Programmation d'un interpréteur abstrait certifié en logique constructive

David Cachera* — **David Pichardie****

** ENS Cachan/IRISA – en délégation à l'INRIA, Centre Rennes - Bretagne Atlantique*

*** INRIA Rennes - Bretagne Atlantique*
Campus de Beaulieu
35042 Rennes cedex

RÉSUMÉ. *Un analyseur statique permet de déduire automatiquement des propriétés d'un programme à partir de son code. La preuve de correction d'un analyseur repose sur des propriétés sémantiques, et devient difficile à assurer lorsque l'analyse met en œuvre des techniques symboliques complexes. Nous proposons une adaptation de la théorie générale de l'analyse statique par interprétation abstraite au cadre de la logique constructive. L'implémentation de ce formalisme dans l'assistant de preuve Coq permet alors d'extraire automatiquement des analyseurs certifiés. Nous nous intéressons plus particulièrement à un langage impératif simple, et présentons en détail le calcul de point fixe par élargissement/rétrécissement et itération dirigée par la syntaxe.*

ABSTRACT. *A static analyzer aims at automatically deducing program properties by examining its source code. Proving the correctness of an analyzer is based on semantic properties, and becomes difficult to ensure when complex analysis techniques are involved. We propose to adapt the general theory of static analysis by abstract interpretation to the framework of constructive logic. Implementing this formalism into the Coq proof assistant then allows for automatic extraction of certified analyzers. We focus here on a simple imperative language and present the computation of fixpoints by widening/narrowing and syntax-directed iteration techniques.*

MOTS-CLÉS : *Analyse statique, interprétation abstraite, calcul de point fixe, logique constructive, assistant de preuve*

KEYWORDS: *Static Analysis, Abstract Interpretation, Fixpoint Computation, Constructive Logic, Proof Assistant*

2e soumission à *TSI*, le 29 avril 2010

1. Introduction

L'analyse statique par interprétation abstraite a démontré, au cours des dernières années, sa capacité à garantir l'absence de larges classes d'erreurs de programmation dans les logiciels critiques de grande taille. L'analyseur ASTRÉE (Cousot *et al.*, 2005) a ainsi été capable de prouver l'absence d'erreur à l'exécution sur un logiciel de commande de vol électrique de l'ordre du million de lignes de C. Cependant, de tels analyseurs statiques sont des logiciels complexes mettant en jeu des calculs symboliques très délicats. On peut donc s'interroger sur la validité de leurs résultats : une erreur dans l'analyse elle-même ou dans sa mise en œuvre informatique pourrait entraîner la non détection de programmes incorrects. Cette interrogation est d'autant plus légitime lorsqu'un analyseur prétend avoir démontré la sûreté d'un logiciel critique demandant un niveau de certification maximum. Cela conduit naturellement à proposer l'utilisation de méthodes formelles, notamment la preuve de programmes, pour certifier la correction d'analyseurs statiques et obtenir des garanties fortes sur la confiance qu'on peut leur accorder.

La preuve de correction d'un analyseur statique repose sur la sémantique du langage qu'il analyse. Au cours des dernières années, plusieurs travaux de recherches (Klein *et al.*, 2006; Bertot, 2008; Cachera *et al.*, 2005a; Barthe *et al.*, 2001a; Barthe *et al.*, 2001b; Klein *et al.*, 2002; Aydemir *et al.*, 2005; Leroy, 2006) se sont intéressés à la formalisation de preuves de propriétés sémantiques à l'aide d'assistants de preuve. Cet engouement s'explique en partie par la difficulté de rédiger (ou même de vérifier) manuellement une telle preuve, surtout lorsqu'il faut passer à l'échelle d'un langage réaliste comme C ou Java. Même si de tels travaux démontrent que les assistants de preuve modernes ont atteint la maturité nécessaire pour permettre la vérification formelle d'outils réalistes, les analyses statiques sous-jacentes à ces approches restent néanmoins en dessous du niveau de complexité d'un outil comme ASTRÉE qui réalise, par exemple, des abstractions numériques complexes.

La théorie de l'interprétation abstraite inventée par Cousot et Cousot (Cousot *et al.*, 1977; Cousot *et al.*, 1979) fournit non seulement des algorithmes pour la vérification de programme mais aussi un cadre théorique pour justifier la validité sémantique de ces vérifications. Au sein d'un formalisme unifié, elle permet non seulement de montrer la correction d'une analyse statique par rapport à une sémantique de référence, mais fournit aussi les outils méthodologiques pour construire des analyseurs efficaces. Ceux-ci vont tirer parti de l'expressivité de treillis complexes, nécessitant l'utilisation d'opérateurs d'élargissement et rétrécissement pour garantir le calcul d'une surapproximation de la sémantique en un temps fini. Les stratégies d'itération de ces calculs devront être judicieusement choisies afin de conserver le maximum de précision dans ces calculs. La théorie de l'interprétation abstraite fournit également un cadre pour élaborer des analyseurs et des preuves modulaires, cette modularité étant une condition indispensable pour le passage à l'échelle.

Si la formalisation de preuves de propriétés sémantiques à l'aide d'assistants de preuve a bénéficié d'une attention croissante ces dernières années, celle d'analyseurs

statiques dans le cadre général de l'interprétation abstraite n'a pas encore connu le même succès. Dans cet article, nous proposons une approche pour effectuer une telle formalisation avec l'assistant de preuve Coq, illustrée sur un langage impératif très simple. Nous insistons ici plus particulièrement sur les mécanismes d'élargissement/rétrécissement et les itérations dirigées par la syntaxe. L'aspect modulaire de la construction des analyses est assuré notamment par l'utilisation d'une bibliothèque de treillis certifiés ayant fait l'objet d'une publication antérieure (Pichardie, 2008).

Cet article est structuré comme suit : nous présentons en section 2 le cadre général de l'interprétation abstraite, avant de préciser ce que nous en retenons pour notre formalisation en Coq. Nous montrons en section 3 comment calculer des points fixes grâce à des opérateurs d'élargissement/rétrécissement définis de façon constructive. Dans la section 4, nous considérons un langage impératif simple pour illustrer la construction d'analyseurs certifiés. Nous présentons un ensemble de travaux connexes en section 5, avant de conclure par une discussion sur les choix effectués dans ce travail.

2. Interprétation abstraite et logique constructive

La vérification de programme se fonde sur une sémantique formelle du langage considéré. Comme aucune propriété non triviale de cette sémantique pour un programme quelconque n'est calculable, la théorie de l'interprétation abstraite fournit un cadre mathématique général pour concevoir une sémantique approchée. L'abstraction permet de concentrer l'effort de preuve sur une sélection de propriétés pertinentes pour le problème à résoudre. Si l'abstraction est suffisamment forte (en sélectionnant suffisamment peu de propriétés), la sémantique abstraite qui en résulte devient calculable, permettant 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, au prix bien sûr d'une perte de complétude.

L'interprétation abstraite fournit un outil de conception de sémantiques abstraites, mais aussi et surtout un moyen de prouver leur correction vis-à-vis de la sémantique concrète initiale. Les trois composants 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, l'implémentation de l'analyseur pourra se faire suivant un squelette algorithmique générique.

Finalement, l'interprétation abstraite nous fournit un moyen de définir la meilleure sémantique abstraite : étant données une fonction sur les domaines sémantiques concrets, et une abstraction de ces domaines concrets vers les domaines abstraits, elle

fournit la définition d'une fonction abstraite qui représente la meilleure approximation correcte de la fonction concrète.

Notre objectif dans ce travail est d'assurer la correction d'analyseurs statiques à l'aide d'un assistant de preuve. La théorie de l'interprétation abstraite fournit un cadre général avec des propriétés intéressantes, mais parfois difficiles à formaliser avec un tel outil. La présence de fonctions non calculables ou de propriétés nécessitant des preuves non constructives, comme nous le verrons dans cette section, rend cette théorie inadaptée à notre but. De plus, les résultats généraux sur l'optimalité des abstractions vont au delà de notre objectif. Il nous a donc fallu adapter la théorie à nos besoins, en affaiblissant certaines définitions ou hypothèses, afin de les adapter aux possibilités des assistants de preuve en général, et de Coq en particulier. Dans cette section, nous rappelons d'abord brièvement la théorie générale de l'interprétation abstraite, puis nous montrons comment nous avons sélectionné les caractéristiques qui nous intéressaient afin de prouver la correction d'analyseurs statiques dans une logique constructive.

2.1. *Le cadre de l'interprétation abstraite*

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). Un tel domaine sémantique $\mathcal{P}(\mathcal{D})$ est muni d'une structure de treillis complet (c'est-à-dire un ensemble muni d'un ordre partiel, tel que toute partie admet une bonne supérieure) $(\mathcal{P}(\mathcal{D}), \subseteq, \cup, \cap)$. 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 $P_1 \subseteq P_2$, alors P_1 est une propriété plus précise que P_2 car elle décrit moins de comportements possibles. Par exemple dans $\mathcal{P}(\mathbb{Z})$, la propriété « être nul » est plus précise que la propriété « être pair » car $\{0\} \subseteq \{2z \mid z \in \mathbb{Z}\}$.

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 des propriétés possibles. Une propriété abstraite sera une approximation correcte d'une propriété concrète si elle est moins précise, par rapport à l'ordre du treillis considéré (nous nous restreignons au cas des sur-approximations). Par exemple dans $\mathcal{P}(\mathbb{Z})$, la propriété « être inférieur à -3 » pourra être sur-approchée par la propriété « être négatif ». La question qui se pose naturellement est alors « qu'est-ce qu'une bonne abstraction ? ». En d'autres termes, quels sont les meilleurs choix pour définir l'ensemble des propriétés abstraites ? L'« hypothèse raisonnable » proposée par Patrick et Radhia Cousot dans leur article fondateur (Cousot *et al.*, 1979) est la suivante : pour toute propriété Q, l'ensemble de toutes les approximations correctes de Q doit posséder un plus petit élément. De plus, les propriétés abstraites sont définies en tant qu'éléments d'un treillis distinct, au lieu d'être un sous-ensemble de celui des propriétés concrètes. Ceci donne naissance à la notion de connexion de Galois.

Définition 2.1 Connexion de Galois.
Soit $(L_1, \sqsubseteq_1, \bigsqcup_1, \bigsqcap_1)$ *et* $(L_2, \sqsubseteq_2, \bigsqcup_2, \bigsqcap_2)$ *deux treillis complets. Un couple de fonctions* $\alpha \in L_1 \to L_2$ *et* $\gamma \in L_2 \to L_1$ *forme une* connexion de Galois *si*

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

La fonction α est un opérateur d'*abstraction* (nécessairement monotone) qui exprime la meilleure abstraction d'une propriété concrète et γ, une fonction (nécessairement monotone) de *concrétisation* qui exprime la propriété concrète représentée par un élément abstrait. Dans cette section, nous supposerons que les sémantiques concrètes et abstraites sont exprimées dans les deux treillis respectifs $(\mathcal{A}, \sqsubseteq, \bigsqcup, \bigsqcap)$ et $(\mathcal{A}^\sharp, \sqsubseteq^\sharp, \bigsqcup^\sharp, \bigsqcap^\sharp)$, reliés par une connexion de Galois (α, γ). Étant donnée une sémantique concrète $[\![P]\!] \in \mathcal{A}$ d'un programme P, le comportement approché de P sera calculé via une sémantique abstraite $[\![P]\!]^\sharp \in \mathcal{A}^\sharp$. L'abstraction sera correcte si et seulement si $[\![P]\!] \sqsubseteq \gamma([\![P]\!]^\sharp)$, ou de manière équivalente $\alpha([\![P]\!]) \sqsubseteq^\sharp [\![P]\!]^\sharp$. La sémantique $\alpha([\![P]\!])$ représente donc la meilleure sémantique abstraite que nous pouvons obtenir avec ce choix de (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. L'approximation correcte d'une fonction monotone $f \in \mathcal{A} \to \mathcal{A}$ est définie comme une fonction $f^\sharp$ qui vérifie l'une des assertions équivalentes suivantes : (i) $\alpha \circ f \circ \gamma \mathrel{\dot{\sqsubseteq}^\sharp} f^\sharp$; (ii) $f \circ \gamma \mathrel{\dot{\sqsubseteq}} \gamma \circ f^\sharp$, où $\dot{\sqsubseteq}$ est l'ordre point à point entre fonctions.

Le cas où $f^\sharp = \alpha \circ f \circ \gamma$ fournit une spécification optimale (c'est-à-dire la plus précise) pour $f^\sharp$, mais il s'agit plus d'une spécification que d'une implémentation, puisque la fonction α est rarement calculable. De plus, l'utilisation d'une fonction d'abstraction α cache des preuves, comme le montre l'exemple suivant. Considérons une abstraction numérique à base de congruence modulo 2. Le domaine concret est $(\mathcal{P}(\mathbb{Z}), \subseteq, \bigcup, \bigcap)$ et le domaine abstrait est le treillis basé sur l'ensemble $\{\bot, \overline{0}, \overline{1}, \top\}$ avec $\bot \sqsubseteq \overline{0} \sqsubseteq \top$ et $\bot \sqsubseteq \overline{1} \sqsubseteq \top$. Les fonctions d'abstraction et de concrétisation correspondantes sont définies par

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

où $\mathbb{Z}_0$ et $\mathbb{Z}_1$ représentent respectivement les ensembles d'entiers pairs et impairs. Si nous voulons maintenant calculer $\alpha \circ f \circ \gamma$ pour une fonction calculable donnée f, nous n'aurons aucun problème avec les définitions de γ et de f, mais l'utilisation de α pourra nécessiter une preuve qu'un sous-ensemble donné de $\mathbb{Z}$ n'est inclus ni dans $\mathbb{Z}_0$, ni dans $\mathbb{Z}_1$ pour démontrer que $\top$ est un résultat correct ! Outre le fait que la fonction d'abstraction α doit manipuler des ensembles arbitraires de valeurs concrètes,

la difficulté principale vient du fait qu'elle doit fournir la *meilleure* abstraction d'un tel ensemble, ce qui nécessite généralement un effort de preuve plus important.[1]

2.2. *Un cadre minimaliste*

À partir des observations précédentes, nous allons maintenant restreindre le cadre général de l'interprétation abstraite, afin d'obtenir une théorie qui soit suffisamment puissante pour prouver la correction d'une large classe d'abstractions, mais cependant adaptée à une implémentation en Coq.

2.2.1. *Treillis complets et connexions de Galois en Coq*

Les treillis complets seront formalisés en Coq au moyen d'un type enregistrement. Les champs principaux contiennent le domaine de base du treillis, et la spécification de l'ordre partiel et des opérateurs de plus grande borne inférieure et plus petite borne supérieure. Des champs additionnels contiennent des *preuves* que ces opérateurs respectent leur spécification. S'il est assez facile de prouver des résultats généraux sur les treillis complets en Coq, l'instantiation effective d'une telle structure est beaucoup plus compliquée. Par exemple, si le domaine de base est un type implémentable (celui des entiers de Péano `nat`, par exemple), l'opérateur de plus petite borne supérieure sera une fonction qui pour tout prédicat P (de type `nat → Prop`) calcule la plus petite borne supérieure de tous les éléments satisfaisant P. Mais P est d'un type logique qui n'est pas forcément implémentable. Un tel opérateur, de type `(nat → Prop) → nat` ne pourra être défini en Coq que si la valeur de son résultat ne dépend pas de celle se son argument logique de type `nat → Prop`. Il s'agit d'une restriction nécessaire pour assurer la cohérence du mécanisme d'extraction. Nous devrons donc définir une structure plus pauvre pour pouvoir traiter des domaines implémentables, en particulier les domaines abstraits.

De la même façon, une connexion de Galois est définie par un enregistrement contenant les définitions des fonctions α et γ, avec des preuves que ces fonctions respectent la propriété de la définition 2.1. Cette fois encore, s'il est assez facile d'établir des résultats généraux, il est plus difficile de construire effectivement une connexion de Galois, à cause de la non calculabilité ou simplement de la non constructivité de la fonction d'abstraction. Intuitivement, nous devons définir une fonction α qui associe une propriété abstraite à chaque sous-ensemble d'éléments du domaine concret : α est naturellement vue comme une relation, et nous devons la tranformer en une fonction. Nous pouvons pour cela adopter deux solutions. La première consiste à introduire un axiome dans la théorie, assurant que si une relation est fonctionnelle, alors il existe une fonction de même graphe. La deuxième solution garde α comme une relation, mais utilise le principe du tiers-exclus pour assurer que α est effectivement fonction-

1. D'ailleurs, dans le cadre général de l'interprétation abstraite, pour certaines abstractions on n'utilise pas de fonctions α justement parce que la meilleure abstraction n'existe pas. Par exemple, il n'existe pas de « meilleur » polyèdre permettant d'approcher un disque.

nelle. Nous avons choisi de rester dans un cadre de logique constructive sans axiomes en supprimant les fonctions d'abstraction de notre formalisme, et en reposant uniquement sur l'utilisation de fonctions de concrétisation.

2.2.2. *Le cadre retenu*

Au final, le cadre que nous avons retenu respecte les caractéristiques suivantes.

1) **Une simple structure de treillis pour les domaines abstraits.** Nous relâchons l'hypothèse de complétude afin de pouvoir définir des treillis implémentables en Coq. En contrepartie, l'existence de points fixes pour les fonctions monotones n'est plus garantie. Nous assortissons donc le treillis d'un critère de terminaison pour le calcul (plus précisément l'approximation) effectif de points fixes.

2) **Un domaine concret de la forme** $A = \mathcal{P}(\mathcal{D})$**.** De tels domaines permettent de définir des fonctions de concrétisation à l'aide de relations inductives, et profitent des capacités de Coq à manipuler de telles constructions (principes d'induction). En effet, $\mathcal{P}(\mathcal{D})$ correspond au type `D → Prop`, donc γ est une fonction de type `A# → (D → Prop)`, facile à définir par un type inductif. Ce choix ne semble pas limiter l'expressivité du cadre, en pratique.

3) **Une fonction de concrétisation monotone** $\gamma \in A^\sharp \to A$**, morphisme d'intersection binaire**[2]**.** Le critère de correction devient alors : $f^\sharp$ est une approximation correcte de f si et seulement si $\forall a^\sharp \in A^\sharp, f \circ \gamma(a^\sharp) \sqsubseteq \gamma \circ f^\sharp(a^\sharp)$. Nous imposons que γ soit un morphisme d'intersection pour être capable de combiner deux approximations d'une même valeur concrète. Ceci garantira en particulier que si un élément concret admet un ensemble fini d'approximations correctes, alors il existe une meilleure approximation de cet élément. Ceci est une version finitaire de l'« hypothèse raisonnable » mentionnée page 4.

4) **Utilisation de la notion de post-point fixe.** En Coq, le principe d'induction associé à une définition inductive fournit une caractérisation immédiate des sémantiques sous la forme de plus petits post-points fixes, plutôt que de points fixes. C'est pour cela que nous choisissons cette caractérisation dans notre cadre.

Notons que ce cadre n'est pas la version la plus minimaliste de l'interprétation abstraite. Il est possible de faire encore plus simple, par exemple sans définir d'ordre ni d'intersection sur le domaine abstrait. La correction des opérateurs et des fonctions de transfert abstraits est alors exprimée via la concrétisation (par exemple, $\gamma(x^\sharp) \leq \gamma(x^\sharp \sqcup^\sharp y^\sharp)$) (Cousot *et al.*, 2006).

---

2. *i.e.* $\gamma(x^\sharp \sqcap^\sharp y^\sharp) = \gamma(x^\sharp) \sqcap \gamma(y^\sharp)$. L'inclusion $\sqsubseteq$ de cette égalité est conséquence de la monotonie de γ. Remarquons que dans le cas classique d'une connexion de Galois, la fonction de concrétisation est nécessairement un morphisme d'intersection binaire.

3. Calcul générique de points fixes

3.1. *Calculs de points fixes*

La sémantique d'un programme s'exprime souvent comme le plus petit point fixe d'une fonction monotone. La sémantique abstraite va mimer cette situation, et approcher le point fixe concret par un (post-)point fixe abstrait. Le cadre classique des treillis complets fournit des résultats importants que nous rappelons brièvement ici, avant d'aborder la question des calculs de points fixes en logique constructive.

3.1.1. *Définitions et résultats classiques*

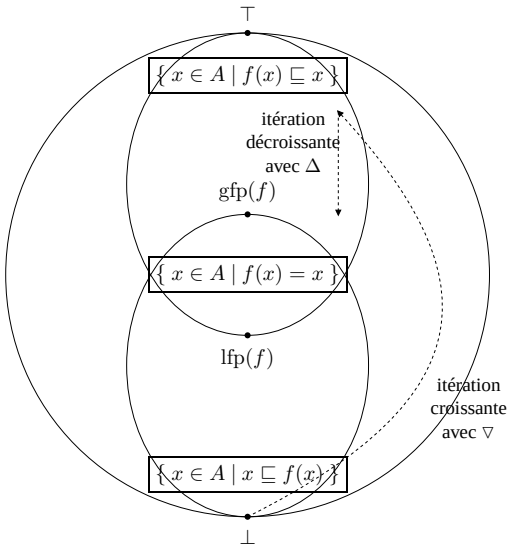

Figure 1. *Positions relatives des points fixes, post-points fixes et pré-points fixes dans un treillis complet – Itération croissante avec élargissement suivie d'une itération décroissante avec rétrécissement.*

Le théorème de Knaster-Tarski 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. Cependant, ce théorème ne dit rien quant à la méthode qui permettrait de calculer un tel point fixe. Lorsque la fonction f est continue, le théorème de Kleene nous donne une caractérisation plus « effective », dans le sens où elle fournit une méthode de calcul. Il s'agit alors d'itérer la fonction f à partir du plus petit élément du treillis, afin d'atteindre le plus petit point fixe. Cette méthode est *semi-complète*, puisque l'itération peut ne pas terminer. Le critère le plus simple permettant d'assurer la terminaison est d'interdire l'existence de chaînes infiniment croissantes (*condition de chaîne ascendante*). Étant donné un pré-point fixe a de f monotone, la suite $a, f(a), \ldots, f^n(a), \ldots$ va alors se stabiliser à partir d'un certain indice, et sa limite sera le plus petit point fixe de f plus grand que a. La structure

de treillis complet n'est même plus nécessaire ici, puique l'existence d'un plus petit point fixe est assurée de façon constructive. Malheureusement, la condition de chaîne ascendante est forte, et surtout impossible à prouver de façon constructive en général : il est en effet possible de montrer (Pichardie, 2005) que, même pour un treillis aussi simple que celui à deux valeurs $\{\bot, \top\}$, il n'existe pas de preuve constructive de cette condition.

En Coq, nous sommes en mesure de démontrer le théorème de Knaster-Tarski pour les treillis complets. Cela se traduit par la définition d'un opérateur `lfp` qui pour un treillis complet et une fonction monotone (et donc sa preuve de monotonie) renvoie le plus petit point fixe de cette fonction. Comme annoncé précédemment, seuls les treillis dont le domaine de base est un type logique (**`Prop`** ou `nat → ` **`Prop`** par exemple) peuvent être munis d'une structure de treillis complet en Coq. Nous construisons donc des treillis complets pour les domaines de la forme `A → ` **`Prop`** ou encore `B → (A → ` **`Prop`**`)`. Nous aurons besoin de ce genre de treillis complet et de leur opérateur de plus petit point fixe pour la sémantique collectrice de la section 4.

3.1.2. *Condition de chaîne ascendante constructive*

Afin d'implémenter le calcul de point fixe en Coq, nous utiliserons donc à la place de la définition standard les notions de relation bien fondée et d'accessibilité (Aczel, 1977).

Définition 3.1 Accessibilité et relation bien fondée.
Soit $\prec$ une relation sur un ensemble A. L'ensemble $\mathrm{Acc}_\prec$ d'éléments accessibles *pour $\prec$ est défini inductivement par*

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

La relation $\prec$ sur A est bien fondée *si tous les éléments de A sont accessibles pour $\prec$.*

La définition formelle de la condition de chaîne ascendante constructive est alors la suivante.

Définition 3.2 Condition de chaîne ascendante constructive.
Un ensemble partiellement ordonné $(A, \sqsubseteq)$ respecte la condition de chaîne ascendante (constructive) *si l'ordre inverse strict associé $\sqsupset$ est bien fondé.*

Nous n'insisterons pas dans cet article sur le calcul de point fixe par récursion bien fondée (Aczel, 1977), dont la formalisation en Coq a été présentée ailleurs (Cachera *et al.*, 2005a). Nous nous concentrerons plutôt sur le calcul d'une approximation des points fixes grâce à une version constructive de la technique d'élargissement et rétrécissement.

3.2. *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. En effet, 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 lorsque le treillis ne vérifie pas la condition de chaîne ascendante, ou bien la condition de chaîne ascendante est vérifiée mais la chaîne d'itération est trop longue pour permettre un calcul efficace, ou encore lorsque le treillis n'est pas complet et que la limite des itérations croissantes n'appartient pas au domaine d'abstraction.

3.2.1. *Technique d'élargissement standard*

La solution proposée par Patrick et Radhia 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 $x_0 = \bot, x_{n+1} = x_n \triangledown f(x_n)$ avec $\triangledown$ un opérateur binaire qui extrapole ses deux arguments. Intuitivement, à la n-ième itération, plutôt que de simplement itérer f, $f(x_n)$ est comparé avec la valeur précédente x_n afin de conjecturer une sur-approximation possible de la limite de la suite des itérés de f. Plus formellement, un *opérateur d'élargissement* $\triangledown$ sur un ensemble partiellement ordonné $(A, \sqsubseteq)$ vérifie $x \sqsubseteq x \triangledown y$, $y \sqsubseteq x \triangledown y$, et pour toute suite croissante $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_n \sqsubseteq \cdots$ la chaîne $y_0 = x_0, y_{n+1} = y_n \triangledown x_{n+1}$ atteint nécessairement un rang k pour lequel $y_k = y_{k+1}$. Si f est un opérateur monotone sur A et a un pré-point fixe de f alors la chaîne $(x_n)_n$ définie par $x_0 = a$ et $x_{k+1} = x_k \triangledown f(x_k)$ atteint en un nombre fini de pas un post-point fixe de f plus grand que a.

Lorsqu'un post-point fixe est atteint, une itération démarrant de ce point permet d'améliorer l'approximation de $\mathrm{lfp}(f)$. En effet, lorsque f est continue, la suite des itérées de f à partir d'un post-point fixe b admet pour limite $(\bigsqcup \{f^n(b)\})$, le plus grand point fixe de f plus petit que b. De plus, chaque étape intermédiaire de calcul est une approximation correcte : $\forall k,\ \mathrm{lfp}(f) \sqsubseteq \mathrm{gfp}_b(f) \sqsubseteq f^k(b) \sqsubseteq b$. Les problèmes de terminaison du calcul sont les mêmes que lors de l'itération croissante.

Une technique d'accélération peut une nouvelle fois être utilisée. L'itération utilisée est cette fois de la forme $x_0 = b, x_{n+1} = x_n \Delta f(x_n)$. 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. Plus formellement, un opérateur binaire $\Delta \in A \times A \to A$ est appelé *opérateur de rétrécissement* s'il vérifie $y \sqsubseteq x \ \Rightarrow\ y \sqsubseteq x \Delta y \sqsubseteq x$, et pour toute suite décroissante $x_0 \sqsupseteq x_1 \sqsupseteq \cdots \sqsupseteq x_n \sqsupseteq \cdots$, la chaîne $y_0 = x_0, y_{n+1} = y_n \Delta x_{n+1}$ atteint nécessairement un rang k pour lequel $y_k = y_{k+1}$. Si f est un opérateur monotone sur A et b un post-point fixe de f alors la chaîne $(x_n)_n$ définie par $x_0 = b$ et $x_{k+1} = x_k \Delta f(x_k)$ atteint en un nombre fini de pas un post-point fixe de f plus petit que b. L'enchaînement des deux itérations est schématisé dans la figure 1.

3.2.2. *Élargissement et rétrécissement constructifs*

L'utilisation de l'opérateur d'élargissement définissant une chaîne ascendante, le critère de terminaison de cet opérateur (ainsi que de celui de rétrécissement) n'est pas adapté aux preuves constructives, comme nous l'avons déjà remarqué en section 3.1.1. Nous proposons maintenant de reformuler ces critères à l'aide de la notion d'accessibilité.

Définition 3.3 Opérateur d'élargissement (version constructive).
Étant donné un ensemble partiellement ordonné $(A, \sqsubseteq)$*, un opérateur binaire* $\triangledown \in A \times A \to A$ *est appelé* opérateur d'élargissement *s'il vérifie les conditions suivantes :*

$$\forall x, y \in A,\ x \sqsubseteq x \triangledown y \qquad [1]$$

$$\forall x, y \in A,\ y \sqsubseteq x \triangledown y \qquad [2]$$

la relation $\prec_\triangledown \subseteq (A \times A) \times (A \times A)$ *définie par*
$\forall x_1, x_2, y_1, y_2 \in A,$
$(x_2, y_2) \prec_\triangledown (x_1, y_1) \iff x_1 \sqsubseteq x_2 \ \wedge\ y_2 = y_1 \triangledown x_2 \ \wedge\ y_2 \neq y_1$ [3]
vérifie $\forall x \in A,\ (x, x)$ *est accessible pour* $\prec_\triangledown$.

Il est à noter que cette définition reste équivalente à la définition classique (Pichardie, 2005). Elle en partage donc également les limites : d'une part, elle est dédiée au calcul de post-points fixes de fonctions monotones, or certaines fonctions abstraites ne sont pas monotones (c'est par exemple le cas de certaines fonctions abstraites utilisées dans l'analyseur ASTRÉE (Cousot *et al.*, 2005)) ; de plus, dans notre contexte, même si un opérateur abstrait est monotone, il peut être intéressant de ne pas avoir à le prouver.

Nous proposons donc une nouvelle définition, permettant de supprimer l'hypothèse de monotonie de la fonction $f^\sharp$. Il suffit pour cela de supprimer l'hypothèse de croissance de la chaîne $x_0, x_1, \ldots, x_n, \ldots$ 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 (Feret, 2005; Miné, 2004). Sur la version constructive, la suppression de cette hypothèse de monotonie revient à supprimer la condition $x_2 \sqsubseteq x_1$ dans la relation d'accessibilité.

3.2.3. *Élargissement et rétrécissement en Coq*

Les définitions précédentes donnent lieu à la définition du type Coq des opérateurs d'élargissement et de rétrécissement à l'aide d'enregistrements.

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

Record widening_operator: Set := WidOp {
```

```
  widen : t → t → t;
  widen_bound1 : ∀ x y : t, order x (widen x y);
  widen_bound2 : ∀ x y : t, order y (widen x y);
  widen_eq1 : ∀ x y z: t, eq x y → eq (widen x z) (widen y z);
  widen_eq2 : ∀ x y z: t, eq x y → eq (widen z x) (widen z y);
  widen_bottom1 : ∀ x : t, eq x (widen x bottom);
  widen_bottom2 : ∀ x : t, eq x (widen bottom x);
  widen_acc_property : ∀ x:t, Acc (widen_rel widen) (x,x)
}.
```

Les champs `widen_eq1` et `widen_eq2` assurent que `widen` est compatible avec l'équivalence `eq` du treillis[3]. Les champs `widen_bottom1` et `widen_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 : l'élément $\bot$ est neutre pour l'opérateur d'élargissement, ce qui signifie qu'aucune accélération n'est effectuée au niveau de $\bot$.

La définition du type des opérateurs de rétrécissement en Coq est similaire.

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

Record narrowing_operator : Set := NarOp {
  narrow : t → t → t;
  narrow_bound1 : ∀ x y, order (narrow x y) x;
  narrow_bound2 : ∀ x y : t, order (meet x y) (narrow x y);
  narrow_eq1 : ∀ x y z: t, eq x y → eq (narrow x z) (narrow y z);
  narrow_eq2 : ∀ x y z: t, eq x y → eq (narrow z x) (narrow z y);
  narrow_acc_property : ∀ x:t, Acc (narrow_rel narrow) (x,x)
}.
```

Alors que les résultats classiques concernant l'opérateur d'élargissement restent valables dans ce cadre constructif avec opérateurs abstraits non monotones, il est nécessaire d'être un peu plus circonspect en ce qui concerne le rétrécissement. Pour pouvoir affirmer que l'itération descendante donne toujours une sur-approximation correcte, il nous faut en effet supposer que l'opérateur abstrait $f^\sharp$ sur lequel nous voulons itérer est lui-même une approximation d'un opérateur concret f (qui lui sera monotone) via une fonction de concrétisation γ monotone et morphisme d'intersection binaire. Plus précisément, nous pouvons énoncer le théorème suivant.

Théorème 1 *Soit f une fonction monotone sur un treillis complet $(A, \sqsubseteq, \sqcup, \sqcap)$, $(A^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp)$ un treillis, γ une fonction de concrétisation monotone, morphisme d'intersection de $A^\sharp$ dans A et $f^\sharp : A^\sharp \to A^\sharp$ une approximation correcte de f. Alors*

3. Les treillis que nous considérons en Coq définissent leur propre relation d'égalité car l'égalité standard de Coq est parfois inadaptée.

l'itération des opérateurs d'élargissement puis de rétrécissement sur $f^\sharp$ fournit en un nombre fini de pas une approximation correcte $a^\sharp$ de $\mathrm{lfp}(f)$.

3.2.4. *Une bibliothèque de treillis pour la construction modulaire d'opérateurs d'élargissement/rétrécissement en Coq*

Les définitions précédentes sont rassemblées dans une signature de module Coq, nommée `LatticeWiden`. Le théorème 1 est utilisé dans un foncteur de module qui fournit un solveur de post-point fixe certifié pour tout module de type `LatticeWiden`.

Mais la dernière difficulté concerne la construction effective de tels modules. Pour des domaines abstraits complexes, la définition des opérateurs d'élargissement/rétrécissement peut être particulièrement difficile en raison de la preuve de terminaison qu'elle occasionne. Pour résoudre cette difficulté, nous avons développé une bibliothèque de modules et de foncteurs Coq pour facilement construire des treillis avec opérateurs d'élargissement/rétrécissement sur des domaines complexes par simple composition de foncteurs et de modules. Cette bibliothèque a été décrite dans une publication précédente (Pichardie, 2008). Le principal foncteur que nous utiliserons dans ce travail est le foncteur `ArrayBinLatticeWiden` qui permet de construire un treillis sur les domaines de type `Word` $\rightarrow$ `L.t` où `L.t` désigne le domaine d'un module `L` de type `LatticeWiden`, paramètre du foncteur. Le type `Word` représente l'ensemble des entiers binaires représentables avec 32 bits. Ce type, défini par nos soins, contient par définition un nombre fini d'éléments. Il s'agit d'une hypothèse importante pour assurer la terminaison des opérateurs d'élargissement/rétrécissement de ce treillis. Pour assurer l'efficacité des fonctions manipulées, nous nous appuyons sur une implémentation fonctionnelle des tableaux (Okasaki *et al.*, 1998).

4. Étude de cas : un interpréteur abstrait pour WHILE

Nous présentons maintenant un interpréteur abstrait pour un langage WHILE jouet. Cet analyseur sera entièrement programmé en Coq, avec un type assurant sa terminaison et sa correction vis-à-vis de la sémantique du langage.

L'analyseur présenté ici s'inspire grandement des algorithmes présentés dans (Cousot, 1999). Nous avons néanmoins entièrement repris les preuves de correction. En effet, Patrick Cousot s'attache 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 la section 2, nous n'utilisons pas de connexions de Galois dans nos développements Coq. Par conséquent, ce genre de dérivation systématique n'est pas à notre portée. Nous avons donc repris les preuves pour les adapter au cadre de l'interprétation abstraite que nous avons retenu. La deuxième différence concerne la sémantique sur laquelle nous nous basons pour prouver la correction de l'interpréteur abstrait, qui est plus standard que les sémantiques habituellement utilisées pour ce type d'analyseur (Cousot, 1999; Miné, 2004; Pichardie, 2005).

4.1. *Présentation du langage*

4.1.1. *Syntaxe du langage*

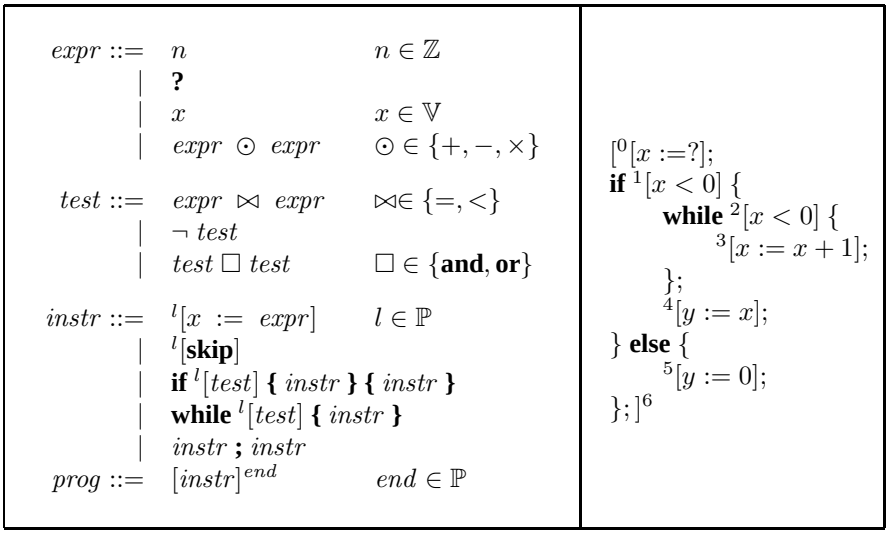

Figure 2. *Syntaxe du langage* WHILE*: définition et exemple.*

Le langage WHILE que nous considérons dans ce chapitre est un langage impératif rudimentaire. Sa syntaxe est résumée dans la figure 2. Un programme WHILE est constitué d'une instruction principale et d'un label de fin. Une instruction est soit l'affectation d'une variable par une expression numérique, soit une conditionnelle **if** . . . **else** . . ., soit une boucle **while**, soit une séquence de deux instructions. Une expression numérique comporte des constantes, des opérations arithmétiques (multiplication, 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, la négation d'un test ou enfin la conjonction ou la disjonction de deux tests. 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 attacher les invariants mémoires calculés par notre interpréteur abstrait à chaque point de programme.

Dans notre formalisation Coq, le type des points de programme $l \in \mathbb{P}$ et des noms de variables $x \in \mathbb{V}$ est le type `Word`. Nous aurions a priori pu prendre n'importe quel type puisqu'aucune opération ne sera effectuée sur ces éléments. Néanmoins, nous allons devoir construire des opérateurs d'élargissement/rétrécissement pour les fonctions de domaine $\mathbb{P}$ et $\mathbb{V}$. Grâce au type `Word` nous pourrons utiliser le foncteur `ArrayBinLatticeWiden` présenté dans la section 3.2.4.

4.1.2. *Sémantique concrète du langage*

Le but de notre analyseur sera de calculer des invariants vérifiés par les variables d'un programme dans les différents états accessibles de son exécution. Nous nous appuyons donc sur une sémantique opérationnelle à petit pas. 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 en figure 3.

Pour pouvoir définir mathématiquement l'ensemble des états accessibles d'un programme, nous aurons besoin de définir au préalable les sémantiques des expressions numériques et des tests (*cf.* figure 3). La sémantique des expressions associe à chaque expression e l'ensemble $[\![e]\!]\uparrow^{\rho}_{\mathrm{expr}}$ 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. L'opérateur $[\![\odot]\!]\uparrow_{\mathrm{op}}$ désigne ici la sémantique usuelle des opérateurs arithmétiques pour $\odot \in \{+, -, \times\}$. La sémantique des tests associe à chaque test t l'ensemble $[\![e]\!]\uparrow^{\rho}_{\mathrm{test}}$ 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 $[\![\cdot]\!]\uparrow^{\rho}_{\mathrm{test}}$. L'opérateur $[\![\bowtie]\!]$ désigne ici la sémantique usuelle des comparaisons arithmétiques pour $\bowtie \in \{=, <\}$. L'opérateur $\neg_{\mathbb{B}}$ (resp. $\vee_{\mathbb{B}}$ et $\wedge_{\mathbb{B}}$) désigne la sémantique booléenne de l'opérateur de négation (resp. de la disjonction et la conjonction).

Enfin, nous utilisons une sémantique opérationnelle structurelle standard (Winskel, 1993; Nielson *et al.*, 1992) et définissons à l'aide de celle-ci l'ensemble des états accessibles au cours de l'exécution d'un programme (voir figure 3). Notre analyseur statique aura pour but de calculer une sur-approximation de ces états accessibles. En général, une telle information peut-être ensuite utilisée dans une deuxième phase pour assurer l'absence de certaines erreurs. Dans le cas présent nous nous concentrons sur la première phase et n'introduisons pas de façon explicite les cas d'erreurs dans notre sémantique.

4.2. *Sémantique collectrice définie par induction sur la syntaxe des programmes*

Avant de présenter notre interpréteur abstrait, nous introduisons une sémantique collectrice intermédiaire qui calcule la même notion d'états accessibles que $[\![\cdot]\!]_{\mathrm{SOS}}$, mais en prenant une forme similaire à l'interpréteur abstrait qui suivra. Cette ressemblance de forme facilitera la preuve de correction de l'interpréteur par rapport à la sémantique collectrice. Cette dernière sera pour sa part reliée formellement à la sémantique standard précédente. Une approche standard consiste à définir la sémantique collectrice et l'interpréteur abstrait comme les solutions de systèmes d'équations attachées à chaque programme. Cette approche a l'avantage d'être relativement indépendante du langage de programmation étudié, en s'appuyant essentiellement sur une notion de graphe de flot de contrôle. Néanmoins la résolution du système d'équa-

Domaines sémantiques

$$\text{Env} \stackrel{\text{def}}{=} \mathbb{V} \to \mathbb{Z} \qquad\qquad \text{State} \stackrel{\text{def}}{=} \mathbb{P} \times \text{Env}$$

Sémantique des expressions

$$\boxed{[\![e]\!]\uparrow^{\rho}_{\text{expr}} \in \mathcal{P}(\mathbb{Z}), \quad e \in \mathit{expr},\ \rho \in \text{Env}}$$

$$\begin{array}{lcl}
[\![n]\!]\uparrow^{\rho}_{\text{expr}} & = & \{\ n\ \} \\
[\![?]\!]\uparrow^{\rho}_{\text{expr}} & = & \mathbb{Z} \\
[\![x]\!]\uparrow^{\rho}_{\text{expr}} & = & \{\ \rho(x)\ \},\ x \in \mathbb{V} \\
[\![e_1 \odot e_2]\!]\uparrow^{\rho}_{\text{expr}} & = & \{\ v_1 [\![\odot]\!]\uparrow_{\text{op}} v_2 \mid v_1 \in [\![e_1]\!]\uparrow^{\rho}_{\text{expr}},\ v_2 \in [\![e_2]\!]\uparrow^{\rho}_{\text{expr}}\ \}
\end{array}$$

Sémantique des tests

$$\boxed{[\![t]\!]\uparrow^{\rho}_{\text{test}} \in \mathcal{P}(\mathbb{B}), \quad t \in \mathit{test},\ \rho \in \text{Env} \quad \mathbb{B} = \{\mathrm{tt}, \mathrm{ff}\}}$$

$$\frac{v_1 \in [\![e_1]\!]\uparrow^{\rho}_{\text{expr}} \quad v_2 \in [\![e_2]\!]\uparrow^{\rho}_{\text{expr}} \quad v_1\ [\![\bowtie]\!]\ v_2}{\mathrm{tt} \in [\![e_1\ \bowtie\ e_2]\!]\uparrow^{\rho}_{\text{test}}}$$

$$\frac{v_1 \in [\![e_1]\!]\uparrow^{\rho}_{\text{expr}} \quad v_2 \in [\![e_2]\!]\uparrow_{\text{expr}}\ \rho \quad \neg\,(v_1\ [\![\bowtie]\!]\ v_2)}{\mathrm{ff} \in [\![e_1\ \bowtie\ e_2]\!]\uparrow^{\rho}_{\text{test}}}$$

$$\frac{b \in [\![t]\!]\uparrow^{\rho}_{\text{test}}}{\neg_{\mathbb{B}} b \in [\![\neg t]\!]\uparrow^{\rho}_{\text{test}}} \qquad \frac{b_1 \in [\![t_1]\!]\uparrow^{\rho}_{\text{test}} \quad b_2 \in [\![t_2]\!]\uparrow^{\rho}_{\text{test}}}{b_1 \wedge_{\mathbb{B}} b_2 \in [\![t_1\ \textbf{and}\ t_2]\!]\uparrow^{\rho}_{\text{test}}} \qquad \frac{b_1 \in [\![t_1]\!]\uparrow^{\rho}_{\text{test}} \quad b_2 \in [\![t_2]\!]\uparrow^{\rho}_{\text{test}}}{b_1 \vee_{\mathbb{B}} b_2 \in [\![t_1\ \textbf{or}\ t_2]\!]\uparrow^{\rho}_{\text{test}}}$$

Sémantique opérationnelle à petit pas

$$\boxed{\Rightarrow \subseteq\ (\mathit{instr} \times \text{Env}) \times ((\mathit{instr} \times \text{Env}) + \text{Env})}$$

$$\frac{v \in [\![a]\!]\uparrow^{\rho}_{\text{expr}}}{({}^{l}[x := a], \rho) \Rightarrow \rho[x \mapsto v]} \qquad \frac{}{({}^{l}[\textbf{skip}], \rho) \Rightarrow \rho}$$

$$\frac{(S_1, \rho) \Rightarrow \rho'}{(S_1;\ S_2, \rho) \Rightarrow (S_2, \rho')} \qquad \frac{(S_1, \rho) \Rightarrow (S_1', \rho')}{(S_1;\ S_2, \rho) \Rightarrow (S_1';\ S_2, \rho')}$$

$$\frac{\mathrm{tt} \in [\![b]\!]\uparrow^{\rho}_{\text{test}}}{(\textbf{if}\ {}^{l}[b]\ \{\ S_1\ \}\ \{\ S_2\ \}, \rho) \Rightarrow (S_1, \rho)} \qquad \frac{\mathrm{ff} \in [\![b]\!]\uparrow^{\rho}_{\text{test}}}{(\textbf{if}\ {}^{l}[b]\ \{\ S_1\ \}\ \{\ S_2\ \}, \rho) \Rightarrow (S_2, \rho)}$$

$$\frac{\mathrm{tt} \in [\![b]\!]\uparrow^{\rho}_{\text{test}}}{(\textbf{while}\ {}^{l}[b]\{\ S\ \}, \rho) \Rightarrow (S;\ \textbf{while}\ {}^{l}[b]\ \{\ S\ \}, \rho)} \qquad \frac{\mathrm{ff} \in [\![b]\!]\uparrow^{\rho}_{\text{test}}}{(\textbf{while}\ {}^{l}[b]\{\ S\ \}, \rho) \Rightarrow \rho}$$

Etats accessibles

$$\boxed{[\![\cdot]\!]_{\text{SOS}} \in \mathit{prog} \to \mathcal{P}(\text{State})}$$

$$[\![[P]^{\text{end}}]\!]_{\text{SOS}} = \left\{ (k, \rho) \;\middle|\; \begin{array}{l} \exists \rho_0 \in \text{Env}, \\ \quad \exists S \in \mathit{instr},\ (P, \rho_0) \Rightarrow^{*} (S, \rho) \text{ et } k = \text{entry}(S) \\ \quad \text{ou}\ \ (P, \rho_0) \Rightarrow^{*} \rho \text{ et } k = \text{end} \end{array} \right\}$$

Figure 3. *Sémantique standard*

tions abstrait à l'aide d'opérateurs d'élargissement/rétrécissement demande une analyse des dépendances du système pour limiter l'utilisation de ces opérateurs aux têtes de boucle du graphe de dépendance. Sans cela, la précision de l'analyseur peut être grandement diminuée. Une formalisation Coq de cette technique demande une preuve délicate pour démontrer que la sélection de points d'élargissement/rétrécissement est suffisamment large pour impliquer la convergence du processus itératif d'approximation de point fixe. Dans ce travail, nous nous inspirons de (Cousot, 1999) et proposons une technique d'itération spécialisée pour les programmes impératifs bien structurés comme ceux de WHILE. Notre interpréteur suit donc une stratégie d'itération dite *dénotationnelle* où chaque boucle interne du programme donne lieu à une recherche de post-point fixe local. Notre sémantique collectrice hérite de cette présentation et est ainsi définie par induction sur la syntaxe des programmes (voir figure 4). Par abus de notation, nous notons ${}^{l}S$ une instruction S dont le premier point de programme est l.

L'objectif de cette sémantique est de calculer pour chaque point de programme l'ensemble des environnements accessibles en ce point. Etant donné un point de programme l et une instruction i, nous notons $\mathcal{C}_l(i)$ l'opérateur qui calcule l'ensemble des états accessibles au cours de l'exécution de i (représenté par un élément $C \in \text{State}^{\text{col}}$), à partir d'un ensemble d'environnements de variables initial à l'entrée de i. Le label l représente le point initial de l'instruction suivante ou le point final du programme s'il n'existe pas d'instruction suivante. Le domaine sous-jacent est donc

$$\text{State}^{\text{col}} = \mathbb{P} \to \mathcal{P}(\text{Env})$$

qui est naturellement muni d'une structure de treillis (complet) $\left(\text{State}^{\text{col}}, \dot{\subseteq}, \dot{\cup}, \dot{\cap}\right)$, pour l'inclusion ensembliste point à point. Pour $C \in \text{State}^{\text{col}}$, $l \in \mathbb{P}$ et $E \in \mathcal{P}(\text{Env})$, nous notons $C \uplus \{l \mapsto E\}$ l'élément de $\text{State}^{\text{col}}$ qui associe $C(l) \cup E$ à l et $C(l')$ à tous les autres points de programme l'. $\{l \mapsto E\}$ est un abus de notation pour $(\lambda l.\emptyset) \uplus \{l \mapsto E\}$. Deux transformateurs de prédicats apparaissent dans la définition de $\mathcal{C}$:

– l'affectation $[\![x := e]\!]_{\text{affect}} \stackrel{\text{def}}{=} \lambda X.\ \{\rho[x \mapsto n] \mid \rho \in X,\ n \in [\![e]\!]\uparrow^{\rho}_{\text{expr}}\}$,

– le test $[\![t]\!]_{\text{test}} \stackrel{\text{def}}{=} \lambda X.\ \{\rho \mid \rho \in X,\ \mathrm{tt} \in [\![t]\!]\uparrow^{\rho}_{\text{test}}\}$,

L'instruction **while** utilise un calcul de plus petit point fixe pour calculer l'invariant I au point de programme l.

Pour définir une telle sémantique en Coq, nous réalisons un point fixe par induction sur la syntaxe, en utilisant pour l'instruction **while**, l'opérateur `lfp` de la section 3.1.1 sur les opérateurs monotones de treillis complets. Ce dernier opérateur nous oblige néanmoins à donner une preuve de monotonie de l'opérateur $\lambda X.E \cup \mathcal{C}_l(S_1)([\![t]\!]_{\text{test}}(X))(l)$ dont on désire le point fixe, mais cette preuve dépend elle-même de l'opérateur $\mathcal{C}$ en cours de définition. Pour résoudre cette difficulté technique, nous définissons le type $\mathcal{P}(\text{Env}) \xrightarrow{m} \text{State}^{\text{col}}$ des opérateurs monotones de $\mathcal{P}(\text{Env})$ vers $\text{State}^{\text{col}}$ et prouvons, en même temps que nous définissons $\mathcal{C}$, que son résultat est toujours un opérateur monotone.

Sémantique collectrice

$$\boxed{\mathcal{C}_l \in \mathit{instr} \to \left(\mathcal{P}(\text{Env}) \xrightarrow{m} \text{State}^{\text{col}}\right), \quad l \in \mathbb{P}}$$

$$\begin{array}{rcl}
\mathcal{C}_{l'}({}^{l}[\textbf{skip}]) & = & \lambda E.\{l \mapsto E\} \uplus \{l' \mapsto E\} \\
\mathcal{C}_{l'}\left({}^{l}[x := e]\right) & = & \lambda E.\{l \mapsto E\} \uplus \{l' \mapsto [\![x := e]\!]_{\text{affect}}(E)\} \\
\mathcal{C}_{l'}\left(\textbf{if}\ {}^{l}[t]\ \{\ {}^{l_1}S_1\ \}\ \{\ {}^{l_2}S_2\ \}\right) & = & \lambda E.\mathcal{C}_{l'}(S_1)([\![t]\!]_{\text{test}}(E)) \dot{\cup} \mathcal{C}_{l'}(S_2)([\![\neg t]\!]_{\text{test}}(E)) \uplus \{l \mapsto E\} \\
\mathcal{C}_{l'}\left(\textbf{while}\ {}^{l}[t]\ \{\ {}^{l_1}S_1\ \}\right) & = & \lambda E.\mathcal{C}_{l}(S_1)([\![t]\!]_{\text{test}}(I)) \uplus \{l \mapsto I\} \uplus \{l' \mapsto [\![\neg t]\!]_{\text{test}}(I)\} \\
 & & \text{avec } I = \text{lfp}(\lambda X.E \cup \mathcal{C}_{l}(S_1)([\![t]\!]_{\text{test}}(X))(l)) \\
\mathcal{C}_{l'}\left({}^{l_1}S_1;\ {}^{l_2}S_2\right) & = & \lambda E.\mathcal{C}_{l_2}(S_1)(E) \dot{\cup} \mathcal{C}_{l'}(S_2)(\mathcal{C}_{l_2}(S_1)(E)(l_2)) \\
 & & \boxed{\mathcal{C} \in \mathit{prog} \to \text{State}^{\text{col}}} \\
\mathcal{C}({}^{l}[P]^{end}) & = & \mathcal{C}_{end}(P)(\text{Env})
\end{array}$$

Sémantique abstraite

$$\boxed{[\![\cdot]\!]^\sharp_l \in \mathit{instr} \to \left(\text{Env}^\sharp \to (\mathbb{P} \to \text{Env}^\sharp)\right), \quad l \in \mathbb{P}}$$

$$\begin{array}{rcl}
[\![{}^{l}[\textbf{skip}]]\!]^\sharp_{l'} & = & \lambda E.\{l \mapsto E\} \uplus^\sharp \{l' \mapsto E\} \\
[\![{}^{l}[x := e]]\!]^\sharp_{l'} & = & \lambda E.\{l \mapsto E\} \uplus^\sharp \{l' \mapsto [\![x := e]\!]^\sharp_{\text{affect}}(E)\} \\
[\![\textbf{if}\ {}^{l}[t]\ \{\ {}^{l_1}S_1\ \}\ \{\ {}^{l_2}S_2\ \}]\!]^\sharp_{l'} & = & \lambda E.[\![S_1]\!]^\sharp_{l'}([\![t]\!]^\sharp_{\text{test}}(E)) \dot{\sqcup}^\sharp [\![S_2]\!]^\sharp_{l'}([\![\neg t]\!]^\sharp_{\text{test}}(E)) \uplus^\sharp \{l \mapsto E\} \\
[\![\textbf{while}\ {}^{l}[t]\ \{\ {}^{l_1}S_1\ \}]\!]^\sharp_{l'} & = & \lambda E.[\![S_1]\!]^\sharp_{l}([\![t]\!]^\sharp_{\text{test}}(I)) \uplus^\sharp \{l \mapsto I\} \uplus^\sharp \{l' \mapsto [\![\neg t]\!]^\sharp_{\text{test}}(I)\} \\
 & & \text{avec } I = \mathit{approx}^\sharp_{\text{lfp}}(\lambda X.E \sqcup^\sharp [\![S_1]\!]^\sharp_{l}([\![t]\!]^\sharp_{\text{test}}(X))(l)) \\
[\![{}^{l_1}S_1;\ {}^{l_2}S_2]\!]^\sharp_{l'} & = & \lambda E.[\![S_1]\!]^\sharp_{l_2}(E) \dot{\sqcup}^\sharp [\![S_2]\!]^\sharp_{l'}([\![S_1]\!]^\sharp_{l_2}(E)(l_2)) \\
 & & \boxed{[\![\cdot]\!]^\sharp \in \mathit{prog} \to (\mathbb{P} \to \text{Env}^\sharp)} \\
[\![{}^{l}[P]^{end}]\!]^\sharp & = & [\![P]\!]^\sharp_{end}(\top^\sharp)
\end{array}$$

Figure 4. *Sémantique collectrice et sémantique abstraite définies par induction sur la syntaxe.*

Nous démontrons ensuite que les états ainsi collectés sont une sur-approximation des états accessibles pour la sémantique standard.

Théorème 2 *Pour tout programme P, $[\![P]\!]_{SOS} \subseteq \{\ (k, \rho) \mid \rho \in \mathcal{C}_k(P)\ \}$.*

L'inclusion inverse est aussi prouvable si l'on suppose que tous les points de programme de P sont distincts, mais cette inclusion n'est pas strictement nécessaire pour prouver que notre interpréteur abstrait réalise une sur-approximation de la sémantique standard. Seule l'inclusion du théorème 2 a été prouvée en Coq.

4.3. *Sémantique abstraite*

Notre sémantique abstraite « mime » la sémantique collectrice en remplaçant l'espace de calcul concret State$^{\text{col}}$ par un treillis abstrait et les opérations concrètes par des opérations abstraites. Le domaine abstrait considéré est de la forme $\mathbb{P} \to \text{Env}^\sharp$ où $Env^\sharp$ est muni d'une structure de treillis $(\text{Env}^\sharp, \sqsubseteq^\sharp, \cup^\sharp, \cap^\sharp)$. Le domaine $\text{Env}^\sharp$ est relié à $\mathcal{P}(\text{Env})$ par une fonction de concrétisation monotone et morphisme d'intersection $\gamma_{\text{Env}} \in \text{Env}^\sharp \to \mathcal{P}(\text{Env})$. Nous définissons alors (voir figure 4) un interpréteur abstrait $[\![\cdot]\!]^\sharp$ qui calcule des propriétés sur les environnements accessibles à chaque point de programme. L'élément $\bot^\sharp$ représente le plus petit élément du treillis, $\top^\sharp$ le plus grand. $[\![x := e]\!]^\sharp_{\text{affect}}$ est une approximation correcte de $[\![x := e]\!]_{\text{affect}}$ et $[\![t]\!]^\sharp$ une approximation correcte de $[\![t]\!]_{\text{test}}$. Toutes ces hypothèses sur le domaine d'abstraction des environnements $\text{Env}^\sharp$ sont rassemblées dans une signature de module Coq nommée `EnvAbstraction` (voir figure 6).

Enfin, $approx^\sharp_{\text{lfp}}$ est un opérateur capable de calculer un post-point fixe d'une fonction de $\text{Env}^\sharp \to \text{Env}^\sharp$. Un tel solveur de post-point fixe est obtenu en combinant notre itérateur générique, présenté en section 3.2.4, et le module de type `LatticeWiden` construit pour représenter $\mathbb{P} \to \text{Env}^\sharp$, grâce au foncteur `ArrayBinLatticeWiden`.

L'interpréteur abstrait de la figure 4 est ensuite défini dans un foncteur de module qui prend en argument un module de type `EnvAbstraction` et où nous démontrons la correction de l'interpréteur vis-à-vis de la sémantique collectrice. Les éléments principaux de ce foncteur sont présentés en figure 5.

Théorème 3 *Pour tout programme P, pour tout $k \in \mathbb{P}$, $\mathcal{C}_k(P) \subseteq \gamma_{Env}\left([\![P]\!]^\sharp(k)\right)$.*

Corollaire 1 *Pour tout programme P, $[\![P]\!]_{SOS} \subseteq \{\ (k, \rho) \mid \rho \in \gamma_{Env}\left([\![P]\!]^\sharp(k)\right)\ \}$.*

```
Module AbSem (AbEnv:EnvAbstraction).
  ...

  Fixpoint AbSem (P:program) (i:instr) (l:pp) :
                    AbEnv.S.t → (array AbEnv.S.Lat.t) := ...

  Theorem analyse_correct_wrt_collect : ... (* Théorème 3 *)
  Theorem analyse_correct : ... (* Corollaire 1 *)
End AbSem.
```

Figure 5. *Foncteur de l'interpréteur abstrait*

```
Module Type EnvAbstraction.

 Declare Module S : LatticeWiden.

 Parameter γ : program → S.Lat.t → environment → Prop.
 Parameter γ_monotone : ∀ P E1 E2 env,
   S.Lat.order E1 E2 → γ P E1 env → γ P E2 env.
 Parameter γ_meet_morph : ∀ P E1 E2 env,
   γ P E1 env → γ P E2 env → γ P (S.Lat.meet E1 E2) env.

 Parameter affect : (* [[x := e]]♯affect *)
    program → S.Lat.t → var → expr → S.Lat.t.
 Parameter affect_correct : ∀ P Env env env' x n e,
   γ P Env env → sem_expr P env e n →
   subst env x n env' → γ P (affect P Env x e) env'.

 Parameter init_env : program → S.Lat.t.
 Parameter init_env_correct : ∀ P (env:environment),
   γ P (init_env P) env.

 Parameter back_test : (* [[t]]♯test *)
    program → test → S.Lat.t → S.Lat.t.
 Parameter back_test_correct : ∀ P t Env env,
   γ P Env env → sem_test P env t true →
   γ P (back_test P t Env) env.

End EnvAbstraction.
```

Figure 6. *Signature abstraite pour l'abstraction des environnements*

4.4. *Abstraction non-relationnelle des environnements*

On distingue généralement deux techniques pour abstraire un ensemble de variables en interprétation abstraite. La première technique, dite *relationnelle*, calcule des relations entre variables. C'est par exemple le cas de l'abstraction par polyèdres (Cousot *et al.*, 1978) qui calcule des relations linéaires entre les variables entières d'un programme. L'autre technique, généralement moins coûteuse, calcule une propriété pour chaque variable, indépendamment des autres. On qualifie ce type d'abstraction de *non-relationnelle*. C'est cette dernière que nous présentons ici. L'abstraction des environnements de variables suit alors la structure des environnements : un environnement abstrait est une fonction entre noms de variables et valeurs numériques abstraites. Nous ne considérons pas d'abstraction numérique spécifique dans cette partie, nous verrons au fur et à mesure les opérateurs et hypothèses nécessaires à une telle abstraction.

Étant donnée une abstraction numérique $(\mathcal{P}(\mathbb{Z}), \subseteq, \cup, \cap) \xleftarrow{\gamma_{\text{Num}}} \left(\text{Num}^\sharp, \sqsubseteq^\sharp_{\text{Num}}, \sqcup^\sharp_{\text{Num}}, \sqcap^\sharp_{\text{Num}}\right)$, nous considérons le treillis standard des fonctions $\mathbb{V} \to \text{Num}^\sharp$ muni de l'ordre point à point (et nous le construisons en Coq en utilisant une nouvelle fois le foncteur `ArrayBinLatticeWiden`). La fonction de concrétisation entre $\mathbb{V} \to \text{Num}^\sharp$ et $\mathcal{P}(\text{Env})$ est définie par

$$\gamma_{\text{Env}}(\rho^\sharp) = \left\{\ \rho \mid \forall x \in \mathit{Var}_P,\ \rho(x) \in \gamma_{\text{Num}}(\rho^\sharp(x))\ \right\}.$$

Cette définition ne concerne que les variables déclarées dans la syntaxe du programme.

La fonction $[\![x := e]\!]^\sharp_{\text{affect}}$ peut être définie par

$$[\![x := e]\!]^\sharp_{\text{affect}}(\rho^\sharp) = \rho^\sharp[x \mapsto [\![e]\!]\uparrow^\sharp_{\text{expr}}]$$

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 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 (voir figure 7) à l'aide des opérateurs suivants de l'abstraction numérique :

– $\mathit{const}^\sharp \in \text{Num}^\sharp$ qui calcule une approximation des constantes.

– $\top_{\text{Num}}$ qui approche n'importe quelle valeur numérique.

– $[\![\odot]\!]\uparrow^\sharp_{\text{op}}$ qui est une approximation de l'opérateur arithmétique $\odot \in \{+, -, \times\}$.

4.4.1. *Construction de* $[\![t]\!]^\sharp_{\text{test}}$

Le deuxième opérateur requis par l'interface est $[\![t]\!]^\sharp_{\text{test}}$, que nous définissons par induction sur t (voir figure 7). Le cas d'un test de la forme $\neg t$ n'y figure pas car nous normalisons au préalable chaque test afin de faire disparaître toutes les occurrences du symbole de négation. Cette manipulation permet de réduire le nombre d'opérateurs abstraits requis pour l'abstraction numérique. Elle peut aussi avoir un impact sur la précision de l'analyse (Cousot, 1999). La définition de cet opérateur introduit trois nouveaux opérateurs, détaillés ci-dessous.

– $\text{reduc}^\sharp \in \text{Env}^\sharp \to \text{Env}^\sharp$ est un opérateur de réduction qui normalise les éléments abstraits vers un représentant partageant la même concrétisation. En effet, lorsque $\gamma_{\text{Num}}(\bot_{\text{Num}})$ 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 variable est associée à $\bot_{\text{Num}}$. L'opérateur $\text{reduc}^\sharp$ défini ci-dessous permet de détecter ce type d'environnements abstraits et de les remplacer par $\bot_{\text{Env}}$.

$$\text{reduc}^\sharp(\rho^\sharp) = \begin{cases} \bot_{\text{Env}} & \text{si } \exists x \in \mathbb{V},\ \rho^\sharp(x) = \bot_{\text{Num}} \\ \rho^\sharp & \text{sinon} \end{cases}$$

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.

$$\forall \rho^\sharp,\ \gamma_{\text{Env}}(\rho^\sharp) \subseteq \gamma_{\text{Env}}(\text{reduc}^\sharp(\rho^\sharp))$$

Évaluation abstraite des expressions

$$\boxed{[\![e]\!]\uparrow^\sharp_{\text{expr}} \in \text{Env}^\sharp \to \text{Num}^\sharp, \quad e \in expr}$$

$$\begin{array}{rcl}
[\![n]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) & = & const^\sharp(n) \\
[\![?]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) & = & \top_{\text{Num}} \\
[\![x]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) & = & \rho^\sharp(x) \\
[\![e_1 \odot e_2]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp) & = & [\![\odot]\!]\uparrow^\sharp_{\text{op}}\left([\![e_1]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp)\right)
\end{array}$$

Évaluation abstraite des tests

$$\boxed{[\![t]\!]^\sharp_{\text{test}} \in \text{Env}^\sharp \to \text{Env}^\sharp, \quad t \in test}$$

$$\begin{array}{rcl}
[\![e_1 \bowtie e_2]\!]^\sharp_{\text{test}}(\rho^\sharp) & = & \text{reduc}^\sharp\left([\![e_1]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n_1^\sharp) \sqcap^\sharp_{\text{Env}} [\![e_2]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n_2^\sharp)\right) \\
\multicolumn{3}{r}{\text{avec } (n_1^\sharp, n_2^\sharp) = [\![\bowtie]\!]\downarrow^\sharp_{\text{comp}}([\![e_1]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp))} \\
[\![t_1 \textbf{ or } t_2]\!]^\sharp_{\text{test}}(\rho^\sharp) & = & [\![t_1]\!]^\sharp_{\text{test}} \sqcup^\sharp_{\text{Env}} [\![t_2]\!]^\sharp_{\text{test}} \\
[\![t_1 \textbf{ and } t_2]\!]^\sharp_{\text{test}}(\rho^\sharp) & = & \text{reduc}^\sharp\left([\![t_1]\!]^\sharp_{\text{test}} \sqcap^\sharp_{\text{Env}} [\![t_2]\!]^\sharp_{\text{test}}\right)
\end{array}$$

Raffinement des expressions

$$\boxed{[\![e]\!]\downarrow^\sharp_{\text{expr}} \in \text{Env}^\sharp \times \text{Num}^\sharp \to \text{Env}^\sharp, \quad e \in expr}$$

$$\begin{array}{rcl}
[\![n]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) & = & \begin{cases} \bot_{\text{Env}} & \text{si } const^\sharp(n) \sqcap^\sharp_{\text{Num}} n^\sharp = \bot_{\text{Num}} \\ \rho^\sharp & \text{sinon} \end{cases} \\
[\![?]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) & = & \rho^\sharp \\
[\![x]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) & = & \text{reduc}^\sharp(\rho^\sharp[x \mapsto \rho^\sharp(x) \sqcap^\sharp_{\text{Num}} n^\sharp]) \\
[\![-e]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) & = & [\![e]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, [\![-]\!]\uparrow^\sharp_{\text{op}}(n^\sharp)) \\
[\![e_1 \odot e_2]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n^\sharp) & = & \text{reduc}^\sharp\left([\![e_1]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n_1^\sharp) \sqcap^\sharp_{\text{Env}} [\![e_2]\!]\downarrow^\sharp_{\text{expr}}(\rho^\sharp, n_2^\sharp)\right) \\
\multicolumn{3}{r}{\text{avec } (n_1^\sharp, n_2^\sharp) = [\![\odot]\!]\downarrow^\sharp_{\text{op}}(n^\sharp, [\![e_1]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp), [\![e_2]\!]\uparrow^\sharp_{\text{expr}}(\rho^\sharp))}
\end{array}$$

Figure 7. *Evaluation abstraite des expressions et des tests*

$\text{reduc}^\sharp$ est en fait une fermeture inférieure, ce qui assure une réduction optimale (car involutive), mais cette propriété n'est pas critique pour la correction de l'abstraction, seulement pour sa précision. 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 surcoût en effort de preuve.

– $[\![\bowtie]\!]\downarrow^\sharp_{\text{comp}} \in \left(\text{Num}^\sharp \times \text{Num}^\sharp\right) \to \left(\text{Num}^\sharp \times \text{Num}^\sharp\right)$ calcule un raffinement de deux valeurs numériques abstraites, sachant qu'elles vérifient une condition $\bowtie$. Par

exemple, pour l'abstraction numérique de parité introduite dans la section 2, $[\![=]\!]\downarrow^\sharp_{\mathrm{comp}}$ $(\overline{0},\overline{1}) = (\bot,\bot)$ car une valeur entière ne peut pas être à la fois paire et impaire.

– $[\![e]\!]\downarrow^\sharp_{\mathrm{expr}} \in \left(\mathrm{Env}^\sharp \times \mathrm{Num}^\sharp\right) \to \mathrm{Env}^\sharp$ 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. Sa correction est formalisée par

$$\forall \rho^\sharp \in \mathrm{Env}^\sharp, n^\sharp \in \mathrm{Num}^\sharp,$$
$$\left\{\ \rho \in \gamma_{\mathrm{Env}}(\rho^\sharp)\ |\exists n \in [\![e]\!]\uparrow^\rho_{\mathrm{expr}} \cap \gamma_{\mathrm{Num}}(n^\sharp)\ \right\} \subseteq \gamma_{\mathrm{Env}}([\![e]\!]\downarrow^\sharp_{\mathrm{expr}}\ (\rho^\sharp, n^\sharp)$$

Sa définition, par induction sur e, est donnée en figure 7. Un dernier opérateur numérique est pour cela nécessaire : $[\![\odot]\!]\downarrow^\sharp_{\mathrm{op}} \in \left(\mathrm{Num}^\sharp \times \mathrm{Num}^\sharp \times \mathrm{Num}^\sharp\right) \to \left(\mathrm{Num}^\sharp \times \mathrm{Num}^\sharp\right)$, qui calcule un raffinement de deux valeurs numériques $n^\sharp_1$ et $n^\sharp_2$ sachant que le résultat de l'opération binaire $\odot$ vaut $n^\sharp$ sur ces valeurs.

Tous les opérateurs requis pour l'abstraction numérique sont présentés, avec leur spécification formelle, dans la signature de module `NumAbstraction` de la figure 8. Nous proposons pour notre analyseur plusieurs implémentations de la signature de module `NumAbstraction` : l'abstraction par signe, l'abstraction par congruence et enfin l'abstraction par intervalle. Il s'agit d'abstractions numériques classiques en interprétation abstraite et nous ne détaillons pas ces implémentations, par manque de place. Le lecteur peut se référer à (Pichardie, 2005) pour plus de détails.

5. Travaux connexes

Cet article fait suite à (Cachera *et al.*, 2005a), où une analyse de type flot de données exprimée sous forme de contraintes avait été formalisée en Coq. Le langage retenu était une version simplifiée de *bytecode* pour Java Card, et la correction de l'analyse prouvée par rapport à une sémantique opérationnelle. Cet article insistait particulièrement sur la bibliothèque certifiée de treillis permettant de construire les analyses de façon modulaire. Le calcul du résultat de l'analyse se faisait à l'aide d'une itération standard de type *round robin*. À la différence du présent article, la fonction de concrétisation était monotone, formalisée via une relation d'approximation $\sim$. Dans (Cachera *et al.*, 2005b), le cadre précédent a été appliqué à une toute autre analyse, puisqu'il s'agissait de garantir l'absence de fuites de mémoire pour un langage de type *bytecode* Java, en prouvant l'absence de boucles inter-procédurales. Dans (Besson *et al.*, 2006), une partie de ce travail a été appliquée au domaine du code porteur de preuve (PCC), fournissant une analyse certifiée par intervalles pour du *bytecode* Java, adaptée de l'analyse de langage While. Dans cet article, il n'était pas nécessaire de prouver la terminaison des calculs itératifs, et les abstractions relationnelles n'étaient pas prises en compte. Par ailleurs, il n'était pas fait mention des difficultés particulières rencontrées lorsque l'on veut adapter le cadre de l'interprétation abstraite à une logique constructive.

Une analyse de langage de type While par interprétation abstraite est présentée en détail par Cousot dans (Cousot, 1999). C'est cette analyse qui sert de base

```
Module Type NumAbstraction.

  Declare Module S : LatticeWiden.

  Parameter γ : S.Lat.t → Z → Prop.
  Parameter γ_monotone : ∀ N1 N2 n,
    S.Lat.order N1 N2 → γ N1 n → γ N2 n.
  Parameter γ_meet_morph : ∀ N1 N2 n,
    γ N1 n → γ N2 n → γ (S.Lat.meet N1 N2) n.

  Parameter BackTest : (* [[⋈]]↓♯comp *)
    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 → γ N1 n1 → γ N2 n2 →
    BackTest c N1 N2 = (N1',N2') → γ N1' n1 ∧ γ N2' n2.

  Parameter SemOp : (* [[⊙]]↑♯op *)
    op → S.Lat.t → S.Lat.t → S.Lat.t.
  Parameter SemOp_correct : ∀ o N1 N2 n1 n2 n,
    γ N1 n1 → γ N2 n2 →
    sem_op o n1 n2 n → γ (SemOp o N1 N2) n.

  Parameter BackSemOp : (* [[e]]↓♯expr *)
    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',
    γ N1 n1 → γ N2 n2 → sem_op o n1 n2 n → γ N n →
    BackSemOp o N N1 N2 = (N1',N2') → γ N1' n1 ∧ γ N2' n2.

  Parameter const : Z → S.Lat.t. (* const♯ *)
  Parameter const_correct : ∀ n, γ (const n) n.

  Parameter top : S.Lat.t. (* ⊤Num *)
  Parameter top_correct : ∀ n, γ top n.

  Parameter bottom_empty : ∀ n, ~ γ S.Lat.bottom n.

End NumAbstraction.
```

Figure 8. *Signature abstraite pour l'abstraction des valeurs numériques*

au présent travail. Une version plus résumée peut être trouvée dans la thèse de Miné (Miné, 2004). Monniaux avait dans (Monniaux, 1998) le projet de formaliser les connexions de Galois en Coq, ce qui a nécessité l'usage de plusieurs axiomes. Il a obtenu ainsi des résultats sur la combinaison de connexions de Galois. Cependant, l'assistant de preuve n'était à l'époque pas doté du même mécanisme d'extraction et n'a pas été en mesure d'extraire des analyseurs à partir des développements réalisés. Plus récemment, Bertot a proposé une formalisation Coq d'interprétation abstraite pour un langage While (Bertot, 2008), ce qui constitue le travail qui se rapproche le plus du nôtre. Son étude se base sur une sémantique axiomatique sous la forme d'un calcul de plus faibles préconditions, générant de façon classique un ensemble de conditions de vérification. L'itération bornée d'opérateurs de sur-approximation puis de rétrécissement permet d'inférer de façon heuristique des invariants pour les boucles. La correction de l'analyse repose sur le fait que les conditions de vérification générées grâce à ces invariants seront vérifiées. L'article ne s'intéresse ni à la terminaison des programmes puisqu'il s'appuie sur un calcul libéral de plus faibles préconditions, ni à celle des analyses. Il suit une approche ad hoc, introduisant les propriétés des opérateurs abstraits au fur et à mesure, et sans proposer de réflexion générale sur l'interprétation abstraite en Coq.

D'autres approches utilisent un assistant de preuve pour certifier la correction d'analyses statiques. En ce qui concerne la vérification de *bytecode*, Barthe et al. (Barthe *et al.*, 2001a) ont montré comment formaliser la vérification de *bytecode* Java, en raisonnant sur une sémantique exécutable du langage. Ils ont ensuite proposé (Barthe *et al.*, 2001b) 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. D'autres approches ont été suivies pour formaliser le vérificateur de *bytecode*. Gerwin Klein et Tobias Nipkow ont utilisé Isabelle/HOL pour formaliser le vérificateur de *bytecode* (Klein *et al.*, 2002). Leurs travaux autour de Java sont considérables puisqu'ils proposent (Klein *et al.*, 2006) 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 une bibliothèque 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. D'autres travaux menés par Leroy et al. (Bertot *et al.*, 2004; Leroy, 2006) montrent aussi que des compilateurs optimisants pour un langage similaire à C peuvent être prouvés corrects en Coq. Finalement, cet assistant de preuve est largement utilisé pour étudier les propriétés des langages de programmation et en particulier de leurs systèmes de types, et Coq fait partie des solutions proposées au défi général de vérification formelle connu sous le nom de *POPLmark* (Aydemir *et al.*, 2005).

6. Conclusion

Nous avons présenté un cadre d'interprétation abstraite adapté à une formalisation dans la logique constructive de Coq. Nous avons pour cela allégé le cadre classique, en ne retenant pas par exemple les hypothèses permettant de prouver l'optimalité des abstractions calculées. Pour autant, les analyseurs construits avec notre cadre fournissent une sur-approximation correcte de la sémantique concrète des programmes. De plus, l'utilisation d'une logique constructive permet d'extraire directement l'analyseur de la preuve de sa spécification, comblant le fossé entre spécification et implémentation de l'analyse.

L'analyseur produit concerne un langage impératif simple de type WHILE. Le développement Coq correspondant à ce cas d'étude comporte environ 10.000 lignes de Coq et permet d'obtenir environ 2500 de lignes de Caml par extraction automatique. Il est disponible en ligne à l'adresse

`http://www.irisa.fr/lande/pichardie/while`

pour être extrait et expérimenté sur quelques exemples. Cet analyseur est générique. Il est en effet possible de choisir l'abstraction numérique adaptée à l'analyse désirée. Il est également possible de brancher une abstraction relationnelle sur l'interface des environnements abstraits, tout en gardant le même mécanisme d'itération sur la syntaxe des programmes.

Ce travail constitue un premier pas vers l'objectif plus ambitieux de certification d'un analyseur complexe comme ASTRÉE, qui traite un langage réel — C en l'occurrence — avec les mêmes techniques d'élargissement/rétrécissement et le même type de stratégie d'itération. Parmi les défis restant à relever pour atteindre cet objectif, il faudra notamment ajouter des domaines abstraits relationnels et gérer les entiers bornés et les flottants.

7. Bibliographie

Aczel P., « An introduction to inductive definitions », *in* J. Barwise (ed.), *Handbook of Mathematical Logic*, North-Holland Publishing Company, p. 739-, 1977.

Aydemir B., Bohannon A., Fairbairn M., Foster J., Pierce B., Sewell P., Vytiniotis D., Washburn G., Weirich S., Zdancewic S., « Mechanized metatheory for the masses: The POPLmark Challenge », *Proc. of TPHOLs 2005*, Springer LNCS vol. 3603, 2005.

Barthe G., Dufay G., Huisman M., de Sousa S. M., « Jakarta: a toolset for reasoning about JavaCard », *Proc. of e-SMART'01*, Springer LNCS vol. 2140, 2001a.

Barthe G., Dufay G., Jakubiec L., Serpette B., de Sousa S. M., « A formal executable semantics of the JavaCard platform », *Proc. ESOP'01*, n° 2028 in *LNCS*, Springer-Verlag, 2001b.

Bertot Y., « Structural abstract interpretation, a formal study using Coq », *LERNET Summer School*, LNCS, Springer, 2008.

Bertot Y., Grégoire B., Leroy X., « A structured approach to proving compiler optimizations based on dataflow analysis », *TYPES'04*, p. 66-81, 2004.

Besson F., Jensen T., Pichardie D., « Proof-carrying code from certified abstract interpretation to fixpoint compression », *Theoretical Computer Science*, vol. 364, n° 3, p. 273-291, 2006.

Cachera D., Jensen T., Pichardie D., Rusu V., « Extracting a data flow analyser in constructive logic », *Theoretical Computer Science*, vol. 342, n° 1, p. 56-78, 2005a.

Cachera D., Jensen T., Pichardie D., Schneider G., « Certified memory usage analysis », *Proc. of 13th International Symposium on Formal Methods (FM'05)*, LNCS, 2005b.

Cousot P., « The calculational design of a generic abstract interpreter », *in* M. Broy, R. Steinbrüggen (eds), *Calculational System Design*, NATO ASI Series F. IOS Press, Amsterdam, 1999.

Cousot P., Cousot R., « Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints », *Proc. of POPL'77*, ACM Press, p. 238-252, 1977.

Cousot P., Cousot R., « Systematic design of program analysis frameworks », *ACM POPL'79*, ACM Press, p. 269-282, 1979.

Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux D., Rival X., « The ASTRÉE analyser », *Proc. of ESOP'05*, vol. 3444 of *LNCS*, Springer, p. 21-30, 2005.

Cousot P., Cousot R., Feret J., Mauborgne L., Miné A., Monniaux D., Rival X., « Combination of Abstractions in the ASTRÉE Static Analyzer », *Post-proc. of ASIAN'06*, vol. 4435 of *LNCS*, Springer, p. 272-300, 2006.

Cousot P., Halbwachs N., « Automatic discovery of linear restraints among variables of a program », *Proc. of POPL'78*, p. 84-97, 1978.

Feret J., Analysis of mobile systems by abstract interpretation, PhD thesis, École Polytechnique, 2005.

Klein G., Nipkow T., « Verified bytecode verifiers », *Theoretical Computer Science*, vol. 298, n° 3, p. 583-626, 2002.

Klein G., Nipkow T., « A machine-checked model for a Java-like language, virtual machine and compiler », *ACM Transactions on Programming Languages and Systems*, vol. 28, n° 4, p. 619-695, 2006.

Leroy X., « Formal certification of a compiler back-end, or: programming a compiler with a proof assistant », *ACM POPL'06*, ACM Press, p. 42-54, 2006.

Miné A., Weakly relational numerical abstract domains, PhD thesis, École Polytechnique, 2004.

Monniaux D., *« Réalisation mécanisée d'interpréteurs abstraits »*, Rapport de DEA, Université Paris VII, 1998.

Nielson H. R., Nielson F., *Semantics with applications: a formal introduction*, John Wiley & Sons, Inc., New York, NY, USA, 1992.

Okasaki C., Gill A., « Fast mergeable integer maps », *Proc. of the ACM SIGPLAN Workshop on ML*, p. 77-86, 1998.

Pichardie D., Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés, PhD thesis, Université Rennes 1, 2005.

Pichardie D., « Building certified static analysers by modular construction of well-founded lattices », *Proc. of FICS'08*, vol. 212 of *ENTCS*, p. 225-239, 2008.

Winskel G., *The formal semantics of programming languages: an introduction*, MIT Press, Cambridge, MA, USA, 1993.

David Cachera *est maître de conférences en informatique à l'École normale supérieure de Cachan – antenne de Bretagne. Ses travaux portent sur la sémantique des langages de programmation. Il s'intéresse plus particulièrement à la certification des analyses statiques et aux aspects quantitatifs des analyses statiques.*

David Pichardie *est chargé de recherche à l'INRIA Rennes dans l'équipe-projet Celtique. Son domaine de recherche est la conception de logiciels sûrs à l'aide de méthodes formelles comme l'analyse statique ou la preuve assistée.*