instrs P),env>>.
```

Cette définition Coq correspond à la définition mathématique $\mathcal{S}_0 = \{debut(P)\} \times \text{Env}$.

L'ensemble des états accessibles est enfin défini à partir de `InitState` et `smallstep`, de la même façon que dans l'exemple 3 du chapitre 3.

```
Inductive ReachableStates (P:program) : state → Prop :=
   reach_init : ∀ st, InitState P st → ReachableStates P st
 | reach_next : ∀ st1 st2,
     ReachableStates P st1 →
```

```
smallstep P st1 st2 →
ReachableStates P st2.
```

Nous modélisons ainsi que la sémantique d'un programme est l'ensemble de ses états accessibles à partir de $\mathcal{S}_0$.

$$[\![P]\!] = \{\ s \mid \exists s_0 \in \mathcal{S}_0,\ s_0 \rightarrow^* s\ \}$$

5.2 Abstraction des états

Nous allons maintenant proposer une sémantique abstraite pour le langage WHILE qui sera une approximation correcte de la sémantique concrète présentée précédemment. Le domaine sémantique laisse apparaître trois grands niveaux hiérarchiques : les états mémoires, les environnements de variables et les valeurs numériques. La conception de notre analyseur va suivre cette même hiérarchie en utilisant une approximation de chaque niveau sémantique. Nous avons choisi de présenter la structure de notre sémantique abstraite en partant du niveau sémantique le plus élevé, celui des états mémoires. Cette présentation verra apparaître progressivement la spécification requise sur l'abstraction des environnements de variables. Nous finirons par l'abstraction des valeurs numériques.

Pour définir une approximation correcte de $[\![P]\!]$, nous appliquons la méthode présentée lors du chapitre 3. Nous proposons dans un premier temps une spécification de la sémantique abstraite $[\![P]\!]^\sharp$ sous la forme d'un post-point fixe, puis nous calculons ce post-point fixe à l'aide des solveurs génériques présentés dans le chapitre 4.

5.2.1 Spécification de $[\![P]\!]^\sharp$

Choisissons tout d'abord l'abstraction que nous voulons réaliser au niveau des états mémoires. Nous utilisons la technique de partitionnement de la section 3.2.3.3 afin d'attacher des invariants mémoires à chaque point de programme. Notre abstraction est ainsi paramétrée par une abstraction sur le domaine sémantique des environnements de variables. Étant donnée une telle abstraction,

$$(\mathcal{P}(\mathrm{Env}), \subseteq, \cup, \cap) \xleftarrow{\gamma_{\mathrm{Env}}} \left(\mathrm{Env}^\sharp, \sqsubseteq^\sharp_{\mathrm{Env}}, \sqcup^\sharp_{\mathrm{Env}}, \sqcap^\sharp_{\mathrm{Env}}\right)$$

nous définissons une abstraction sur les états

$$(\mathcal{P}(\mathbb{P} \times \mathrm{Env}), \subseteq, \cup, \cap) \xleftarrow{\gamma_{\mathrm{Etat}}} \left(\mathrm{Etat}^\sharp, \sqsubseteq^\sharp_{\mathrm{Etat}}, \sqcup^\sharp_{\mathrm{Etat}}, \sqcap^\sharp_{\mathrm{Etat}}\right)$$

en prenant

$$\begin{array}{rcl}
\mathrm{Etat}^\sharp & \stackrel{\text{def}}{=} & \mathbb{P} \rightarrow \mathrm{Env}^\sharp \\
\sqsubseteq^\sharp_{\mathrm{Etat}} & \stackrel{\text{def}}{=} & \lambda s^\sharp_1 \lambda s^\sharp_2.\ \forall p \in \mathbb{P},\ s^\sharp_1(p) \sqsubseteq^\sharp_{\mathrm{Env}} s^\sharp_2(p) \\
\sqcup^\sharp_{\mathrm{Etat}} & \stackrel{\text{def}}{=} & \lambda s^\sharp_1 \lambda s^\sharp_2 \lambda p.\ s^\sharp_1(p) \sqcup^\sharp_{\mathrm{Env}} s^\sharp_2(p) \\
\sqcap^\sharp_{\mathrm{Etat}} & \stackrel{\text{def}}{=} & \lambda s^\sharp_1 \lambda s^\sharp_2 \lambda p.\ s^\sharp_1(p) \sqcap^\sharp_{\mathrm{Env}} s^\sharp_2(p) \\
\gamma_{\mathrm{Etat}}(s^\sharp) & \stackrel{\text{def}}{=} & \left\{\ \langle\!\langle p, \rho \rangle\!\rangle \mid \rho \in \gamma_{\mathrm{Env}}\left(s^\sharp(p)\right)\ \right\}
\end{array}$$

En Coq, le treillis des environnements de variables abstraits est déclaré par un module de type `LatticeSolve`. Cette signature, présentée dans la section 4.5, regroupe un treillis et un critère de terminaison pour les calculs itératifs de point fixe. La définition du treillis $\left(\text{Etat}^\sharp, \sqsubseteq^\sharp_{\text{Etat}}, \sqcup^\sharp_{\text{Etat}}, \sqcap^\sharp_{\text{Etat}}\right)$ est alors réalisée grâce au foncteur de module `ArrayBinSolve`, présenté lors du chapitre 6. Étant donné un module `S` de type `LatticeSolve`, `ArrayBinSolve(S)` et un module de type `LatticeSolve` pour la structure de treillis sur les fonctions de $\mathbb{P}$ vers le type `S.Lat.t` des éléments du treillis de `S`. Ces fonctions sont implémentées à l'aide des tableaux fonctionnels présentés lors du chapitre 2. `ArrayBinSolve(S)` conserve le même critère de terminaison que `S`.

Pour tout programme P, la sémantique abstraite $[\![P]\!]^\sharp$ doit être une approximation correcte de $[\![P]\!]$, c'est à dire :

$$[\![P]\!] \subseteq \gamma_{\text{Etat}}\left([\![P]\!]^\sharp\right)$$

La définition de $[\![P]\!]$ donnée dans la section précédente nous permet de caractériser $[\![P]\!]$ comme le plus petit post-point fixe de l'opérateur

$$F_P = \lambda S.\mathcal{S}_0 \cup \text{post}[\rightarrow_P](S) \in \mathcal{P}(\text{Etat}) \rightarrow \mathcal{P}(\text{Etat})$$

Grâce au théorème 3.3.1, nous pouvons alors spécifier $[\![P]\!]^\sharp$ comme un post-point fixe d'un opérateur $F^\sharp_P \in \text{Etat}^\sharp \rightarrow \text{Etat}^\sharp$, approximation correcte de F_P ($F_P \circ \gamma_{\text{Etat}} \dot{\subseteq} \gamma_{\text{Etat}} \circ F^\sharp_P$) :

$$[\![P]\!]^\sharp \in \left\{ s^\sharp \in \text{Etat}^\sharp \mid F^\sharp_P(s^\sharp) \sqsubseteq^\sharp_{\text{Etat}} s^\sharp \right\}$$

La spécification de $F^\sharp_P$ peut ensuite être découpée en exprimant F_P comme la composition de plusieurs opérateurs plus simples et en utilisant le lemme 3.2 sur la composition des approximations correctes. En effet,

$$\begin{aligned} F_P &= \lambda S.\mathcal{S}_0 \cup \text{post}[\rightarrow_P](S) \\ &= \lambda S.\mathcal{S}_0 \cup \text{post}\left[\bigcup_{j \in I_j} \rightarrow_j\right](S) \\ &= \lambda S.\mathcal{S}_0 \cup \bigcup_{j \in I_j} \text{post}[\rightarrow_j](S) \end{aligned}$$

car l'opérateur post est un morphisme d'union. On peut ainsi chercher un opérateur $F^\sharp_P$ de la forme

$$F^\sharp_P = \lambda s^\sharp.s^\sharp_0 \sqcup^\sharp_{\text{Etat}} \bigsqcup^\sharp_{j \in I_P \text{ Etat}} F^\sharp_j(s^\sharp)$$

avec $s^\sharp_0$ une approximation correcte de l'ensemble des états initiaux ($\mathcal{S}_0 \subseteq \gamma_{\text{Etat}}(s^\sharp_0)$) et chaque fonction $F^\sharp_j$, une approximation correcte de l'opérateur post$[\rightarrow_j]$ (pour tout indice j de I_P, post$[\rightarrow_j]\left(\gamma_{\text{Etat}}(s^\sharp)\right) \sqsubseteq^\sharp_{\text{Etat}} \gamma_{\text{Etat}}(F^\sharp_j(s^\sharp))$).

Puisque $\mathcal{S}_0$ est de la forme $\{debut(P)\} \times \text{Env}$, nous pouvons prendre $s_0^\sharp$ égal à $\bot_{\text{Etat}}[debut(P) \mapsto \text{init\_env}_P]$ avec $\text{init\_env}_P \in \text{Env}^\sharp$ une approximation correcte de $\text{Env} \in \mathcal{P}(\text{Env})$.

Chaque opérateur $F_j^\sharp$ peut être pris de la forme

$$F_j^\sharp = \lambda s^\sharp.\bot_{\text{Etat}}[p_j^{\text{out}} \mapsto \text{post}_j^\sharp(s^\sharp(p_j^{\text{in}}))]$$

avec p_j^{out}, p_j^{in} et $\text{post}_j^\sharp$ définies par

j	p_j^{out}	p_j^{in}	$\text{post}_j^\sharp$
$(p_1, x := e, p_2, 1)$	p_1	p_2	$[\![x := e]\!]^\sharp_{\text{affect}}$
$(p_1, \textbf{if } t \{ b_1 \} \{ b_2 \}, p_2, 1)$	p_1	$debut(b_1)$	$[\![t]\!]\downarrow^\sharp$
$(p_1, \textbf{if } t \{ b_1 \} \{ b_2 \}, p_2, 2)$	p_1	$debut(b_2)$	$[\![\bar{t}]\!]\downarrow^\sharp$
$(p_1, \textbf{if } t \{ b_1 \} \{ b_2 \}, p_2, 3)$	$fin(b_1)$	p_2	$\lambda p^\sharp.p^\sharp$
$(p_1, \textbf{if } t \{ b_1 \} \{ b_2 \}, p_2, 4)$	$fin(b_2)$	p_2	$\lambda p^\sharp.p^\sharp$
$(p_1, \textbf{while } t \{ b \}, p_2, 1)$	p_1	$debut(b)$	$[\![t]\!]\downarrow^\sharp$
$(p_1, \textbf{while } t \{ b \}, p_2, 2)$	p_1	p_2	$[\![\bar{t}]\!]\downarrow^\sharp$
$(p_1, \textbf{while } t \{ b \}, p_2, 3)$	$fin(b)$	p_1	$\lambda p^\sharp.p^\sharp$

Ces définitions font apparaître trois nouveaux opérateurs.

- $\bar{t}$ désigne ici la négation du test t, définie par induction sur la structure de t. La correction de cet opérateur s'exprime par

  $$\forall t \in test, \{ \neg b \mid b \in [\![t]\!]\uparrow_{\text{test}} \} \subseteq [\![\bar{t}]\!]\uparrow_{\text{test}}$$

  L'inclusion est suffisante pour prouver la correction de la sémantique abstraite, mais l'égalité ensembliste est facilement établie.
- $[\![x := e]\!]^\sharp_{\text{affect}} \in \text{Env}^\sharp \to \text{Env}^\sharp$ calcule l'environnement abstrait obtenu après l'affectation d'une expression e à une variable x. Un tel opérateur doit donc vérifier

  $$\forall x \in \mathbb{V}, \forall e \in expr, \forall \rho^\sharp \in \text{Env}^\sharp$$
  $$\left\{ \rho[x \mapsto v] \mid \rho \in \gamma_{\text{Env}}(\rho^\sharp) \text{ et } v \in [\![e]\!]\uparrow^\rho_{\text{expr}} \right\} \subseteq \gamma_{\text{Env}}\left([\![x := e]\!]^\sharp_{\text{affect}}(\rho^\sharp)\right)$$

- $[\![t]\!]\downarrow^\sharp \in \text{Env}^\sharp \to \text{Env}^\sharp$ calcule une approximation des environnements vérifiant un test t.

  $$\forall t \in test, \forall \rho^\sharp \in \text{Env}^\sharp$$
  $$\left\{ \rho \mid \rho \in \gamma_{\text{Env}}(\rho^\sharp) \text{ et } T \in [\![t]\!]\uparrow^\rho_{\text{test}} \right\} \subseteq \gamma_{\text{Env}}\left([\![t]\!]\downarrow^\sharp_{\text{test}}(\rho^\sharp)\right)$$

  Un tel opérateur permettra de raffiner les invariants mémoires qui succèdent à un test.

Le premier opérateur est indépendant de l'abstraction considérée sur les environnements de variables. Les deux autres viennent s'ajouter aux propriétés qui doivent être vérifiées par l'abstraction des environnements de variables. Toutes ces propriétés sont résumées dans une interface de module Coq appelée `EnvAbstraction` et présentée dans la figure 5.11. Une abstraction d'environnement est ainsi spécifiée comme un module comportant

- un module `S` de type `LatticeSolve`.
- une fonction de concrétisation `gamma`, ainsi que les deux propriétés assurant l'hypothèse de morphisme d'intersection.
- les opérateurs `affect` ($[\![x := e]\!]^\sharp_{\text{affect}}$), `back_test` ($[\![t]\!]\downarrow^\sharp_{\text{test}}$) et `init_env` (init_env) présentés précédemment, avec leurs preuves de correction. Des propriétés de monotonie sont de plus requises sur `back_test` et `affect`. Elle sont nécessaires pour assurer la monotonie globale de $F^\sharp_P$. Cette monotonie ne sera utilisée que lorsque le critère de terminaison des itérations de point fixe sera soit une condition de chaîne ascendante, soit une utilisation d'opérateurs d'élargissement/rétrécissement standards. Ceci explique l'hypothèse sur `S.termination_flag` dans ces deux énoncés.

5.2.2 Calcul de $[\![P]\!]^\sharp$

Le calcul final de $[\![P]\!]^\sharp$ demande de réunir toutes les fonctions F_j pour $j \in I_P$. Ce calcul est effectué à l'aide de deux fonctions mutuellement récursives sur les blocs et les instructions, qui énumèrent ainsi tous les éléments de I_P.

Plusieurs techniques sont ensuite possibles pour le calcul de $[\![P]\!]^\sharp$.

- La méthode la plus simple consiste à seulement définir un vérificateur de post-points fixes qui prend comme arguments un programme et un état abstrait, et renvoie un booléen égal à vrai si l'état abstrait est un post-point fixe de $F^\sharp_P$.

```
Lemma check : ∀ (P:program) (hint:S.Lat.t),
 { (ReachableStates P)<=<(gamma P hint) }+{True}.
```

  L'opérateur infixe `<=<` dénote ici l'inclusion de propriété logique. `P <=< Q` est ainsi équivalent à `∀ x, P x → Q x`. Il s'agit du codage standard de l'inclusion ensembliste en théorie des types. Nous utilisons une nouvelle fois les booléens riches de Coq. Si `check` renvoie vraie, alors l'état abstrait `hint` est une approximation correcte de $[\![P]\!]$. Sinon on ne peut rien conclure.

- La deuxième méthode consiste à calculer $F^\sharp_P$ puis à chercher (ou à approcher) son plus petit post-point fixe. Nous utilisons pour cela le solveur de post-point fixe présenté dans le chapitre précédent. Pour certains critères de terminaison du treillis des états abstraits, une preuve de la monotonie de $F^\sharp_P$ sera nécessaire.

```
Lemma solve : ∀ P,
  { St:S.Lat.t | (ReachableStates P) <=< (gamma P St) }.
```

- Si $F^\sharp_P$ est monotone, nous pouvons chercher un post-point fixe $s^\sharp$ du système

$$\begin{cases} s^\sharp_0 \sqsubseteq^\sharp_{\text{Etat}} s^\sharp \\ F^\sharp_j(s^\sharp) \sqsubseteq^\sharp_{\text{Etat}} s^\sharp, \ \forall j \in I_P \end{cases}$$

```
Module Type EnvAbstraction.

  Declare Module S : LatticeSolve.

  Parameter gamma : program → S.Lat.t → environment → Prop.
  Parameter gamma_monotone : ∀ P E1 E2 env,
    S.Lat.order E1 E2 →
    gamma P E1 env →
    gamma P E2 env.
  Parameter gamma_meet_morph : ∀ P E1 E2 env,
    gamma P E1 env → gamma P E2 env →
    gamma P (S.Lat.meet E1 E2) env.

  Parameter affect : program → S.Lat.t → VarName → expr → S.Lat.t.
  Parameter affect_correct : ∀ P Env env env' x n e,
    gamma P Env env →
    sem_expr P env e n →
    subst env x n env' →
    gamma P (affect P Env x e) env'.
  Parameter affect_monotone :
    S.termination_flag <> ACCELERATING_NOT_MONOTONE →
    ∀ P E1 E2 x e,
    S.Lat.order E1 E2 →
    S.Lat.order (affect P E1 x e) (affect P E2 x e).

  Parameter init_env : program → S.Lat.t.
  Parameter init_env_correct : ∀ P (env:environment),
    gamma P (init_env P) env.

  Parameter back_test : program → test → S.Lat.t → S.Lat.t.
  Parameter back_test_correct : ∀ P t Env env,
    gamma P Env env →
    sem_test P env t true →
    gamma P (back_test P t Env) env.
  Parameter back_test_monotone :
    S.termination_flag <> ACCELERATING_NOT_MONOTONE →
    ∀ P t E1 E2,
    S.Lat.order E1 E2 →
    S.Lat.order (back_test P t E1) (back_test P t E2).

End EnvAbstraction.
```

FIG. 5.11 – Signature abstraite pour l'abstraction des environnements

en utilisant les opérateurs `pfp_list` des sections 4.2.3.3 et 4.3.4.2. Ces opérateurs permettent de calculer un post-point fixe commun d'une liste de fonctions, en itérant chaque fonction par alternance. Le type Coq du programme correspondant est le même que précédemment, mais le calcul itératif est plus rapide.

– La dernière méthode utilise l'itération chaotique de la section 4.4.2. L'ensemble des triplets $\left((p_j^{out}, p_j^{in}, post_j^\sharp)\right)_{j\in I_P}$ doit pour cela être collecté de manière à ce que les hypothèses du théorème 4.4.2 soient vérifiées. Ce dernier théorème n'est cependant pas prouvé en Coq, et ainsi aucune preuve formelle n'est nécessaire pour instancier le solveur. En contrepartie, le résultat final doit être vérifié par la fonction `check` précédente. Notre preuve « papier » du théorème 4.4.2 nous assure néanmoins que le résultat sera toujours accepté par `check`.

Au final, nous disposons d'un analyseur certifié paramétré par une abstraction correcte des environnements de variables.

```
Module StateLift (AbEnv:EnvAbstraction).

...

Lemma solve : ∀ P,
   { St:S.Lat.t | (ReachableStates P)<=<(gamma P St) }.

End StateLift.
```

Comme nous l'avions annoncé, seule la correction de l'analyse est prouvée formellement. La précision de l'analyse pourra être évaluée expérimentalement avec les exemples de la section 5.5.

5.3 Abstraction des environnements

Nous allons maintenant présenter deux implémentations possibles et similaires de la signature de module `EnvAbstraction`, la seconde étant une optimisation de la première basée sur la technique *d'itération réductive* proposée dans [Cou99]. Ces deux implémentations seront paramétrées par une abstraction des valeurs numériques, dont nous verrons au fur et à mesure les hypothèses nécessaires.

L'abstraction des environnements de variables que nous avons choisie suit la structure des environnements : un environnement abstrait est une fonction entre noms de variables et valeurs numériques abstraites. Étant donnée une abstraction numérique

$$(\mathcal{P}(\mathbb{Z}), \subseteq, \cup, \cap) \xleftarrow{\gamma_{Num}} \left(Num^\sharp, \sqsubseteq^\sharp_{Num}, \sqcup^\sharp_{Num}, \sqcap^\sharp_{Num}\right)$$

nous prenons le treillis standard des fonctions de $\mathbb{V} \to Num^\sharp$ muni de l'ordre point à point. La fonction de concrétisation entre $\mathbb{V} \to Num^\sharp$ et $\mathcal{P}(Env)$ est définie par

$$\gamma_{Env}(\rho^\sharp) = \left\{\ \rho \mid \forall x \in Var_P,\ \rho(x) \in \gamma_{Num}(\rho^\sharp(x))\ \right\}$$

Cette définition ne parle que des variables déclarées dans la syntaxe du programme. Comme nous l'avons déjà fait remarquer, seules ces variables sont utilisées au cours de l'exécution du programme.

La construction du treillis abstrait est une nouvelle fois basée sur l'utilisation du foncteur de module `ArrayBinSolve`.

5.3.1 Construction de $[\![x := e]\!]^\sharp_{\text{affect}}$

La fonction $[\![x := e]\!]^\sharp_{\text{affect}}$ peut être définie par

$$[\![x := e]\!]^\sharp_{\text{affect}}(\rho^\sharp) = \rho^\sharp[x \mapsto [\![e]\!]\uparrow^\sharp_{\text{expr}}],\ \forall \rho^\sharp \in \text{Env}^\sharp$$

avec $[\![e]\!]\uparrow^\sharp_{\text{expr}} \in \text{Env}^\sharp \to \text{Num}^\sharp$ une fonction qui calcule une approximation de l'évaluation d'une expression arithmétique. Cette fonction doit ainsi vérifier :

$$\left\{ v \mid \exists \rho \in \gamma_{\text{Env}}(\rho^\sharp),\ v \in [\![e]\!]\uparrow^\rho_{\text{expr}} \right\} \subseteq \gamma_{\text{Num}}\left([\![e]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp)\right)\ \forall e \in \mathit{expr}, \forall \rho^\sharp \in \text{Env}^\sharp$$

Elle est définie par induction sur la structure de l'expression e.

$$\begin{aligned}
[\![n]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) &= \text{const}^\sharp(n) \\
[\![?]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) &= \top_{\text{Num}} \\
[\![x]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) &= \rho^\sharp(x) \\
[\![-e]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) &= [\![-]\!]\uparrow^\sharp_{\text{op}}([\![e]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp)) \\
[\![e_1 \odot e_2]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) &= [\![\odot]\!]\uparrow^\sharp_{\text{op}}\left([\![e_1]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp)\right)
\end{aligned}$$

Plusieurs opérateurs et constantes de l'abstraction numérique sont pour cela nécessaires :

- $\text{const}^\sharp$ calcule une approximation des constantes.

$$\forall n \in \mathbb{Z},\ \{n\} \subseteq \gamma_{\text{Num}}(\text{const}^\sharp(n))$$

- $\top_{\text{Num}}$ approche n'importe quelle valeur numérique.

$$\mathbb{Z} \subseteq \gamma_{\text{Num}}(\top_{\text{Num}})$$

- $[\![-]\!]\uparrow^\sharp_{\text{op}}$ est une approximation correcte de l'opposé d'un entier relatif.

$$\forall n^\sharp \in \text{Num}^\sharp,\ \{\ -n \mid n \in \gamma_{\text{Num}}(n^\sharp)\ \} \subseteq \gamma_{\text{Num}}([\![-]\!]\uparrow^\sharp_{\text{op}}(n^\sharp))$$

- $[\![\odot]\!]\uparrow_{\text{op}}$ est une approximation de l'opérateur arithmétique $\odot \in \{+, -, \times\}$.

$$\begin{aligned}
&\forall \odot \in \{+, -, \times\}, \forall n_1^\sharp, n_2^\sharp \in \text{Num}^\sharp, \\
&\quad \{\ n_1 [\![\odot]\!]\uparrow_{\text{op}} n_2 \mid n_1 \in \gamma_{\text{Num}}(n_1^\sharp),\ n_2 \in \gamma_{\text{Num}}(n_2^\sharp)\ \} \subseteq \gamma_{\text{Num}}([\![\odot]\!]\uparrow^\sharp_{\text{op}}(n_1^\sharp, n_2^\sharp))
\end{aligned}$$

5.3.2 Construction de $[\![t]\!]\downarrow^\sharp_{test}$

Le deuxième opérateur requis par l'interface est $[\![t]\!]\downarrow^\sharp_{test}$. Afin de simplifier sa définition, il est utile de normaliser les tests. Nous définissons ainsi un nouveau type de test :

$$\begin{array}{rcll} \overline{test} ::= & expr\ c\ expr & & c \in \{<, =\} \\ | & test\ \overline{\textbf{and}}\ expr & & \\ | & test\ \overline{\textbf{or}}\ expr & & \end{array}$$

Une fonction $normalise \in test \to \overline{test}$ est ensuite définie pour transformer chaque test $t \in test$ en une version équivalente dans $\overline{test}$. La correction de *normalise* est spécifiée par

$$\forall t \in test,\ [\![t]\!]\uparrow_{test} \subseteq [\![normalise(t)]\!]\uparrow_{\overline{test}}$$

$[\![\cdot]\!]\uparrow_{\overline{test}}$ désigne ici la version de $[\![\cdot]\!]\uparrow_{test}$ restreinte à $\overline{test}$. Encore une fois, l'égalité logique est démontrable mais pas nécessaire. Grâce à cette normalisation, les opérateurs manipulant des tests peuvent être uniquement définis pour les constructions $<$, $=$, **and** et **or**. L'opérateur $[\![t]\!]\downarrow^\sharp_{test}$ est ainsi défini en composant *normalise* avec une version de $[\![t]\!]\downarrow^\sharp_{test}$ spécialisée pour les tests de $\overline{test}$.

$$[\![t]\!]\downarrow^\sharp_{test} \stackrel{\text{def}}{=} [\![normalise(t)]\!]\downarrow^\sharp_{\overline{test}},\ \forall t \in test$$

$[\![t]\!]\downarrow^\sharp_{\overline{test}}$ est ensuite définie par induction sur t à partir de trois nouveaux opérateurs $\text{reduc}^\sharp \in \text{Env}^\sharp \to \text{Env}^\sharp$, $[\![c]\!]\downarrow^\sharp_{comp} \in \left(\text{Num}^\sharp \times \text{Num}^\sharp\right) \to \left(\text{Num}^\sharp \times \text{Num}^\sharp\right)$ et $[\![e]\!]\downarrow^\sharp_{expr} \in \left(\text{Env}^\sharp \times \text{Num}^\sharp\right) \to \text{Env}^\sharp$.

$$\begin{array}{rcl} [\![e_1\ c\ e_2]\!]\downarrow^\sharp_{\overline{test}}(\rho^\sharp) & = & \text{reduc}^\sharp\left([\![e_1]\!]\downarrow^\sharp_{expr}(\rho^\sharp, n_1^\sharp) \sqcap^\sharp_{Env} [\![e_2]\!]\downarrow^\sharp_{expr}(\rho^\sharp, n_2^\sharp)\right) \\ & & \text{avec } (n_1^\sharp, n_2^\sharp) = [\![c]\!]\downarrow^\sharp_{comp}([\![e_1]\!]\uparrow^\sharp_{expr}(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp_{expr}(\rho^\sharp)) \\ [\![t_1\ \textbf{or}\ t_2]\!]\downarrow^\sharp_{\overline{test}}(\rho^\sharp) & = & [\![t_1]\!]\downarrow^\sharp_{\overline{test}} \sqcup^\sharp_{Env} [\![t_2]\!]\downarrow^\sharp_{\overline{test}} \\ [\![t_1\ \textbf{and}\ t_2]\!]\downarrow^\sharp_{\overline{test}}(\rho^\sharp) & = & \text{reduc}^\sharp\left([\![t_1]\!]\downarrow^\sharp_{\overline{test}} \sqcap^\sharp_{Env} [\![t_2]\!]\downarrow^\sharp_{\overline{test}}\right) \end{array}$$

$[\![c]\!]\downarrow^\sharp_{comp}(n_1^\sharp, n_2^\sharp)$ calcule un raffinement de deux valeurs numériques abstraites, sachant qu'elles vérifient une condition c. Plus formellement :

$$\left\{(n_1, n_2) \mid n_1 \in \gamma_{Num}(n_1^\sharp),\ n_2 \in \gamma_{Num}(n_2^\sharp),\ n_1 [\![c]\!]\uparrow_{comp} n_2\right\} \subseteq \gamma_{Num}(m_1^\sharp) \times \gamma_{Num}(m_2^\sharp)$$
$$\text{avec } (m_1^\sharp, m_2^\sharp) = [\![c]\!]\downarrow^\sharp_{comp}(n_1^\sharp, n_2^\sharp)$$

$\text{reduc}^\sharp$ est un opérateur de réduction (voir section 3.2.3.1). En effet, lorsque $\gamma_{Num}(\bot_{Num})$ est égal à $\emptyset$, plusieurs éléments de $\text{Env}^\sharp$ représentent la même propriété sur $\mathcal{P}(\text{Env})$. Il s'agit des fonctions pour lesquelles au moins une des images est égale à $\bot_{Num}$.

$$\gamma_{Num}(\bot_{Num}) = \emptyset \ \Rightarrow\ \forall \rho^\sharp \in \text{Env}^\sharp,\ \left(\exists x \in \text{Var}_P,\ \rho^\sharp(x) = \bot_{Num}\right) \Rightarrow \gamma_{Env}(\rho^\sharp) = \emptyset$$

L'opérateur $\text{reduc}^\sharp$ permet de détecter ce type d'environnements abstraits et de les remplacer par $\bot_{\text{Env}}$.

$$\text{reduc}^\sharp(\rho^\sharp) = \begin{cases} \bot_{\text{Env}} & \text{si } \exists x \in \text{Var}_P,\ \rho^\sharp(x) = \bot_{\text{Num}} \\ \rho^\sharp & \text{sinon} \end{cases}$$

Pour montrer la correction de l'analyse, seul l'aspect conservatif de $\text{reduc}^\sharp$ vis-à-vis de γ_{Env} doit être prouvé en Coq.

$$\forall \rho^\sharp,\ \gamma_{\text{Env}}(\rho^\sharp) \subseteq \gamma_{\text{Env}}(\text{reduc}^\sharp(\rho^\sharp))$$

$\text{reduc}^\sharp$ est en fait une fermeture inférieure. Cette propriété n'est pas critique pour la correction de l'abstraction, mais seulement pour sa précision. Si nous suivions scrupuleusement ce qui a été présenté lors de la section 3.2.3.1, nous devrions remplacer chaque opérateur abstrait $f^\sharp \in \text{Env}^\sharp \to \text{Env}^\sharp$ par $\text{reduc}^\sharp \circ f^\sharp \circ \text{reduc}^\sharp$. Cependant, nous pouvons économiser certaines occurrences de $\text{reduc}^\sharp$ car les fonctions abstraites seront toujours utilisées sur des éléments du domaine $\text{reduc}^\sharp(\text{Env}^\sharp)$. Certains opérateurs (comme $\sqcup^\sharp$) peuvent même être restreints à ce domaine ($f^\sharp\left(\text{reduc}^\sharp(\text{Env}^\sharp)\right) \subseteq \text{reduc}^\sharp(\text{Env}^\sharp)$). Nous pouvons dans ce cas nous dispenser d'utiliser $\text{reduc}^\sharp$. En ce qui concerne le développement de preuve en Coq, un tel opérateur est très facile à utiliser puisque nous devons juste prouver sa correction vis à vis de γ_{Env}. Nous gagnons ainsi en précision pour notre analyse, avec un faible sur-coût au niveau preuve. Il aurait en effet été beaucoup plus lourd de raffiner chaque opérateur abstrait afin de le forcer à avoir une image dans $\text{reduc}^\sharp(\text{Env}^\sharp)$.

Le dernier opérateur utilisé pour la définition de $[\![t]\!]\downarrow^\sharp_{\text{test}}$ est

$$[\![e]\!]\downarrow^\sharp_{\text{expr}} \in \left(\text{Env}^\sharp \times \text{Num}^\sharp\right) \to \text{Env}^\sharp$$

$[\![e]\!]\downarrow^\sharp_{\text{expr}}\ (\rho^\sharp, n^\sharp)$ calcule un raffinement de l'environnement abstrait $\rho^\sharp$, sachant que l'expression e s'évalue par la valeur numérique abstraite $n^\sharp$ dans cet environnement. $[\![e]\!]\downarrow^\sharp_{\text{expr}}$ est définie par induction sur e :

$$\begin{aligned}
[\![n]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) &= \begin{cases} \bot_{\text{Env}} & \text{si } \text{const}^\sharp(n) \sqcap^\sharp_{\text{Num}} n^\sharp = \bot_{\text{Num}} \\ \rho^\sharp & \text{sinon} \end{cases} \\
[\![?]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) &= \rho^\sharp \\
[\![x]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) &= \text{reduc}^\sharp(\rho^\sharp[x \mapsto \rho^\sharp(x) \sqcap^\sharp_{\text{Num}} n^\sharp]) \\
[\![-e]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) &= [\![e]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, [\![-]\!]\uparrow^\sharp_{\text{op}}(n^\sharp)) \\
[\![e_1 \odot e_2]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) &= \text{reduc}^\sharp\left([\![e_1]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n_1^\sharp) \sqcap^\sharp_{\text{Env}} [\![e_2]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n_2^\sharp)\right) \\
&\quad \text{avec } (n_1^\sharp, n_2^\sharp) = [\![\odot]\!]\downarrow^\sharp_{\text{op}}(n^\sharp, [\![e_1]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp))
\end{aligned}$$

Un dernier opérateur numérique est pour cela nécessaire :

$$[\![\odot]\!]\downarrow^\sharp_{\text{op}} \in \left(\text{Num}^\sharp \times \text{Num}^\sharp \times \text{Num}^\sharp\right) \to \left(\text{Num}^\sharp \times \text{Num}^\sharp\right)$$

$[\![\odot]\!]\downarrow^\sharp_{\text{op}}(n^\sharp, n_1^\sharp, n_2^\sharp)$ calcule un raffinement de deux valeurs numériques $n_1^\sharp$ et $n_2^\sharp$ sachant que le résultat de l'opération binaire $\odot$ vaut $n^\sharp$ sur elles.

$$\begin{array}{l}\forall n^\sharp, n_1^\sharp, n_2^\sharp \in \text{Num}^\sharp,\\ \quad \left\{ (n_1, n_2) \mid n_1 \in \gamma_{\text{Num}}(n_1^\sharp),\ n_2 \in \gamma_{\text{Num}}(n_2^\sharp),\ (n_1 [\![\odot]\!]\uparrow_{\text{op}} n_2) \in \gamma_{\text{Num}}(n^\sharp) \right\}\\ \qquad \subseteq \gamma_{\text{Num}}(m_1^\sharp) \times \gamma_{\text{Num}}(m_2^\sharp)\\ \qquad\qquad\qquad\qquad \text{avec } (m_1^\sharp, m_2^\sharp) = [\![\odot]\!]\downarrow^\sharp_{\text{op}}(n^\sharp, n_1^\sharp, n_2^\sharp)\end{array}$$

La correction de $[\![e]\!]\downarrow^\sharp_{\text{expr}}$ est formalisée par

$$\begin{array}{l}\forall \rho^\sharp \in \text{Env}^\sharp, n^\sharp \in \text{Num}^\sharp,\\ \quad \left\{ \rho \in \gamma_{\text{Env}}(\rho^\sharp) \mid \exists n \in [\![e]\!]\uparrow^\rho_{\text{expr}} \cap \gamma_{\text{Num}}(n^\sharp) \right\} \subseteq \gamma_{\text{Env}}([\![e]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp)\end{array}$$

5.3.3 Signature des abstractions numériques

La construction des opérateurs précédents a fait apparaître plusieurs pré-requis sur l'abstraction numérique. Nous en déduisons l'interface Coq pour les modules des abstractions numériques, présentée dans la figure 5.12.

Au final, les opérateurs abstraits sur les environnements sont regroupés dans un foncteur de type

```
Module EnvNotRelational (AN : NumAbstraction) : EnvAbstraction.
```

c'est à dire, un foncteur prenant un module de type `NumAbstraction` en argument et retournant un module de type `EnvAbstraction` en résultat.

5.3.4 Raffinement des tests par itération réductive

Nous proposons une deuxième implémentation de la signature `EnvAbstraction`. Le treillis des environnements abstraits considéré est toujours le même, mais nous ajoutons un calcul de réduction itérative pour affiner l'information calculée sur les tests. Cette technique a été introduite par Philippe Granger dans [Gra92], et est présentée dans l'analyseur de Patrick Cousot [Cou99]. Nous la reformulons ici, en nous concentrons une nouvelle fois sur la correction sémantique de l'approche.

L'idée consiste à améliorer la précision de l'opérateur $[\![t]\!]\downarrow_{\overline{\text{test}}}$ défini précédemment. Si nous mettons la propriété vérifiée par $[\![t]\!]\downarrow_{\overline{\text{test}}}$ sous la forme suivante

$$\begin{array}{l}\forall \rho \in \text{Env}, \forall t \in \mathit{test}, T \in [\![t]\!]\downarrow^\rho_{\text{test}} \Rightarrow\\ \quad \forall \rho^\sharp \in \text{Env}^\sharp, \rho \in \gamma_{\text{Env}}(\rho^\sharp) \Rightarrow \rho \in \gamma_{\text{Env}}([\![t]\!]\downarrow^\sharp_{\overline{\text{test}}}(\rho^\sharp))\end{array}$$

il est alors facile de démontrer que

$$\begin{array}{l}\forall n \in \mathbb{N}, \forall \rho \in \text{Env}, \forall t \in \mathit{test}, T \in [\![t]\!]\downarrow^\rho_{\text{test}} \Rightarrow\\ \quad \forall \rho^\sharp \in \text{Env}^\sharp, \rho \in \gamma_{\text{Env}}(\rho^\sharp) \Rightarrow \rho \in \gamma_{\text{Env}}([\![t]\!]\downarrow^{\sharp n}_{\overline{\text{test}}}(\rho^\sharp))\end{array}$$

```
Module Type NumAbstraction.

  Declare Module S : LatticeSolve.

  Parameter gamma : S.Lat.t → Z → Prop.
  Parameter gamma_monotone : ∀ N1 N2 n,
    S.Lat.order N1 N2 → gamma N1 n → gamma N2 n.
  Parameter gamma_meet_morph : ∀ N1 N2 n,
    gamma N1 n → gamma N2 n → gamma (S.Lat.meet N1 N2) n.

  Parameter BackTest : comp' → S.Lat.t → S.Lat.t → S.Lat.t*S.Lat.t.
  Parameter BackTest_correct : ∀ c N1 N2 N1' N2' n1 n2,
    sem_comp' c n1 n2 → gamma N1 n1 → gamma N2 n2 →
    BackTest c N1 N2 = (N1',N2') → gamma N1' n1 ∧ gamma N2' n2.
  Parameter BackTest_monotone_fst :
    S.termination_flag <> ACCELERATING_NOT_MONOTONE →
    ∀ c N1 N2 N1' N2', S.Lat.order N1 N1' → S.Lat.order N2 N2' →
    S.Lat.order (fst (BackTest c N1 N2)) (fst (BackTest c N1' N2')).
  Parameter BackTest_monotone_snd : ...

  Parameter Minus : S.Lat.t → S.Lat.t.
  Parameter Minus_correct : ∀ N n, gamma N n → gamma (Minus N) (-n).
  Parameter Minus_monotone :
    S.termination_flag <> ACCELERATING_NOT_MONOTONE →
    ∀ N1 N2, S.Lat.order N1 N2 → S.Lat.order (Minus N1) (Minus N2).

  Parameter SemOp : op → S.Lat.t → S.Lat.t → S.Lat.t.
  Parameter SemOp_correct : ∀ o N1 N2 n1 n2 n,
    gamma N1 n1 → gamma N2 n2 →
    sem_op o n1 n2 n → gamma (SemOp o N1 N2) n.
  Parameter SemOp_monotone :
     S.termination_flag <> ACCELERATING_NOT_MONOTONE →
    ∀ o N1 N2 N1' N2', S.Lat.order N1 N1' → S.Lat.order N2 N2' →
    S.Lat.order (SemOp o N1 N2) (SemOp o N1' N2').

  Parameter BackSemOp :
               op → S.Lat.t → S.Lat.t → S.Lat.t → S.Lat.t*S.Lat.t.
  Parameter BackSemOp_correct : ∀ o N N1 N2 n1 n2 n N1' N2',
    gamma N1 n1 → gamma N2 n2 → sem_op o n1 n2 n → gamma N n →
    BackSemOp o N N1 N2 = (N1',N2') → gamma N1' n1 ∧ gamma N2' n2.
  Parameter BackSemOp_monotone_fst :
    S.termination_flag <> ACCELERATING_NOT_MONOTONE →
    ∀ o N1 N2 N1' N2' N N', S.Lat.order N1 N1' →
    S.Lat.order N2 N2' →  S.Lat.order N N' →
    S.Lat.order (fst (BackSemOp o N N1 N2))
                (fst (BackSemOp o N' N1' N2')).
  Parameter BackSemOp_monotone_snd : ...

  Parameter const : Z → S.Lat.t.
  Parameter const_correct : ∀ n, gamma (const n) n.

  Parameter top : S.Lat.t.
  Parameter top_correct : ∀ n, gamma top n.

  Parameter bottom_empty : ∀ n, ¬ gamma S.Lat.bottom n.

End NumAbstraction.
```

FIG. 5.12 – Signature abstraite pour l'abstraction des valeurs numériques

Toute itérée de $[\![t]\!]\downarrow^{\sharp}_{test}$ fournit ainsi une autre implémentation correcte de $[\![t]\!]\downarrow^{\sharp}_{test}$. En pratique, $[\![t]\!]\downarrow^{\sharp}_{test}$ doit être réductive pour être utile, en terme de précision. Dans ce cas, $[\![t]\!]\downarrow^{\sharp\mathbf{n}}_{test}$ est au moins aussi précise que $[\![t]\!]\downarrow^{\sharp}_{test}$:

$$\forall t \in \mathit{test}, \forall \rho^{\sharp} \in \mathrm{Env}^{\sharp}$$
$$\{\ \rho \mid \rho \in \gamma_{\mathrm{Env}}(\rho^{\sharp}) \text{ et } T \in [\![t]\!]\uparrow^{\rho}_{test}\ \}$$
$$\subseteq \cdots \subseteq \gamma_{\mathrm{Env}}\left([\![t]\!]\downarrow^{\sharp\mathbf{n}}_{test}(\rho^{\sharp})\right) \subseteq \cdots \subseteq \gamma_{\mathrm{Env}}\left([\![t]\!]\downarrow^{\sharp}_{test}(\rho^{\sharp})\right)$$

En pratique, pour une valeur initiale $\rho^{\sharp}$, nous pouvons donc améliorer le résultat $[\![t]\!]\downarrow^{\sharp}_{test}(\rho^{\sharp})$ en itérant $[\![t]\!]\downarrow^{\sharp}_{test}$. L'itération peut être arrêtée dès qu'un point fixe est atteint, puisque plus aucune précision n'est alors a attendre des itérées suivantes. Si le treillis ne vérifie pas la condition de chaîne descendante, l'itération peut décroître indéfiniment. Un opérateur de rétrécissement peut alors être utilisé pour assurer la convergence. Dans notre implémentation, nous nous sommes contentés de borner le nombre d'itérations par une constante, car à tout instant de l'itération, le résultat courant est une approximation correcte.

Nous construisons ainsi un deuxième foncteur

```
Module EnvNotRelationalWithGuardReduction
        (AN : NumAbstraction) : EnvAbstraction.
```

qui permet de construire une abstraction correcte des environnements de variable, à partir d'une abstraction numérique correcte. Les opérateurs abstraits de ce module sont au moins aussi précis que ceux proposés dans le module `EnvNotRelational`, mais les itérations réductives effectuées peuvent les rendre plus coûteuses en terme de calcul. La section 5.5 présentera une illustration du gain de précision apporté.

5.4 Abstraction numérique

Il nous reste maintenant à proposer des implémentations pour la signature de module `NumAbstraction` des abstractions numériques.

5.4.1 Abstraction par signe

La première implémentation que nous proposons abstrait les valeurs numériques par leurs signes [CC76]. Le diagramme de Hasse du treillis abstrait est présenté dans la figure 5.13. La fonction de concrétisation $\gamma_{\mathrm{Num}} : \mathrm{Num}^{\sharp} \to \mathcal{P}(\mathbb{Z})$ associée à l'abstraction

est définie par

$$\begin{aligned}
\gamma_{\mathrm{Num}}(\bot) &= \emptyset \\
\gamma_{\mathrm{Num}}(-) &= \{\, z \mid z < 0 \} \\
\gamma_{\mathrm{Num}}(0) &= \{0\} \\
\gamma_{\mathrm{Num}}(+) &= \{\, z \mid z > 0 \} \\
\gamma_{\mathrm{Num}}(-_0) &= \{\, z \leq 0 \mid \} \\
\gamma_{\mathrm{Num}}(+_0) &= \{\, z \geq 0 \mid \} \\
\gamma_{\mathrm{Num}}(\top) &= \mathbb{Z}
\end{aligned}$$

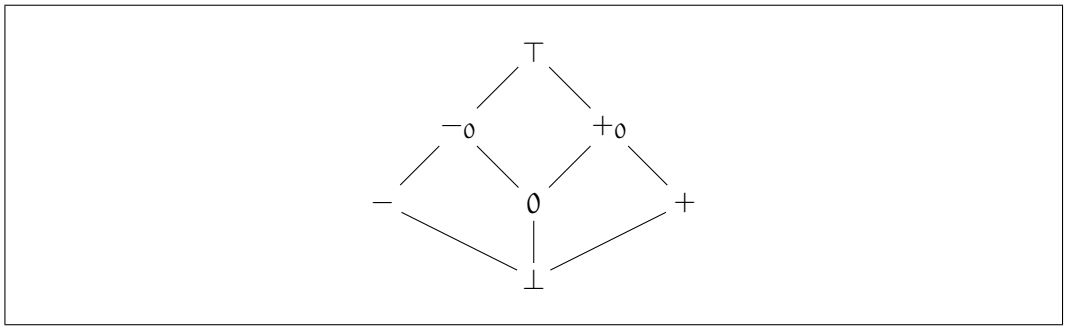


FIG. 5.13 – Diagramme de Hasse de l'abstraction par signe

La construction de ce treillis en Coq est relativement simple. Le type de base est un simple type énuméré. La plupart des preuves se font en examinant toutes les valeurs possibles des éléments. Le langage de tactique de Coq permet d'écrire des scripts de preuves indépendants du nombre d'éléments définis dans le type énuméré. Les preuves deviennent ainsi suffisamment robustes pour pouvoir rajouter un élément dans le domaine abstrait et rejouer tous les scripts avec succès.

Le treillis vérifie la condition de chaîne ascendante. Pour le prouver en terme d'accessibilité, il suffit de prouver que chaque élément du treillis est accessible. Un script de preuve permet à nouveau de prouver ce résultat, indépendemment du nombre et de la nature des éléments du domaine abstrait : pour chaque élément, on parcourt récursivement toute la chaîne des éléments plus grands. La preuve se termine à chaque fois sur l'accessibilité de $\top$, qui est triviale (pas d'élément strictement plus grand).

Les opérateurs abstraits du module suivent les règles des signes standards. Certains opérateurs demandent une construction un peu fastidieuse en raison du grand nombre de cas à distinguer. $[\![\odot]\!]\downarrow^{\sharp}_{\mathrm{op}}$ prend, par exemple, trois arguments. Si nous fixons la valeur de son premier argument à $+$, il reste encore 7^2 cas !. Le tableau suivant présente ces

différents cas[2].

$[\![\odot]\!]^\sharp_{\downarrow op}(+, n_1^\sharp, n_2^\sharp)$		$n_2^\sharp$						
		$\bot$	$-$	0	$+$	$-_0$	$+_0$	$\top$
$n_1^\sharp$	$\bot$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$
	$-$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(-,+)$	$(\bot,\bot)$	$(\bot,\bot)$	$(-,+)$
	0	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(0,+)$	$(\bot,\bot)$	$(0,+)$	$(0,+)$
	$+$	$(\bot,\bot)$	$(+,-)$	$(+,0)$	$(+,+)$	$(+,-_0)$	$(+,+_0)$	$(+,\top)$
	$-_0$	$(\bot,\bot)$	$(\bot,\bot)$	$(\bot,\bot)$	$(-_0,0)$	$(\bot,\bot)$	$(-_0,+_0)$	$(-_0,+)$
	$+_0$	$(\bot,\bot)$	$(+,-)$	$(+,0)$	$(+_0,+)$	$(+,-_0)$	$(+_0,+_0)$	$(+_0,\top)$
	$\top$	$(\bot,\bot)$	$(+,-)$	$(+,0)$	$(\top,+)$	$(+,-_0)$	$(\top,+_0)$	$(+_0,+_0)$

L'utilisation du filtrage (avec le joker _) permet de restreindre un peu le nombre de cas lors de la définition de la fonction. Coq élimine cependant en interne tous les jokers des filtrages. Les preuves manipulent alors la version exhaustive des filtrages. Une utilisation avancée du système de tactique de Coq est de ce fait indispensable pour éviter des scripts de preuves de longueur proportionnelle aux nombres de cas.

5.4.2 Abstraction par congruence

Une autre abstraction assez simple peut être proposée : l'abstraction par congruence [Gra89]. L'abstraction par parité rencontrée lors du chapitre 3 en est un cas particulier. Le module correspondant à cette abstraction est paramétré par un entier a strictement positif[3]. Le diagramme de Hasse du treillis abstrait est présenté dans la figure 5.14. La fonction de concrétisation $\gamma_{\text{Num}} : \text{Num}^\sharp \to \mathcal{P}(\mathbb{Z})$ associée à l'abstraction est définie par

$$\begin{aligned} \gamma_{\text{Num}}(\bot) &= \emptyset \\ \gamma_{\text{Num}}(i) &= \{\, z \mid z \equiv i \bmod a \,\},\ \forall i \in \{0, \ldots, a-1\} \\ \gamma_{\text{Num}}(\top) &= \mathbb{Z} \end{aligned}$$

5.4.3 Abstraction par intervalle

La dernière abstraction numérique que nous proposons utilise les intervalles [CC76]. Le domaine abstrait est celui des intervalles à coefficient dans $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, +\infty\}$. $\bot$ désigne l'intervalle vide.

$$\text{Int} \stackrel{\text{def}}{=} \{\, [a, b] \mid a, b \in \overline{\mathbb{Z}},\ a \leq b \,\} \cup \{\bot\}$$

L'ordre considéré correspond à l'inclusion ensembliste.

$$\begin{aligned} \bot &\sqsubseteq_{\text{Int}} I,\ \forall I \in \text{Int} \\ [a, b] &\sqsubseteq_{\text{Int}} [c, d] \iff c \leq a \text{ et } b \leq d \end{aligned}$$

[2] Pour les arguments $n_1^\sharp = +_0$ et $n_2^\sharp = 0$ le résultat $(+, 0)$ de $[\![\odot]\!]^\sharp_{\downarrow op}(+, n_1^\sharp, n_2^\sharp)$ s'interprète par « si la somme d'un entier positif ou nul et d'un entier nul est un entier strictement positif, alors le premier entier est strictement positif ».

[3] Pour respecter les contraintes du système de module de Coq et de Caml, cet entier est encapsulé dans un module

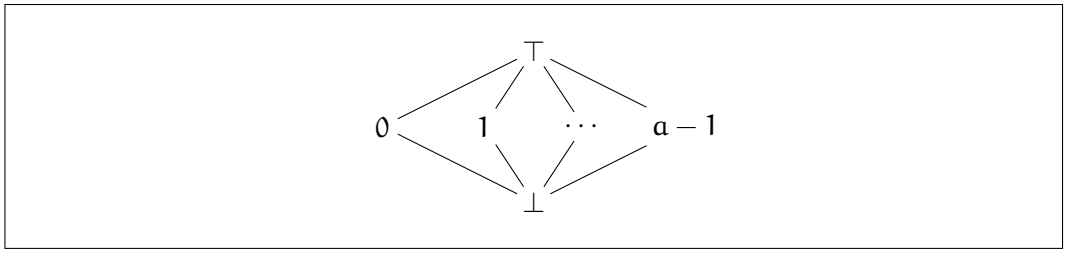


FIG. 5.14 – Diagramme de Hasse de l'abstraction par congruence, paramétrée par $a > 0$

L'union et l'intersection utilise les opérateur de mininum et de maximum binaire sur $\mathbb{Z}$, étendus à $\overline{\mathbb{Z}}$.

$$\begin{aligned} I \sqcup_{\text{Int}} \bot &= \bot \sqcup_{\text{Int}} I = I, \ \forall I \in \text{Int} \\ [a,b] \sqcup_{\text{Int}} [c,d] &= [\min(a,c), \max(b,d)] \end{aligned}$$

$$\begin{aligned} I \sqcap_{\text{Int}} \bot &= \bot \sqcap_{\text{Int}} I = \bot, \ \forall I \in \text{Int} \\ [a,b] \sqcap_{\text{Int}} [c,d] &= \begin{cases} [\max(a,c), \min(b,d)] & \text{si } \max(a,c) \leq \min(b,d) \\ \bot & \text{sinon} \end{cases} \end{aligned}$$

La fonction de concrétisation envoie chaque intervalle vers l'ensemble des entiers qu'il contient.

$$\begin{aligned} \gamma_{\text{Int}}(\bot) &= \emptyset \\ \gamma_{\text{Int}}([a,b]) &= \{ \, z \in \mathbb{Z} \mid a \leq z \text{ et } z \leq b \, \} \end{aligned}$$

Nous listons maintenant les différents opérateurs abstraits implémentés sur ce domaine. Toutes ces fonctions renvoient $\bot$ si l'un de leurs arguments vaut $\bot$, nous ne considérons donc que les intervalles de la forme $[a,b]$. Nous utiliserons parfois la fonction $\rho \in (\overline{\mathbb{Z}} \times \overline{\mathbb{Z}}) \to \text{Int}$ définie par

$$\rho(a,b) = \begin{cases} [a,b] & \text{si } a \leq b, \\ \bot & \text{sinon} \end{cases}$$

pour s'assurer que les intervalles calculés sont cohérents.

$$
\begin{aligned}
[\![+]\!]^\sharp_{\mathrm{op}}\,([a,b],[c,d]) &= [a+c, b+d]\\
[\![-]\!]^\sharp_{\mathrm{op}}\,([a,b],[c,d]) &= [a-d, b-c]\\
[\![\times]\!]^\sharp_{\mathrm{op}}\,([a,b],[c,d]) &= [\min(ac,ad,bc,bd), \max(ac,ad,bc,bd)]
\end{aligned}
$$

$$
[\![-]\!]^\sharp_{\mathrm{op}}\,([a,b]) = [-b,-a]
$$

$$
\begin{aligned}
[\![+]\!]^\sharp_{\mathrm{op}}\downarrow\,([a,b],[c,d],[e,f]) &= (\rho(\max(c,a-f),\min(d,b-e)),\\
&\qquad \rho(\max(e,a-d),\min(f,b-c)))\\
[\![-]\!]^\sharp_{\mathrm{op}}\downarrow\,([a,b],[c,d],[e,f]) &= (\rho(\max(c,a+e),\min(d,b+f)),\\
&\qquad \rho(\max(e,c-b),\min(f,d-a)))\\
[\![*]\!]^\sharp_{\mathrm{op}}\downarrow\,([a,b],[c,d],[e,f]) &= ([c,d],[e,f])
\end{aligned}
$$

$$
\begin{aligned}
[\![=]\!]^\sharp_{\mathrm{comp}}\downarrow\,([a,b],[c,d]) &= ([a,b]\sqcap_{\mathrm{Int}}[c,d], [a,b]\sqcap_{\mathrm{Int}}[c,d])\\
[\![<]\!]^\sharp_{\mathrm{comp}}\downarrow\,([a,b],[c,d]) &= ([a,b]\sqcap_{\mathrm{Int}}[-\infty,d-1], [a+1,+\infty]\sqcap_{\mathrm{Int}}[c,d])\\
\mathrm{const}(n)^\sharp &= [n,n]\\
\top &= [-\infty,+\infty]
\end{aligned}
$$

Ce treillis ne vérifie pas la condition de chaîne ascendante : la suite infinie

$$\bot, [0,0], [-1,1], \ldots, [-n,n], \ldots$$

est strictement croissante. Nous utilisons par conséquent des opérateurs d'élargissement/rétrécissement pour forcer la convergence des itérations de points fixes. Notre développement Coq utilise les opérateurs traditionnels d'élargissement/rétrécissement.

$$
\begin{aligned}
\bot\nabla I &= I\nabla\bot = I, \forall I \in \mathrm{Num}^\sharp\\
[a,b]\nabla[c,d] &= [\text{si } a \le c \text{ alors } a \text{ sinon } -\infty,\\
&\qquad \text{si } d \le b \text{ alors } b \text{ sinon } +\infty]\ \forall a,b,c,d \in \overline{\mathbb{Z}}
\end{aligned}
$$

$$
\begin{aligned}
\bot\Delta I &= I\Delta\bot = I, \forall I \in \mathrm{Num}^\sharp\\
[a,b]\Delta[c,d] &= [\text{si } a = -\infty \text{ alors } c \text{ sinon } a,\\
&\qquad \text{si } b = +\infty \text{ alors } d \text{ sinon } b]\ \forall a,b,c,d \in \overline{\mathbb{Z}}
\end{aligned}
$$

5.5 Extraction

Au final, plusieurs interpréteurs abstraits certifiés peuvent être extraits, en combinant les modules et les foncteurs précédemment présentés. Ainsi, nous pouvons construire un interpréteur abstrait à base d'intervalles avec le module suivant :

```
Module IntervalAnalysis.

  Module E := EnvNotRelational IntervalAbstraction.
  Module S := StateLift E.

  Definition solve : ∀ P,
  { St:S.S.Lat.t | (ReachableStates P) <=< (S.gamma P St) } := S.solve.

End IntervalAnalysis.
```

L'extraction de l'ensemble des modules produit un fichier de 6000 lignes de code Caml. Le tableau suivant indique la répartition du nombre de lignes de code en fonction du rôle joué dans l'analyse.

sujet	**nombres de lignes Caml**	**nombres de lignes Coq**
points fixes	400	2000
signes	1900	1300
congruences	300	900
intervalles	1000	3000
manipulation des tableaux	800	2500
environnements	300	1000
itérations chaotiques	150	500
divers	850	2600
analyseur principal	300	1200
total	6000	15000

L'abstraction des signes prend à elle seule 1900 lignes car le code extrait contient des filtrages exhaustifs.

Le code extrait est inséré dans un programme Caml proposant un parseur de programme WHILE et un enlumineur syntaxique (*pretty-printer*) de valeurs abstraites. Voici quelques exemples d'utilisation de cet analyseur. Le programme peut être téléchargé et testé [Whi].

Améliorations apportées grâce aux itérations réductives Un premier exemple va nous permettre de présenter l'apport de la technique d'itération réductive.

```
% ./while -num sign PROG/p12.c
sign analysis
        // I =  top   J =  top
J = ?;
        // I =  top   J =  top
I = -5;
        // I =  top   J =  top
while (I * J < 0 and I <= 0) {
        // I = {-,0}  J =  top
  I = I + 1;
        // I =  top   J =  top
};
        // I =  top   J =  top
```

```
% ./while -num sign
          -reduction PROG/p12.c
sign analysis with guard reduction
        // I =  top   J =  top
J = ?;
        // I =  top   J =  top
I = -5;
        // I =  top   J =  top
while (I * J < 0 and I <= 0) {
        // I =   -    J =   +
  I = I + 1;
        // I =  top   J =   +
};
        // I =  top   J =  top
```

Dans l'exemple de gauche, nous n'utilisons pas cette technique. Très peu d'information est alors générée après le test de la boucle **while**. Dans l'exemple de droite l'itération permet de raffiner à la fois l'invariant sur `I` et sur `J`.

Comparaison de précision entre signes et intervalles Même lorsque les invariants calculés avec des intervalles sont directement représentables par l'abstraction des signes, l'analyse par intervalle peut donner une information plus précise. Dans cet exemple, les intervalles démontrent que la valeur finale de `Y` est O, alors que les signes en sont incapables. L'imprécision de l'analyse par signe provient d'une approximation non optimale de l'affectation `X = X+1`. Pour analyser plus finement cet exemple, il faudrait ajouter une abstraction de l'opérateur successeur dans nos abstractions numériques. Contrairement à l'analyse par signe, l'analyse par intervalle est capable de traiter tous les programmes similaires à celui de cet exemple, où 0 est remplacé par une constante quelconque.

```
% ./while  -num interval /PROG/p2.c
interval analysis
      // Y = [-oo,+oo]  X = [-oo,+oo]
X = ?;
      // Y = [-oo,+oo]  X = [-oo,+oo]
if X < 0 {
      // Y = [-oo,+oo]  X = [-oo, 0 ]
  while X < 0 {
      // Y = [-oo,+oo]  X = [-oo, -1]
    X = X + 1;
      // Y = [-oo,+oo]  X = [-oo, 0 ]
  };
      // Y = [-oo,+oo]  X = [ 0 , 0 ]
  Y = X;
      // Y = [ 0 , 0 ]  X = [ 0 , 0 ]
} else {
      // Y = [-oo,+oo]  X = [ 0 ,+oo]
  Y = 0;
      // Y = [ 0 , 0 ]  X = [ 0 ,+oo]
};
      // Y = [ 0 , 0 ]  X = [ 0 ,+oo]
```

```
% ./while -num sign PROG/p2.c
sign analysis
          // Y =  top   X =  top
X = ?;
          // Y =  top   X =  top
if X < 0 {
          // Y =  top   X =  top
  while X < 0 {
          // Y =  top   X =   -
    X = X + 1;
          // Y =  top   X =  top
  };
          // Y =  top   X = {+,0}
  Y = X;
          // Y = {+,0}  X = {+,0}
} else {
          // Y =  top   X = {+,0}
  Y = 0;
          // Y =   0    X = {+,0}
};
          // Y = {+,0}  X = {+,0}
```

Comparaison des techniques d'itération avec élargissement/rétrécissement

```
% ./while -num interval PROG/p1.c
interval analysis
      // J = [-oo,+oo]  I = [-oo,+oo]
I = 2;
      // J = [-oo,+oo]  I = [ 2 , 2 ]
J = 0;
      // J = [ 0 , 19]  I = [ 2 , 24]
while I + J < 21 {
      // J = [ 0 , 18]  I = [ 2 , 20]
  if 9 < I {
      // J = [ 0 , 18]  I = [ 10, 20]
    I = I + 4;
      // J = [ 0 , 18]  I = [ 14, 24]
  } else {
      // J = [ 0 , 18]  I = [ 2 , 9 ]
    I = I + 2;
      // J = [ 0 , 18]  I = [ 4 , 11]
    J = J + 1;
      // J = [ 1 , 19]  I = [ 4 , 11]
  };
      // J = [ 0 , 19]  I = [ 4 , 24]
};
      // J = [ 0 , 19]  I = [ 2 , 24]
```

```
% ./while -num interval
          -naive PROG/p1.c
naive solver with interval analysis
      // J = [-oo,+oo]  I = [-oo,+oo]
I = 2;
      // J = [-oo,+oo]  I = [ 2 , 2 ]
J = 0;
      // J = [-oo, 19]  I = [ 2 ,+oo]
while I + J < 21 {
      // J = [-oo, 18]  I = [ 2 ,+oo]
  if 9 < I {
      // J = [-oo, 18]  I = [ 10,+oo]
    I = I + 4;
      // J = [-oo, 18]  I = [ 14,+oo]
  } else {
      // J = [-oo, 18]  I = [ 2 , 9 ]
    I = I + 2;
      // J = [-oo, 18]  I = [ 4 , 11]
    J = J + 1;
      // J = [-oo, 19]  I = [ 4 , 11]
  };
      // J = [-oo, 19]  I = [ 4 ,+oo]
};
      // J = [-oo, 19]  I = [ 2 ,+oo]
```

Cet exemple illustre les possibilités de l'itération chaotique. À gauche, l'itération chaotique présentée dans le chapitre précédent calcule des intervalles bornés lorsque c'est possible, alors que l'itération sans restriction sur les points d'élargissement (à droite) génère plusieurs intervalles non-bornés.

L'itération chaotique peut néanmoins être trop imprécise sur certains programmes.

```
% ./while -num interval PROG/p10.c
interval analysis
      // k = [-oo,+oo]  i = [-oo,+oo]
i = 0;
      // k = [-oo,+oo]  i = [ 0 , 0 ]
k = 0;
      // k = [ 0 ,+oo]  i = [ 0 , 10]
while k < 10 {
      // k = [ 0 , 9 ]  i = [ 0 , 10]
  i = 0;
      // k = [ 0 ,+oo]  i = [ 0 , 10]
  while i < 10 {
      // k = [ 0 ,+oo]  i = [ 0 , 9 ]
    i = i + 1;
      // k = [ 0 ,+oo]  i = [ 1 , 10]
  };
      // k = [ 0 ,+oo]  i = [ 10, 10]
  k = k + 1;
      // k = [ 1 ,+oo]  i = [ 10, 10]
};
      // k = [ 10,+oo]  i = [ 0 , 10]
```

```
% ./while -num interval
          -extern widen ./PROG/p10.c
external solver with widening/narrowing
Valid post-fix point !
      // k = [-oo,+oo]  i = [-oo,+oo]
i = 0;
      // k = [-oo,+oo]  i = [ 0 , 0 ]
k = 0;
      // k = [ 0 , 10]  i = [ 0 , 10]
while k < 10 {
      // k = [ 0 , 9 ]  i = [ 0 , 10]
  i = 0;
      // k = [ 0 , 9 ]  i = [ 0 , 10]
  while i < 10 {
      // k = [ 0 , 9 ]  i = [ 0 , 9 ]
    i = i + 1;
      // k = [ 0 , 9 ]  i = [ 1 , 10]
  };
      // k = [ 0 , 9 ]  i = [ 10, 10]
  k = k + 1;
      // k = [ 1 , 10]  i = [ 10, 10]
};
      // k = [ 10, 10]  i = [ 0 , 10]
```

Dans cet exemple, l'itération chaotique certifiée ne permet pas de détecter que k reste borné (exemple de gauche). Cette perte de précision est due à la stratégie itérative que nous adoptons dans notre itérateur chaotique. Une meilleur précision peut généralement être obtenue en itérant en priorité les équations des boucles les plus imbriquées. Nous proposons un solveur[4] externe (non certifié) capable de trouver les plus petits intervalles pour k. Son résultat est cependant validé par le vérificateur de post-point fixe certifié.

5.6 Conclusions

Nous avons présenté dans ce chapitre le premier analyseur certifié de ce manuscrit. Grâce au caractère modulaire de notre développement, c'est en fait plusieurs analyseurs qui sont ici certifiés. La théorie de l'interprétation abstraite est en effet d'une grande aide pour proposer des interfaces génériques entre les différents composants d'une analyse[5].

[4] Ce solveur utilise la structure des programmes While en suivant une sémantique plus dénotationnelle des programmes. Chaque boucle **while** fait appel à une résolution de post-point fixe. Cette technique d'itération est présentée dans [Cou99] et utilisée dans [CCF+05].

[5] L'interface des environnements de variables est par exemple compatible avec la plupart des abstractions relationnelles existantes [Min04, CH78].

Les algorithmes utilisés dans cet analyseur sont ceux proposés dans [Cou99]. Les preuves ont néanmoins été revues pour se concentrer uniquement sur la correction sémantique, et non sur la précision. L'opérateur de réduction $\text{reduc}^\sharp$ n'apparaissait pas distinctement dans [Cou99], chaque opérateur abstrait étant optimisé spécifiquement. Il semble beaucoup plus économique en terme de preuve de recourir à un opérateur générique comme $\text{reduc}^\sharp$ pour améliorer la précision de certains opérateurs abstraits.

Le cadre que nous avons choisi pour développer des interpréteurs abstraits montre ainsi sa capacité à produire de véritables analyseurs certifiés. Une partie difficile du développement réside dans l'utilisation du foncteur `ArrayBinSolve` pour construire le treillis des fonctions. Il s'agit d'une construction particulièrement délicate en raison des preuves de terminaison nécessaires et du soin qu'il faut apporter aux structures de données employées[6]. La construction de ce foncteur sera présentée dans le chapitre 6.

Le langage WHILE est un parfait « terrain de jeu » pour l'interprétation abstraite numérique. Il reste cependant assez éloigné d'un langage réaliste. Nous aborderons, dans le chapitre 8, l'analyse d'un langage plus réaliste comme le bytecode Java. Les techniques d'analyse que nous avons utilisées sont cependant non-triviales et n'avaient jamais fait l'objet de certification formelle jusqu'à présent.

Il nous apparaît important d'insister sur la conception modulaire de cette analyse. Du point de vue de programmation, la réutilisation de composants logiciels est appréciable. Du point de vue de la preuve formelle, elle devient indispensable pour pouvoir gérer de gros projets comme le notre. Sans modularité, un petit changement dans un algorithme peut avoir des répercussions sur plusieurs milliers de lignes de preuves. Les interfaces de module font ainsi office de portes « coupe-feu ». Nous verrons dans le chapitre 7 que le découpage modulaire d'une preuve de correction sémantique d'une analyse n'est pas toujours aussi évident, par exemple pour l'analyse des langages à objet. Nous proposerons alors des techniques spécifiques pour, malgré tout, découper suffisamment la preuve de correction.

[6] Le choix des structures de données prendra vraiment de l'importance lorsque nous aborderons l'efficacité des analyseurs extraits sur des plus gros programmes que les programmes WHILE.

Chapitre 6

Composition constructive de treillis

Les analyseurs statiques certifiés que nous développons dans nos travaux reposent sur une structure de treillis. Dans le chapitre 4, nous avons présenté certains critères permettant de calculer (ou d'approcher) des points fixes de fonctions sur ces treillis. Ces critères assurent principalement la terminaison des algorithmes employés.

Nous allons maintenant présenter une technique modulaire pour construire des treillis en Coq, chaque treillis vérifiant un critère de terminaison de calcul itératif de point fixe. L'idée consiste à proposer une librairie de foncteurs de treillis qui permettra de construire des treillis arbitrairement complexes par simple composition de ces foncteurs sur des treillis de base. Les structures ainsi construites posséderont des preuves de terminaison très difficiles à obtenir sans un tel découpage modulaire.

Les preuves de terminaison sont généralement une partie très délicate dans un développement formel. L'utilisation de la notion d'accessibilité pour définir les critères de terminaison rend cette tâche encore plus ardue car les propriétés à prouver sont alors peu intuitives. Plusieurs travaux récents présentent des techniques pour définir des fonctions récursives générales en théorie des types [Bov02, Bal02], mais ils donnent rarement des outils ou des méthodes pour construire des preuves d'accessibilité. Les résultats connus sur le prédicat d'accessibilité sont principalement réunis dans les travaux de Paulson [Pau86]. Ce chapitre décrit les preuves les plus difficiles de notre travail de thèse. Il a notamment été nécessaire de prouver un nouveau résultat général sur le prédicat d'accessibilité.

Nous allons dans un premier temps présenter une sélection des différents foncteurs de treillis que nous proposons dans notre bibliothèque. Les sections 6.2 et 6.3 présenteront ensuite les preuves de terminaison que nous avons dû établir.

6.1 Différents foncteurs de treillis

Nous allons maintenant présenter les différentes combinaisons de treillis que nous proposons dans notre librairie. Les différents résultats mathématiques utilisés dans cette section admettent des preuves assez simples que nous ne préciserons pas.

6.1.1 Produit de treillis

Le première combinaison de base permet de former le produit de deux treillis. Le résultat mathématique suivant nous assure en effet qu'une structure de treillis peut être prise sur le produit des deux domaines, grâce à l'ordre composante par composante.

Théorème 6.1.1. *Si* $(A, \sqsubseteq_A, \sqcup_A, \sqcap_A)$ *et* $(B, \sqsubseteq_B, \sqcup_B, \sqcap_B)$ *sont deux treillis alors le quadruplet* $(A \times B, \sqsubseteq_{A\times B}, \sqcup_{A\times B}, \sqcap_{A\times B})$ *défini par*

$$\begin{aligned}(a_1, b_1) \sqsubseteq_{A\times B} (a_2, b_2) &\stackrel{def}{=} a_1 \sqsubseteq_A a_2 \ \wedge \ b_1 \sqsubseteq_B b_2\\ (a_1, b_1) \sqcup_{A\times B} (a_2, b_2) &\stackrel{def}{=} (a_1 \sqcup_A a_2, b_1 \sqcup_B b_2)\\ (a_1, b_1) \sqcap_{A\times B} (a_2, b_2) &\stackrel{def}{=} (a_1 \sqcap_A a_2, b_1 \sqcap_B b_2)\end{aligned}$$

est un treillis.

Si, de plus, $(A, \sqsubseteq_A, \sqcup_A, \sqcap_A)$ *admet un plus petit élément* $\bot_A$ *et* $(B, \sqsubseteq_B, \sqcup_B, \sqcap_B)$ *admet un plus petit élément* $\bot_B$*, alors* $(A \times B, \sqsubseteq_{A\times B}, \sqcup_{A\times B}, \sqcap_{A\times B})$ *admet pour plus petit élément* $\bot_{A\times B} = (\bot_A, \bot_B)$.

Ces résultats mathématiques se traduisent en Coq à l'aide d'un foncteur de module. Étant donnés deux modules `L1` et `L2` vérifiant une signature de treillis (signature présentée dans la section 4.2.3), `ProdLattice` construit un module de type `Lattice` dont le domaine de base est le produit des deux domaines de base de `L1` et `L2` et dont les différents composants suivent les définitions du théorème 6.1.1. L'entête du foncteur est ainsi définie par

```
Module ProdLattice (L1:Lattice) (L2:Lattice) <: Lattice
  with Definition t := L1.t * L2.t
  with Definition eq := fun (x y : L1.t * L2.t) ⇒
    L1.eq (fst x) (fst y) ∧ L2.eq (snd x) (snd y)
  with Definition order := fun (x y : L1.t * L2.t) ⇒
    L1.order (fst x) (fst y) ∧ L2.order (snd x) (snd y)
  with Definition join := fun x y ⇒
    (L1.join (fst x) (fst y),L2.join (snd x) (snd y))
  with Definition meet := fun x y ⇒
    (L1.meet (fst x) (fst y),L2.meet (snd x) (snd y))
  with Definition bottom := (L1.bottom,L2.bottom).
```

La notation `<: Lattice` impose que le module construit vérifie une signature de treillis. Les définitions données avec le mot clé **`with`** permettent de préciser les définitions choisies dans le module. Seuls les objets de type preuve ont une valeur non précisée.

La programmation Coq des différents éléments de `ProdLattice` est ensuite aisée.

6.1.2 Somme parallèle disjointe de treillis-partiels

Si $(A, \sqsubseteq_A)$ et $(B, \sqsubseteq_B)$ sont deux ensembles partiellement ordonnés, il est possible de munir la somme disjointe $A + B$ d'un ordre partiel comparant uniquement les éléments de même type (voir figure 6.1).

Lemme 6.1. ***(Poset somme parallèle)*** *Étant donnés deux ensembles partiellement ordonnés* $(A, \sqsubseteq_A)$ *et* $(B_B, \sqsubseteq_B)$*, le doublet* $(A + B, \sqsubseteq_{A+B})$ *défini par*

$$\sqsubseteq_{A+B} = \{(a_1, a_2) \mid a_1 \sqsubseteq_A a_2\} \cup \{(b_1, b_2) \mid b_1 \sqsubseteq_B b_2\}$$

est un ensemble partiellement ordonné.

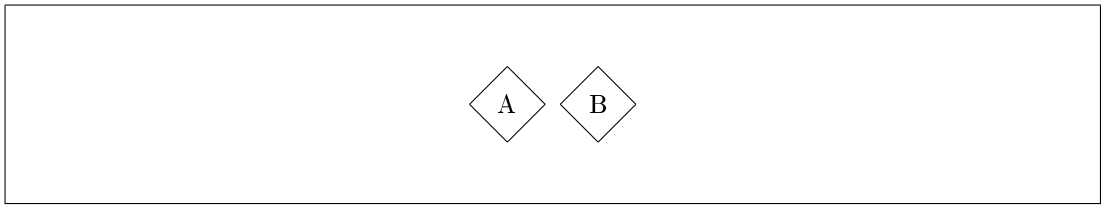


FIG. 6.1 – Poset somme parallèle

Cette combinaison ne conserve pas la structure de treillis. Il n'existe, par exemple, pas de borne supérieure pour un doublet de la forme (a, b) avec $a \in A$ et $b \in B$. Nous proposons par conséquent un foncteur de la forme

```
Module SumPartialLattice (L1 L2:PartialLattice)  <: PartialLattice
```

où `PartialLattice` est une signature de *treillis partiel* dans laquelle les opérateurs binaires $\sqcup$ et $\sqcap$ ne sont pas définies sur les éléments incomparables (voir figure 6.2).

Le passage entre treillis partiel et treillis standard s'effectue à l'aide de deux foncteurs. Le premier transforme un treillis standard en un treillis partiel en gardant le même domaine de base et le même ordre.

```
Module PartialLattice_of_Lattice (L:Lattice) <: PartialLattice
  with Definition t := L.t
  with Definition eq := L.eq
  with Definition order := L.order
  with Definition join := fun x y ⇒ Some (L.join x y)
  with Definition meet := fun x y ⇒ Some (L.meet x y).

...

End PartialLattice_of_Lattice.
```

Le second foncteur transforme un treillis partiel en un treillis standard en ajoutant deux éléments $\top$ et $\bot$ au domaine de base.

```
Module Lattice_of_PartialLattice (L:PartialLattice) <: Lattice
  with Definition t := lift_top_bot.t L.t
  with ...

...

End Lattice_of_PartialLattice.
```

```
Module Type PartialLattice.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  Parameter eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter order : t → t → Prop.
  Parameter order_refl : ∀ x y : t, eq x y → order x y.
  Parameter order_antisym : ∀ x y : t,
    order x y → order y x → eq x y.
  Parameter order_trans : ∀ x y z : t,
    order x y → order y z → order x z.
  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

  Parameter join  : t → t → option t.
  Parameter join_bound1 : ∀ x y j : t,
    join x y = Some j →  order x j.
  Parameter join_bound2 : ∀ x y j : t,
    join x y = Some j →  order y j.
  Parameter join_least_upper_bound : ∀ x y z : t,
    order x z → order y z →
    ∃ j, join x y = Some j ∧ order j z.

  Parameter meet  : t → t → option t.
  Parameter meet_bound1 : ∀ x y m : t,
    meet x y = Some m → order m x.
  Parameter meet_bound2 : ∀ x y m : t,
    meet x y = Some m → order m y.
  Parameter meet_greatest_lower_bound : ∀ x y z : t,
    order z x → order z y →
    ∃ m, meet x y = Some m ∧ order z m.
End PartialLattice.
```

FIG. 6.2 – Signature des treillis partiels en Coq

6.1.3 Treillis des fonctions

Nous allons maintenant aborder une construction très importante : les fonctions. Beaucoup d'analyses reposent en effet sur un treillis de fonction. La technique de partitionnement de la section 3.2.3.3 est par exemple indispensable pour pouvoir attacher des informations à chaque point de programme et elle repose sur un treillis de fonctions.

6.1.3.1 Implémentation des fonctions

L'implémentation des fonctions est un point crucial pour l'efficacité du code extrait. Nous proposons par conséquent de séparer la structure de données choisie pour implémenter les fonctions de la construction de la structure de treillis.

Une signature de module pour décrire les implémentations de fonctions. Une implémentation de fonction est décrite par une signature `Func_FiniteSet_Lattice` (voir figure 6.3). Cette signature contient un module `A` respectant la signature `FiniteSet` (associé au domaine des fonctions). Nous reviendrons plus en détail sur cette signature dans la sous-section 6.2.3.1. Nous pouvons nous contenter de considérer un module de ce type comme un ensemble fini (modélisé par un type `A.t`) muni d'une relation d'équivalence `A.eq`. Le module `B` représente le treillis associé au codomaine des fonctions. Le type `t` est le type de la structure de donnée abstraite utilisée pour représenter des fonctions. La fonction `get` permet d'accéder à la fonction (élément de `A.t→B.t`) associée à un élément de type `t` : si `F:t` est une représentation de fonction, si `a:A.t` un élément du domaine, `(get F a)` représente l'image associée, de type `B.t`. Grâce à cette fonction `get`, on peut définir l'équivalence `eq` et l'ordre partiel `order` associés[1]. Aucune preuve que ces relations forment des relations d'équivalence (respectivement d'ordres partiels) valides n'est requise, car il s'agit de conséquences directes de leurs définitions. Ces propriétés pourront donc être établies pour toute implémentation. Trois propriétés nécessitent cependant une preuve spécifique à chaque implémentation :

- la propriété `eq_refl` qui revient à imposer que les résultats de `get` respectent la relation d'équivalence `A.eq` prise sur `A.t`,
- `eq_dec` et `order_dec` sont des implémentations des tests d'équivalence et d'ordre sur les structures de type `t`.

Étant donné un opérateur binaire `f : B.t → B.t → B.t` sur le codomaine `B`, les opérateurs `map2` et `map2'` permettent de construire une nouvelle fonction à partir de deux autres fonctions `t1` et `t2`. La fonction ainsi construite est équivalente à $\lambda x.f(t_1(x), t_2(x))$. Les différences entre `map2` et `map2'` seront précisées dans la section suivante.

Deux implémentations distinctes. Une telle construction a l'avantage de laisser libre l'implémentation choisie pour les fonctions. Notre bibliothèque propose ainsi le choix entre deux types d'implémentations :

[1] Nous utilisons pour cela une particularité du système de module de Coq : pouvoir placer des définitions dans la signature de modules.

```
Module Type Func_FiniteSet_Lattice.

  Declare Module A : FiniteSet.
  Declare Module B : Lattice.

  Parameter t : Set.

  Parameter get : t → A.t → B.t.

  Definition eq : t → t → Prop := fun f1 f2 ⇒
    ∀ a1 a2 : A.t, A.eq a1 a2 → B.eq (get f1 a1) (get f2 a2).

  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Definition order : t → t → Prop := fun f1 f2 ⇒
    ∀ a1 a2 : A.t, A.eq a1 a2 → B.order (get f1 a1) (get f2 a2).

  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

  Parameter map2 : (B.t → B.t → B.t) → t → t → t.
  Parameter get_map2 : ∀ (f : B.t → B.t → B.t),
    (∀ x : B.t, B.eq x (f B.bottom x)) →
    (∀ x : B.t, B.eq x (f x B.bottom)) →
    ∀ (t1 t2 : t) (a : A.t),
      B.eq (get (map2 f t1 t2) a) (f (get t1 a) (get t2 a)).

  Parameter map2' : (B.t → B.t → B.t) → t → t → t.
  Parameter get_map2' : ∀ (f : B.t → B.t → B.t),
    (∀ x : B.t, B.eq B.bottom (f B.bottom x)) →
    (∀ x : B.t, B.eq B.bottom (f x B.bottom)) →
    ∀ (t1 t2 : t) (a : A.t),
      B.eq (get (map2' f t1 t2) a) (f (get t1 a) (get t2 a)).

  Parameter bottom : t.
  Parameter bottom_eq_bottom : ∀ a, B.eq (get bottom a) B.bottom.

End Func_FiniteSet_Lattice.
```

FIG. 6.3 – Signature des implémentations de fonctions en Coq

– la première implémentation est spécifique aux fonctions dont le domaine est de type `Word` (les entiers binaires bornés, présentés dans la section 5.1). Nous utilisons pour cela un stockage dans un arbre, en utilisant un entier binaire (de type `positive`) pour représenter sa position (technique présentée dans le chapitre 2).
– la deuxième construction permet de prendre comme domaine n'importe quel ensemble fini (éventuellement construit à l'aide des foncteurs présentés précédemment). Elle est basée sur une implémentation abstraite des maps de Caml. Nous proposons actuellement une implémentation sous forme de liste triée, en réutilisant les travaux de Filliâtre et Letouzey [FL04] sur les ensembles finis. Nous avons aussi entrepris une implémentation plus efficace sous forme d'arbre équilibré (les AVL utilisés dans [FL04]), inachevée à l'heure actuelle.

6.1.3.2 Construction du treillis de fonction

La construction d'un treillis de fonction est ensuite réalisée par un foncteur prenant en argument une implémentation des fonctions.

```
Module FuncLattice (F : Func_FiniteSet_Lattice) <: Lattice
  with Definition t := F.t
  with Definition eq := F.eq
  with Definition order := F.order.
```

La définition des opérateurs `join` et `meet` utilise les opérateurs `map2` et `map2'` de l'interface `Func_FiniteSet_Lattice`. `map2` et `map2'` diffèrent par leur traitement de la valeur $\bot$. Celle-ci jouant un rôle particulier dans nos implémentations de fonctions, il est intéressant de pouvoir tenir compte du comportement d'un opérateur binaire f vis à vis de $\bot$ pour optimiser la fonction calculée par $\mathrm{map}_2(f)$. Dans le cas de l'implémentation par arbres binaires et clés binaires, toute feuille représente la fonction valant $\bot$ pour toute clé. Selon que f est *stricte* ($f(\bot, x) = f(x, \bot) = \bot$) ou *absorbante* ($f(\bot, x) = f(x, \bot) = x$) vis à vis de $\bot$, le calcul de $\mathrm{map}_2(f)$ peut être simplifié lorsqu'une feuille est atteinte. La figure 6.4 présente un exemple du traitement spécifique à apporter dans chaque cas.

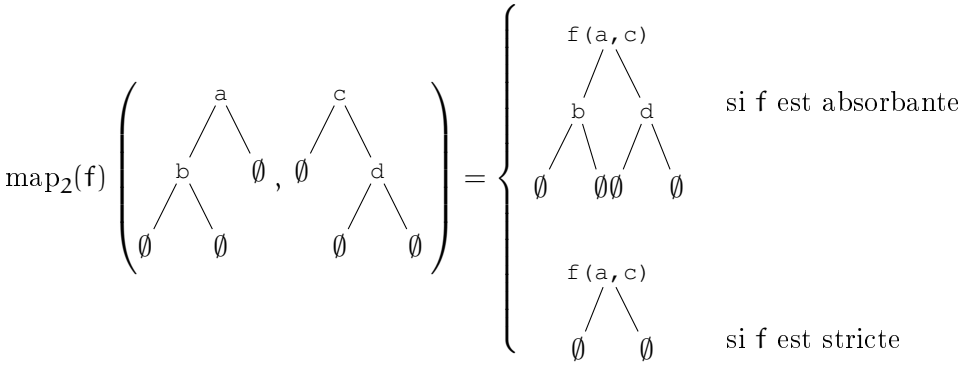


FIG. 6.4 – $\mathrm{map}_2(f)$ selon que f soit stricte ou absorbante

6.1.4 D'autres foncteurs

Nous avons présenté plusieurs foncteurs : le produit et la somme disjointe de treillis, ainsi que les treillis de fonctions. D'autres foncteurs sont actuellement proposés dans notre librairie de foncteurs : la somme étagée de deux treillis (voir figure 6.5(a)) et le treillis des listes à valeurs dans un treillis (seules les listes de même taille sont comparables, voir figure 6.5(b)). Ces foncteurs ne sont pas détaillés davantage car leurs constructions sont similaires à celles des foncteurs précédents.

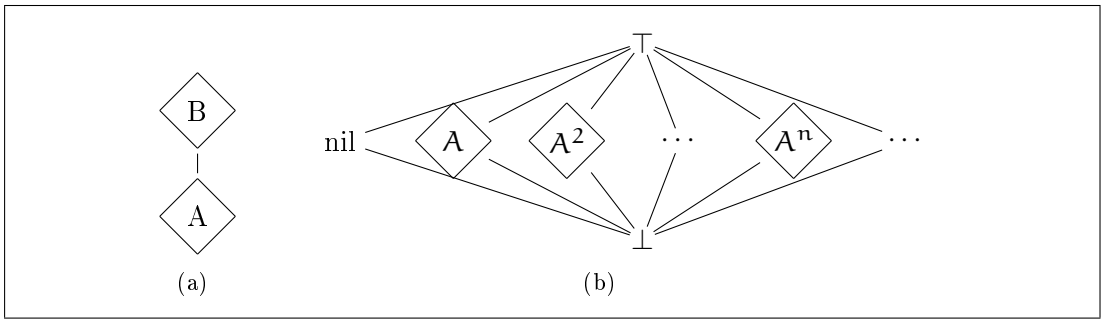


FIG. 6.5 – Le treillis de la somme étagée et le treillis des listes

6.1.5 Construction de treillis par injection dans un autre treillis

Les différents foncteurs précédents permettent de construire des treillis (ou des foncteurs de treillis) complexes par simple composition de foncteurs. Les structures mathématiques sont alors valides d'après leur type, mais du point de vue de la programmation, les structures de données et les fonctions utilisées peuvent être peu élégantes, voir inefficaces. L'utilisation du foncteur de produit binaire de treillis peut par exemple être itérée pour construire le produit n-aire de treillis, avec n une constante entière. Le domaine sous jacent sera néanmoins de la forme $\cdot \times (\cdot \times \cdots (\cdot \times \cdot))$, ce qui n'est pas particulièrement satisfaisant. Nous proposons de reprogrammer ces treillis, tout en conservant leur preuve de validité.

Une nouvelle implémentation d'un treillis est représentée par un module respectant la signature `NewImplemLattice` présentée dans la figure 6.6. Un tel module contient un treillis `L` que l'on souhaite reprogrammer. Le type sur lequel la nouvelle structure de treillis reposera est désigné par `t`. Les relations binaires `eq` et `order` sont définies avec leurs tests de décision `eq_dec` et `order_dec`. Les hypothèses `order_eq1` et `order_eq2` assurent que `eq` est bien la relation d'équivalence associée à `order`. Les preuves nécessaires pour la construction d'un treillis sur `t` et `order` sont obtenues grâce à l'injection `i` prise entre `t` et `L.t`. Nous imposons en effet que la relation `order` soit équivalente à `L.order` (à injection près). L'ensemble représenté par `t` est ainsi isomorphe (pour l'ordre `L.order`) à un sous-treillis de `L`.

Il est important de remarquer que la construction d'un tel module demande très peu de preuves. Les seules preuves nécessaires concernent l'injection `i` et la commutativité des anciens opérateurs avec les nouveaux.

```
Module Type NewImplemLattice.

  Declare Module L : Lattice.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter order : t → t → Prop.
  Parameter order_eq1 : ∀ x y : t,
    eq x y → order x y ∧ order y x.
  Parameter order_eq2 : ∀ x y : t,
    order x y → order y x → eq x y.
  Parameter order_dec : ∀ x y : t, {order x y}+{¬ order x y}.

  Parameter join  : t → t → t.
  Parameter meet  : t → t → t.

  Parameter i : t → L.t.
  Parameter i_order_preserve : ∀ x y,
    order x y → L.order (i x) (i y).
  Parameter inv_i_order_preserve : ∀ x y,
    L.order (i x) (i y) → order x y.

  Parameter join_commut : ∀ x y,
    L.eq (i (join x y)) (L.join (i x) (i y)).
  Parameter meet_commut : ∀ x y,
    L.eq (i (meet x y)) (L.meet (i x) (i y)).

  Parameter bottom : t.
  Parameter bottom_is_bottom : L.eq L.bottom (i bottom).

End NewImplemLattice.
```

FIG. 6.6 – Signature des nouvelles implémentations de treillis

Un foncteur permet ensuite de construire un treillis à partir d'une nouvelle implémentation quelconque :

```
Module Lattice_of_NewImplemLattice (N:NewImplemLattice) <: Lattice
 with Definition t := N.t
 with Definition eq := N.eq
 with Definition order := N.order
 with Definition join := N.join
 with Definition meet := N.meet
 with Definition bottom := N.bottom.

...

End Lattice_of_NewImplemLattice.
```

Exemple 9. *Étant donnés trois treillis* `L1`, `L2` *et* `L3` *nous pouvons définir le foncteur qui construit le treillis de la somme disjointe de ces trois treillis, en ajoutant un plus petit élément et un plus grand élément, comme le montre le diagramme de Hasse de la figure 6.7. Un tel treillis peut être rapidement construit par combinaison des précédents*

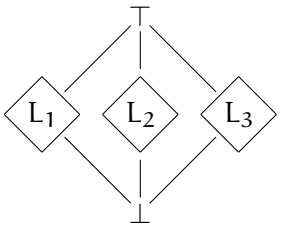


FIG. 6.7 – Diagramme de Hasse de la somme disjointe de trois treillis

foncteurs :

```
Module L <: Lattice := Lattice_of_PartialLattice(
    SumPartialLattice(
     SumPartialLattice(PartialLattice_of_Lattice(L1),
                       PartialLattice_of_Lattice(L2)),
     PartialLattice_of_Lattice(L3))).
```

Chaque treillis est transformé en treillis partiels. On construit ensuite la somme disjointe des trois treillis partiels obtenus en composant deux fois le foncteur de somme disjointe binaire. Le treillis partiel obtenu est ensuite complété avec des éléments $\bot$ et $\top$. La structure de treillis souhaitée est ainsi obtenue. Le type de base construit n'est cependant pas particulièrement agréable à utiliser pour programmer différentes fonctions abstraites, puisqu'il est de la forme $((L_1+L_2)+L_3)^{\top}_{\bot}$. Notre technique de reprogrammation de treillis permet d'implémenter le même treillis en prenant un type de base de la forme

```
  Inductive t : Set :=
     bot : t
   | in1 : L1 → t
   | in2 : L2 → t
```

```
| in3 : L3 → t
| top : t.
```

Le module de type `NewImplemLattice` *nécessaire à cette construction est facilement construit. L'injection utilisée dans ce module est dans ce cas particulier un isomorphisme.*

6.2 Construction de treillis vérifiant la condition de chaîne ascendante

Nous allons maintenant nous intéresser à la condition de chaîne ascendante qu'un treillis peut vérifier. Pour chacun des foncteurs présentés dans la section précédente, nous prouvons que si les treillis pris en argument vérifient la condition de chaîne ascendante alors le treillis construit la vérifie aussi. Ces preuves reposent sur des résultats généraux du prédicat d'accessibilité que nous rappelons dans la section 6.2.1. Nous présenterons ensuite deux de ces preuves, pour la construction, d'une part du produit de treillis, et d'autre part du treillis de fonctions.

6.2.1 Résultats généraux sur le prédicat d'accessibilité

Le résultat de base concernant l'accessibilité est bien entendu le principe d'induction qui lui est attaché.

Lemme 6.2. ***(Principe d'induction bien fondé)*** *Soit $\mathcal{P}$ une propriété sur un ensemble A et $\prec$ une relation sur A, si $\forall x \in \mathrm{Acc}_\prec,\ (\forall y \in A,\ y \prec x \ \Rightarrow\ \mathcal{P}(y)) \ \Rightarrow\ \mathcal{P}(x)$ alors $\forall x \in \mathrm{Acc}_\prec,\ \mathcal{P}(x)$.*

Les résultats suivants sont donnés par Paulson dans [Pau86] et présents dans la bibliothèque standard de Coq.

Lemme 6.3. *Pour toutes relations R_1, R_2 sur un ensemble A, si $R_1 \subseteq R_2$ alors $\mathrm{Acc}_{R_2} \subseteq \mathrm{Acc}_{R_1}$.*

Lemme 6.4. *Pour toute relation R_B sur un ensemble B, pour toute fonction $f : A \to B$ avec A un ensemble, la relation R_A définie sur A par*

$$R_A = \{(a_1, a_2) \mid (f(a_1), f(a_2)) \in R_B\}$$

vérifie $\mathrm{Acc}_{R_B} \subseteq \mathrm{Acc}_{R_A}$.

Ces deux lemmes peuvent être englobés par le résultat suivant, conséquence des énoncés précédents.

Lemme 6.5. *Pour toute relation R_A sur un ensemble A, R_B sur un ensemble B, pour toute fonction $f : A \to B$ vérifiant*

$$\forall (a_1, a_2) \in R_A, (f(a_1), f(a_2)) \in R_B$$

on a $\forall a \in A,\ f(a) \in \mathrm{Acc}_{R_B} \ \Rightarrow\ a \in \mathrm{Acc}_{R_A}$.

f est appelée *fonction de mesure.*

Lemme 6.6. *Pour toute relation* R *sur un ensemble* A, $\mathrm{Acc}_R = \mathrm{Acc}_{R^+}$ *avec* R^+ *la fermeture transitive de* R.

Lemme 6.7. ***(Produit lexicographique)*** *Pour toutes relations* R_A, R_B *sur les ensembles respectifs* A *et* B, *si* R_A *et* R_B *sont bien fondées, alors le* produit lexicographique R^{lex} *défini par*

$$R^{lex} = \{((a_1, b_1), (a_2, b_2)) \mid (a_1, a_2) \in R_A \ \vee \ (a_1 = a_2 \ \wedge (b_1, b_2) \in R_B)\}$$

est bien fondé lui aussi.

Ce résultat peut être quelque peu généralisé en remplaçant $a_1 = a_2$ par $a_1 \equiv a_2$ avec $\equiv$ une relation d'équivalence compatible à droite avec R_A ($\forall a_1, a_2, a_3 \in A, \ (a_1, a_2) \in R_A \wedge a_2 \equiv a_3 \Rightarrow (a_1, a_3) \in R$).

6.2.2 Produit de treillis vérifiant la condition de chaîne ascendante

La condition de chaîne ascendante ne concerne que le poset sous-jacent à un treillis. Nous nous concentrons donc sur cette structure. Nous prenons des posets munis d'une relation d'équivalence $\equiv$ pour laquelle l'ordre du poset est antisymétrique, comme dans nos définitions Coq des treillis. L'utilisation d'une relation d'équivalence demande parfois d'étendre les résultats de la librairie standard lorsque ceux-ci utilisent l'égalité standard.

Lemme 6.8. *Étant donnés deux posets* $(A, \equiv_A, \sqsubseteq_A)$ *et* $(B, \equiv_B, \sqsubseteq_B)$, *si* $\sqsupset_A$ *et* $\sqsupset_B$ *sont bien fondées, alors* $\sqsupset_{A\times B}$ *est bien fondée.*

Preuve : la preuve repose sur les résultats standards donnés en section 6.2.1. Il suffit en effet de remarquer que $\sqsupset_{A\times B}$ est une sous-relation de l'ordre lexicographique entre $\sqsupset_A$ et $\sqsupset_B$ (en prenant $\equiv_A$ comme relation d'équivalence sur A) :

$$\begin{array}{l}\forall a_1, a_2 \in A, \ \forall b_1, b_2 \in B, \\ \quad (a_1, b_1) \sqsupset_{A\times B} (a_2, b_2) \ \Rightarrow \ a_1 \sqsupset_A a_2 \ \vee (a_1 \equiv_A a_2 \wedge b_1 \sqsupset_B b_2)\end{array}$$

Nous pouvons alors conclure à l'aide des lemmes 6.3 et 6.7. □

6.2.3 Treillis de fonction et condition de chaîne ascendante

Dans cette partie, nous établissons que si le treillis `B` d'une implémentation de fonction de type `Func_FiniteSet_Lattice` vérifie la condition de chaîne ascendante, alors le treillis de fonction construit par le foncteur `FuncLattice` vérifie lui aussi la condition de chaîne ascendante. L'hypothèse faite sur le domaine `A.t` des fonctions est ici indispensable. Si `A.t` n'était pas fini, nous pourrions en effet construire une chaîne strictement croissante infinie.

Exemple 10. *Considérons les fonctions de* $\mathbb{N} \to \{\bot, \top\}$ *et la suite* $(f_n)_{n\in\mathbb{N}}$ *définie par*

$$\forall n, x \in \mathbb{N},\ f_n(x) = \begin{cases} \bot & \textit{si } x < n \\ \top & \textit{sinon} \end{cases}$$

Cette suite est strictement croissante.

Nous allons présenter dans un premier temps la signature `FiniteSet` et les foncteurs qui permettent de construire des ensembles finis. Nous aborderons ensuite la preuve de condition de chaîne ascendante.

6.2.3.1 Les ensembles finis

```
Module Type FiniteSet.

  Parameter t : Set.

  Parameter eq : t → t → Prop.
  Parameter eq_refl : ∀ x : t, eq x x.
  Parameter eq_sym : ∀ x y : t, eq x y → eq y x.
  Parameter eq_trans : ∀ x y z : t, eq x y → eq y z → eq x z.
  Parameter eq_dec : ∀ x y : t, {eq x y}+{¬ eq x y}.

  Parameter cardinal : Z.
  Parameter cardinal_positive : cardinal > 0.

  Parameter inject : t → Z.
  Parameter nat2t : Z → t.
  Parameter inject_bounded : ∀ x : t,
    0 <= (inject x) < cardinal.
  Parameter inject_nat2t : ∀ n : Z,
    0 <= n < cardinal → inject (nat2t n) = n.
  Parameter inject_injective : ∀ x y : t,
    inject x = inject y → eq x y.
  Parameter inject_comp_eq : ∀ x y : t,
    eq x y → inject x = inject y.

End FiniteSet.
```

Fig. 6.8 – La signature des ensembles finis non vides

La signature `FiniteSet` est présentée dans la figure 6.8. Elle contient un type de base `t`, une relation d'équivalence `eq` sur `t` et un entier relatif `cardinal` tel que `t` soit en bijection avec la partie $[\![0, \texttt{cardinal} - 1]\!]$ de $\mathbb{Z}$. Cette bijection est exprimée à l'aide d'une injection `inject` de `t` vers $[\![0, \texttt{cardinal} - 1]\!]$ et son inverse `nat2t`.

Afin de pouvoir construire des modules de signature `Finiteset`, nous proposons des foncteurs d'ensembles finis :

– le foncteur `ProdFiniteSet` permet de faire le produit de deux ensembles finis

– le foncteur `ListFiniteSet` permet de construire les listes de longueur inférieures à un paramètre k et dont les éléments prennent leurs valeurs dans un ensemble fini. Le paramètre k doit être encapsulé dans un module pour être passé en paramètre au foncteur.

Un module de base `Word` de signature `FiniteSet` représente les mots binaires d'au plus 32 bits (basée sur le type `positive` de Coq). Ce module est fréquemment utilisé dans nos formalisations d'analyses statiques pour nommer les différents composants d'un programme (nom de variable, nom de champs, nom de classes, ...).

6.2.3.2 Condition de chaîne ascendante sur les fonctions

Étant donnée une implémentation `F` de fonction sur un treillis `B`, la construction de la preuve de chaîne ascendante est basée sur l'utilisation du lemme 6.5 avec une mesure f envoyant les éléments du domaine `F.t` vers les fonctions de $\mathbb{N} \to$ `B.t`.

```
f := fun (x : F.t) ⇒ fun (n : nat) ⇒ get x (F.A.nat2t (Z_of_nat n))
```

La relation bien fondée utilisée sur $\mathbb{N} \to$ `B.t` est le produit lexicographique généralisé : étant donnés un ensemble A et une relation $\prec$ bien fondée sur A, la relation $\prec_n$ définie sur $\mathbb{N} \to A$ par

$$f_1 \prec_n f_2 \iff \exists k,\ k < n \wedge \forall i, i < n \Rightarrow f_1(i) = f_2(i) \wedge f_1(k) \prec f_2(k)$$

est bien fondée pour tout entier n. Un récurrence sur n permet de prouver ce résultat à partir de la bonne fondaison du produit lexicographique standard.

6.3 Construction de treillis avec opérateurs d'élargissement et de rétrécissement

L'utilisation d'opérateurs d'élargissement et de rétrécissement demande de prouver la propriété qui assure la terminaison des itérations de point fixe utilisant ces opérateurs. Comme nous l'avons présenté dans le chapitre 4, cette propriété est de la forme $\forall x, (x, x) \in \mathrm{Acc}_{\prec_\nabla}$, avec $\prec_\nabla$ la relation associée à l'opérateur d'élargissement ∇ (avec la définition standard ou non). Les opérateurs de rétrécissement demandent le même type de propriété.

Contrairement à la section précédente, les preuves de conservation de la propriété de terminaison par application des foncteurs de treillis ne sont plus des applications directes de résultats classiques. Nous allons présenter en détail le cas du produit, qui illustre bien les difficultés rencontrées.

Nous étudierons dans un premier temps le cas des opérateurs d'élargissement standard. Le lemme clé à établir est alors

Lemme 6.9. *Étant donné deux posets* $(A, \equiv_A, \sqsubseteq_A)$ *et* $(B, \equiv_B, \sqsubseteq_B)$*, et deux opérateurs binaires* ∇_A *et* ∇_B *respectivement sur* A *et* B*, si les relations* $\prec_{\nabla_A}$ *et* $\prec_{\nabla_B}$ *associées vérifient* $\forall a \in A, (a,a) \in \mathrm{Acc}_{\prec_{\nabla_A}}$ *et* $\forall b \in B, (b,b) \in \mathrm{Acc}_{\prec_{\nabla_B}}$ *alors l'opérateur* $\nabla_{A\times B}$ *défini par*

$$(a_1, b_1)\ \nabla_{A\times B}\ (a_2, b_2) = (a_1 \nabla_A a_2, b_1 \nabla_B b_2)$$

vérifie $\forall c \in A \times B, (c,c) \in \mathrm{Acc}_{\prec_{\nabla_{A\times B}}}$.

La notation $\prec_\nabla$ a été définie à la page 75 par

$$(x_1, y_1) \prec_\nabla (x_2, y_2) \quad \overset{\text{def}}{\Longleftrightarrow} \quad x_2 \sqsubseteq x_1 \ \wedge\ y_1 = y_2 \nabla x_1 \ \wedge\ y_1 \not\equiv y_2$$

Ce résultat demande une preuve trop technique pour être établi directement (en partie parce que la relation porte sur des couples). Nous pouvons cependant généraliser un peu le résultat. L'idée consiste à exprimer $\prec_{\nabla_{A\times B}}$ comme un produit lexicographique de $\prec_{\nabla_A}$ et $\prec_{\nabla_B}$. Le résultat à établir est alors de la forme

$$\forall a \in \mathrm{Acc}_{\lhd_A},\ \forall b \in \mathrm{Acc}_{\lhd_B},\ (a,b) \in \mathrm{Acc}_{\lhd^{lex}_{A\times B}}$$

avec $\lhd_A$ qui joue le rôle de $\prec_{\nabla_A}$ et $\lhd_B$ celui de $\prec_{\nabla_B}$. Nous allons maintenant prouver que ce résultat est faux pour des relations $\lhd_A$ et $\lhd_B$ quelconques. Nous proposerons ensuite une définition plus contraignante du produit lexicographique pour laquelle le résultat pourra être établi.

Lemme 6.10. *Soit* $a \in \mathrm{Acc}_{\lhd_A}$ *et* $b \in \mathrm{Acc}_{\lhd_B}$*, s'il existe* $b' \in B$ *tel que* $b' \notin \mathrm{Acc}_{\lhd_B}$ *et* $a' \in A$ *tel que* $a' \lhd_A a$ *alors* $(a,b) \notin \mathrm{Acc}_{\lhd^{lex}_{A\times B}}$.

Preuve : Supposons que $(a,b) \in \mathrm{Acc}_{\lhd^{lex}_{A\times B}}$ et montrons qu'alors $b' \in \mathrm{Acc}_{\lhd_B}$. Puisque $(a',b') \lhd^{lex}_{A\times B} (a,b)$ on a $(a',b') \in \mathrm{Acc}_{\lhd^{lex}_{A\times B}}$. Mais ceci implique justement $b' \in \mathrm{Acc}_{\lhd_B}$. En effet, si on considère la fonction $f : B \to (A \times B)$ définie par $f(x) = (a', x)$, on a bien

$$\forall b_1, b_2 \in B,\ b_1 \lhd_B b_2 \ \Rightarrow\ f(b_1) \lhd^{lex}_{A\times B} f(b_2)$$

donc on peut appliquer le lemme 6.5 avec $f(b') = (a', b')$ pour conclure. □

Le problème se situe ici dans la possibilité de prendre n'importe quel b' pour obtenir un prédécesseur (a', b') de (a, b). Le cas $a_1 \lhd_A a_2$ de la définition de $\lhd^{lex}_{A\times B}$ est donc trop souple, il faut nécessairement imposer une contrainte sur le domaine B pour pouvoir prouver un résultat de la forme de celui attendu. Cette contrainte s'exprime à l'aide d'une relation $\blacktriangleleft_B$ qui apparaît dans le cas $a_1 \lhd_A a_2$. Le nouveau produit lexicographique prend alors la forme

$$(a_1, b_1) \lhd^{\text{lex}} (a_2, b_2) \iff \begin{array}{c} (a_1 \lhd_A a_2 \wedge b_1 \blacktriangleleft_B b_2) \\ \vee \\ (a_1 = a_2 \ \wedge b_1 \lhd_B b_2) \end{array}$$

en imposant sur $\lhd_B$ et $\blacktriangleleft_B$ une propriété empêchant de construire n'importe quel b' comme précédemment : si $b_2 \blacktriangleleft_B b_1$ et $b_1 \in \mathrm{Acc}_{\lhd_B}$ alors b_2 doit rester dans $\mathrm{Acc}_{\lhd_B}$.

Nous allons prendre une condition suffisante assez simple (n'utilisant pas la notion d'accessibilité) mais malgré tout relativement générale :

$$\forall b_1, b_2, b_3 \in B, b_1 \lhd_B b_2 \wedge b_2 \blacktriangleleft_B b_3 \;\Rightarrow\; b_1 \lhd^+ b_3$$

Nous pouvons même rendre la définition symétrique et englober le cas de $\sqsubseteq_{A\times B}$ (où $a_1 = a_2$ avait été remplacé par $a_1 \equiv_A a_2$) en introduisant une relation $\blacktriangleleft_A$ vérifiant une propriété similaire à $\blacktriangleleft_B$. Ceci nous amène à définir *le produit lexicographique étendu.*

Définition 6.3.1. Produit lexicographique étendu.
Étant donnés deux paires de relations $\lhd_A$ *et* $\blacktriangleleft_A$ *sur un ensemble* A, $\lhd_B$ *et* $\blacktriangleleft_B$ *sur* B. *Le* produit lexicographique étendu *est la relation* $\lll^{lex(\lhd_A,\lhd_B,\blacktriangleleft_A,\blacktriangleleft_B)}$ *sur* $A \times B$ *définie par*

$$(a_1, b_1) \lll^{lex(\lhd_A,\lhd_B,\blacktriangleleft_A,\blacktriangleleft_B)} (a_2, b_2) \iff \begin{array}{c} (a_1 \lhd_A a_2 \;\wedge\; b_1 \blacktriangleleft_B b_2) \\ \vee \\ (a_1 \blacktriangleleft_A a_2 \;\wedge\; b_1 \lhd_B b_2) \end{array}$$

avec les conditions suivantes

$$\forall a_1, a_2, a_3 \in A,\; a_1 \lhd_A a_2 \;\wedge\; a_2 \blacktriangleleft_A a_3 \;\Rightarrow\; a_1 \lhd_A^+ a_3 \tag{6.1}$$
$$\forall b_1, b_2, b_3 \in B,\; b_1 \lhd_B b_2 \;\wedge\; b_2 \blacktriangleleft_B b_3 \;\Rightarrow\; b_1 \lhd_B^+ b_3 \tag{6.2}$$

Quand le contexte sera non ambiguë nous noterons plus simplement $\lll$ cette relation.
Nous pouvons d'ores et déjà proposer un exemple de produit lexicographique étendu.

Exemple 11.

$$(a_1, b_1) \sqsupset_{A\times B} (a_2, b_2) \iff \begin{array}{c} (a_1 \sqsupset_A a_2 \wedge b_1 \sqsupseteq b_2) \\ \vee \\ (a_1 \equiv_A a_2 \wedge b_1 \sqsupset b_2) \end{array}$$

et on a

$$\forall a_1, a_2, a_3 \in A,\; a_1 \sqsupset_A a_2 \;\wedge\; a_2 \equiv_A a_3 \;\Rightarrow\; a_1 \sqsupset_A a_3$$

et

$$\forall b_1, b_2, b_3 \in B,\; b_1 \sqsupset_B b_2 \;\wedge\; b_2 \sqsupseteq_B b_3 \;\Rightarrow\; b_1 \sqsupset_B^+ b_3$$

Donc $\sqsupset_{A\times B} = \lll^{lex(\sqsupset_A,\sqsupset_B,\equiv_A,\sqsupseteq_B)}$.

Nous en venons au théorème principal de cette section.

Théorème 6.3.1. *Si* $\lhd_A$, $\lhd_B$, $\blacktriangleleft_A$ *et* $\blacktriangleleft_B$ *vérifient les hypothèses de la définition 6.3.1 alors, pour tout* $a \in \mathrm{Acc}_{\lhd_A}$ *et* $b \in \mathrm{Acc}_{\lhd_B}$, $(a, b) \in \mathrm{Acc}_{\lll}$

Preuve : La forme de cet énoncé encourage à utiliser une induction bien fondée avec

$$\forall a \in A,\; \mathcal{P}(a) \stackrel{\text{def}}{=} \forall b \in \mathrm{Acc}_{\lhd_B},\; (a, b) \in \mathrm{Acc}_{\lll}$$

Mais l'hypothèse d'induction générée sera alors trop faible car uniquement utilisable avec un élément $b \in \mathrm{Acc}_{\lhd_B}$.

Nous renforçons donc tout naturellement le but courant en prouvant

$$\forall a \in \mathrm{Acc}_{\lhd_A^+},\ \forall b_1 \in \mathrm{Acc}_{\lhd_B},\ \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow\ (a, b_2) \in \mathrm{Acc}_{\lhd\!\!\blacktriangleleft} \tag{6.3}$$

avec $\blacktriangleleft_B^*$ la fermeture réflexive transitive de $\blacktriangleleft_B$. Ce résultat est suffisant pour prouver notre théorème puisque $\blacktriangleleft_B^*$ est réflexive et que d'après le lemme 6.6, $\mathrm{Acc}_{\lhd_A} = \mathrm{Acc}_{\lhd_A^+}$. Pour prouver (6.3), nous utilisons une induction bien fondée sur a et $\lhd_A^+$. Nous considérons un élément $a_1 \in \mathrm{Acc}_{\lhd_A^+}$ tel que

$$\begin{array}{l}\forall a_2 \in A,\ a_2 \lhd_A^+ a_1\ \Rightarrow \\ \quad \forall b_1 \in \mathrm{Acc}_{\lhd_B},\ \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow\ (a_2, b_2) \in \mathrm{Acc}_{\lhd\!\!\blacktriangleleft}\end{array} \tag{6.4}$$

et cherchons à prouver $\forall b_1 \in \mathrm{Acc}_{\lhd_B},\ \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow\ (a_1, b_2) \in \mathrm{Acc}_{\lhd}$. Là encore la preuve par induction sur b_1 serait prématurée, sans généraliser auparavant le résultat. Nous prouvons donc le résultat suivant

$$\begin{array}{l}\forall b_1 \in \mathrm{Acc}_{\lhd_B^+}, \\ \quad \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow \\ \quad \forall a_3 \in A,\ a_3 \blacktriangleleft_A^* a_1\ \Rightarrow\ (a_3, b_2) \in \mathrm{Acc}_{\lhd\!\!\blacktriangleleft}\end{array} \tag{6.5}$$

Ce résultat est suffisant car, d'une part $\mathrm{Acc}_{\lhd_B} = \mathrm{Acc}_{\lhd_B^+}$ et d'autre part $\blacktriangleleft_A^*$ est réflexif. Nous prouvons donc (6.5) par induction bien fondée sur b_1. Nous considérons ainsi un élément $b_1 \in \mathrm{Acc}_{\lhd_B^+}$ tel que

$$\begin{array}{l}\forall b_3 \in B,\ b_3 \lhd_B^+ b_1\ \Rightarrow \\ \quad \forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_3,\ \Rightarrow \\ \quad \forall a_3 \in A,\ a_3 \blacktriangleleft_A^* a_1\ \Rightarrow\ (a_3, b_2) \in \mathrm{Acc}_{\lhd\!\!\blacktriangleleft}\end{array} \tag{6.6}$$

et cherchons à prouver

$$\forall b_2 \in B,\ b_2 \blacktriangleleft_B^* b_1,\ \Rightarrow\ \forall a_3 \in A,\ a_3 \blacktriangleleft_A^* a_1\ \Rightarrow\ (a_3, b_2) \in \mathrm{Acc}_{\lhd\!\!\blacktriangleleft}$$

Nous supposons donc

$$b_1 \in \mathrm{Acc}_{\lhd_B^+} \tag{6.7}$$

$$b_2 \blacktriangleleft_B^* b_1 \tag{6.8}$$

$$a_3 \blacktriangleleft_A^* a_1 \tag{6.9}$$

et cherchons à établir $(a_3, b_2) \in \mathrm{Acc}_{\lhd\!\!\blacktriangleleft}$. Il nous faut pour cela considérer un prédécesseur (a_4, b_4) quelconque de (a_3, b_2)

$$(a_4, b_4) \lhd\!\!\blacktriangleleft (a_3, b_2) \tag{6.10}$$

et prouver que $(a_4, b_4) \in \mathrm{Acc}_{\lhd\!\!\blacktriangleleft}$. D'après la définition de $\lhd\!\!\blacktriangleleft$, deux cas résultent de l'hypothèse (6.10).

– Cas 1 :

$$a_4 \lhd_A a_3 \tag{6.11}$$
$$b_4 \blacktriangleleft_B b_2 \tag{6.12}$$

Nous pouvons alors utiliser l'hypothèse d'induction (6.4). Trois conditions doivent être vérifiées
– $a_4 \lhd_A a_1$: vrai grâce aux hypothèses (6.1), (6.9) et (6.11).
– $b_1 \in \mathrm{Acc}_{\lhd_B}$: vrai grâce à (6.7) et au résultat général $\mathrm{Acc}_{\lhd_B^+} = \mathrm{Acc}_{\lhd_B}$
– $b_4 \blacktriangleleft_B^* b_1$: vrai d'après (6.12) et (6.8)

– Cas 2 :

$$a_4 \blacktriangleleft_A a_3 \tag{6.13}$$
$$b_4 \lhd_B b_2 \tag{6.14}$$

Cette fois, nous pouvons utiliser l'hypothèse d'induction (6.6). Trois conditions doivent être vérifiées
– $b_4 \lhd_A^+ b_1$: vrai grâce aux hypothèses (6.2) , (6.8) et (6.14)
– $b_4 \blacktriangleleft_B^* b_4$: vrai par réflexivité de $\blacktriangleleft_B^*$
– $a_4 \blacktriangleleft_A^* a_1$: vrai grâce aux hypothèses (6.9) et (6.13)

Le résultat est donc établi. □

Grâce à ce théorème, nous pouvons maintenant établir la preuve du lemme 6.9.

Preuve du lemme 6.9 : Considérons le produit lexicographique étendu

$$\lll^{\mathrm{lex}(\prec_{\triangledown_A}, \prec_{\triangledown_B}, \preccurlyeq_{\triangledown_A}, \preccurlyeq_{\triangledown_B})}$$

sur $(A \times A) \times (B \times B)$ avec $\preccurlyeq_{\triangledown_A}$ et $\preccurlyeq_{\triangledown_B}$ définies par

$$(x_1, y_1) \preccurlyeq_{\triangledown_A} (x_2, y_2) \iff x_2 \sqsubseteq_A x_1 \ \wedge \ y_1 \equiv y_2$$

et

$$(x_1, y_1) \preccurlyeq_{\triangledown_B} (x_2, y_2) \iff x_2 \sqsubseteq_B x_1 \ \wedge \ y_1 \equiv_B y_2 \triangledown_B x_1$$

Il est facile de vérifier que les hypothèses (6.1) et (6.2) de la définition 6.3.1 sont vérifiées.

Il suffit alors d'utiliser le lemme 6.5 en prenant une mesure $f : (A \times B) \times (A \times B) \to (A \times A) \times (B \times B)$ définie par $f((a_1, b_1), (a_2, b_2)) = ((a_1, a_2), (b_1, b_2))$ entre la relation $\prec_{\triangledown_{A \times B}}$ sur $(A \times B) \times (A \times B)$ et $\lll^{\mathrm{lex}(\prec_{\triangledown_A}, \prec_{\triangledown_B}, \preccurlyeq_{\triangledown_A}, \preccurlyeq_{\triangledown_B})}$ pour conclure. □

Le théorème 6.3.1 permet de traiter de la même façon les opérateurs de rétrécissement ainsi que la version non standard des opérateurs d'élargissement et de rétrécissement.

La preuve concernant les treillis de fonctions est une nouvelle fois basée sur une version généralisée du produit lexicographique. Les mêmes arguments que dans la section 6.2.3.2 sont donc utilisés en remplaçant le produit lexicographique standard par notre produit lexicographique étendu.

6.4 Conclusion

Nous avons présenté dans ce chapitre une partie importante de notre travail de thèse. Comme nous l'avions expliqué dans les chapitres précédents, une analyse statique certifiée repose sur un treillis muni d'un critère de terminaison pour un algorithme itératif de calcul de point fixe. Afin de pouvoir rapidement développer des analyses, il était primordial de proposer une technique modulaire de construction de ces treillis.

Au final, chaque foncteur est proposé avec deux versions : une version qui combine les modules de type `Lattice` et une autre version qui combine les modules de type `LatticeSolve` (treillis munis d'un critère de terminaison). Pour cette dernière version, les preuves présentées dans les sections 6.2 et 6.3 sont utilisées. La technique de reprogrammation de treillis, présentée dans la section 6.1.5, permet de raffiner le type de base d'un treillis construit par composition de foncteurs. Cette technique permet aussi de conserver la preuve de terminaison d'un treillis de type `LatticeSolve`.

La contribution majeure de ce chapitre concerne les preuves de terminaison que nous avons développées dans notre librairie. Comme nous l'avions remarqué dans le chapitre 4, extraire une analyse statique qui termine n'est pas indispensable puisque l'on peut se contenter d'extraire un vérificateur de post-point fixe. Notre contribution concerne donc plus le domaine des assistants de preuve que celui de l'analyse statique certifiée. Nous montrons en effet qu'une bonne façon d'aborder des preuves complexes de terminaison consiste à modulariser leurs constructions. Le découpage de ce ces preuves en petits composants permet non seulement de simplifier la preuve finale, mais aussi les preuves futures que l'on peut être amené à faire, en arranger différemment les différents composants. Une telle construction modulaire de preuve a le mérite de mettre des preuves de terminaison complexes à la portée d'utilisateurs non experts : la combinaison de foncteurs est beaucoup plus facile que la manipulation du prédicat d'accessibilité !

Chapitre 7

Preuves modulaires à l'aide de concrétisations paramétrées

L'interprétation abstraite propose un cadre théorique permettant de construire et de prouver la correction des analyses statiques. Une des particularités de ce cadre réside dans ses capacités à proposer des constructions modulaires et réutilisables. Cette caractéristique est particulièrement prometteuse lorsqu'on désire construire une preuve mécanisée de la correction d'une analyse. De telles preuves doivent en effet être réalisées *in-extenso*, avec un niveau de détails particulièrement élevé. L'architecture globale de tels développements de preuves est donc un point critique, tout particulièrement lorsqu'on s'attaque à des analyses réalistes pour des langages de programmation réalistes.

Le slogan "prouver c'est programmer" prend encore une fois tout son sens ici : les leçons apprises dans le développement des logiciels s'appliquent à la construction de preuves formelles. En abstrayant certaines parties du développement avec des interfaces précises, on peut proposer des constructions modulaires plus simples à construire (car indépendantes des détails plus techniques des autres composants) et réutilisables. On simplifie ainsi le problème de preuve global en le ramenant à un ensemble de plusieurs problèmes plus simples. De plus, en implémentant de façon différente un composant, on peut modifier le développement global tout en conservant sa cohérence.

Nous nous intéressons dans ce chapitre à la construction modulaire de la preuve mécanisée d'une analyse statique pour un langage « réaliste ». Nous verrons que, dans ce contexte, certaines constructions modulaires classiques sont difficilement utilisables. Notre contribution consiste à proposer une extension élégante de la définition de ces constructions pour permettre leur usage sur des analyses réalistes. Le cadre d'application de notre technique est restreint à une certaine classe de langages de programmation et de techniques d'abstraction, mais nous montrons qu'il est néanmoins suffisant pour appréhender des langages comme le BYTECODE JAVA pour des analyses de références non-triviales. La contribution principale de ce travail se situe donc sur l'organisation modulaire de la preuve de correction d'une analyse statique pour la rendre plus facilement exprimable dans un assistant de preuve.

Nous allons tout d'abord présenter dans la section 7.1 un exemple classique de

construction modulaire issu de la théorie de l'interprétation abstraite. La section 7.2 présentera un autre outil théorique : les concrétisations paramétrées. Nous verrons que l'utilisation de telles concrétisations peut rendre difficile la modularité d'une preuve de correction. Nous proposons de résoudre ce problème grâce à la notion de foncteurs de concrétisation (section 7.3). Nous présenterons la méthode de preuve associée à l'utilisation de ces foncteurs puis nous conclurons dans la section 7.4.

7.1 Construction modulaire des connexions

Rappelons que le but d'une analyse statique est de calculer une approximation de la sémantique $[\![\mathsf{P}]\!]$ d'un programme P quelconque. Nous supposons que la sémantique concrète est exprimée dans un treillis de la forme $(\mathcal{D}, \subseteq, \sqcup, \sqcap)$. L'approximation considérée est à valeurs dans un domaine abstrait $\mathcal{D}^\sharp$ muni d'une structure de treillis $\left(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp\right)$. La correction de l'approximation est spécifiée par une fonction de concrétisation γ, morphisme d'intersection, appartenant à $\mathcal{D}^\sharp \to \mathcal{D}$. Tous ces éléments forment ce que nous appellerons une connexion (en référence aux connexions de Galois). Nous noterons une telle connexion de la façon suivante :

$$(\mathcal{D}, \subseteq, \sqcup, \sqcap) \xleftarrow{\gamma} \left(\mathcal{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp\right)$$

Pour abréger, nous nous permettrons parfois la notation plus synthétique $\mathcal{D} \xleftarrow{\gamma} \mathcal{D}^\sharp$.

La théorie de l'interprétation abstraite nous apprend comment composer certaines connexions à partir d'autres. Ainsi une abstraction des environnements (fonctions partielles[1] entre variables et valeurs) d'un langage de programmation

$$(\mathcal{P}(\mathrm{Var} \rightharpoonup \mathrm{Val}), \subseteq, \cup, \cap) \xleftarrow{\gamma^{\mathrm{Env}}} \left(\mathrm{Env}^\sharp, \sqsubseteq_{\mathrm{Env}}, \sqcup_{\mathrm{Env}}, \sqcap_{\mathrm{Env}}\right)$$

peut être paramétrée par une abstraction quelconque des valeurs

$$(\mathcal{P}(\mathrm{Val}), \subseteq, \cup, \cap) \xleftarrow{\gamma^{\mathrm{Val}}} \left(\mathrm{Val}^\sharp, \sqsubseteq_{\mathrm{Val}}, \sqcup_{\mathrm{Val}}, \sqcap_{\mathrm{Val}}\right)$$

Formellement, une telle abstraction d'environnement est définie par

Définition 7.1.1. Connexion générique d'environnement.
Une connexion générique d'environnement *est une fonctionnelle qui associe à toute connexion* $(\mathcal{P}(Val), \subseteq, \cup, \cap) \xleftarrow{\gamma^{Val}} \left(Val^\sharp, \sqsubseteq_{Val}, \sqcup_{Val}, \sqcap_{Val}\right)$
un quadruplet $\left(\left(Env^\sharp, \sqsubseteq_{Env}, \sqcup_{Env}, \sqcap_{Env}\right), \gamma_{Env}, \mathrm{get}^\sharp, \mathrm{subst}^\sharp\right)$ *avec*

- $\left(Env^\sharp, \sqsubseteq_{Env}, \sqcup_{Env}, \sqcap_{Env}\right)$ *un treillis*
- $\gamma \in Env^\sharp \to \mathcal{P}(Var \to Val)$ *une concrétisation morphisme d'intersection entre* $Env^\sharp$ *et les environnements*

[1] Nous utiliserons le symbole $\rightharpoonup$ pour désigner les fonctions partielles.

- $\text{get}^\sharp \in \mathit{Env}^\sharp \times \mathit{Var} \to \mathit{Val}^\sharp$ *une approximation correcte de la fonction d'accès à une valeur d'une variable*

$$\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \forall x \in \mathit{Var},\quad \left\{ \rho(x) \;\middle|\; \rho \in \gamma(\rho^\sharp) \right\} \subseteq \gamma^{\mathit{Val}}(\text{get}^\sharp(\rho^\sharp, x))$$

- $\text{subst}^\sharp \in \mathit{Env}^\sharp \times \mathit{Var} \times \mathit{Val}^\sharp \to \mathit{Env}^\sharp$ *une approximation correcte de la fonction de substitution d'une valeur dans une variable*

$$\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \forall x \in \mathit{Var},\ \forall \nu^\sharp \in \mathit{Val}^\sharp,\quad \left\{ \rho[x \mapsto \nu] \;\middle|\; \begin{array}{l} \rho \in \gamma(\rho^\sharp) \\ \nu \in \gamma^{\mathit{Val}}(\nu^\sharp) \end{array} \right\} \subseteq \gamma(\text{subst}^\sharp(\rho^\sharp, x, \nu^\sharp))$$

Une connexion générique d'environnement construit donc un treillis abstrait, une fonction de concrétisation et deux approximations correctes des fonctions primitives pour manipuler les environnements, pour une abstraction des valeurs donnée. Un exemple classique de telle connexion générique est l'abstraction non relationnelle standard.

Lemme 7.1. *La fonctionnelle qui associe à toute connexion*

$$(\mathcal{P}(\mathit{Val}), \subseteq, \cup, \cap) \xleftarrow{\gamma^{\mathit{Val}}} \left(\mathit{Val}^\sharp, \sqsubseteq_{\mathit{Val}}, \sqcup_{\mathit{Val}}, \sqcap_{\mathit{Val}}\right)$$

le quadruplet $\left(\left(\mathit{Env}^\sharp, \sqsubseteq_{\mathit{Env}}, \sqcup_{\mathit{Env}}, \sqcap_{\mathit{Env}}\right), \gamma_{\mathit{Env}}, \text{get}^\sharp, \text{subst}^\sharp\right)$ *avec*

- $\mathit{Env}^\sharp = \mathit{Var} \to \mathit{Val}^\sharp$
- $\sqsubseteq_{\mathit{Env}} = \dot{\sqsubseteq}_{\mathit{Val}}$
- $\sqcup_{\mathit{Env}} = \lambda\rho_1^\sharp\lambda\rho_2^\sharp\lambda x.\rho_1^\sharp(x) \sqcup_{\mathit{Val}} \rho_2^\sharp(x)$,
- $\sqcap_{\mathit{Env}} = \lambda\rho_1^\sharp\lambda\rho_2^\sharp\lambda x.\rho_1^\sharp(x) \sqcap_{\mathit{Val}} \rho_2^\sharp(x)$,
- $\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \gamma_{\mathit{Env}}(\rho^\sharp) = \left\{ \rho \;\middle|\; \forall x \in \mathit{Var},\ \rho(x) \in \gamma^{\mathit{Val}}(\rho^\sharp(x)) \right\}$
- $\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \forall x \in \mathit{Var},\ \text{get}^\sharp(\rho^\sharp, x) = \rho^\sharp(x)$
- $\forall \rho^\sharp \in \mathit{Env}^\sharp,\ \forall x \in \mathit{Var},\ \forall \nu^\sharp \in \mathit{Val}^\sharp,\ \text{subst}^\sharp(\rho^\sharp, x, \nu^\sharp) = \rho^\sharp[x \mapsto \nu^\sharp]$

est une connexion générique d'environnement.

La construction est bien paramétrée par n'importe quelle connexion sur les valeurs du langage. Au final, on obtient donc deux « boîtes noires » : une boîte qui contient l'abstraction des valeurs et une autre qui contient l'abstraction des environnements. Tout l'enjeu de ce travail est de pouvoir utiliser ces boîtes dans la preuve globale de l'analyse, indépendamment de leur contenu, en se basant uniquement sur leurs interfaces. Cette modularisation est très importante dans le cadre d'un développement vérifié par ordinateur car

- les preuves pour construire l'abstraction des environnements ne dépendent pas de la nature (plus ou moins complexe) de γ^{Val} et sont par conséquent plus simples,
- il est facile de proposer plusieurs abstractions de valeur et d'en déduire les abstractions d'environnement sans refaire de preuve,
- si dans la construction globale de l'analyse on n'utilise pas plus de propriétés sur les environnements abstraits que celles imposées dans la définition d'une abstraction d'environnement (on n'ouvre pas la boîte), on pourra proposer plusieurs implémentations sans compromettre (et donc avoir à revérifier) la validité globale de l'analyse.

Cette construction nécessite cependant d'avoir des abstractions de la forme $\gamma^{\mathrm{Val}} \in \mathrm{Val}^\sharp \to \mathcal{P}(\mathrm{Val})$: une valeur abstraite doit exprimer des propriétés sur les valeurs concrètes. Ce qui, à première vue, n'apparaît pas comme une limitation se révèle pourtant problématique lorsque γ^{Val} a besoin d'un « contexte » pour être exprimé formellement.

Exemple 12. *Si les valeurs du langage sont des références à des objets alloués dynamiquement dans un tas, une abstraction de ces références par l'ensemble des noms de classe des objets associés, sera relative à un tas.*

$$\forall \mathsf{s} \in \mathcal{P}(\mathit{Class}),\ \gamma(\mathsf{s}) = \left\{(\mathsf{h}, \mathrm{loc}) \ \middle|\ \mathrm{loc} \in \mathrm{dom}(\mathsf{h}) \ \wedge\ \mathrm{class}(\mathsf{h}(\mathrm{loc})) \in \mathsf{s} \right\} \subseteq \mathit{Tas} \times \mathit{Val}$$

où $\mathit{Tas} \stackrel{def}{=} \mathit{Val} \rightharpoonup \mathit{Objet}$. *Ce type de concrétisation est généralement noté sous une forme paramétrée plus élégante*

$$\begin{array}{l}\forall \mathsf{h} \in \mathit{Tas},\ \forall \mathsf{s} \in \mathcal{P}(\mathit{Class}),\\ \quad \gamma_{\mathsf{h}}(\mathsf{s}) = \left\{\mathrm{loc} \ \middle|\ \mathrm{loc} \in \mathrm{dom}(\mathsf{h}) \ \wedge\ \mathrm{class}(\mathsf{h}(\mathrm{loc})) \in \mathsf{s} \right\} \subseteq \mathit{Tas} \times \mathit{Val}\end{array}$$

Nous allons maintenant présenter formellement ce type de concrétisations et montrer comment elles s'utilisent dans une preuve de correction.

7.2 Concrétisations paramétrées

Les concrétisations que nous étudions ici dépendent d'un contexte. Chaque valeur abstraite exprime ainsi une relation entre un objet concret et un élément du contexte. Nous nous intéressons donc aux connexions de la forme

$$(\mathcal{P}(\mathsf{C} \times \mathsf{D}), \subseteq, \cup, \cap) \stackrel{\gamma}{\leftarrow} (\mathsf{D}^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp)$$

avec C le domaine du contexte.

Exemple 13. *On retrouve le même type de concrétisation que dans l'exemple 12 lorsqu'on abstrait les références par la super-classe de tous les objets associés (c'est l'abstraction prise dans le vérificateur de bytecode Java)*

$$\forall \tau \in \mathit{Class},\ \gamma(\tau) = \left\{(\mathsf{h}, \mathrm{loc}) \ \middle|\ \mathrm{loc} \in \mathrm{dom}(\mathsf{h}) \ \wedge\ \mathrm{class}(\mathsf{h}(\mathrm{loc})) \prec_{\mathsf{P}} \tau \right\} \subseteq \mathit{Tas} \times \mathit{Val}$$

où $\prec_{\mathsf{P}}$ *est la relation de sous-typage associée à la hiérarchie de classe du programme* P.

Exemple 14. *Une abstraction plus précise que l'abstraction par ensemble de classes peut être obtenue en abstrayant par points de création [JM79, RMR01]. La définition formelle de la fonction de concrétisation est alors relative à une trace partielle d'exécution.*

Une trace partielle est une suite non vide $< pc_0, \mathsf{m}_0 > :: \cdots :: < pc_n, \mathsf{m}_n >$ *d'états, chaque état étant constitué d'un point de programme* pc_i *et d'une mémoire* m_i. *Si l'instruction située en un point de programme* pc *est une création d'objet de classe* cl *(événement noté* $\mathrm{instrAt}(pc) = \mathtt{new}\ cl$*), une nouvelle adresse* $\mathrm{newObject}(cl, \mathsf{m})$ *est allouée dans la mémoire* m *pour y stocker un objet de classe* cl.

La concrétisation associée à l'abstraction par points de création est alors

$$\forall s \in \mathcal{P}(\mathbb{P}),$$
$$\gamma(s) = \left\{ (< pc_0, m_0 > :: \cdots :: < pc_n, m_n >, \mathrm{loc}) \;\middle|\; \begin{array}{l} \exists k \in \{0, \ldots, n\}, \\ \quad pc_k \in s \\ \quad \mathrm{instrAt}(pc_k) = \texttt{new } cl \\ \quad \mathrm{newObject}(cl, m_k) = \mathrm{loc} \end{array} \right\} \subseteq \mathit{Trace} \times \mathit{Val}$$

Ce type de concrétisation peut être représenté sous une forme **paramétrée** équivalente. Nous noterons γ^{param} la fonction de $C \to D^\sharp \to \mathcal{P}(D)$ définie par

$$\forall c \in C,\ \forall d^\sharp \in D^\sharp,\ \gamma_c^{\mathrm{param}}(d^\sharp) = \{d \mid (c, d) \in \gamma(d^\sharp)\}$$

Nous omettrons la notation $\cdot^{\mathrm{param}}$ quand le contexte permettra de le faire sans ambiguïté.

7.2.1 Utilisation d'une connexion générique avec une concrétisation paramétrée

La version paramétrée de ces concrétisations permet « d'oublier » l'aspect relationnel puisque γ_c agit comme une concrétisation classique entre $\mathcal{P}(D)$ et $D^\sharp$. On retrouve ainsi le cadre d'application des constructions modulaires de la section précédente : une abstraction paramétrée $\mathcal{P}(\mathrm{Val}) \xleftarrow{\gamma_c^{\mathrm{Val}}} \mathrm{Val}^\sharp$ (avec c un élément fixé de C) peut être utilisée pour instancier une connexion générique d'environnement. Nous obtenons alors un quadruplet $\left(\left(\mathrm{Env}^\sharp, \sqsubseteq_{\mathrm{Env}}, \sqcup_{\mathrm{Env}}, \sqcap_{\mathrm{Env}}\right), \gamma_c^{\mathrm{Env}}, \mathrm{get}^\sharp, \mathrm{subst}^\sharp\right)$ avec $\mathrm{get}^\sharp$ vérifiant, par exemple

$$\forall c \in C,\ \forall \rho^\sharp \in \mathrm{Env}^\sharp,\ \forall x \in \mathrm{Var}, \quad \left\{ \rho(x) \;\middle|\; \rho \in \gamma_c^{\mathrm{Env}}(\rho^\sharp) \right\} \subseteq \gamma_c^{\mathrm{Val}}(\mathrm{get}^\sharp(\rho^\sharp, x))$$

Une connexion générique d'environnement est donc capable d'utiliser une concrétisation paramétrée pour produire une concrétisation d'environnement paramétrée avec ces opérateurs abstraits de base $\mathrm{get}^\sharp$ et $\mathrm{subst}^\sharp$. Il faut cependant remarquer que les propriétés de correction vérifiées par ces opérateurs sont relatives à un même contexte c. Comme nous allons le voir maintenant, il s'agira d'une limitation majeure pour effectuer la preuve de correction des fonctions de transfert de façon modulaire.

7.2.2 Prouver la correction de fonctions de transfert abstraites

La difficulté d'utilisation des concrétisations paramétrées apparaît lors de la preuve de correction des fonctions de transfert abstraites (correspondant à l'abstraction de chaque instruction d'un langage de programmation). Nous allons maintenant exposer ce point à l'aide d'un exemple de langage avec des variables et des allocations dynamiques. L'état mémoire d'un tel langage est de la forme $\mathrm{Mem} \stackrel{\mathrm{def}}{=} \mathrm{Tas} \times \mathrm{Env}$ avec $\mathrm{Tas} \stackrel{\mathrm{def}}{=} \mathrm{Val} \rightharpoonup \mathrm{Objet}$, $\mathrm{Env} \stackrel{\mathrm{def}}{=} \mathrm{Var} \rightharpoonup \mathrm{Val}$ et Val le domaine des valeurs du langage, réduit ici (pour simplifier le discours) à des entrées du tas.

Puisque la mémoire est séparée en deux structures différentes, il est naturel de l'abstraire à l'aide de deux objets abstraits distincts. Considérons donc une première connexion $\mathcal{P}(\text{Tas}) \xleftarrow{\gamma^{\text{Tas}}} \text{Tas}^\sharp$ quelconque. Pour les environnements de variables, supposons que l'abstraction des valeurs ait nécessité une paramétrisation de la fonction de concrétisation : l'abstraction est alors de la forme

$$\left(\mathcal{P}(\text{Env}) \xleftarrow{\gamma^{\text{Env}}_{\mathsf{h}}} \text{Env}^\sharp\right)_{\mathsf{h}\in\text{Tas}}$$

Au niveau global, la concrétisation d'un couple $(\mathsf{h}^\sharp, \rho^\sharp)$ de valeurs abstraites sera

$$\gamma\left(\mathsf{h}^\sharp, \rho^\sharp\right) = \left\{(\mathsf{h}, \rho) \;\middle|\; \begin{array}{l} \mathsf{h} \in \gamma^{\text{Tas}}(\mathsf{h}^\sharp) \\ \rho \in \gamma^{\text{Env}}_{\mathsf{h}}(\rho^\sharp) \end{array}\right\} \subseteq \text{Tas} \times \text{Env}$$

La paramétrisation de la concrétisation fait donc apparaître une notion de produit dépendant dans le domaine concret.

Au niveau de la mémoire, toute fonction de transfert sera de la forme

$$\begin{array}{rccc} \mathcal{F} : & \text{Tas} \times \text{Env} & \to & \text{Tas} \times \text{Env} \\ & (\mathsf{h}, \rho) & \mapsto & (\mathsf{f}(\mathsf{h}, \rho), \mathsf{g}(\mathsf{h}, \rho)) \end{array}$$

Pour abstraire correctement une fonction de transfert, il faut donc proposer une fonction $\mathcal{F}^\sharp$ de la forme

$$\begin{array}{rccc} \mathcal{F}^\sharp : & \text{Tas}^\sharp \times \text{Env}^\sharp & \to & \text{Tas}^\sharp \times \text{Env}^\sharp \\ & (\mathsf{h}^\sharp, \rho^\sharp) & \mapsto & \left(\mathsf{f}^\sharp(\mathsf{h}^\sharp, \rho^\sharp), \mathsf{g}^\sharp(\mathsf{h}^\sharp, \rho^\sharp)\right) \end{array}$$

vérifiant le critère de correction « classique »

$$\forall(\mathsf{h}^\sharp, \rho^\sharp) \in \text{Tas}^\sharp \times \text{Env}^\sharp,$$
$$\left\{(\mathsf{f}(\mathsf{h}, \rho), \mathsf{g}(\mathsf{h}, \rho)) \;\middle|\; \begin{array}{l} \mathsf{h} \in \gamma^{\text{Tas}}(\mathsf{h}^\sharp) \\ \rho \in \gamma^{\text{Env}}_{\mathsf{h}}(\rho^\sharp) \end{array}\right\} \subseteq \left\{(\mathsf{h}', \rho') \;\middle|\; \begin{array}{l} \mathsf{h}' \in \gamma^{\text{Tas}}(\mathsf{f}^\sharp(\mathsf{h}^\sharp, \rho^\sharp)) \\ \rho' \in \gamma^{\text{Env}}_{\mathsf{h}'}(\mathsf{g}^\sharp(\mathsf{h}^\sharp, \rho^\sharp)) \end{array}\right\}$$

Ce critère peut être réduit de manière équivalente à la conjonction des deux propriétés suivantes

$$\begin{array}{l} \forall(\mathsf{h}^\sharp, \rho^\sharp) \in \text{Tas}^\sharp \times \text{Env}^\sharp, \\ \forall(\mathsf{h}, \rho) \in \gamma^{\text{Tas}}(\mathsf{h}^\sharp) \times \gamma^{\text{Env}}_{\mathsf{h}}(\rho^\sharp), \quad \mathsf{f}(\mathsf{h}, \rho) \in \gamma^{\text{Tas}}(\mathsf{f}^\sharp(\mathsf{h}^\sharp, \rho^\sharp)) \end{array} \tag{7.1}$$

$$\begin{array}{l} \forall(\mathsf{h}^\sharp, \rho^\sharp) \in \text{Tas}^\sharp \times \text{Env}^\sharp, \\ \forall(\mathsf{h}, \rho) \in \gamma^{\text{Tas}}(\mathsf{h}^\sharp) \times \gamma^{\text{Env}}_{\mathsf{h}}(\rho^\sharp), \quad \mathsf{g}(\mathsf{h}, \rho) \in \gamma^{\text{Env}}_{\mathsf{f}(\mathsf{h},\rho)}(\mathsf{g}^\sharp(\mathsf{h}^\sharp, \rho^\sharp)) \end{array} \tag{7.2}$$

Nous touchons alors un point important de ce travail : le critère (7.2) est problématique car il contient deux instances $\gamma^{\text{Env}}_{\mathsf{h}}$ et $\gamma^{\text{Env}}_{\mathsf{f}(\mathsf{h},\rho)}$ a priori distinctes de la concrétisation d'environnement de variable. Or, les propriétés assurées par le constructeur d'abstraction d'environnement ne sont relatives qu'à une seule fonction de concrétisation (et donc relative à un seul contexte à la fois). On ne peut donc pas espérer construire $\mathsf{g}^\sharp$ et prouver sa correction par simple composition des propriétés assurées par l'abstraction d'environnement.

Il serait cependant souhaitable de pouvoir toujours utiliser des constructeurs génériques. La solution que nous proposons de suivre est basée sur une hypothèse forte CHANGER[2] faite sur f et γ^{Env}. Nous allons réduire la preuve de (7.2) à deux résultats, l'un portant sur f, l'autre sur g :

$$\forall (\mathsf{h}, \rho) \in \mathrm{Tas} \times \mathrm{Env},\ \gamma^{\mathrm{Env}}_{\mathsf{h}} \dot{\subseteq} \gamma^{\mathrm{Env}}_{\mathsf{f}(\mathsf{h},\rho)} \tag{7.3}$$

$$\begin{array}{l} \forall (\mathsf{h}^\sharp, \rho^\sharp) \in \mathrm{Tas}^\sharp \times \mathrm{Env}^\sharp, \\ \forall (\mathsf{h}, \rho) \in \gamma^{\mathrm{Tas}}(\mathsf{h}^\sharp) \times \gamma^{\mathrm{Env}}_{\mathsf{h}}(\rho^\sharp), \quad \mathsf{g}(\mathsf{h}, \rho) \in \gamma^{\mathrm{Env}}_{\mathsf{h}}(\mathsf{g}^\sharp(\mathsf{h}^\sharp, \rho^\sharp)) \end{array} \tag{7.4}$$

Le critère (7.4) ne contient plus qu'une seule instance $\gamma^{\mathrm{Env}}_{\mathsf{h}}$ de la concrétisation d'environnement (contrairement au critère (7.2)). On peut donc à nouveau espérer pouvoir utiliser les propriétés des opérateurs $\mathrm{get}^\sharp$ et $\mathrm{subst}^\sharp$ définis dans les abstractions d'environnement.

Le critère (7.3) semble cependant reprendre les mêmes défauts que (7.2). Nous allons maintenant proposer une variante de la définition d'abstraction d'environnement qui nous permettra d'établir 7.3 sans avoir à connaître l'abstraction exacte prise pour les environnements ni faire apparaître la notion de contexte dans l'interface des abstractions d'environnement.

7.3 Foncteurs (monotones) de concrétisation

Le changement que nous proposons de faire dans la définition des connexions génériques est basé sur la notion de foncteurs de concrétisation : des opérateurs qui transforment des concrétisations en d'autres concrétisations.

7.3.1 Exemple et définition

Un exemple de tel foncteur a déjà été esquissé dans le lemme 7.1.

$$\begin{array}{rccc} \Gamma : & \left((\mathrm{Val}^\sharp, \sqcap_{\mathrm{Val}}) \xrightarrow{\sqcap} (\mathcal{P}(\mathrm{Val}), \cap)\right) & \to & \left((\mathrm{Env}^\sharp, \sqcap_{\mathrm{Env}}) \xrightarrow{\sqcap} (\mathcal{P}(\mathrm{Env}), \cap)\right) \\ & \gamma^{\mathrm{Val}} & \mapsto & \rho^\sharp \mapsto \left\{\rho \;\middle|\; \forall \mathsf{x} \in \mathrm{Var},\ \rho(\mathsf{x}) \in \gamma(\rho^\sharp(\mathsf{x}))\right\} \end{array}$$

$\xrightarrow{\sqcap}$ désigne ici les fonctions morphismes d'intersection. Ce genre de foncteur est sous-jacent à la plupart des constructeurs de connexion paramétrés par d'autres connexions que l'on peut trouver dans la littérature sur l'interprétation abstraite. Nous allons cependant nous intéresser à une propriété qui est rarement mise en avant : leur monotonie.

Définition 7.3.1. Foncteur de concrétisation.
Étant donnés quatre treillis $(\mathsf{A}, \sqsubseteq_{\mathsf{A}}, \sqcup_{\mathsf{A}}, \sqcap_{\mathsf{A}})$, $\left(\mathsf{A}^\sharp, \sqsubseteq_{\mathsf{A}^\sharp}, \sqcup^\sharp_{\mathsf{A}}, \sqcap^\sharp_{\mathsf{A}}\right)$, $(\mathsf{B}, \sqsubseteq_{\mathsf{B}}, \sqcup_{\mathsf{B}}, \sqcap_{\mathsf{B}})$ *et* $\left(\mathsf{B}^\sharp, \sqsubseteq_{\mathsf{B}^\sharp}, \sqcup^\sharp_{\mathsf{B}}, \sqcap^\sharp_{\mathsf{B}}\right)$, *un* foncteur de concrétisation *est un opérateur* Γ *appartenant à*

$$\left(\left(\mathsf{A}^\sharp, \sqcap_{\mathsf{A}^\sharp}\right) \xrightarrow{\sqcap} (\mathsf{A}, \sqcap_{\mathsf{A}})\right) \to \left(\left(\mathsf{B}^\sharp, \sqsubseteq_{\mathsf{B}^\sharp}\right) \xrightarrow{\sqcap} (\mathsf{B}, \sqsubseteq_{\mathsf{B}})\right)$$

[2] Nous verrons en section 7.3.3 que cette hypothèse est néanmoins vérifiée dans plusieurs cas d'étude réalistes.

qui vérifie la propriété de monotonie suivante

$$\forall \gamma_1, \gamma_2 \in \left(\left(A^\sharp, \sqcap_{A^\sharp}\right) \xrightarrow{\sqcap} (A, \sqsubseteq_A)\right), \quad \gamma_1 \mathrel{\dot{\sqsubseteq}_A} \gamma_2 \quad \Rightarrow \quad \Gamma(\gamma_1) \mathrel{\dot{\sqsubseteq}_B} \Gamma(\gamma_2)$$

Un foncteur de concrétisation conserve donc la précision relative de deux concrétisations. Il s'agit d'une propriété assez naturelle qui est vérifiée par de nombreux foncteurs de concrétisations de la littérature (la construction générique d'abstractions faiblement relationnelles proposée par Antoine Miné [Min04] en est un bon exemple). Autant que nous le sachions, cette propriété n'avait jamais été explicitement utilisée auparavant.

Nous allons utiliser cette notion de foncteur pour proposer une nouvelle définition des connexions génériques d'environnement.

Définition 7.3.2. Connexion générique d'environnement (nouvelle version). *Une* connexion générique d'environnement *est une fonctionnelle qui associe à tout treillis* $\left(Val^\sharp, \sqsubseteq_{Val}, \sqcup_{Val}, \sqcap_{Val}\right)$ *un quadruplet* $\left(\left(Env^\sharp, \sqsubseteq_{Env}, \sqcup_{Env}, \sqcap_{Env}\right), \Gamma^{Env}, \mathrm{get}^\sharp, \mathrm{subst}^\sharp\right)$ *avec*

- $\left(Env^\sharp, \sqsubseteq_{Env}, \sqcup_{Env}, \sqcap_{Env}\right)$ *un treillis,*
- $\Gamma \in \left(Val^\sharp \xrightarrow{\sqcap} \mathcal{P}(Val)\right) \to \left(Env^\sharp \xrightarrow{\sqcap} \mathcal{P}(Var \to Val)\right)$ *un foncteur* **monotone** *de concrétisation*
- $\mathrm{get}^\sharp \in Env^\sharp \times Var \to Val^\sharp$ *une approximation correcte de la fonction d'accès à une valeur d'une variable*

$$\begin{array}{l}\forall \gamma \in \left(Val^\sharp \xrightarrow{\sqcap} \mathcal{P}(Val)\right), \\ \forall \rho^\sharp \in Env^\sharp,\ \forall x \in Var, \quad \left\{\rho(x) \;\middle|\; \rho \in \Gamma^{Env}(\gamma)\,(Env^\sharp)\right\} \subseteq \gamma(\mathrm{get}^\sharp(\rho^\sharp, x))\end{array}$$

- $\mathrm{subst}^\sharp \in Env^\sharp \times Var \times Val^\sharp \to Env^\sharp$ *une approximation correcte de la fonction de substitution d'une valeur dans une variable*

$$\begin{array}{l}\forall \gamma \in \left(Val^\sharp \xrightarrow{\sqcap} \mathcal{P}(Val)\right), \\ \forall \rho^\sharp \in \rho^\sharp,\ \forall x \in Var,\ \forall v^\sharp \in Val^\sharp, \\ \qquad \left\{\rho[x \mapsto v] \;\middle|\; \begin{array}{l}\rho \in \Gamma^{Env}(\gamma)\,(\rho^\sharp) \\ v \in \gamma(v^\sharp)\end{array}\right\} \subseteq \Gamma^{Env}(\gamma)\,(\mathrm{subst}^\sharp(\rho^\sharp, x, v^\sharp))\end{array}$$

La modification que nous apportons ici se situe au niveau de la fonction de concrétisation qui n'est plus fixée mais paramétrée par n'importe quelle fonction de concrétisation sur les valeurs. En ce qui concerne les opérateurs $\mathrm{get}^\sharp$ et $\mathrm{subst}^\sharp$, la quantification sur toute concrétisation de valeur n'apporte pas de preuve supplémentaire puisque dans la version précédente, la concrétisation γ_{Val} était déjà quelconque. On peut donc affirmer que cette nouvelle interface n'est pas une restriction forte vis-à-vis de l'ancienne : seule la propriété de monotonie de Γ^{Env} est vraiment ajoutée, mais c'est une propriété couramment vérifiée par les foncteurs de concrétisations standards.

Nous allons maintenant expliquer pourquoi cette interface est suffisamment riche pour pouvoir être utilisée dans la preuve du critère 7.3.

7.3.2 Utiliser la propriété fonctorielle dans les preuves

Avec notre nouvelle définition de connexion générique d'environnement, la concrétisation γ^{Env} utilisée dans l'exemple de la section 7.2 est maintenant de la forme

$$\gamma^{\mathrm{Env}} = \Gamma^{\mathrm{Env}}\left(\gamma^{\mathrm{Val}}\right)$$

Le critère 7.3 peut ainsi être réduit en une propriété sur γ^{Val}.

Lemme 7.2. *Si* $\gamma^{Env} = \Gamma^{Env}\left(\gamma^{Val}\right)$ *avec* Γ^{Env} *un foncteur de concrétisation, alors le critère*

$$\forall(\mathsf{h},\rho) \in Tas \times Env,\ \gamma^{Val}_{\mathsf{h}} \dot{\subseteq} \gamma^{Val}_{\mathsf{f}(\mathsf{h},\rho)} \tag{7.5}$$

implique

$$\forall(\mathsf{h},\rho) \in Tas \times Env,\ \gamma^{Env}_{\mathsf{h}} \dot{\subseteq} \gamma^{Env}_{\mathsf{f}(\mathsf{h},\rho)}$$

Preuve : C'est une conséquence directe de la propriété de monotonie de Γ^{Env}. □

Nous obtenons ainsi un nouveau critère (7.5) structurellement plus petit : il ne concerne que l'abstraction des valeurs. Cela peut donc être vu comme une *propriété de conservation*. La concrétisation $\gamma^{\mathrm{Val}}_{\mathsf{f}(\mathsf{h},\rho)}$ associée avec la transformation du tas h doit donc contenir au moins toutes les propriétés de l'original $\gamma^{\mathrm{Val}}_{\mathsf{h}}$. C'est une propriété forte que notre définition des connexions génériques nous permet de transférer au niveau des connexions de valeur, sans sacrifier la généricité des connexions d'environnement.

Il nous reste maintenant à expliquer comment ce type de propriété peut être prouvé au niveau des abstractions.

7.3.3 Établir la propriété de conservation

Dans le contexte d'un preuve de correction exhaustive d'une analyse statique, il y aura autant d'obligations de preuve de la forme de (7.5) que de fonctions utilisées dans les différentes fonctions de transfert du langage étudié. Nous proposons de factoriser ces preuves en découpant ces conditions en deux. Ce découpage est effectué à l'aide d'un pré-ordre bien choisi sur le domaine des contextes.

Nous allons avoir besoin d'une notion de *concrétisation paramétrée monotone*

Définition 7.3.3. Concrétisation paramétrée monotone.
Étant donné une relation de pré-ordre $\preceq_{\mathsf{C}}$ *sur* $\mathsf{C} \times \mathsf{C}$, *une concrétisation paramétrée* $\gamma \in \mathsf{C} \to \mathsf{D}^\sharp \to \mathcal{P}(\mathsf{D})$ *est* monotone *par rapport à* $\preceq_{\mathsf{C}}$ *si*

$$\forall(\mathsf{c}_1,\mathsf{c}_2) \in \mathsf{C}^2,\ \mathsf{c}_1 \preceq_{\mathsf{C}} \mathsf{c}_2 \ \Rightarrow\ \gamma_{\mathsf{c}_1} \dot{\subseteq} \gamma_{\mathsf{c}_2}$$

Lemme 7.3. *Soit* $\mathcal{S} \subseteq (Tas \times Env) \to Tas$ *un ensemble de fonctions. Soit* γ^{Val} *une concrétisation paramétrée sur les valeurs et* $\preceq_{Tas}$ *un pré-ordre sur le domaine Tas. Si*

$$\gamma^{Val} \textit{ est monotone par rapport à } \preceq_{Tas} \tag{7.6}$$

et

$$\forall f \in \mathcal{S},\ \forall (h, \rho) \in \mathit{Tas} \times \mathit{Env},\ h \preceq_{\mathit{Tas}} f(h, \rho) \tag{7.7}$$

alors

$$\forall f \in \mathcal{S},\ \forall (h, \rho) \in \mathit{Tas} \times \mathit{Val},\ \gamma_h^{\mathit{Val}} \dot{\subseteq} \gamma_{f(h,\rho)}^{\mathit{Val}}$$

Comme nous l'avons expliqué précédemment, cette méthode de preuve n'est pas applicable pour toutes les concrétisations paramétrées de valeurs. La restriction principale concerne l'existence d'un pré-ordre adéquat sur le domaine des contextes.

La propriété (7.6) est vérifiée pour n'importe quel choix de $\preceq_{\text{Tas}}$ qui soit la fermeture réflexive transitive d'une sous-relation de la relation $\prec_{\gamma^{\text{Val}}}$ définie par

$$\prec_{\gamma^{\text{Val}}} = \left\{ (h_1, h_2) \mid \gamma_{h_1}^{\text{Val}} \dot{\subseteq} \gamma_{h_2}^{\text{Val}} \right\}$$

La deuxième propriété (7.7) doit être vérifiée pour toutes les fonctions de transfert de l'ensemble $\mathcal{S}$. Il suffit dans ce cas de considérer une relation $\preceq_{\text{Tas}}$ qui soit la fermeture réflexive transitive d'une sur-relation de la relation $\prec_{\mathcal{S}}$ définie par

$$\prec_{\mathcal{S}} = \left\{ (h, f(h, \rho)) \;\middle|\; \begin{array}{l} f \in \mathcal{S} \\ h \in \text{Tas} \end{array} \right\}$$

Les hypothèses sont donc vérifiées par tout pré-ordre de la forme $\preceq_{\text{Tas}} = \prec_{\text{Tas}}^*$ avec $\prec_{\text{Tas}}$ une relation vérifiant

$$\prec_{\mathcal{S}} \quad \subseteq \quad \prec_{\text{Tas}} \quad \subseteq \quad \prec_{\gamma^{\text{Val}}}$$

On voit clairement ici que l'existence d'une telle relation n'est pas du tout assurée dans le cas général. Elle est néanmoins assurée pour les exemples d'abstraction que nous avions donnés précédemment :

- Pour l'exemple 12, il suffit de prendre

$$\preceq_{\text{Tas}} = \left\{ (h_1, h_2) \;\middle|\; \begin{array}{l} \text{dom}(h_1) \subseteq \text{dom}(h_2) \\ \forall \text{loc} \in \text{dom}(h_1),\ \text{class}(h_1(\text{loc})) = \text{class}(h_2(\text{loc})) \end{array} \right\}$$

  La concrétisation de valeurs γ définie pour cet exemple est alors bien monotone par rapport à ce pré-ordre, puisque si h_1 et h_2 sont des tas vérifiant $h_1 \preceq_{\text{Tas}} h_2$, pour tout ensemble de classe s, si loc appartient à $\gamma_{h_1}(s)$ alors loc appartient à $\text{dom}(h_1)$ et $\text{class}(h_1(\text{loc}))$ appartient à s. Or $\text{dom}(h_1)$ est inclus dans $\text{dom}(h_2)$, donc loc appartient à $\text{dom}(h_2)$ et puisque $\text{class}(h_1(\text{loc})) = \text{class}(h_2(\text{loc}))$, on peut aussi affirmer que $\text{class}(h_2(\text{loc}))$ appartient à s. On a donc démontré que loc appartient à $\gamma_{h_2}(s)$.
  La propriété 7.7 sera quant à elle assurée par toute fonction de transfert qui ne supprime pas d'objet dans le tas, ni ne modifie leur classe. C'est effectivement le cas de toutes les fonctions de transfert de langages comme JAVA ou BYTECODE JAVA en l'absence de ramasseur de miette.
- Le même pré-ordre peut être utilisé pour traiter l'exemple 13

– Pour l'exemple 14, le contexte n'est plus un tas mais une trace partielle. La relation $\preceq_{\text{Trace}}$ suivante suffit alors :

$$\preceq_{\text{Trace}} = \{(\mathtt{tr}_1, \mathtt{tr}_2) \mid \mathtt{tr}_1 \text{ est un préfixe de } \mathtt{tr}_2\}$$

En effet, si $\mathtt{tr}_1$ est une trace partielle préfixe d'une trace partielle $\mathtt{tr}_2$, toutes les allocations d'objet apparaissant dans $\mathtt{tr}_1$ apparaissent aussi dans $\mathtt{tr}_2$. Ce qui assure la monotonie de γ par rapport à $\preceq_{\text{Trace}}$.
Pour le critère 7.7, il suffit de s'assurer que toutes les fonctions de transfert se contentent d'empiler des états en fin de traces partielles, ce qui est effectivement le cas dans les langages de programmation standards.

7.3.4 Résumé de la méthode de preuve

Nous résumons maintenant notre méthode de preuve pour établir la correction d'une fonction abstraite $\mathcal{F}^\sharp$ par rapport à une fonction $\mathcal{F}$ (exemple pris dans la section 7.2.2).

– Le critère de correction est partagé en deux critères équivalents (7.1) et (7.2). (7.1) permet de faire une preuve modulaire car il n'utilise qu'un seul contexte. Ce n'est pas le cas de (7.2).
– Le critère (7.2) est donc découpé en deux critères suffisants (7.4) et (7.3). (7.4) peut être établi à l'aide d'une connexion générique.
– Pour prouver (7.3), nous introduisons la notion de foncteur de concrétisation et un pré-ordre adéquat. (7.3) est ainsi séparé entre les critères (7.6) et (7.7). (7.6) repose uniquement sur l'abstraction prise sur les valeurs. (7.7) concerne la sémantique du langage étudié et peut être établi pour plusieurs fonctions de transfert, si le pré-ordre est bien choisi.

7.4 Travaux relatifs et conclusions

Nous avons proposé une technique de découpage modulaire de la preuve de correction d'une analyse statique. Notre approche permet une abstraction forte des différents composants de la preuve. La preuve gagne ainsi en simplicité, puisque chaque composant se base sur un petit nombre de propriétés des autres composants, et elle gagne aussi en généricité puisqu'un composant peut être implémenté de diverses façons sans que la correction globale de l'analyse soit compromise. Notre approche se démarque des autres travaux par l'usage de concrétisations paramétrées qui est un frein important pour la modularisation. La notion de foncteur monotone de concrétisation et de concrétisation monotone vis-à-vis d'un pré-ordre bien choisi nous permet de dépasser cette difficulté sans sacrifier la générécité des interfaces des composants.

La notion de concrétisation paramétrée est un outil occasionnellement utilisé en interprétation abstraite. Elle apparaît notamment dans la thèse d'Isabelle Pollet [Pol04]. Dans ces travaux, l'interprétation abstraite de programmes Java est présentée à l'aide de concrétisations paramétrées afin de relier les valeurs concrètes et les valeurs abstraites. Le contexte utilisé en paramètre est une relation entre adresses mémoires de la mémoire

dynamique. Le découpage proposé permet de faire varier le type de relation utilisé, mais la notion de contexte est omniprésente. Notre utilisation des foncteurs de concrétisation permet de faire apparaître les paramètres des concrétisations uniquement au moment opportun. Nous pouvons ainsi proposer des connexions qui ne dépendent pas du type de contexte utilisé.

En ce qui concerne les preuves modulaires, peu de travaux présentent une approche aussi modulaire que la nôtre. Cela s'explique par le fait qu'au niveau d'une preuve « papier », rien n'empêche de présenter une version spécialisée d'un domaine abstrait (par exemple les environnements de variable). On se convainc souvent facilement que la preuve de correction est similaire en changeant de contexte. Cela n'est plus possible au niveau de détail requis par un assistant de preuve. Il est alors nécessaire de proposer un bon découpage des constituants d'une preuve. Moins les constituants demandent d'hypothèses sur les autres, plus il est facile de les ré-utiliser pour des problèmes différents. On économise ainsi un temps certain de preuve interactive. Concernant les constructions modulaires, nous pouvons néanmoins citer les travaux de Patrick et Radhia Cousot [CC94] où de nombreuses connexions génériques sont proposées, dans le cadre des connexions de Galois. La propriété de monotonie identifiée dans nos travaux n'y est cependant pas présentée. Elle ne semble en effet pas utile pour le découpage modulaire utilisé, moins relationnel que celui que nous utilisons.

Nous avons expérimenté notre technique sur la preuve de correction d'une analyse statique paramétrable pour un langage de bytecode avec allocation dynamique d'enregistrements. Cette analyse sera présentée dans le prochain chapitre. L'aspect mécanisé de la preuve donne tout son intérêt à notre approche car elle propose un découpage précis et souple de l'effort de preuve. Ce chapitre témoigne ainsi du besoin supplémentaire en modularité que peut occasionner une preuve par ordinateur, comparée à une preuve traditionnelle.

Chapitre 8

Analyseur de bytecode Java certifié

Nous nous intéressons maintenant à l'analyse de programmes Java, au format bytecode, en nous basant sur les techniques développées dans les chapitres précédents. Le bytecode Java constitue un bon exemple de langage réaliste pour lequel il existe un réel besoin d'analyses statiques, afin d'assurer la sécurité des applets téléchargées sur la toile, ou pour les application embarquées sur les cartes à puces (bancaires ou téléphoniques) [Ler03].

Il existe de nombreux formalismes pour décrire la sémantique d'un langage de programmation. Chaque choix est plus ou moins facile à utiliser dans un assistant de preuve. Dans le cadre du bytecode Java, nous bénéficions d'une sémantique opérationnelle simple (qui suit de près le fonctionnement de la machine virtuelle) issue du manuel de référence [LY99][1]. Ce type de sémantique est particulièrement adapté aux définitions inductives de Coq. Nous pouvons ainsi nous concentrer sur la partie analyse, plutôt que sur la partie sémantique.

Nous ne présentons pas un cas d'étude mais plusieurs. Chacun de ces exemples constitue une analyse certifiée spécialisée sur un fragment du langage Java (noyau impératif, appels de méthode, couche objet, ...).

- La première analyse concerne le fragment impératif de Java et sera présentée dans la section 8.1. Il s'agit d'une analyse d'intervalle appliquée à la vérification des bornes de tableau. Nous adaptons pour cela l'analyse WHILE présentée dans le chapitre 5 en utilisant une abstraction qui « décompile » les expressions au niveau de la pile d'opérande.
- La deuxième analyse s'intéresse à la couche objet et aux appels virtuels de méthodes (section 8.2). Il s'agit cette fois d'une analyse inter-procédurale.
- Nous présenterons ensuite dans la section 8.3 l'analyse de référence basée sur les techniques du chapitre 7. L'analyse proposée à l'avantage d'être modulaire et de permettre plusieurs compromis entre précision et efficacité.
- Nous finirons par une analyse d'usage mémoire dans la section 8.4. Bien que basée sur une algorithme de graphe assez simple, cette analyse demande une preuve de

[1] Nous nous sommes largement aidé de la sémantique, plus formelle, proposée par Igor Siveroni [Siv04] pour une représentation intermédiaire du langage.

correction complexe basée sur une sémantique de traces partielles.

Chaque étude de cas suit le même schéma standard :

1. Nous définissons la sémantique opérationnelle du langage avec une relation de transition $\rightarrow_P$ entre états mémoires de la machine virtuelle. $[\![P]\!]$ désigne l'ensemble des états accessibles pour cette relation à partir d'un état initial.
2. L'analyse repose sur une structure de treillis $(A, \sqsubseteq, \sqcup, \sqcap)$ que nous construisons à l'aide des techniques modulaires du chapitre 6.
3. Nous définissons une fonction de concrétisation γ entre domaines abstraits et domaines concrets
4. Nous spécifions l'analyse sous la forme d'un système d'inéquations[2] (ou contraintes)
5. La correction de ce système est établie en prouvant le résultat suivant

$$\forall P,\ \forall s^\sharp,\ P \vdash s^\sharp \ \Rightarrow\ [\![P]\!] \subseteq \gamma(s^\sharp) \tag{8.1}$$

   où $P \vdash s^\sharp$ exprime que l'état abstrait $s^\sharp$ vérifie le système d'inéquations attaché au programme P. Cette preuve est toujours basée sur le lemme intermédiaire (de type *subject reduction*) suivant

$$\forall s^\sharp \in A, \forall s_1 \in \gamma(s^\sharp),\ \forall s_2, s_1 \rightarrow_P s_2 \Rightarrow s_2 \in \gamma(s^\sharp)$$

   En pratique, il y a autant de cas dans la preuve de cet énoncé, que de règles dans la définition de $\rightarrow_P$. Chaque cas demande généralement une inéquation (exceptionnellement plusieurs) du système d'inéquation spécifiant l'analyse.
6. Un algorithme de collecte des différentes inéquations d'un programme est proposé, puis couplé avec les solveurs génériques proposés dans le chapitre 4 pour calculer une solution $[\![P]\!]^\sharp$ vérifiant

$$\forall P, P \vdash [\![P]\!]^\sharp \tag{8.2}$$

7. Grâce aux résultats (8.1) et (8.2), nous obtenons une analyse statique certifiée $[\![\cdot]\!]^\sharp$ calculant des approximations correctes de la sémantique concrète d'un programme

$$\forall P, [\![P]\!] \subseteq \gamma([\![P]\!]^\sharp)$$

8. $[\![P]\!]^\sharp$ peut éventuellement être utilisé pour un dernier calcul afin de vérifier si les informations calculées sur $[\![P]\!]$ permettent d'assurer que l'exécution de P est conforme à une certaine politique de sécurité.

Nous allons présenter maintenant ces différentes études de cas en insistant sur les points spécifiques (fragment du langage traité, domaines abstraits, fonctions de concrétisation, système d'inéquations) de chacune.

[2] Comparés aux systèmes d'équations, les systèmes d'inéquations permettent un meilleur découpage de la preuve de correction. En effet, chaque inéquation correspond à une transition sémantique. Si nécessaire, une technique standard permet de transformer au moment de la résolution, un système d'inéquations en un système d'équations, en rassemblant les équations portant sur les mêmes variables.

8.1 Analyse d'intervalle sur un fragment impératif

Cette analyse est une adaptation de l'analyse du langage WHILE présentée dans le chapitre 5. Nous nous restreignons par conséquent au fragment impératif et intra-procédural de Java. Les programmes analysés n'utilisent pas d'objet, mais peuvent manipuler des tableaux. En Java, les tableaux sont des cas particuliers d'objets car ils sont alloués dynamiquement. L'analyse d'intervalles que nous réalisons sur les variables d'un programme nous permet de vérifier statiquement si les accès à un tableau respectent la taille qui lui a été allouée.

8.1.1 Syntaxe et sémantique du langage

La figure 8.1 présente un exemple de programme Java que notre analyseur est capable de traiter (sous sa forme compilée). Dans cet exemple, trois appels de méthodes statiques apparaissent. Tous ces appels sont remplacés par un bytecode équivalent lors du parcours du fichier `.class` associé au programme.

- `Input.init();` initialise le générateur aléatoire de la classe `Input`,
- `Input.read_int();` une valeur entière tirée aléatoirement,
- `Tab.print_tab(t);` affichage des éléments d'un tableau `t`.

```
class BubbleSort {

    public static void main(String[] argv ) {
        int i,j,tmp,n;
        n = 10;
        int[] t = new int[n];
        Input.init();
        for (i=0;i<n;i++) {
            t[i] = Input.read_int();
        };
        Tab.print_tab(t);
        for (i=0; i<n-1; i++) {
            for (j=0; j<n-1-i; j++)
                if (t[j+1] < t[j]) {
                    tmp = t[j];
                    t[j] = t[j+1];
                    t[j+1] = tmp;
                }
        };
        Tab.print_tab(t);
    }

}
```

FIG. 8.1 – Tri par bulle en Java

Les programmes que nous analysons respectent la syntaxe suivante :

$$
\begin{array}{lcl}
\mathit{pgm} & ::= & (\mathit{pc}\ \mathit{instr})^* \\
\mathit{instr} & ::= & \text{Nop} \mid \text{Ipush } i \mid \text{Pop} \mid \text{Dup} \\
& \mid & \text{Ineg} \mid \text{Iadd} \mid \text{Isub} \mid \text{Imult} \\
& \mid & \text{Iload } x \mid \text{Istore } x \mid \text{Iinc } x\ n \\
& \mid & \text{Newarray} \mid \text{Arraylength} \mid \text{Iaload} \mid \text{Iastore} \\
& \mid & \text{Goto } \mathit{pc} \\
& \mid & \text{If\_icmp } \mathit{cond}\ \mathit{pc} \quad \mathit{cond} \in \{\text{eq}, \text{ne}, \text{lt}, \text{le}, \text{gt}, \text{ge}\} \\
& \mid & \text{Iinput}
\end{array}
$$

Un programme est une séquence de doublets $(\mathit{pc}, \mathit{instr})$ exprimant qu'une instruction *instr* se trouve au point de programme *pc*. Ce type de programme s'exécute sur une machine virtuelle munie d'une pile d'opérandes, d'un ensemble de variables locales et d'un tas. Le jeu d'instructions contient des opérations pour manipuler la pile (dépilement et empilement de valeurs, opérations arithmétiques, ...), les variables locales (lire, écrire, incrémenter) et le tas (créer un tableau, obtenir la taille d'un tableau, lire et modifier un élément dans un tableau). Le flot de contrôle peut être modifié inconditionnellement avec Goto ou conditionnellement avec la série d'instructions If_icmp *cond* qui comparent les éléments en tête de la pile d'opérandes et réalisent un branchement en conséquence. L'instruction Iinput empile une valeur quelconque sur la pile d'opérandes (elle correspond à l'appel de la méthode statique `Input.read_int()`).

Nous donnons à ce langage une sémantique opérationnelle directement extraite de la sémantique de Java [LY99], en enlevant les détails sans rapport avec le jeu d'instructions considéré. Les différents domaines sémantiques sont formellement définis par

$$
\begin{array}{rcl}
\text{Val} & = & \mathsf{U} \mid \text{num } \mathsf{n} \mid \text{ref loc} \quad \mathsf{n} \in \mathbb{Z},\ \text{loc} \in \text{Reference} \\
\text{Pile} & = & \text{Val}^* \\
\text{LocalVar} & = & \text{Var} \rightharpoonup \text{Val} \\
\text{Tableau} & = & (\text{taille} : \mathbb{Z}) \times ([\mathsf{0}, \text{taille}{-}\mathsf{1}] \to \mathbb{Z}) \\
\text{Tas} & = & \text{Reference} \rightharpoonup \text{Tableau} \\
\text{Etat} & = & (\mathbb{P} \times \text{Tas} \times \text{Pile} \times \text{LocalVar}) + \{\Omega\}
\end{array}
$$

Cette sémantique opère ainsi sur des états de la forme $\langle\!\langle \mathit{pc}, \mathsf{h}, \mathsf{s}, \mathsf{l} \rangle\!\rangle$ avec *pc* le point de programme courant (pris dans un ensemble fini $\mathbb{P}$), h le tas contenant les tableaux alloués (une fonction partielle entre adresses mémoire et tableaux), s une pile d'opérandes (une liste de valeurs) et l un ensemble de variables locales (une fonction partielle entre noms de variable et valeurs). Un tableau est modélisé par une paire formée de la taille courante du tableau et de la fonction d'accès aux éléments (une fonction partielle entre les indices positifs, strictement inférieurs à la taille du tableau et des valeurs numériques). Notre langage ne manipule que deux types de valeurs : les valeurs numériques (prises dans $\mathbb{Z}$) et les adresses mémoires du tas (prises dans un ensemble Reference). Une valeur spéciale, notée U représente une valeur non initialisée. Nous nous restreignons aux tableaux de valeurs numériques. Un état spécial Ω est utilisé pour représenter les erreurs d'exécution. Les différentes causes d'erreurs sont l'utilisation d'une variable non initialisée, la création

d'un tableau de taille négative ou l'accès (lecture ou écriture) en dehors des bornes d'un tableau.

La sémantique opérationnelle d'un programme P est ensuite définie par une relation de transition $\rightarrow_P$ entre états. Deux états s_1 et s_2 sont en relation $(s_1 \rightarrow_P s_2)$ si et seulement si la machine virtuelle peut passer de l'état s_1 à l'état s_2 lors de l'exécution du programme P. Une partie des transitions de la relation $\rightarrow_P$ sont données dans la figure 8.2.

$$\frac{\text{instrAt}(p_1, \text{Nop } n, p_2)}{\langle\langle p_1, h, s, l\rangle\rangle \rightarrow_P \langle\langle p_2, h, s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{Ipush } n, p_2)}{\langle\langle p_1, h, s, l\rangle\rangle \rightarrow_P \langle\langle p_2, h, (\text{num } n) :: s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{Pop}, p_2)}{\langle\langle p_1, h, v :: s, l\rangle\rangle \rightarrow_P \langle\langle p_2, h, s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{Dup}, p_2)}{\langle\langle p_1, h, v :: s, l\rangle\rangle \rightarrow_P \langle\langle p_2, h, v :: v :: s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{Iload } x, p_2) \quad l(x) = \text{num } n}{\langle\langle p_1, h, s, l\rangle\rangle \rightarrow_P \langle\langle p_2, h, (\text{num } n) :: s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{If\_icmplt } p, p_2) \quad n_1 < n_2}{\langle\langle p_1, h, (\text{num } n_2) :: (\text{num } n_1) :: s, l\rangle\rangle \rightarrow_P \langle\langle p, h, s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{If\_icmplt } p, p_2) \quad \neg\, n_1 < n_2}{\langle\langle p_1, h, (\text{num } n_2) :: (\text{num } n_1) :: s, l\rangle\rangle \rightarrow_P \langle\langle p_2, h, s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{Arraylength}, p_2) \quad h(\text{loc}) = a}{\langle\langle p_1, h, (\text{ref loc}) :: s, l\rangle\rangle \rightarrow_P \langle\langle p_2, h, (\text{num } a.\text{taille}) :: s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{Iaload}, p_2) \quad h(\text{loc}) = a \quad 0 \leq i < a.\text{taille}}{\langle\langle p_1, h, (\text{num } i) :: (\text{ref loc}) :: s, l\rangle\rangle \rightarrow_P \langle\langle p_2, h, (\text{num } a[i]) :: s, l\rangle\rangle}$$

$$\frac{\text{instrAt}(p_1, \text{Iaload}, p_2) \quad h(\text{loc}) = a \quad \neg\, 0 \leq i < a.\text{taille}}{\langle\langle p_1, h, (\text{num } i) :: (\text{ref loc}) :: s, l\rangle\rangle \rightarrow_P \Omega}$$

FIG. 8.2 – Quelques règles de la sémantique opérationnelle du langage

À l'initialisation de la machine virtuelle, toutes les variables locales possèdent une valeur non initialisée, la pile d'opérandes est vide et le tas ne contient aucun tableau.

En appelant $\mathcal{S}_0$ l'ensemble de ces états initiaux, nous pouvons ainsi définir l'ensemble des états accessibles au cours d'une exécution d'un programme P par

$$[\![P]\!] = \{\ s \mid \exists s_0 \in \mathcal{S}_0,\ s_0 \rightarrow^* s\ \}$$

Un programme P sera considéré comme *sûr* si l'état Ω ne peut pas être atteint au cours de son exécution :

$$\text{Sûr}(P) \stackrel{\text{def}}{=} \Omega \notin [\![P]\!]$$

Le but de notre analyse statique est de prouver qu'un programme est sûr.

8.1.2 Domaines abstraits

Nous utilisons pour cela le domaine abstrait suivant

$$\begin{aligned}
\text{Int} &= \{\ [a,b] \mid a \in \overline{\mathbb{Z}},\ b \in \overline{\mathbb{Z}},\ a \leq b\ \} \\
\text{Num}^\sharp &= \text{Ref}^\sharp = \text{Int}_\bot \\
\text{Val}^\sharp &= \left(\text{Num}^\sharp + \text{Ref}^\sharp + U^\sharp\right)^\top_\bot \\
\text{Expr}[\text{Val}^\sharp] &= \text{const } n \mid \text{var } x \mid \text{abs } v^\sharp \mid \text{unop op } e \mid \text{binop op } e_1\ e_2 \\
&\qquad e, e_1, e2 \in \text{Expr}[\text{Val}^\sharp] \\
\text{Pile}^\sharp &= (\text{Expr}^*)^\top_\bot \\
\text{LocalVar}^\sharp &= \text{Var} \rightarrow \text{Val}^\sharp \\
\text{Etat}^\sharp &= \mathbb{P} \rightarrow \left(\text{Pile}^\sharp \times \text{LocalVar}^\sharp\right)
\end{aligned}$$

Les valeurs numériques sont ainsi abstraites par des intervalles. Une référence à un tableau est abstraite par une abstraction numérique (à base d'intervalles) de sa longueur. L'abstraction des valeurs est la somme (selon le sens défini lors du chapitre 6) des abstractions des valeurs numériques, des références et d'une valeur abstraite spéciale $U^\sharp$ pour représenter la valeur U. Le domaine $\text{Expr}[\text{Val}^\sharp]$ permet de décompiler les expressions utilisées dans les programmes sources. Nous le présenterons dans la section 8.1.3. Les autres domaines abstraits sont plus standards : les piles d'opérandes sont abstraites par des listes d'expressions abstraites et les ensembles de variables locales par des fonctions associant une valeur abstraite à chaque nom de variable. Aucun domaine abstrait n'est requis pour abstraire le tas : nous ne cherchons pas à analyser finement le contenu des tableaux dans cette analyse. Un état abstrait est une fonction qui associe à chaque point de programme une pile abstraite et un ensemble abstrait de variables locales. La concrétisation correspondante est définie par

$$\gamma_{\text{Etat}}(st^\sharp) = \left\{ \langle\!\langle p, h, v_1 :: \cdots :: v_n, l \rangle\!\rangle \;\middle|\; \begin{array}{l} st^\sharp(p) = \left(e_1 :: \cdots :: e_n, l^\sharp\right) \\ v_1 \in \gamma^{h,l}_{\text{Expr}}(e_1), \ldots, v_n \in \gamma^{h,l}_{\text{Expr}}(e_n) \\ l \in \gamma^{h}_{\text{LocalVar}}(l^\sharp) \end{array} \right\}$$

La concrétisation des variables locales est paramétrée par un tas h car les variables locales abstraites contiennent des valeurs abstraites dont la concrétisation nécessite elle-même un tas h.

8.1.3 Décompilation des expressions

Le domaine Expr[Val$^\sharp$] permet de décompiler les expressions utilisées dans les programmes sources. Ce domaine est ainsi constitué d'expressions syntaxiques formées d'opérateurs binaires et unaires du langage source, de constantes numériques, de variables mais aussi de valeurs abstraites. De telles valeurs abstraites apparaissent lorsque la décompilation d'une expression n'est plus possible, ou inutile pour la suite de l'analyse.

Exemple 15. *Un exemple de construction et d'utilisation de ces expressions est présenté dans la figure 8.3. Avant le point de programme 7, la variable* `j` *est initialisée avec une valeur numérique quelconque, tandis que la variable* `i` *vaut 100. Les instructions suivantes construisent l'expression* `j+i` *afin de pouvoir évaluer la garde de l'expression conditionnelle* `if (j+i>3) { .. }`*. En empilant sur la pile d'opérandes abstraite les expressions* (*var* j), (*var* i) *puis* (*binop* + (*var* j) (*var* i))*, nous sommes ainsi capables d'inférer que la valeur numérique située en tête de pile avant le point de programme 11 est égale à la somme des valeurs présentes dans* `i` *et* `j`*. Cela nous permet de raffiner les valeurs possibles de la variable* `j` *après le point de programme 11. Sans cette technique nous serons seulement capables d'inférer que les deux valeurs en sommet de pile au niveau du point 11 sont dans les intervalles* $[-\infty, +\infty]$ *et* $[100, 100]$*. Aucun raffinement sur la variable* `j` *ne serait alors possible.*

L'ordre imposé sur les expressions dépend de l'ordre pris sur le domaine des valeurs abstraites de Val$^\sharp$. Deux expressions sont en relation pour cet ordre si elles ont la même structure et si les valeurs abstraites de chaque position de l'expression sont en relation pour $\sqsubseteq_{\text{Val}}$. L'ordre $\sqsubseteq_{\text{Expr}}$ est formellement défini par les règles inductives suivantes :

$$\frac{n = m}{(\text{const } n) \sqsubseteq_{\text{Expr}} (\text{const } m)} \qquad \frac{x = y}{(\text{var } x) \sqsubseteq_{\text{Expr}} (\text{var } y)}$$

$$\frac{v_1^\sharp \sqsubseteq_{\text{Val}} v_2^\sharp}{(\text{abs } v_1^\sharp) \sqsubseteq_{\text{Expr}} (\text{abs } v_2^\sharp)} \qquad \frac{op_1 = op_2 \quad e_1 \sqsubseteq_{\text{Expr}} e_2}{(\text{unop } op_1\ e_1) \sqsubseteq_{\text{Expr}} (\text{unop } op_2\ e_2)}$$

$$\frac{op_1 = op_2 \quad e_1 \sqsubseteq_{\text{Expr}} e_2 \quad e_3 \sqsubseteq_{\text{Expr}} e_4}{(\text{binop } op_1\ e_1\ e_3) \sqsubseteq_{\text{Expr}} (\text{binop } op_2\ e_2\ e_4)}$$

La fonction de concrétisation de ce domaine abstrait dépend de la fonction de concrétisation γ_{Val} utilisée pour le domaine Val des valeurs. La concrétisation de certaines expressions n'a de sens que pour un tas h et un ensemble de variables locales l donnés. Nous utilisons donc une fonction de concrétisation paramétrée par deux éléments quelconques h et l[3]. La fonction de concrétisation γ_{Val} est elle-même paramétrée par un

[3]Bien que nous utilisions ici des concrétisations paramétrées, nous ne pouvons pas appliquer les techniques du chapitre 7 sur cette analyse. En effet, γ_{Expr} n'est pas monotone vis à vis de son paramètre l : par exemple

$$\gamma^{h,l}_{\text{Expr}}(\text{var } x) \not\subseteq \gamma^{h,l[x\mapsto v]}_{\text{Expr}}(\text{var } x)$$

si $v \neq l(x)$. Il s'agit donc d'une abstraction trop relationnelle sur Tas × LocalVar × Val pour pouvoir

exemple source

```
int i = 100;
int j = Input.read_int();
if (j+i>3) { .. }
```

version bytecode

```
...
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <>
 7 : iload j
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <(var j)>
 8 : iload i
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <(var j)::(var i)>
 9 : iadd
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <(binop + (var j) (var i))>
10 : ipush 3
    // [j↦[-oo,+oo] ; i↦[100,100]]
    // <(binop + (var j) (var i))::(const 3)>
11 : if_icmple 16
    // [j↦[-96,+oo] ; i↦[100,100]]
    // <>
...
```

FIG. 8.3 – Exemple d'analyse

tas h car le domaine des valeurs abstraites contient le domaine des références abstraites et car ce dernier nécessite une telle concrétisation paramétrée. La figure 8.4 propose la définition de cette fonction γ_{Expr}.

$$\gamma^{h,l} : \text{Expr}[\text{Val}^\sharp] \to \mathcal{P}(\text{Val}) \qquad \text{avec } l \in \text{LocalVar}, h \in \text{Tas}$$

$$\begin{aligned}
\gamma^{h,l}_{\text{Expr}}(\text{const } m) &= \{ (\text{num } m) \} \\
\gamma^{h,l}_{\text{Expr}}(\text{var } x) &= \{ l(x) \} \\
\gamma^{h,l}_{\text{Expr}}\left(\text{abs } v^\sharp\right) &= \gamma^{h}_{\text{Val}}(v^\sharp) \\
\gamma^{h,l}_{\text{Expr}}(\text{unop taille } e) &= \left\{ (\text{num } a.\text{taille}) \mid (\text{ref loc}) \in \gamma^{h,l}_{\text{Expr}}(e),\ h(\text{loc}) = a \right\} \\
\gamma^{l}_{\text{Expr}}(\text{unop } - \ e) &= \left\{ (\text{num } (-n)) \mid (\text{num } n) \in \gamma^{l}_{\text{Expr}}(e) \right\} \\
\gamma^{h,l}_{\text{Expr}}(\text{binop } + \ e_1\ e_2) &= \left\{ (\text{num } (n_1 + n_2)) \;\middle|\; \begin{array}{l}(\text{num } n_1) \in \gamma^{h,l}_{\text{Expr}}(e_1) \\ (\text{num } n_2) \in \gamma^{h,l}_{\text{Expr}}(e_2)\end{array} \right\} \\
\gamma^{h,l}_{\text{Expr}}(\text{binop } - \ e_1\ e_2) &= \left\{ (\text{num } (n_1 - n_2)) \;\middle|\; \begin{array}{l}(\text{num } n_1) \in \gamma^{h,l}_{\text{Expr}}(e_1) \\ (\text{num } n_2) \in \gamma^{h,l}_{\text{Expr}}(e_2)\end{array} \right\} \\
\gamma^{h,l}_{\text{Expr}}(\text{binop } \times \ e_1\ e_2) &= \left\{ (\text{num } (n_1 \times n_2)) \;\middle|\; \begin{array}{l}(\text{num } n_1) \in \gamma^{h,l}_{\text{Expr}}(e_1) \\ (\text{num } n_2) \in \gamma^{h,l}_{\text{Expr}}(e_2)\end{array} \right\} \\
\gamma^{h,l}_{\text{Expr}}(\text{binop tab } e_1\ e_2) &= \left\{ (\text{num } (a[n])) \;\middle|\; \begin{array}{l}(\text{ref loc}) \in \gamma^{h,l}_{\text{Expr}}(e_1),\ h(\text{loc}) = a \\ (\text{num } n) \in \gamma^{h,l}_{\text{Expr}}(e_2)\end{array} \right\}
\end{aligned}$$

FIG. 8.4 – Fonction de concrétisation pour le domaine $\text{Expr}[\text{Val}^\sharp]$

8.1.4 Spécification, calcul puis post-traitement de la sémantique abstraite

L'analyse est ensuite spécifiée comme la solution $\text{st}^\sharp$ d'un système d'inéquations, chaque inéquation étant associée à une instruction du programme. La figure 8.5 présente quelques unes de ces inéquations. Nous ré-utilisons ici la technique d'analyse arrière des tests, telle que nous l'avons présentée dans le chapitre 5 sur l'analyse du langage WHILE. Nous ré-utilisons notamment l'opérateur $[\![\cdot]\!]\downarrow^\sharp_{\text{test}}$ d'analyse arrière des tests.

Les solutions de ce système nous donnent des invariants mémoires attachés à chaque point du programme. Un simple parcours de ces invariants permet ensuite de vérifier si aucun état d'erreur ne pourra être atteint au cours de l'exécution du programme. Nous caractérisons pour cela l'ensemble $\text{PredErreur}_P \subseteq \mathcal{P}(\text{Etat})$ des états prédécesseurs de l'état Ω.

$$\text{PredErreur}_P \stackrel{\text{def}}{=} \{ \text{st} \in [\![P]\!] \mid \text{st} \to_P \Omega \}$$

utiliser notre technique. L'analyse de la section 8.3 proposera un meilleur cadre d'application pour le chapitre 7.

$$\frac{\text{instrAt}(p_1, \text{Ipush } n, p_2) \quad st^\sharp(p_1) = (s^\sharp_{p_1}, l^\sharp_{p_1})}{st^\sharp(p_2) \sqsupseteq \left((\text{const } n) :: s^\sharp_{p_1}, l^\sharp_{p_1}\right)}$$

$$\frac{\text{instrAt}(p_1, \text{If\_icmplt } p, p_2) \quad st^\sharp(p_1) = (e_2 :: e_1 :: s^\sharp_{p_1}, l^\sharp_{p_1})}{st^\sharp(p) \sqsupseteq \left(s^\sharp_{p_1}, [\![e_1 < e_2]\!]\downarrow^\sharp_{\text{test}} (l^\sharp_{p_1})\right)}$$

$$\frac{\text{instrAt}(p_1, \text{If\_icmplt } p, p_2) \quad st^\sharp(p_1) = (e_2 :: e_1 :: s^\sharp_{p_1}, l^\sharp_{p_1})}{st^\sharp(p_2) \sqsupseteq \left(s^\sharp_{p_1}, [\![e_1 \geq e_2]\!]\downarrow^\sharp_{\text{test}} (l^\sharp_{p_1})\right)}$$

FIG. 8.5 – Quelques unes des inéquations associées aux instructions d'un programme

Nous programmons ensuite (toujours en Coq), un vérificateur d'invariants abstraits $\text{verif} \in \text{Etat}^\sharp \to \text{bool}$ qui assure qu'aucun état prédécesseur de Ω ne peut être atteint :

$$\forall st^\sharp \in \text{Etat}^\sharp,\ \text{verif}(st^\sharp) = \mathsf{T} \ \Rightarrow\ \gamma_{\text{Etat}}(st^\sharp) \cap \text{PredErreur}_P = \emptyset$$

En notant $[\![P]\!]^\sharp$ la solution du système d'inéquations calculée par nos solveurs itératifs (à base d'élargissements et de rétrécissements puisque nous utilisons une abstraction numérique par intervalles), nous avons

$$P^\sharp \subseteq \gamma_{\text{Etat}}([\![P]\!]^\sharp)$$

Combiné avec verif nous obtenons ainsi un outil certifié qui assure la sûreté d'un programme, lorsqu'il répond T :

$$\text{verif}([\![P]\!]^\sharp) = \mathsf{T} \Rightarrow \text{Sûr}(P)$$

8.1.5 Expériences

La figure 8.6 présente les invariants calculés sur l'exemple présenté précédemment. Pour plus de lisibilité nous présentons une partie des invariants calculés au niveau du bytecode, en les replaçant (à la main) au niveau du programme source. Les invariants permettent de démontrer que ce programme est sûr (la fonction verif renvoie T). L'analyse infère en effet que les indices `i` et `j` restent dans des intervalles bornés ($[0, 19]$ pour `i` et $[0, 18]$ pour `j`) au moment de leurs utilisation pour accéder au tableau `t`. Le temps de calcul de l'analyseur extrait sur cet exemple d'une centaine de bytecode est inférieur à une seconde. Nous avons expérimenté cette analyse sur d'autre programmes de manipulation de tableau (le tri par tas, le tri rapide, le produit de polynôme, l'algorithme Floyd-Wharshall et le produit de convolution de vecteurs). Les temps de calculs pour ces programmes sont tous inférieurs à une seconde (pour un nombre de bytecodes oscillant entre 100 et 400), mais seul le produit de convolution est prouvé sûr. Pour les autres programmes, il nous faut avoir recours à un solveur externe (qui gère mieux les boucles imbriquées) pour prouver la sûreté. Le post-point fixe calculé par le solveur externe est néanmoins vérifié par notre analyseur certifiée, ce qui assure la correction sémantique du résultat.

```
      // [j ↦    U    ; n ↦    U    ; i ↦    U    ; t ↦      U     ]
n = 20;
      // [j ↦    U    ; n ↦ [20, 20]; i ↦    U    ; t ↦      U     ]
int[] t = new int[n];
      // [j ↦    U    ; n ↦ [20, 20]; i ↦    U    ; t ↦ int[20, 20]]
   Input.init();
      // [j ↦    U    ; n ↦ [20, 20]; i ↦ [ 0, 20]; t ↦ int[20, 20]]
   for (i=0;i<n;i++) {
      // [j ↦    U    ; n ↦ [20, 20]; i ↦    U    ; t ↦ int[20, 20]]
       t[i] = Input.read_int();
      // [j ↦    U    ; n ↦ [20, 20]; i ↦ [ 0, 19]; t ↦ int[20, 20]]
    };
      // [j ↦    U    ; n ↦ [20, 20]; i ↦ [20, 20]; t ↦ int[20, 20]]
    Tab.print_tab(t);
      // [j ↦    U    ; n ↦ [20, 20]; i ↦ [ 0, 20]; t ↦ int[20, 20]]
   for (i=0; i<n-1; i++) {
      // [j ↦  topV   ; n ↦ [20, 20]; i ↦ [ 0, 18]; t ↦ int[20, 20]]
       for (j=0; j<n-1-i; j++)
      // [j ↦ [ 0, 18]; n ↦ [20, 20]; i ↦ [ 0, 18]; t ↦ int[20, 20]]
           if (t[j+1] < t[j]) {
      // [j ↦ [ 0, 18]; n ↦ [20, 20]; i ↦ [ 0, 18]; t ↦ int[20, 20]]
             tmp = t[j];
      // [j ↦ [ 0, 18]; n ↦ [20, 20]; i ↦ [ 0, 18]; t ↦ int[20, 20]]
             t[j] = t[j+1];
      // [j ↦ [ 0, 18]; n ↦ [20, 20]; i ↦ [ 0, 18]; t ↦ int[20, 20]]
             t[j+1] = tmp;
      // [j ↦ [ 0, 18]; n ↦ [20, 20]; i ↦ [ 0, 18]; t ↦ int[20, 20]]
           }
      // [j ↦ [ 1, 19]; n ↦ [20, 20]; i ↦ [ 0, 18]; t ↦ int[20, 20]]
   };
      // [j ↦  topV   ; n ↦ [20, 20]; i ↦ [19, 19]; t ↦ int[20, 20]]
   Tab.print_tab(t);
      // [j ↦  topV   ; n ↦ [20, 20]; i ↦ [19, 19]; t ↦ int[20, 20]]
```

FIG. 8.6 – Résultat de l'analyse du tri bulle, réinjecté dans le programme source. Les accès au tableau `t` sont ainsi démontrés comme étant tous valides.

8.2 Analyse de classe inter-procédurale

Nous présentons maintenant une analyse inter-procédurale consacrée à la couche objet du langage Java.

8.2.1 Sémantique

Par rapport à l'analyse précédente, nous ne traitons plus des tableaux, mais nous gérons cette fois la manipulation des objets, les appels et retours de méthode. Nous rajoutons pour cela cinq instructions :

- new $\mathtt{cl}$ pour créer un nouvel objet de classe $\mathtt{cl}$ dans le tas et placer une référence à cet objet en tête de pile,
- putfield $\mathtt{f}$ pour modifier le champ $\mathtt{f}$ d'un objet dont la référence se trouve en deuxième position sur la pile, la nouvelle valeur du champ se trouvant en première position sur la pile,
- getfield $\mathtt{f}$ pour placer la valeur du champ $\mathtt{f}$ d'un objet dont la référence se trouve en première position sur la pile,
- invokevirtual $\mathtt{m}_{\mathtt{id}}$ pour réaliser un appel virtuel de la méthode $\mathtt{m}_{\mathtt{id}}$ en déterminant dynamiquement quelle implémentation de $\mathtt{m}_{\mathtt{id}}$ utiliser, en fonction de la classe de l'objet en tête de pile,
- return pour terminer l'exécution de la méthode courante et transférer le résultat à la méthode appelante.

Un programme est désormais constitué d'une liste de classes. Chaque classe possède un nom, une liste de noms de champs (les champs déclarés pour cette classe) et une liste de méthodes. Une méthode comprend un nom, son nombre d'arguments et une liste d'instructions.

Le domaine sémantique correspondant à ce langage de bytecode est maintenant de la forme suivante.

$$
\begin{array}{rcll}
\text{Val} & = & \text{num } \mathtt{n} & \mathtt{n} \in \mathbb{Z} \\
 & | & \text{ref loc} & \text{loc} \in \text{Reference} \\
 & | & \text{null} & \\
\text{Pile} & = & \text{Val}^* & \\
\text{LocalVar} & = & \text{Var} \rightharpoonup \text{Val} & \\
\text{EtatLocal} & = & \mathbb{P} \times \text{NomMethode} \times \text{LocalVar} \times \text{Pile} & \\
\text{PileAppel} & = & \text{EtatLocal}^* & \\
\text{Objet} & = & \text{NomClasse} \times (\text{NomChamp} \rightharpoonup \text{Val}) & \\
\text{Tas} & = & \text{Reference} \rightharpoonup \text{Objet} & \\
\text{Etat} & = & \text{Tas} \times \text{EtatLocal} \times \text{PileAppel} &
\end{array}
$$

Un tas est désormais une fonction partielle entre adresses mémoires et objets. Chaque objet possède une classe et une fonction partielle associant noms de champs et valeurs. La nouveauté majeure concerne l'apparition d'une pile d'appels dans les états. Un état est maintenant de la forme $\langle\langle \mathtt{h}, \langle \mathtt{m}, pc, \mathtt{l}, \mathtt{s}\rangle, sf\rangle\rangle$ avec $\mathtt{h}$ un tas, $\langle \mathtt{m}, pc, \mathtt{l}, \mathtt{s}\rangle$ l'état local courant et sf une pile d'appels comprenant une liste d'états locaux. Un état courant

comme $\langle \mathsf{m}, \mathsf{p}, \mathsf{l}, \mathsf{s} \rangle$ est constitué du nom de la méthode courante m, du point de programme courant p, d'un ensemble de variables locales l et d'une pile d'opérandes s. Les états locaux en suspens dans la pile d'appels correspondent aux différents contextes d'appel de méthode qui ont mené jusqu'au point d'exécution courant. Avec un tel domaine sémantique, la règle sémantique associée à l'instruction Ipush devient

$$\frac{\text{instrAt}(\mathsf{m}, pc, \text{Ipush } \mathsf{n})}{\langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc, \mathsf{l}, \mathsf{s} \rangle, sf \rangle\!\rangle \rightarrow_{\mathsf{P}} \langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc + 1, \mathsf{l}, (\text{num } \mathsf{n}) :: \mathsf{s} \rangle, sf \rangle\!\rangle}$$

Toutes les instructions intra-procédurales du langage de bytecode présentées dans la section précédente s'adaptent de la même façon à cette version inter-procédurale de la sémantique. La gestion des objets est modélisée par les règles sémantiques suivantes.

$$\frac{\begin{array}{c}\text{instrAt}(\mathsf{m}, pc, \text{new } \mathsf{cl}) \\ \exists \mathsf{c} \in \text{classes}(\mathsf{P}) \text{ avec } \text{NomClasse}(\mathsf{c}) = \mathsf{cl} \quad (\mathsf{h}', \mathsf{loc}) = \text{newObject}(\mathsf{cl}, \mathsf{h})\end{array}}{\langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc, \mathsf{l}, \mathsf{s} \rangle, sf \rangle\!\rangle \rightarrow_{\mathsf{P}} \langle\!\langle \mathsf{h}', \langle \mathsf{m}, pc + 1, \mathsf{l}, \mathsf{loc} :: \mathsf{s} \rangle, sf \rangle\!\rangle}$$

$$\frac{\text{instrAt}(\mathsf{m}, pc, \text{putfield f}) \quad \mathsf{h}(\mathsf{loc}) = \mathsf{o} \quad \mathsf{o}' = \mathsf{o}[\mathsf{f} \mapsto \mathsf{v}]}{\langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc, \mathsf{l}, \mathsf{v} :: \text{loc} :: \mathsf{s} \rangle, sf \rangle\!\rangle \rightarrow_{\mathsf{P}} \langle\!\langle \mathsf{h}[\mathsf{loc} \mapsto \mathsf{o}'], \langle \mathsf{m}, pc + 1, \mathsf{l}, \mathsf{s} \rangle, sf \rangle\!\rangle}$$

$$\frac{\text{instrAt}(\mathsf{m}, pc, \text{getfield f}) \quad \mathsf{h}(\mathsf{loc}) = \mathsf{o}}{\langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc, \mathsf{l}, \mathsf{loc} :: \mathsf{s} \rangle, sf \rangle\!\rangle \rightarrow_{\mathsf{P}} \langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc + 1, \mathsf{l}, \text{valChamp}(\mathsf{o}, \mathsf{f}) :: \mathsf{s} \rangle, sf \rangle\!\rangle}$$

classes(P) représente l'ensemble des classes déclarées dans le programme P. L'expression newObject(cl, h) représente un nouveau tas h' et une nouvelle adresse mémoire loc$'$ telles que h' est une copie du tas h auquel on a ajouté un nouvel objet de classe cl à l'adresse loc$'$. La fonction valChamp permet d'accéder à la valeur du champ d'un objet.

Les seules règles modifiant la pile d'appels sont celles qui concernent l'appel et le retour de méthodes.

$$\frac{\begin{array}{c}\text{instrAt}(\mathsf{m}, pc, \text{invokevirtual } \mathsf{M}) \\ \mathsf{h}(loc) = \mathsf{o} \qquad \mathsf{m}' = \text{methodLookup}(\mathsf{M}, \text{class}(\mathsf{o})) \\ \mathsf{f}' = \langle \mathsf{m}', 1, \mathsf{V}, \varepsilon \rangle \qquad\qquad \mathsf{f}'' = \langle \mathsf{m}, pc, \mathsf{l}, \mathsf{s} \rangle\end{array}}{\langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc, \mathsf{l}, \mathsf{loc} :: \mathsf{V} :: \mathsf{s} \rangle, sf \rangle\!\rangle \rightarrow_{\mathsf{P}} \langle\!\langle \mathsf{h}, \mathsf{f}', \mathsf{f}'' :: sf \rangle\!\rangle}$$

$$\frac{\text{instrAt}(\mathsf{m}, pc, \text{return f})}{\langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc, \mathsf{l}, \mathsf{v} :: \mathsf{s} \rangle, \langle \mathsf{m}', pc', \mathsf{l}', \mathsf{s}' \rangle :: sf \rangle\!\rangle \rightarrow_{\mathsf{P}} \langle\!\langle \mathsf{h}, \langle \mathsf{m}', pc' + 1, \mathsf{l}', \mathsf{v} :: \mathsf{s}' \rangle, sf \rangle\!\rangle}$$

Une instruction invokevirtual M où M est un nom de méthode ne peut s'exécuter que sur un état de la forme $\langle\!\langle \mathsf{h}, \langle \mathsf{m}, pc, \mathsf{l}, \mathsf{loc} :: \mathsf{V} :: \mathsf{s} \rangle, sf \rangle\!\rangle$ où V est une liste de paramètres pour la méthode appelée. La méthode appelée est calculée dynamiquement en utilisant

la classe de l'objet pointé par la référence placée en tête de la pile d'opérandes. Cette résolution dynamique est réalisée en parcourant la hiérarchie de classes du programme P. La recherche commence au niveau des méthodes de la classe de l'objet o : si aucune implémentation de M n'y est trouvée, la recherche est relancée sur la super-classe, et ainsi de suite. Puisque que nous considérons des programmes complets (toutes les classes nécessaires à l'exécution sont présentes dans le programme), cette recherche peut être tabulée en associant statiquement à chaque identificateur de méthode M et chaque classe c l'implémentation de la méthode associée. Ce type d'optimisation a déjà été formalisé dans un de nos précédents travaux [GJKP04]. Nous nous contentons de supposer ici qu'une telle tabulation est fournie avec le programme P. Lorsque le nom m' de la méthode à appeler est trouvé, un nouvel état local est créé avec le premier point de programme de la méthode, un ensemble de variables locales initialisé avec les paramètres V et une pile d'opérandes vide. L'ancien état local est placé en tête de la pile d'appels, en supprimant auparavant (ref loc) :: V de la pile d'opérandes.

Une instruction return se contente de restaurer le premier état local de la pile d'appels, en passant au point de programme suivant l'appel et en empilant le résultat de la méthode (la tête de la pile d'opérandes de l'ancien état local) en tête de la pile d'opérandes du nouvel état local.

8.2.2 Domaines abstraits

Le domaine abstrait proposé pour cette analyse est décrit par les définitions suivantes.

$$
\begin{array}{rcl@{\qquad}rcl}
\mathrm{Num}^\sharp & := & \mathbb{Z}^\top_\bot & \mathrm{Ref}^\sharp & := & \mathcal{P}(\mathrm{NomClasse}) \\
\mathrm{Val}^\sharp & := & \left(\mathrm{Ref}^\sharp + \mathrm{Num}^\sharp\right)^\top_\bot & \mathrm{Pile}^\sharp & := & \left(\mathrm{Val}^{\sharp *}\right)^\top_\bot \\
\mathrm{LocalVar}^\sharp & := & \mathrm{Var} \to \mathrm{Val}^\sharp & \mathrm{Objet}^\sharp & := & \mathrm{NomChamp} \to \mathrm{Val}^\sharp \\
\mathrm{Tas}^\sharp & := & \mathrm{NomClasse} \to \mathrm{Objet}^\sharp & & & \\
\mathrm{Etat}^\sharp & := & \mathrm{Tas}^\sharp \times \left(\mathrm{NomMethode} \times \mathbb{P} \to \mathrm{LocalVar}^\sharp\right) & & & \\
 & & \quad \times \left(\mathrm{NomMethode} \times \mathbb{P} \to \mathrm{Pile}^\sharp\right) & & &
\end{array}
$$

Le domaine abstrait des valeurs numériques est le treillis des constantes de Kildall. Une référence est abstraite par la classe de l'objet auquel elle se réfère dans le tas. Une abstraction d'un ensemble de référence est donc un ensemble de noms de classes. Les domaines des valeurs, des piles d'opérandes et des variables locales sont les mêmes que dans l'analyse précédente (on conserve la même structure que dans le domaine concret). L'analyse utilise cette fois une abstraction non triviale du tas : une fonction entre noms de classes et objets abstraits, chaque objet abstrait étant une fonction entre noms de champs et valeurs abstraites. Cette abstraction consiste à réunir tous les objets possédant la même classe en un seul objet abstrait. La figure 8.7 présente un exemple d'utilisation de cette abstraction.

Le domaine global est un produit de trois domaines. Le premier domaine correspond à l'abstraction du tas. Il s'agit ici d'une analyse du tas insensible au flot puisqu'un seul tas abstrait est calculé pour tout le programme. Les deux autres domaines correspondent

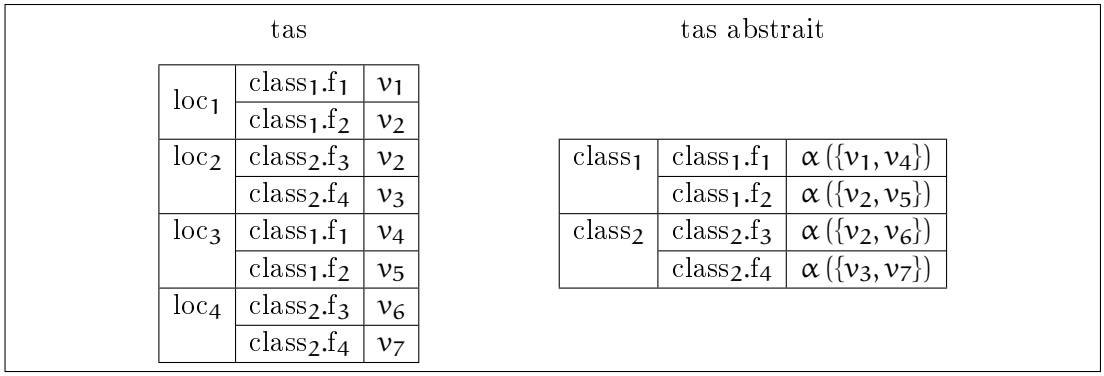


FIG. 8.7 – Exemple d'abstraction du tas

à l'abstraction de la pile d'opérandes et aux variables locales. Il s'agit cette fois d'une analyse sensible au flot : pour chaque instruction (repérée par un couple (m, pc)) on attache une pile abstraite et des variables locales abstraites. Le treillis final vérifie la condition de chaîne ascendante.

8.2.3 Spécification de la sémantique abstraite

La nature du domaine abstrait incite à présenter la spécification de l'analyse sous la forme d'un système d'inéquations portant sur un tas abstrait $H^\sharp$ et un ensemble de piles abstraites $\left(S^\sharp_{m,pc}\right)$ et de tableau de variables locales abstraites $\left(L^\sharp_{m,pc}\right)$ avec (m, pc) les différentes positions du programme où l'on trouve une instruction. Nous présentons maintenant trois exemples d'inéquations, les autres pouvant être consultées dans l'annexe de [CJPR05].

– une instruction putfield f placée en un point (m, pc) sera associée à l'inéquation

$$\begin{array}{l}\text{si } S^\sharp_{m,pc} = v^\sharp :: loc^\sharp :: s^\sharp \text{ alors}\\ \quad s^\sharp \sqsubseteq_{\text{Pile}} S^\sharp_{m,pc+1}\\ \quad L^\sharp_{m,pc} \sqsubseteq_{\text{LocalVar}} L^\sharp_{m,pc+1}\\ \quad \forall cl \in loc^\sharp,\ \bot_{\text{Tas}}[cl \mapsto \bot_{\text{Objet}}[f \mapsto v^\sharp]] \sqsubseteq_{\text{Tas}} H^\sharp\end{array}$$

– pour une instruction return placée en un point (m, pc), nous utilisons simplement l'inéquation

$$S^\sharp_{m,pc} \sqsubseteq_{\text{Pile}} S^\sharp_{m,\text{fin}(m)}$$

où $\text{fin}(m)$ est un point de programme artificiellement ajouté à la méthode m afin d'y réunir toutes les piles d'opérandes abstraites atteintes aux différents points de retour de la méthode,

– pour une instruction invokevirtual M en un point (m, pc), nous utilisons

$$
\begin{array}{l}
\text{si } S^\sharp_{m,pc} = \text{loc}^\sharp :: v^\sharp_1 :: \cdots :: v^\sharp_n :: s^\sharp \text{ alors} \\
\forall cl \in \text{loc}^\sharp, \text{si } S^\sharp_{m',\text{fin}(m')} = v^\sharp_{\text{ret}} :: s^\sharp_{\text{ret}} \text{ alors} \\
\quad L^\sharp_{m,pc} \sqsubseteq_{\text{LocalVar}} L^\sharp_{m,pc+1} \\
\quad v^\sharp :: s^\sharp \sqsubseteq_{\text{Pile}} S^\sharp_{m,pc+1} \\
\quad \bot_{\text{LocalVar}}[0 \mapsto \{cl\}, 1 \mapsto v^\sharp_1, \ldots, n \mapsto v^\sharp_n] \sqsubseteq_{\text{LocalVar}} L^\sharp_{m',1} \\
\quad \text{nil}^\sharp \sqsubseteq_{\text{Pile}} S^\sharp_{m',1} \\
\quad \text{avec } m' = \text{methodLookup}(M, cl)
\end{array}
$$

8.2.4 Expériences

Cette analyse sert essentiellement à raffiner la connaissance sur le graphe de flot de contrôle d'un programme Java. Elle permet en effet de calculer statiquement une approximation de la résolution des appels virtuels. Si au moment d'un appel virtuel, l'abstraction de l'objet en tête de pile est un singleton, nous pouvons « dévirtualiser » cet appel puisque nous connaissons alors statiquement sa cible. Ce type d'information a de multiples applications : on optimise le programme en remplaçant certains appels virtuels par des appels statiques, on peut « inliner » certaines méthodes pour améliorer certaines analyses statiques insensibles aux contextes, ...

La figure 8.8 présente un exemple[4] de programme où les cibles des appels virtuels sont a priori multiples puisque la fonction `foo` est définie dans chacune des classes `A`, `B` et `C`.

Le résultat de l'analyse sur le programme de bytecode correspondant est quant à lui présenté dans la figure 8.9. Entre chaque ligne d'instructions, nous faisons figurer la pile abstraite et les variables locales abstraites du point de programme courant.

Grâce à cette analyse, les appels virtuels de la méthode `foo` sont statiquement résolus. L'analyse inter-procédurale réalisée permet alors de calculer statiquement le résultat final du programme principal.

8.3 Analyse de référence modulaire

L'analyse précédente regroupe dans le tas abstrait tous les objets possédant la même classe. Il s'ensuit que, si l'on accède aux champs d'un objet, on récupère une approximation de tous les champs des objets de même classe. On perd ainsi beaucoup d'informations. Pour gagner en précision, nous proposons de partitionner le tas, non plus selon les classes des objets, mais selon leurs points de création. Une référence est alors abstraite par le point de création de l'objet auquel elle réfère. Le programme du nouvel analyseur est similaire au précédent (on manipule désormais des points de programme à la place de noms de classes), mais il est très difficile d'adapter aussi facilement la preuve de correction (notamment parce qu'il faut désormais raisonner sur les traces partielles

[4] Cet exemple est issu d'un exposé de Barbara G.Ryder [Ryd03] sur l'analyse de référence dans les langages à objet. Les invariants donnés sont ceux calculés par notre analyseur extrait.

```
static int main(){
  B b1 = new B();
  A a1 = new A();
  return b1.f(b1)+b1.g(b1);
}
class A {
  static void f(A a2){ return a2.foo();}
  static void g(B b2){
    B b3 = b2;
    b3 = new C();
    return b3.foo();
  }
  int foo(){ return 0;}
}
class B extends A{
  int foo() { return 1;}
}
class C extends B{
  int foo() { return 2;}
}
```

FIG. 8.8 – Programme source pour l'exemple d'analyse de classe

accessibles, plutôt que sur les états accessibles, voir l'exemple 14 du chapitre 7). Cette difficulté a motivé nos travaux sur la construction modulaire de preuves de correction d'analyse, présentés dans le chapitre 7.

Cette technique de preuve du chapitre 7 a été expérimentée sur un langage de bytecode similaire à celui de la section précédente, mais sans appel et retour de méthode. L'analyse ainsi formalisée est beaucoup plus modulaire que la précédente. Elle calcule des invariants sur les états accessibles, pour chaque point de programme. Le domaine abstrait est de la forme

$$\text{Etat}^\sharp = \mathbb{P} \to \left(\text{Tas}^\sharp \times \text{LocalVar}^\sharp \times \text{Pile}^\sharp\right)$$

avec $\text{Tas}^\sharp$, $\text{LocalVar}^\sharp$ et $\text{Pile}^\sharp$ des domaines abstraits génériques pour le tas, les variables locales et la pile d'opérandes. Cette analyse est paramétrée par cinq connections génériques (pour les valeurs, les piles d'opérandes, les variables locales, les objets et les tas) et deux abstractions de base (une abstraction numérique et une abstraction paramétrée pour les références). Nous avons développé plusieurs instanciations possibles pour les différentes connections de l'analyse :

- Entiers : abstraction par le type, abstraction des constantes, signes et congruences,
- Références : abstraction par la classe (comme dans la section précédente) et abstraction par le point de création,
- Valeurs : somme des abstractions numériques et des abstractions de références, plus éventuellement des constantes abstraites pour représenter la valeur null, une valeur non initialisée, *etc.*...

```
main                                        A.foo
              <>     [1↦bot;0↦bot]                 bot     [0↦bot]
  0 : new B                                   0 : push 0
           <{B}>     [1↦bot;0↦bot]                 bot     [0↦bot]
  1 : store 0                                 1 : return
              <>     [1↦bot;0↦{B}]                 bot     [0↦bot]
  2 : new A                                 B.foo
           <{A}>     [1↦bot;0↦{B}]                  <>     [0↦{B}]
  3 : store 1                                 0 : push 1
              <>     [1↦{A};0↦{B}]               <{1}>     [0↦{B}]
  4 : load 0                                  1 : return
           <{B}>     [1↦{A};0↦{B}]               <{1}>     [0↦bot]
  5 : load 0                                C.foo
     <{B}::{B}>     [1↦{A};0↦{B}]                  <>     [0↦{C}]
  6 : invokevirtual f                         0 : push 2
           <{1}>     [1↦{A};0↦{B}]               <{2}>     [0↦{C}]
  7 : load 0                                  1 : return
     <{B}::{1}>     [1↦{A};0↦{B}]               <{2}>     [0↦bot]
  8 : load 0
 <{B}::{B}::{1}>     [1↦{A};0↦{B}]
  9 : invokevirtual g
     <{2}::{1}>     [1↦{A};0↦{B}]
 10 : numop plus
           <{3}>     [1↦{A};0↦{B}]
A.f
              <>     [0↦{B};1↦{B}]
  0 : load 1
           <{B}>     [0↦{B};1↦{B}]
  1 : invokevirtual foo
           <{1}>     [0↦{B};1↦{B}]
  2 : return
           <{1}>     [0↦bot;1↦bot]
A.g
              <>     [2↦bot;0↦{B};1↦{B}]
  0 : load 1
           <{B}>     [2↦bot;0↦{B};1↦{B}]
  1 : store 2
              <>     [2↦{B};0↦{B};1↦{B}]
  2 : new C
           <{C}>     [2↦{B};0↦{B};1↦{B}]
  3 : store 2
              <>     [2↦{C};0↦{B};1↦{B}]
  4 : load 1
           <{B}>     [2↦{C};0↦{B};1↦{B}]
  5 : load 2
     <{C}::{B}>     [2↦{C};0↦{B};1↦{B}]
  6 : invokevirtual foo
     <{2}::{B}>     [2↦{C};0↦{B};1↦{B}]
  7 : return
     <{2}::{B}>     [2↦bot;0↦bot;1↦bot]
```

FIG. 8.9 – Exemple de programme bytecode analysé

- Piles d'opérandes, variables locales, objets : nous proposons le choix entre une abstraction structurelle (qui mime la structure concrète) ou une seule valeur abstraite pour représenter tous les éléments de la structure[5],
- Tas : il s'agit de la connexion la plus difficile. Nous proposons une abstraction de la forme $\mathsf{P} \to \text{Objet}^\sharp$ avec P un ensemble fini. Le domaine abstrait des références doit alors être de la forme $\mathcal{P}(\mathsf{P})$ pour pouvoir être utilisé avec cette connexion générique.

Ce cas d'étude témoigne de l'intérêt des techniques modulaires de preuves pour proposer une gamme d'analyses certifiées, s'étalant entre précision et efficacité, au lieu d'une seule analyse difficilement modifiable. Dans ce cas précis, modulariser l'algorithme de l'analyseur était facile, mais modulariser sa preuve de correction était beaucoup plus difficile.

8.4 Analyse d'usage mémoire

La dernière analyse à laquelle nous nous sommes intéressé est une analyse d'usage mémoire.

8.4.1 Une sémantique de traces partielles

Nous considérons un langage de bytecode avec manipulations d'objets et de tableaux, et avec des appels et des retours de méthodes (ce langage est en quelque sorte l'union des différents langages présentés dans ce chapitre).

Le but de cette analyse est de prouver que l'exécution d'un programme ne contient qu'un nombre fini d'allocations. Nous considérons pour cela une sémantique de traces partielles.

$$[\![\mathsf{P}]\!] = \left\{ \mathsf{s}_0 :: \mathsf{s}_1 :: \cdots :: \mathsf{s}_n \in \text{Etat}^+ \;\middle|\; \begin{array}{l} \mathsf{s}_0 \in \mathcal{S}_0 \wedge \\ \forall \mathsf{k} < \mathsf{n},\ \exists \mathsf{i},\ \mathsf{s}_\mathsf{k} \to_\mathsf{i} \mathsf{s}_{\mathsf{k}+1} \end{array} \right\} \in \mathcal{P}(\text{Etat}^+)$$

Nous noterons Trace l'ensemble des traces partielles (Etat^+). Un programme aura une utilisation mémoire bornée s'il vérifie le prédicat suivant

$$\text{MemLim}(\mathsf{P}) \stackrel{\text{def}}{=} \exists \mathit{Max}_{\text{new}},\ \forall \mathsf{t} \in [\![\mathsf{P}]\!],\ |\mathsf{t}|_{\text{new}} < \mathit{Max}_{\text{new}}$$

$|\mathsf{t}|_{\text{new}}$ désigne ici le nombre de fois où une instruction new est exécutée dans la trace partielle t.

8.4.2 Algorithmes et domaines abstraits

L'analyse à laquelle nous nous sommes intéressé utilise un simple algorithme de parcours de graphe. Le défi scientifique concerne ici davantage la preuve de correction que l'algorithme. Nous présentons notre algorithme comme un enchaînement d'analyses calculant des approximations correctes de l'ensemble de traces partielles accessibles par un programme. Nous présentons maintenant ces quatre sous-analyses.

[5] Ce type d'abstraction, bien qu'assez brutale, est parfois utilisée pour accélérer le calcul des analyses.

1. Pour chaque méthode $\mathtt{m}$, nous calculons une approximation des ancêtre de $\mathtt{m}$: l'ensemble des méthodes qui appellent $\mathtt{m}$ directement (appel à $\mathtt{m}$) ou indirectement (appel à une méthode qui appelle elle-même $\mathtt{m}$). Le treillis considéré est de la forme
$$\left(\text{NomMethode} \to \mathcal{P}(\text{NomMethode}), \dot{\subseteq}, \dot{\cup}, \dot{\cap}\right)$$
avec une concrétisation définie par
$$\begin{array}{rl}\gamma_{\text{Anc}} : & (\text{NomMethode} \to \mathcal{P}(\text{NomMethode})) \longrightarrow \mathcal{P}(\text{Trace}) \\ & \mathsf{X} \mapsto \left\{ \mathtt{t} \in \text{Trace} \;\middle|\; \begin{array}{l} \text{pour tout } \langle\langle \mathtt{h}, \langle \mathtt{m}, pc, \mathtt{l}, \mathtt{s}\rangle, sf\rangle\rangle \text{ dans } \mathtt{t} \\ \text{pour tout } \mathtt{m}' \text{ apparaissant dans } sf,\ \mathtt{m}' \in \mathsf{X}(\mathtt{m}) \end{array} \right\}\end{array}$$
Nous notons $[\![\mathsf{P}]\!]^\sharp_{\text{Anc}}$ le résultat de cette analyse.
2. Grâce à ce calcul, nous calculons une sur-approximation de l'ensemble des méthodes qui sont dans un cycle d'appels potentiellement récursifs (ou accessibles depuis un tel cycle). Le treillis considéré est simplement
$$(\mathcal{P}(\text{NomMethode}), \subseteq, \cup, \cap)$$
Pour une méthode $\mathtt{m}$ donnée et une trace partielle $\mathtt{t}$, nous disons (et notons $\text{PileAppelSur}(\mathtt{m}, \mathtt{t})$ ce prédicat) que $\mathtt{m}$ est toujours exécutée avec une pile d'appel sûre dans la trace $\mathtt{t}$ si pour tous les états de $\mathtt{t}$, de la forme $\langle\langle \mathtt{h}, \langle \mathtt{m}, pc, \mathtt{l}, \mathtt{s}\rangle, sf\rangle\rangle$, $\mathtt{m}$ n'apparaît pas dans la pile d'appel sf, et toutes les méthodes de sf sont distinctes.
$$\begin{array}{rl}\gamma_{\text{MutRec}} : & \mathcal{P}(\text{NomMethode}) \longrightarrow \mathcal{P}(\text{Trace}) \\ & \mathsf{X} \mapsto \left\{ \mathtt{t} \in \text{Etat}^+ \;\middle|\; \begin{array}{l} \text{pour tout } \mathtt{m} \in \text{NomMethode, si } \mathtt{m} \notin \mathsf{X}, \\ \text{alors la propriété } \text{PileAppelSur}(\mathtt{m}, \mathtt{t}) \text{ est vérifiée} \end{array} \right\}\end{array}$$
Nous notons $[\![\mathsf{P}]\!]^\sharp_{\text{MutRec}}$ le résultat de cette analyse.
3. Nous calculons ensuite les boucles intra-procédurales. Pour chaque méthode $\mathtt{m}$ et chaque point de programme pc de $\mathtt{m}$, nous calculons une sur-approximation $[\![\mathsf{P}]\!]^\sharp_{\text{Pred}}(\mathtt{m}, pc)$ de l'ensemble des points de programmes de $\mathtt{m}$ qui sont atteint avant $(\mathtt{m}, pc)$ au cours d'une exécution de $\mathtt{m}$ (les prédécesseurs de $(\mathtt{m}, \mathtt{pc})$). L'analyse opère donc sur le treillis
$$\left(\text{NomMethode} \times \mathbb{P} \to \mathcal{P}(\mathbb{P}), \dot{\subseteq}, \dot{\cup}, \dot{\cap}\right)$$
avec une concrétisation définie par
$$\begin{array}{rl}\gamma_{\text{Pred}} : & (\text{NomMethode} \times \mathbb{P} \to \mathcal{P}(\mathbb{P})) \longrightarrow \mathcal{P}(\text{Trace}) \\ & \mathsf{X} \mapsto \left\{ \mathtt{t} \in \text{Trace} \;\middle|\; \begin{array}{l} \text{pour tout préfixe } \mathtt{t}' ::: \langle\langle \mathtt{h}, \langle \mathtt{m}, pc, \mathtt{l}, \mathtt{s}\rangle, sf\rangle\rangle \text{ de } \mathtt{t}, \\ \text{si } \text{PileAppelSur}(\mathtt{m}, \mathtt{t}) \\ \text{alors } \mathit{courant}(\mathtt{t}', \mathtt{m}) \subseteq \mathsf{X}(\mathtt{m}, pc) \end{array} \right\}\end{array}$$
avec $\mathit{courant}(\mathtt{t}', \mathtt{m})$ l'ensemble des points de programmes qui apparaissent dans l'exécution courante de $\mathtt{m}$, dans la trace $\mathtt{t}'$.

4. Nous calculons ensuite une sur-approximation de l'ensemble des méthodes qui sont appelées directement ou indirectement depuis une zone de boucle intra-procédurale (zone détectée grâce à l'analyse $[\![\mathsf{P}]\!]^\sharp_{\text{Pred}}$). Le treillis associé est encore une fois

$$(\mathcal{P}(\text{NomMethode}), \subseteq, \cup, \cap)$$

Étant données deux méthodes $\mathtt{m}$ et $\mathtt{m}'$ et une trace partielle $\mathtt{t}$, notons par $\text{UnAppel}(\mathtt{m}, \mathtt{m}', \mathtt{t})$ le fait que $\mathtt{m}$ est appelé au plus une fois dans chaque exécution de $\mathtt{m}'$, dans la trace $\mathtt{t}$. La concrétisation de cette analyse (appelée $[\![\mathsf{P}]\!]^\sharp_{\text{AppelBoucle}}$) est alors définie par

$$\gamma_{\text{AppelBoucle}} : \mathcal{P}(\text{NomMethode}) \longrightarrow \mathcal{P}(\text{Trace})$$
$$\mathsf{X} \mapsto \left\{ \mathtt{t} \in \text{Trace} \;\middle|\; \begin{array}{l} \text{pour tout préfixe } \mathtt{t}' ::: \langle\langle \mathtt{h}, \langle \mathtt{m}, pc, \mathtt{l}, \mathtt{s}\rangle, sf\rangle\rangle \text{ de } \mathtt{t}, \\ \text{si PileAppelSur}(\mathtt{m}, \mathtt{t}) \text{ et } \mathtt{m} \notin \mathsf{X}, \\ \text{alors pour tout méthode } \mathtt{m}', \\ \text{UnAppel}(\mathtt{m}, \mathtt{m}', \mathtt{t}) \text{ est vérifié} \end{array} \right\}$$

Toutes ces analyses sont facilement programmables à l'aide d'algorithmes de parcours de graphe. Pour les inscrire dans le cadre théorique retenu dans nos travaux, nous les spécifions à l'aide de systèmes d'inéquations. Nous profitons ainsi de la méthode de preuve résumée en introduction pour accomplir la preuve de correction vis à vis de la sémantique de traces partielles. Le calcul de $[\![\mathsf{P}]\!]^\sharp_{\text{Anc}}$ est par exemple simplement caractérisé par la contrainte

$$\frac{(\mathtt{m}, pc) : \text{invokevirtual } \mathtt{m}_{\text{ID}} \quad \mathtt{m}' \in \text{implem}(\mathsf{P}, \mathtt{m}_{\text{ID}})}{[\![\mathsf{P}]\!]^\sharp_{\text{Anc}}(\mathtt{m}) \cup \{\mathtt{m}\} \subseteq [\![\mathsf{P}]\!]^\sharp_{\text{Anc}}(\mathtt{m}')}$$

avec $\text{implem}(\mathsf{P}, \mathtt{m}_{\text{ID}})$ une sur-approximation de l'ensemble des méthodes susceptibles d'être atteintes par l'appel virtuel invokevirtual $\mathtt{m}_{\text{ID}}$ (information grossièrement obtenue par un parcours de la hiérarchie de classe).

La dernière étape consiste à utiliser les résultats précédents pour vérifier l'usage mémoire du programme. Il nous suffit pour cela de parcourir l'ensemble des points $(\mathtt{m}, pc)$ où une allocation est réalisée (avec une instruction new[6]) et de vérifier la conjonction des trois condition suivantes

- $\mathtt{m} \notin [\![\mathsf{P}]\!]^\sharp_{\text{MutRec}}$: $\mathtt{m}$ n'est pas dans une boucle inter-procédurale d'appels récursifs (éventuellement mutuels),
- $\mathtt{m} \notin [\![\mathsf{P}]\!]^\sharp_{\text{AppelBoucle}}$: $\mathtt{m}$ n'est jamais appelé depuis une zone de boucle intra-procédurale,
- $pc \notin [\![\mathsf{P}]\!]^\sharp_{\text{Pred}}$: l'instruction en $(\mathtt{m}, pc)$ n'est pas dans une boucle interne.

La partie la plus difficile de ce cas d'étude concerne la preuve de correction de cette dernière étape. Il nous faut montrer qu'un programme passant avec succès les tests abstraits attachés à chacune de ses instructions new, vérifie le prédicat MemLim. Le raisonnement suit les étapes suivantes, en ne considérant que les méthodes qui vérifient les tests précédents :

[6] Nous confondons ici allocation d'objet et allocation de tableaux.

1. le nombre d'occurrences d'une instruction new dans une trace partielle t est caractérisé comme la somme sur toutes les méthodes et tous les point de programme du nombres $|\mathsf{t}|^{\mathsf{m},pc}_{\text{new}}$ d'occurrence de l'instruction new dans t et dans un état positionné en (m, pc)

$$\forall \mathsf{t} \in [\![\mathsf{P}]\!],\ |\mathsf{t}|_{\text{new}} = \sum_{\mathsf{m}} |\mathsf{t}|^{\mathsf{m},pc}_{\text{new}}$$

2. grâce aux hypothèses faites sur le résultat de l'analyse, ce nombre $|\mathsf{t}|^{\mathsf{m},pc}_{\text{new}}$ est majoré par le nombre d'exécutions de m dans la trace t (noté $\mathit{Exec}(\mathsf{m}, \mathsf{t})$)

$$\forall \mathsf{t} \in [\![\mathsf{P}]\!],\ |\mathsf{t}|^{\mathsf{m},pc}_{\text{new}} \leq \mathit{Exec}(\mathsf{m}, \mathsf{t})$$

3. toujours sous les même hypothèses, nous prouvons une relation entre les différents $\mathit{Exec}(\mathsf{m}, \mathsf{t})$, $\mathsf{m} \in \text{NomMethode}$.

$$\mathit{Exec}(\mathsf{m}, \mathsf{t}) \leq \sum_{\mathsf{m} \in \mathit{Appel}(\mathsf{m}')} \mathit{Exec}(\mathsf{m}', \mathsf{t}) \cdot \mathit{Max}_{\texttt{invoke}}(\mathsf{m}').$$

   avec $\mathit{Appel}(\mathsf{m}')$ l'ensemble des méthodes appelées par m' et $\mathit{Max}_{\text{invokevirtual}}(\mathsf{m}')$ le nombre d'instruction invokevirtual apparaissant dans m',

4. cette dernière relation nous permet de prouver l'existence d'une borne $\mathit{Max}_{\text{exec}}$ pour $\mathit{Exec}(\mathsf{m}, \mathsf{t})$

$$\forall \mathsf{t} \in [\![\mathsf{P}]\!],\ \mathit{Exec}(\mathsf{m}, \mathsf{t}) \leq \mathit{Max}_{\text{exec}}$$

5. tous ces résultats nous permettent finalement de borner $|\mathsf{t}|_{\text{new}}$

$$\forall \mathsf{t} \in [\![\mathsf{P}]\!],\ |\mathsf{t}|_{\text{new}} = \sum_{\mathsf{m},pc} |\mathsf{t}|^{\mathsf{m},pc}_{\text{new}} \leq \mathit{MethodMax}_{\mathsf{P}} \cdot \mathit{PointProgMax}_{\mathsf{P}} \cdot \mathit{Max}_{\texttt{invoke}}$$

   avec $\mathit{MethodMax}_{\mathsf{P}}$ le nombre de méthodes dans le programme P et $\mathit{PointProgMax}_{\mathsf{P}}$ le nombre maximum de points de programme par méthode.

Il s'agit ici d'une preuve[7] purement sémantique qui demande des raisonnement combinatoires non triviaux pour le niveau de détail requis dans un assistant de preuve. Cet exemple illustre les difficultés relatives à la dernière étape d'une analyse statique : tirer parti des informations calculées pour prouver une propriété sur l'exécution d'un programme (prouver qu'une optimisation est alors possible, qu'un certain type d'erreurs ne pourra jamais se produire, *etc...*). Les algorithmes utilisés peuvent être plus moins complexes et reposent directement sur des propriétés sémantiques du langage. Le cadre théorique est moins balisé que pour les étapes précédentes, ce qui explique, à notre avis, ces difficultés.

[7] Plus de détails peuvent être trouvés dans l'article [CJPS05] présentant cette analyse.

8.4.3 Expériences

Cette analyse est assez restrictive, puisqu'elle rejette beaucoup de programmes valides. Elle démontre néanmoins la capacité de notre cadre à raisonner sur des analyses de natures variées.

Elle nous permet aussi de démontrer l'efficacité des structures de données employées dans notre bibliothèque de treillis. Elle repose en effet sur la manipulations d'ensembles finis. Ces ensembles sont codés comme des fonctions à valeurs booléennes, ce qui revient à utiliser des représentations binaires portant des booléens dans leur feuilles. Nous avons expérimenté l'efficacité de ces manipulations ensemblistes en testant cette analyse sur des gros programmes. Nous utilisons pour cela un générateur aléatoire de programmes en bytecode, qui construit des programmes de taille arbitraire. Les programmes générés ne sont pas forcément cohérents (en particulier, ils ne passent pas le vérificateur de bytecode Java) mais la simplicité de l'analyse (beaucoup d'instructions sont traitées comme des instructions nop) nous permet de valider cette expérience : les temps mesurés reflètent bien les temps de calcul sur des programmes bien formés de taille équivalente. Les résultats mesurés sont donnés dans la figure 8.10. La première ligne présente une évolution du temps d'exécution en fixant le nombre de lignes de bytecode par méthode et en augmentant le nombre de méthodes. Dans la deuxième ligne, le nombre de méthodes est fixé mais le nombre de lignes par méthode augmente. La première colonne présente les temps d'exécution tandis que la deuxième présente l'occupation mémoire. Les temps d'exécution sont très bons : moins de deux minutes sont nécessaires pour analyser plus de 100.000 lignes de code. L'occupation mémoire est moins bonne. Il serait sans doute utile d'utiliser une représentation plus compacte pour les ensembles calculés par $[\![\mathsf{P}]\!]^\sharp_{\mathrm{Pred}}$, car les éléments de ces ensembles sont souvent des entiers consécutifs.

8.5 Conclusions

Nous avons présenté différents cas d'études qui démontrent la capacité de notre cadre à traiter des analyses variées pour un langage réaliste comme le bytecode Java. La première analyse est une analyse d'intervalle pour assurer la validité des accès aux tableaux. C'est le cas d'étude qui repose le plus sur le cadre génériques de notre travail (notamment avec l'utilisation d'opérateurs d'élargissement et de rétrécissement). Afin d'adapter l'analyse par intervalle des programmes WHILE au bytecode Java, nous avons proposé une technique originale[8] de décompilation abstraite des expressions. Une application de cette analyse pour le *proof carrying code* est en cours de développement (nous y reviendrons dans les perspectives du chapitre 9). La deuxième analyse constitue le premier cas d'étude abordé au cours de notre thèse. Cette analyse est inspiré de l'analyse proposée par René Rydhof Hansen [Han02]. Ce travail a donné lieu à une publication [CJPR04] (ainsi qu'à une version étendue [CJPR05]). L'analyse d'usage mémoire de la dernière section a elle aussi donné lieu à une publication [CJPS05].

[8] Parallèlement à nos travaux, une technique similaire a été proposée par Martin Wildmoser, Amine Chaieb et Tobias Nipkow [WCN05].

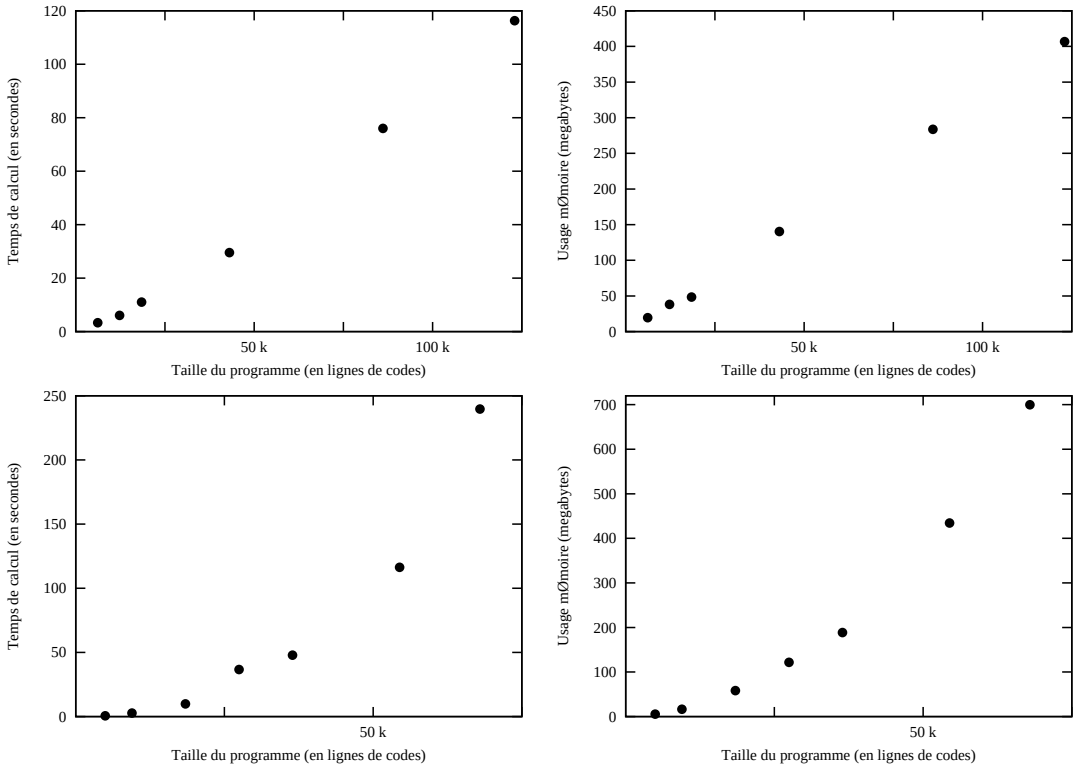


FIG. 8.10 – Mesure de performance de l'analyse d'usage mémoire. La première ligne présente une évolution du temps d'exécution en fixant le nombre de lignes de bytecode par méthode et en augmentant le nombre de méthodes. Dans la deuxième ligne, le nombre de méthodes est fixé mais le nombre de lignes par méthode augmente.

Plusieurs aspects du langage Java ont été laissés de côté : les exceptions, les sous-routines et le parallélisme. Les exceptions nous semblent tout à fait abordables. Nous avons déjà entamé une extension de l'analyse de classe gérant les exceptions. Il nous paraît pour cela intéressant de définir une sémantique intermédiaire plus adaptée à l'analyse des exceptions que la sémantique standard. Les sous-routines posent un problème assez difficile à résoudre, alors que leur intérêt pour le langage Java n'est forcément démontrés. Gerwin Klein et Martin Wildmoser [KW03] proposent déjà une formalisation complète du traitement des sous-routines en Java. Le parallélisme constitue le dernier morceau que nous n'avons pas du tout abordé pour l'instant.

Nous disposons au final de trois types d'analyses : une analyse numérique (à base d'intervalles), une analyse de classe[9] (pour raffiner le graphe de flot de contrôle) et une analyse d'usage mémoire. La combinaison de ces analyses serait sans doute intéressante puisque l'analyse de classe permettrait d'améliorer l'analyse d'usage mémoire en enlevant certains cycles impossibles en raison des appels virtuels de méthode. Une analyse numérique serait aussi nécessaire pour compter le nombre d'allocations effectuées dans les boucles. Une analyse relationnelle entre les variables du programme et la taille du tas serait alors indispensable [Hub05]. Nous y reviendrons dans les perspectives de notre travail de thèse, au cours du prochain chapitre.

[9] Nous y incluons l'analyse modulaire de la section 8.3 qui permet d'optimiser la précision de l'analyse de classe.

Chapitre 9

Conclusions

9.1 Bilan sur nos travaux

La première étape de notre travail a été de proposer un cadre basé sur l'interprétation abstraite et adapté à la preuve de correction et à l'extraction d'analyseurs statiques dans l'assistant de preuve Coq. Nous avons choisi de ne pas utiliser la notion de connexion de Galois dans nos travaux. Comme nous l'avons expliqué dans le chapitre 3, la construction d'une connexion de Galois, même la plus simple, est difficile en Coq. Cette difficulté peut être en partie expliquée par le fait que les connexions de Galois servent à relier deux mondes de natures différentes : le monde logique de la sémantique concrète et le monde de la sémantique abstraite, sur lequel les calculs de l'analyse sont réalisés. Si le passage du second au premier ne pose pas de problème (grâce à une fonction de concrétisation) le passage inverse est beaucoup plus difficile car le système de types de Coq impose des restrictions fortes sur le construction d'objets informatifs à partir d'objets logiques. Cette restriction est néanmoins nécessaire pour assurer la cohérence du mécanisme d'extraction, outil que nous utilisons activement dans nos travaux. Nous avons donc renoncé à utiliser la notion de fonction d'abstraction et nous nous sommes concentrés sur les fonctions de concrétisation. La pertinence de ce choix a été démontrée par les différents cas d'étude que nous avons été en mesure de proposer.

Pour ce qui concerne la partie plus algorithmique de l'interprétation abstraite, nous avons fait moins de concessions. Nous avons ainsi été capables d'utiliser les techniques d'accélération de convergence basées sur la notion d'opérateurs d'élargissement et de rétrécissement. Ces techniques ont été présentées dans le chapitre 4. Disposer d'outils génériques pour implémenter les analyses est définitivement une grande chance pour un développement formel. Les cas d'étude que nous avons formalisés montrent que le plus gros du travail réside ainsi dans la spécification d'une analyse, et non dans la résolution de cette spécification (*i.e.* le calcul d'une solution d'un système d'inéquations).

Ces techniques génériques de calcul itératif demandent cependant de prouver, pour chaque analyse, une propriété de terminaison. Ces propriétés requièrent les preuves les plus techniques de notre travail. Elles constituaient a priori un frein important pour le développement d'analyses certifiées par un utilisateur Coq néophyte. Notre démarche

a alors été de proposer dans le chapitre 6 une librairie de foncteurs permettant de construire de façon compositionnelle des treillis, munis d'un critère de terminaison. La simple combinaison de foncteurs permet ainsi de prouver des propriétés de terminaison extrêmement techniques. Nous pensons qu'il s'agit là d'une contribution qui dépasse le cadre des analyses statiques certifiées. Les preuves de terminaison constituent un passage souvent technique, mais néanmoins nécessaire, dans les assistants de preuve. Nous proposons ici de placer ces preuves de terminaison au même niveau que la construction du domaine de base des algorithmes (dans notre cas, des treillis) et indépendamment des calculs exacts effectués dans les algorithmes (dans notre cas, les inéquations spécifiant la sémantique abstraite).

Cette librairie de treillis constitue néanmoins la seule véritable portion de développement Coq réellement réutilisable pour une large palette d'analyses et de langages de programmation. Il nous est apparu beaucoup plus difficile de développer une construction modulaire, et indépendante d'une analyse et d'un langage, pour les preuves de correction des spécifications de sémantique abstraite. Cette difficulté est notamment apparue en essayant de découper la preuve de correction de nos analyses sur le bytecode Java. Dans ce contexte, nous avons introduit la notion de foncteur de concrétisation pour permettre l'utilisation de constructions génériques indépendantes de leur contexte d'utilisation. Nous donnons cependant une solution partielle à ce problème puisque la technique que nous proposons dans le chapitre 7 reste cependant dédiée à un certain type d'abstraction. Nous l'avons néanmoins appliquée à une analyse de références standard pour un langage de bytecode objet comme Java. Une réflexion plus poussée est encore nécessaire pour comprendre quel est le cadre d'application général de cette technique pour les analyses relationnelles.

Nous avons finalement entrepris plusieurs cas d'études pour illustrer les différentes techniques présentées dans cette thèse. Tous ces cas d'études, présentés dans le chapitre 5 et dans le chapitre 8, sont basés sur la bibliothèque de treillis. La figure 9.1 récapitule la taille de ces différent cas d'études, en nombre de lignes de Coq. Nous excluons à chaque fois la partie dédiées aux treillis qui repose sur notre librairie. L'analyse modulaire du langage While comporte le plus grand nombre de lignes parce qu'elle contient plusieurs abstractions numériques (intervalles, signes, congruences) « gourmandes » en preuve.

sujet	**nombres de lignes Coq**
Librairie de treillis	13000
Analyse modulaire du langage While	8500
Analyse de classe	6500
Analyse de référence avec preuve modulaire	4000
Analyse d'usage mémoire	5000
Analyse d'intervalle sur du bytecode	5000
Total	42000

Fig. 9.1 – Nombre de lignes de Coq développées durant cette thèse

9.2 Perspectives

Nous présentons maintenant les perspectives de nos travaux.

Connexions de Galois L'utilisation des connexions de Galois semble actuellement inadaptée à la logique de Coq. Ces structures ne nous ont, au final, pas trop manqué car nous nous sommes concentrés sur la preuve de correction des analyses. Les connexions de Galois proposent cependant une fonctionnalité fort utile. La possibilité de dériver la définition d'un opérateur abstrait $f^\sharp$ à partir de sa spécification optimale $\alpha \circ f \circ \gamma$. Les notes de cours de Patrick Cousot sur la formalisation d'un interpréteur abstrait pour le langage WHILE proposent de nombreux exemples d'utilisation de cette technique. Le niveau de détail qui y est donné laisse penser que nous ne sommes plus très loin d'une preuve vérifiable par un ordinateur. Encore faut-il disposer d'un assistant de preuve apte à raisonner sur ce type de déduction logique.

Il nous semble intéressant de proposer un assistant dédié à ce type de tâche. Un assistant basé sur la théorie des ensembles pourrait convenir, mais serait peut-être encore trop général pour la tâche spécifique qui nous intéresse. Afin de conserver une bonne fiabilité dans cet assistant, nous pourrions proposer un mécanisme de traduction des raisonnements de cet assistant vers Coq. Cet assistant dédié permettrait de faire une construction progressive de $f^\sharp$ de la forme suivante

$$\begin{array}{rcl} \alpha \circ f \circ \gamma(x) & = & exp_1 \\ & & \dots \\ & = & exp_k \\ & \sqsubseteq^\sharp & exp_{k+1} \\ & & \dots \\ & = & f^\sharp(x) \end{array}$$

Les étapes où $\sqsubseteq^\sharp$ apparaît indiquent une perte de précision. Il est intéressant de pouvoir faire clairement apparaître le moment précis où de telles décisions sont prises. Pour transformer ce raisonnement en une preuve Coq, nous ne gardons que le sens suffisant pour la correction de $f^\sharp$:

$$\begin{array}{rcl} \alpha \circ f \circ \gamma(x) & \sqsubseteq^\sharp & exp_1 \\ & & \dots \\ & \sqsubseteq^\sharp & exp_k \\ & \sqsubseteq^\sharp & exp_{k+1} \\ & & \dots \\ & \sqsubseteq^\sharp & f^\sharp(x) \end{array}$$

Il nous faut ensuite éliminer les utilisations de la fonction d'abstraction, qui n'existera plus dans la preuve Coq produite. Nous utilisons pour cela les propriétés des connexions

de Galois. Le raisonnement précédent est ainsi équivalent à

$$\begin{aligned} f \circ \gamma(x) &\sqsubseteq \gamma(exp_1) \\ &\cdots \\ &\sqsubseteq \gamma(exp_k) \\ &\cdots \\ &\sqsubseteq \gamma \circ f^\sharp(x)) \end{aligned}$$

Nous obtenons alors une preuve, plus facile à traduire en Coq. Nous ne donnons ici que l'idée de base de cette perspective de travail. Nous n'avons pas étudié en profondeur les limites de ce type de transformation, ni la totalité des règles de déduction nécessaires pour construire un tel assistant de preuve dédié.

***Proof carrying code* (PCC)** Une application directe de nos travaux concerne le *proof carrying code* [Nec97]. Cette technique permet à un utilisateur craintif de télécharger un programme accompagné d'une preuve. Cette preuve atteste que le programme vérifie une certaine politique de sécurité. L'utilisateur dispose d'un outil de vérification pour attester que la preuve est valide pour le programme fourni.

Rappelons que la preuve de correction de nos analyses suit essentiellement deux étapes. La première consiste à définir un système d'inéquations pour spécifier la sémantique abstraite de l'analyse. Nous prouvons ensuite que les solutions de ce système sont des approximations correctes de la sémantique d'un programme. En notant $F^\sharp_P$ la fonctionnelle associée au système d'inéquations, cette étape peut être formalisée par l'énoncé suivant

$$\forall P,\ \forall s^\sharp,\quad F^\sharp_P(s^\sharp) \sqsubseteq^\sharp s^\sharp \ \Rightarrow\ [\![P]\!] \sqsubseteq \gamma(s^\sharp) \tag{9.1}$$

La deuxième étape consiste ensuite à construire un solveur (noté solve) pour ce système d'inéquations.

$$\forall P,\ F^\sharp_P(\mathrm{solve}(P)) \sqsubseteq^\sharp \mathrm{solve}(P)$$

Optionnellement, une dernière vérification peut être réalisée sur les solutions de ce système (par une fonction à valeur booléenne $\hat{\mathrm{Sur}}^\sharp$) pour vérifier si une certaine propriété Sûr est assurée par le programme.

$$\forall P, \forall s^\sharp, [\![P]\!] \sqsubseteq \gamma(s^\sharp) \wedge \hat{\mathrm{Sur}}^\sharp(P)(s^\sharp) = T \ \Rightarrow\ \hat{\mathrm{Sur}}(P) \tag{9.2}$$

En combinant ces trois étapes nous obtenons ainsi un vérificateur de programme certifié $\hat{\mathrm{Sur}}^\sharp \circ \mathrm{solve}$.

$$\forall P,\ \hat{\mathrm{Sur}}^\sharp \circ \mathrm{solve}(P) = T \ \Rightarrow\ \hat{\mathrm{Sur}}(P)$$

Dans le cadre du *proof carrying code*, la programmation certifiée de solve est inutile. Nous pouvons en effet proposer une architecture PCC dans laquelle les certificats abstraits sont des éléments du domaine abstrait. Un vérificateur de preuve est alors défini par

$$\mathrm{verif}(P, s^\sharp) \stackrel{\mathrm{def}}{=} F^\sharp_P(s^\sharp) \sqsubseteq^\sharp s^\sharp \wedge \hat{\mathrm{Sur}}(P)^\sharp(s^\sharp) = T$$

Dans le scénario classique du PCC, un producteur veut envoyer un programme P à un consommateur. Le consommateur veut être certain que P vérifie la propriété Sûr et il ne fait pas confiance au producteur. Le producteur envoie donc avec un P un certificat $s^\sharp$ qu'il a calculé avec une version non-nécessairement certifiée de $F^\sharp_P$ et solve. Le consommateur n'a alors plus qu'à appliquer verif sur P et $s^\sharp$. Si la réponse de ce calcul est T, il a alors l'assurance que le programme P est sûr. Les propriétés (9.1) et (9.2) impliquent en effet la correction formelle du vérificateur de preuve verif

$$\forall P,\ \forall s^\sharp,\ \mathrm{verif}(P, s^\sharp) = T \ \Rightarrow\ \mathrm{Sûr}(P)$$

Pour parler de véritable architecture PCC, encore faut-il proposer des certificats de petites tailles et des temps de vérification courts. En collaboration avec Frédéric Besson et Thomas Jensen, nous avons développé une telle architecture. Les expériences menées sur l'analyse de bytecode Java par intervalle démontrent l'intérêt de cette technique (voir figure 9.2). Nous concilions ainsi les qualités de l'architecture PCC classique (telle qu'elle a été proposée par Georges Necula) avec celle du *foundational proof carrying code* [App01] proposé par Andrew Appel. Nous permettons en effet d'accompagner le vérificateur de sa preuve de correction, elle même vérifiable par le vérificateur de preuve Coq. Cette preuve de correction est directement formulée vis-à-vis de la sémantique du programme. Elle peut être envoyée par le producteur comme phase d'initialisation des communications.

programme	**taille du fichier .java**	**taille du fichier .class**	**taille du certificat**	**temps de vérification du certificat**
BubbleSort	440	528	32	0.015
HeapSort	1044	858	63	0.050
QuickSort	1078	965	124	0.060
ConvolutionProduct	378	542	52	0.010
FloydWharshall	417	596	134	0.020
PolynomProduct	509	604	87	0.010

FIG. 9.2 – Temps de vérification (en seconde) des certificats et taille des fichiers (en bytes) pour l'architecture PCC certifiée (vérification des accès aux tableaux pour le bytecode Java)

Utilisation d'une analyse certifiée comme une procédure de décision Une dernière perspective de notre travail concerne cette fois la preuve de programme dans les assistants de preuve. Dans nos travaux de thèse, l'assistant de preuve se met au service de l'analyse statique pour démontrer sa correction. Pourquoi ne pas inverser les rôles ?

Si l'on désire faire une preuve interactive sur un programme de bytecode Java, nous pouvons raisonner sur la sémantique $[\![P]\!]$ telle que nous l'avons définie en Coq, durant nos travaux. Ce type de preuve interactive s'annonce fastidieux, mais admettons que nous

ayons prouvé une méthode de preuve par pré-condition/post-condition qui nous allège un peu le travail. Nous pouvons tout à fait lancer une analyse statique pour obtenir des invariants sur les états accessibles de P, susceptibles de nous aider à accomplir notre preuve interactive. Comme nous travaillons dans un assistant de preuve muni d'un vérificateur de preuve, nous ne pouvons pas utiliser n'importe quel analyseur statique. Il faut en effet être capable de construire un terme de preuve assurant que les informations calculées par l'analyse sont correctes.

Une fois encore nos travaux de thèse s'appliquent directement puisque nous avons programmé des analyseurs complets en Coq. Dans leurs versions richement typées, ces analyseurs calculent une preuve de la correction de leur résultat. Dans le cas où le calcul de ces analyses serait trop long pour le moteur de réduction de Coq[1] nous pouvons reléguer le calcul de point fixe à un outil externe et seulement effectuer une vérification de point fixe en Coq. Cette approche est encore largement prospective, mais il y a fort à parier que les assistants de preuve auront un jour où l'autre besoin de ce type de procédure de décision pour les aider à affronter des programmes toujours plus grands. La collaboration entre assistants de preuve et analyseurs statiques n'est pas terminée !

[1] Celui ci a cependant récemment fait des progrès grâce à la machine virtuelle proposée par Benjamin Grégoire et Xavier Leroy [GL02].

Annexe A

Fichiers Coq

Tous ces fichiers sont regroupés sur la page web [Web] associée à ce document.

Chapitre 2 : Programmation dans le Calcul des Constructions Inductives

- `bin.v` : exemple de manipulation de types dépendants avec la notion d'entiers binaires de taille bornée
- `treemap.v` : codage des maps par des arbres binaires et des clés binaires

Chapitre 3 : Spécification d'analyse statique par interprétation abstraite

- `complete.v` : définition des treillis complets et preuves des théorèmes 3.1.1 et 3.2.2
- `galois.v` : définition des connexions de Galois et preuves de certaines propriétés de base
- `mod2.v` : construction de la connexion de Galois associée à l'exemple 2
- `pfp.v` : expression de l'ensemble des états accessibles comme un plus petit point fixe

Chapitre 4 : Calcul et approximation de points fixes

- `lattice_def.v` : définition de la signature de module `Lattice`, définitions générales sur les treillis, résultats généraux sur les treillis.
- `solve_wf.v` : solveur de (post-)points fixes dans les treillis vérifiant la condition de chaîne ascendante.
- `solve_widen_classic.v` : solveur de post-points fixes avec les opérateurs d'élargissement/rétrécissement standards.
- `solve_widen.v` : solveur de post-points fixes avec les opérateurs d'élargissement/-rétrécissement dans leurs versions modifiées.

- `solve.v` : définition de la signature `Solve` des treillis munis d'un critère de terminaison, résultats généraux.
- `acc_chaotic.v` : preuves de terminaisons pour l'itérateur chaotique.
- `chaotic.v` : itérateur chaotique.

Chapitre 5 : Un interpréteur abstrait modulaire pour un langage While

- `semantic.v` : syntaxe et sémantique du langage
- `comp.v` : normalisation des tests
- `collect.v` : collecte de tous les indices d'un programme pour génerer un système d'équation
- `AbStateLift.v` : abstraction des états
- `AbEnvNotRelational.v` : abstraction des environnements de variable
- `AbEnvNotRelationalWithTGuardReduction.v` : abstraction (avec réduction itérative des guardes) des environnements de variable
- `z_extra_props.v` : résultats généraux sur les entiers relatifs
- `AbSign.v` et `sign.v` : analyse par signe
- `AbModulo.v` : analyse par congruence
- `interval.v`, `interval_op.v` et `AbInterval.v` : analyse d'intervalle
- `AbstractionsSignatures.v` : signatures des abstractions d'environnements de variables et des valeurs numériques
- `analysers.v` : instantiation et extraction de plusieurs analyseurs certifiés

Chapitre 6 : Composition constructive de treillis

- `wf_prop.v` : résultats généraux sur le prédicat d'accessibilité
- `widen_classic_prop.v` : résultats généraux sur les opérateurs d'élargissement/rétrécissement
- `partial_lattice.v` : notion de treillis partiel
- `new_implem.v` : re-programmation de treillis
- `add_option.v` : résultats permettant de rajouter un plus petit ou un plus grand élément à un treillis
- `Word.v` : mots binaires de taille bornée
- `TabTree.v` : tableaux fonctionnelles implémentés avec des arbres
- `BinTree.v` : tableaux fonctionnelles implémentés avec des arbres de hauteurs bornées
- `Single.v` : treillis réduit à un seul élément
- `SumFunctors.v` : Somme disjointe de treillis
- `Sum3Functors.v` : Somme disjointe de 3 treillis
- `ProdFunctors.v` : Produit de treillis
- `ListFunctors.v` : Liste de treillis
- `FuncSignature.v` : Signature des implémentations de fonctions

– `FuncFunctors.v` : treillis de fonctions pour une implémentation quelconque
– `ArrayBinFunctors.v` : treillis de fonctions pour l'implémentation par arbre binaire

Bibliographie

[Acz77] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, pages 739–. North-Holland Publishing Company, 1977.

[App01] Andrew W. Appel. Foundational proof-carrying code. In *Proc. of 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 247–258. IEEE Press, 2001.

[Ath] K. Arkoudas and others. Athena. http://www.pac.lcs.mit.edu/athena.

[Bal02] Antonia Balaa. *Fonctions récursives générales dans le calcul des constructions*. PhD thesis, Université de Nice - Sophia Antipolis, 2002. ftp://ftp-sop.inria.fr/lemme/Antonia.Balaa/main.ps.gz.

[Bar91] H. Barendregt. Lambda calculi with types. Technical report, Catholic University Nijmegen, 1991. In Handbook of Logic in Computer Science, Vol II.

[BC04] Yves Bertot and Pierre Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[BDHdS01] Gilles Barthe, Guillaume Dufay, Marieke Huisman, and Simão Melo de Sousa. Jakarta : A Toolset for Reasoning about JavaCard. In *Proc. of International Conference on Research in Smart Cards, E-smart 2001*, number 2140 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[BDJ+01] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, and Simão Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In *Proc. of 10th European Symposium on Programming, ESOP'01*, number 2028 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[Ber01] Yves Bertot. Formalizing a JVML Verifier for Initialization in a Theorem Prover. In *Proc. of 13th International Conference on Computer Aided Verification, CAV 2001*, number 2102 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[BN04] Gilles Barthe and Leonor Prensa Nieto. Formally verifying information flow type systems for concurrent and thread systems. In *Proc. of the 2004 ACM*

Workshop on Formal Methods in Security Engineering, FMSE 2004, pages 13–22, 2004.

[Bou92] François Bourdoncle. *Sémantiques des Langages Impératifs d'Ordre Supérieur et Interprétation Abstraite*. PhD thesis, École Polytechnique, 1992. http://www.exalead.com/Francois.Bourdoncle/these.html.

[Bov02] Ana Bove. *General Recursion in Type Theory*. PhD thesis, Department of Computing Science, Chalmers University of Technology, 2002. http://www.cs.chalmers.se/~bove/Papers/phd_thesis.ps.gz.

[Cam] The Objective Caml language. http://caml.inria.fr/.

[CBR02] Ludovic Casset, Lilian Burdy, and Antoine Requet. Formal Development of an embedded verifier for Java Card Byte Code. In *Proc. of IEEE Int. Conference on Dependable Systems & Networks (DSN)*, 2002.

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

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, POPL'77*, pages 238–252. ACM Press, 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, ACM POPL'79*, pages 269–282. ACM Press, 1979.

[CC92a] Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3) :103–179, 1992.

[CC92b] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4) :511–547, 1992.

[CC92c] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the Fourth International Symposium, PLILP'92*, pages 269–295. LNCS, 1992.

[CC94] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 95–112, Toulouse, France, 16–19 May 1994. IEEE Computer Society Press, Los Alamitos, California.

[CCF+05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer-Verlag, 2005.

[CGD04] Solange Coupet-Grimal and William Delobel. A uniform and certified approach for two static analyses. In *Types for Proofs and Programs, International Workshop, TYPES 2004*, volume 3839, pages 115–137, 2004.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, POPL'78*, pages 84–97, 1978.

[CH88] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 1988.

[CJPR04] David Cachera, Thomas Jensen, David Pichardie, and Vlad Rusu. Extracting a Data Flow Analyser in Constructive Logic. In *Proc. of 13th European Symposium on Programming (ESOP'04)*, number 2986 in Lecture Notes in Computer Science, pages 385–400. Springer-Verlag, 2004.

[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. Extended version of [CJPR04].

[CJPS05] David Cachera, Thomas Jensen, David Pichardie, and Gerardo Schneider. Certified memory usage analysis. In *Proc. of 13th International Symposium on Formal Methods (FM'05)*, number 3582 in Lecture Notes in Computer Science, pages 91–106. Springer-Verlag, 2005.

[CM95] Guy Cousineau and Michel Mauny. *Approche Fonctionnelle de la Programmation*. Collection Informatique. Ediscience, 1995.

[Coq] The Coq Proof Assistant. http://coq.inria.fr/.

[Cou78] Patrick Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes*. PhD thesis, Thèse d'état ès sciences mathématiques, Université scientifique et médicale de Grenoble, France, 1978.

[Cou99] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[CPM90] Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Colog'88*, 1990.

[CPS02] Laurent Chicli, Loïc Pottier, and Carlos Simpson. Mathematical quotients and quotient types in coq. In *Proc. of Second International Workshop on Types for Proofs and Programs, TYPES 2002*, pages 95–107, 202.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[Fer05] Jérôme Feret. *Analysis of mobile systems by abstract interpretation*. PhD thesis, École Polytechnique, 2005. http://www.di.ens.fr/~feret/thesis/thesis.pdf.

[FL04] Jean-Christophe Filliâtre and Pierre Letouzey. Functors for Proofs and Programs. In *Proc. of 13th European Symposium on Programming (ESOP'04)*, number 2986 in Lecture Notes in Computer Science, pages 370–384. Springer-Verlag, 2004.

[GBL04] Benjamin Grégoire, Yves Bertot, and Xavier Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In *Types for Proofs and Programs, International Workshop, TYPES 2004*, volume 3839, pages 66–81, 2004.

[GJKP04] Thomas Genet, Thomas Jensen, Vikash Kodati, and David Pichardie. A Java Card CAP converter in PVS. In Jens Knoop and Wolf Zimmermann, editors, *Proc. of 2nd International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2003), Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2004.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *International Conference on Functional Programming 2002*, pages 235–246, 2002.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs an Types*. Cambridge Tracts in Theoretical Computer Science 7, Cambridge University Press, 1989.

[Gra89] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30 :165–190, 1989.

[Gra92] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. In *12th Foundations of Software Technology and Theoretical Computer Science Conference,*, pages 68–79, 1992.

[Han02] René Rydhof Hansen. Flow Logic for Carmel. Technical Report SECSAFE-IMM-001, Danish Technical University, 2002.

[Har96] John Harrison. Hol light : A tutorial introduction. In *Proc. of the First International Conference on Formal Methods in Computer-Aided Design, FMCAD'96*, pages 265–269, 1996.

[HKPM04] G. Huet, G. Kahn, and Ch. Paulin-Mohring. *The* Coq *Proof Assistant - A tutorial - Version 8.0*, April 2004.

[How80] H.A. Howard. The formulae-as-types notion of constructions. In J. Seldin and J.R. Hindley, editors, *to H.B. Curry : Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[Hub05] Laurent Hubert. Analyse de ressources pour bytecode java. Stage de 4ème année de l'INSA de Rennes, projet Lande, IRISA, 2005.

[JM79] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of lisp-like structures. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages, POPL'79*, pages 244–256, 1979.

[Kle03] Gerwin Klein. *Verified Java bytecode verification*. PhD thesis, Technische Universität München, 2003.

[KN02] Gerwin Klein and Tobias Nipkow. Verified Bytecode Verifiers. *Theoretical Computer Science*, 298(3) :583–626, 2002.

[KN04] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, March 2004. To appear in ACM TOPLAS.

[KW03] Gerwin Klein and Martin Wildmoser. Verified bytecode subroutines. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'03)*, volume 2758 of *Lecture Notes in Computer Science*, pages 55–70. Springer Verlag, September 2003.

[Ler03] Xavier Leroy. Java bytecode verification : algorithms and formalizations. *Journal of Automated Reasonning*, 30(3-4) :235–269, 2003.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end, or : programming a compiler with a proof assistant. In *Proc. of the 33th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'06*, pages 42–54. ACM Press, 2006.

[Let04] Pierre Letouzey. *Programmation fonctionelle certifiée, L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université de Paris-Sud, 2004.

[LMC03] Sorin Lerner, Todd D. Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *Proc. of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI'03*, pages 220–231, 2003.

[LMRC05] Sorin Lerner, Todd D. Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proc. of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'05*, pages 364–377, 2005.

[LY99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1999.

[Min04] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, 2004. http://www.di.ens.fr/~mine/these/these-color.pdf.

[Mon98] David Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998.

[Nau05] David A. Naumann. Verifying a secure information flow analyzer. In *Proc. of 18th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2005*, volume 3603, pages 211–226, 2005.

[Nec97] George C. Necula. Proof-carrying code. In *Proc. of 24th ACM Symp. on Principles of Programming Languages (POPL'97)*, pages 106–119. ACM Press, 1997.

[Nor88] Bengt Nordström. Terminating general recursion. *BIT*, 28(3), 1988. http://www.cs.chalmers.se/~bengt/papers/genrec.pdf.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[OG98] Chris Okasaki and Andrew Gill. Fast mergeable integer maps. In *Proc. of the ACM SIGPLAN Workshop on ML*, pages 77–86, 1998.

[ORS92] Sam Owre, John M. Rushby, and Natarajan Shankar. PVS : A prototype verification system. In *Proc. of the 11th International Conference on Automated Deduction, CADE'92*, pages 748–752, 1992.

[Pau86] Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2(4) :325–355, December 1986.

[Pic05] David Pichardie. Modular proof principles for parameterized concretizations. In *Procòf 2nd International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS 2005)*, Lecture Notes in Computer Science. Springer-Verlag, 2005. to Appear.

[Pol04] Isabelle Pollet. *Towards a generic framework for the abstract interpretation of Java*. PhD thesis, Université catholique de Louvain, Belgium, 2004.

[Pus99] Cornelia Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In W. Rance Cleaveland, editor, *Proc. of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS'99*, volume 1579 of *Lecture Notes in Computer Science*, pages 89–103. Springer-Verlag, 1999.

[RMR01] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annoted constraints. In *Proc. of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 200*, pages 43–55, 2001.

[Ryd03] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented languages. In *Proc. of International Conference on Compiler Construction (CC'03)*, Lecture Notes in Computer Science. Springer-Verlag, 2003. Invited talk, slides available online at http://www.cs.rutgers.edu/~ryder/CC03InvitedNew.pdf.

[SA05] Alexandru Salcianu and Konstantine Arkoudas. Machine-Checkable Correctness Proofs for Intra-procedural Dataflow Analyses. In *Proc. of 4th International Workshop on Compiler Optimization Meets Compiler Verification (COCV 2005)*, volume 141 of *Electronic Notes in Theoretical Computer Science*, pages 53–68. Elsevier, 2005.

[Siv04] Igor Siveroni. Operational semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58(1-2), 2004.

[Tar55] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5 :285–309, 1955.

[The04] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. http://coq.inria.fr.

[War42] M. Ward. The closure operators of a lattice. *Annals Math.*, 43 :191–196, 1942.

[WCN05] Martin Wildmoser, Amine Chaieb, and Tobias Nipkow. Bytecode analysis for proof carrying code. In *Proceedings of the 1st Workshop on Bytecode Semantics, Verification and Transformation, Electronic Notes in Computer Science*, 2005.

[Web] Page web accompagnant ce manuscript. http://www.irisa.fr/lande/pichardie/these.

[Whi] Interpréteur abstrait certifié pour WHILE. http://www.irisa.fr/lande/pichardie/these/While.

[WL93] Pierre Weis and Xavier Leroy. *Le langage Caml*. Sciences Sup. Dunod, 1993.

Liste des définitions

Résumé

Nous nous intéressons dans cette thèse à la preuve formelle de correction des analyses statiques. Nous nous basons sur la théorie de l'interprétation abstraite qui présente une analyse statique comme une sémantique approchée d'un programme. Nous utilisons l'assistant de preuve Coq qui permet d'extraire le contenu calculatoire d'une preuve constructive. L'implémentation Caml certifiée d'une analyse peut ainsi être extraite de la preuve d'existence, pour tout programme, d'une approximation correcte de la sémantique concrète de ce programme.

Nous présentons un cadre théorique fondé sur l'interprétation abstraite et permettant le développement formel d'une large gamme d'analyses statiques. Une bibliothèque Coq de construction modulaire de treillis est ensuite proposée. Des preuves complexes de terminaison de calcul itératif de point fixe peuvent ainsi être construites par simple composition de foncteurs. Plusieurs cas d'études pour l'analyse de programme en bytecode Java sont présentés.

Mots cléfs : analyse statique, assistants de preuve, interprétation abstraite, Coq.

Abstract

This thesis deals with machine checked proofs of soundness of static analysis. We rely on the theory of abstract interpretation, where static analyses are presented as an approximated semantics of a program. We use the Coq proof assistant which allows for extracting the computational content of a constructive proof. A Caml implementation can hence be extracted from a proof of existence, for any program, of a correct approximation of the concrete program semantics.

We present a theoretical framework based on abstract interpretation allowing for the formal development of a broad range of static analyses. A Coq library for the modular construction of lattice is then proposed. Complex proofs for termination of iterative fixpoint computations can hence be constructed by simple composition of lattice functors. Several case studies for the analysis of Java bytecode are presented.

Key words : static analysis, proof assistants, abstract interpretation, Coq.