Dérivation symbolique d'interpréteur abstrait en Coq

Dominique Barbe
ENS Rennes,
L3, Université Rennes 1,
Parcours Recherche et Innovation

Rapport du stage effectué à l'INRIA Rennes dans l'équipe Celtique sous la direction de David Pichardie.

Résumé

La fiabilité des logiciels est une question importante, tant du point de vue économique (coûts de développement) que du point de vue humain (informatique embarquée). Il existe deux grandes méthodes pour certifier des logiciels : la vérification à la main à l'aide d'un assistant de preuve ou la vérification automatique par un autre outil. Dans ce stage, nous nous sommes intéressés à la fusion de ces deux méthodes, et en particulier à utiliser entièrement le cadre théorique utilisé pour l'analyse de programmes.

Mots-clefs : Assistant de preuve, analyse statique, interprétation abstraite.

Table des matières

Introduction

Comment faire confiance aux programmes informatiques incorporés dans les objets qui nous entourent ? Un bug, c'est fréquent : on en voit sur les téléphones portables, sur des écrans dans un hall, sur le terminal d'une imprimante, etc. Mais n'importe quel programme informatique peut y être sujet : par exemple ceux utilisés dans les centrales nucléaires, ou embarqués dans les avions. Une faille dans de tels logiciels peut avoir de graves conséquences ; leur coût de développement est donc très élevé, puisque de nombreux moyens sont mis en œuvre pour éviter toute erreur.

On prend en exemple un programme C simple :

```
int t[25] = {0} ;
int x = y = 0 ;
while(y < 10) {
  y = y + 1 ;
  if (x < 25)
    { x = x + y ; }
  else
    { x = x - y ; }
}
t[x] = 1 ;
```

On peut se poser quelques question concernant son comportement : le programme termine-t-il ? L'accès à la case x du tableau à la dernière ligne est-il dangereux (x est-il bien un indice du tableau) ? Il existe plusieurs techniques permettant de répondre à cette question : prouver manuellement la correction du programme, faire des tests, etc. Mais l'idéal serait de pouvoir automatiser la recherche de réponses à ce genre de questions, afin de pouvoir passer à l'échelle et analyser des programmes bien plus imposants, dont la vérification à la main ne peut être faite de façon sûre ou est associée à un coût ou une durée élevée.

On pose le problème de façon plus générale. On s'intéresse à la sémantique des programmes. Étant donné le code d'un programme (sa syntaxe), quelles sont ses exécutions possibles dans tous les environnement d'exécution (sa sémantique) ? On veut travailler ce problème de façon algorithmique : existe-t-il un programme qui puisse calculer la sémantique d'un programme quelconque ? L'objectif est d'en déduire la réponses à des questions qui ressemblent à celles ci-dessous :

- déterminer si un programme s'arrête ;
- déterminer si un programme renvoie-t-il bien les valeurs que l'on souhaite ;
- déterminer les zones de la mémoire auxquelles le programme accède ;
- déterminer les zones mémoires réservées par le programme (afin de les libérer).

Le problème de l'arrêt est indécidable : il n'existe pas de programme qui puisse décider si un un programme quelconque s'arrête sur une entrée donnée ; le premier exemple est condamné. En fait on peut énoncer un résultat plus général. Le théorème de Rice annonce qu'on ne peut décider d'une propriété non triviale d'un programme à partir de son code (pour les langages de programmation avec la même expressivité que les machines de Turing). Vouloir déterminer algorithmiquement la sémantique des programmes est donc voué à l'échec, si l'on considère le problème dans sa globalité.

Le problème ne peut être résolu de façon totale. Deux pistes sont alors envisageables : soit abandonner la recherche d'une automatisation complète, soit construire des algorithmes capables d'analyser au moins une partie des programmes.

Dans le premier cas, on peut utiliser un assistant de preuve, comme Coq [1] ou Isabelle [2], pour certifier certains aspects d'un programme. Un assistant de preuve est un logiciel permettant de formaliser et de démontrer des théorèmes mathématiques ; en particulier, on peut les appliquer à la démonstration de propriétés sur l'exécution de programmes informatiques. Certes, ces méthodes ne sont pas automatiques : il faut guider le logiciel lors de la preuve. Mais les assistants de preuve reposent pour la plupart sur des noyaux réduits, basés sur des théories mathématiques générales ; ils sont donc considérés comme fiables. Ils permettent donc de s'assurer qu'il n'y a pas de fautes dans les raisonnements effectués. De plus, certains assistants de preuve fournissent quand même quelques outils d'automatisation dans des cadres précis.

Dans le second cas, on cherche des algorithmes capables d'analyser automatiquement des programmes. Il y a deux grandes familles de méthodes d'analyse de programmes. En analyse dynamique, on exécute le code, et on regarde ses effets. Cette méthode a l'avantage d'être simple à mettre en œuvre. En revanche, elle ne permet pas de répondre à toutes les questions (par exemple celle de l'arrêt), et il est difficile de l'appliquer de façon exhaustive. En analyse statique, on essaie de prouver des propriétés sur l'exécution d'un programme, sans jamais l'exécuter, c'est-à-dire en examinant son code, et ce de façon automatique. Ici, on s'intéresse à l'analyse statique par interprétation abstraite, qui sera détaillée plus loin.

Un analyseur statique est un programme qui lit un code et qui tente d'en déduire sa sémantique. Mais l'analyseur est lui aussi un programme, parfois très complexe. Qui va donc vérifier l'analyseur ? Il faut trouver un point de départ où mettre sa confiance. Un solution est de mélanger les deux méthodes présentées : utiliser un assistant de preuve pour construire l'analyseur ; ici, on se réfère à l'assistant de preuve Coq, dont on reparlera ci-après. C'est cette idée qui a été explorée pendant ce stage. L'assistant de preuve va permettre de vérifier pas à pas les différents mécanismes de l'analyseur au cours de sa construction : le but est d'obtenir un analyseur correct par construction.

Pour construire l'analyseur, on se base sur le cadre théorique suivant : l'interprétation abstraite. Il s'agit d'une théorie d'approximation de la sémantique des programmes. En interprétation abstraite, on distingue un monde concret, permettant d'exprimer des propriétés sur les programmes, et un monde abstrait, où on approxime de telles propriétés. Ces deux mondes sont reliés par deux fonctions, α, la fonction d'abstraction, et γ, la fonction de concrétisation. En particulier, ces fonctions permettent de spécifier ce que sont une approximation correcte et une approximation optimale. Nous avons implémenté cette théorie en Coq pour pouvoir construire l'analyseur.

1 Préliminaires

1.1 L'assistant de preuve Coq

Coq est un assistant de preuve développé depuis les années 80. C'est un outil permettant de réaliser des preuves mathématiques, automatisant certains aspects, mais nécessitant toujours qu'un humain guide le raisonnement. Coq repose sur un noyau qui utilise la théorie du calcul des constructions inductives [6] pour s'assurer que les expressions sont bien typées et que les preuves implémentées sont correctes.

Le développement de Coq s'inscrit dans une logique plus large de vérification systématique des outils informatiques ou des preuves mathématiques. Il a par exemple servi à développer CompCert, un compilateur C vérifié. Un autre exemple est l'implémentation et la vérification de la preuve du théorème des 4 couleurs ou du théorème de Feit-Thomson [5].

Coq est muni d'un mécanisme d'extraction de code Caml à partir de code Coq. Ainsi, une fois l'analyseur construit en Coq et vérifié, on pourra l'exporter dans un langage plus réaliste (du point de vue performances).

Nous allons brièvement montrer comment fonctionne Coq. Contrairement à la plupart des autres langages, Coq ne propose pas directement les types courants (entiers, flottants , booléens, etc). A la place, il est muni d'un système de types inductifs qui va permettre de définir très rapidement les types dont on aura besoin. Voyons par exemple les entiers : en entier naturel est soit 0, soit le successeur d'un entier (définition liée à l'arithmétique de Péano). En Coq, cela donne :

```
Inductive nat : Type :=
  | O
  | S (n : nat).
```

De la même façon, on peut définir des prédicats. Nous implémentons la relation d'ordre usuelle de la façon suivante :

— O est plus petit que tout entier ;
— si on a $n \leq m$, alors on en déduit $(n+1) \leq (m+1)$.

On traduit ceci en Coq par :

```
Inductive le_nat : nat → nat → Prop :=
  | le_nat_O : forall n, le_nat O n
  | le_nat_S : forall n1 n2, le_nat n1 n2 → le_nat (S n1) (S n2).
```

Nous avons défini un type et un prédicat, nous allons maintenant définir une fonction : l'addition sur les entiers. Du fait de la définition inductive des entiers, il faut définir cette fonction de façon récursive :

— pour tout m, on a $0 + m = m$;
— si on a $n + m$, alors on en déduit $(n+1) + m$.

```
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O ⇒ m
  | S n' ⇒ S (plus n' m)
  end.
```

Remarque : nous avons utilisé le même mot-clef pour définir les entiers et la relation d'ordre : "Inductive". C'est parce que ces deux objets, bien que de types différents, on étés tous deux définis par induction.

Nous avons défini une fonction, nous allons maintenant prouver un théorème sur cette fonction :

$$\forall (n; m) \in \mathbf{N}^2 \quad n \leq n + m$$

Les preuves se font de façon interactive en Coq : on annonce un théorème, et Coq nous montre le but associé. Tant que tous les buts n'ont pas été résolus, on applique des "tactiques" qui transforment le but, le divisent en plusieurs autres, ou le résolvent.

```
Theorem n_le_n_plus_m :
  forall (n m : nat), le_nat n (plus n m).
Proof.
  (* Le but courant est : forall n m : nat, le_nat n (plus n m) *)
  intros n m.
  induction n.
  (* Cette tactique a genere deux buts :
     1) le_nat 0 (plus 0 m)
     2) le_nat (S n) (plus (S n) m) *)
  - (* On s'interesse au premier cas. *)
    simpl.
    (* Apres simplification, le but est : le_nat 0 m *)
    apply le_nat_0.
    (* Le premier but est resolu *)
  - (* On s'interesse au second cas. Cette fois, on a une hypothese :
       IHn : le_nat n (plus n m) *)
    simpl.
    (* Apres simplification, le but est : le_nat (S n) (S (plus n m)) *)
    apply le_nat_S.
    (* Le constructeur le_nat_S transforme le but en : le_nat n (plus n m) *)
    apply IHn.
    (* L'hypothese d'induction prouve le but. *)
Qed.
```

Maintenant que les rudiments de Coq ont été abordés, nous passons à un autre sujet : la théorie mathématique de l'interprétation abstraite, que nous allons implémenter et utiliser en Coq.

1.2 Interprétation abstraite

1.2.1 Introduction

L'interprétation abstraite est une théorie formalisée par Patrick et Radhia Cousot à partir des années 70. Elle fournit un cadre théorique permettant de formaliser des méthodes d'analyse statique. L'idée de base d'approximer la sémantique des programmes. En effet, puisqu'elle est en général non calculable, on va simplifier les opérations que font les programmes ; en perdant en précision, on espère gagner en décidabilité et garder une sémantique assez précise pour pouvoir répondre aux questions qui nous intéressent.

L'interprétation abstraite permet de calculer en différents points du programme des propriétés sur ses variables. De telles propriétés peuvent être "x est pair", "$0 \leq x < 100$ ou "$x^2 \in (2\mathbf{Z} + 7) \cap \mathbf{P}$". De façon générale, pour $x \in \mathbf{Z}$, une propriété sur x est un élément de $\wp(\mathbf{Z})$. Puisque

le monde des propriétés est trop complexe ($\wp(\mathbf{Z})$ est "trop gros"), on cherche un monde plus restreint.

Un exemple souvent utilisé est celui de l'abstraction par signe : au lieu de regarder toutes les propriétés sur les entiers, on ne garde que les suivantes : $\emptyset$, $\{0\}$, $\mathbf{Z}_-$, $\mathbf{Z}_+$ *et* $\mathbf{Z}$ (on rappelle qu'une propriété est un élément de $\wp(\mathbf{Z})$). Ces propriétés forment le monde abstrait. Puisqu'on ne considère plus que ces propriétés-là, il faut trouver un représentant (i.e une approximation) pour toutes les autres propriétés. On fait le choix suivant de la sur-approximation : l'objectif de l'interprétation abstraite est de faire de l'analyse de programmes pour les certifier corrects. Sous cet angle, il vaut mieux être trop prudent que pas assez ; autrement dit, il vaut mieux avoir des faux positifs (l'analyseur averti d'une erreur qui n'apparaît pas en pratique) que des faux négatifs (l'analyseur certifie le programme sans erreurs alors qu'il y en a). Il faut donc abstraire chaque élément de $\wp(\mathbf{Z})$ par le plus petit élément de $\{\emptyset, \{0\}, \mathbf{Z}_-, \mathbf{Z}_+ \mathbf{Z}\}$ qui le contient. En effet, il faut choisir un élément qui le contient, puisqu'il correspond à une propriété plus générale (on fait le choix de la sur-approximation). Il faut aussi choisir le plus petit élément possible, pour rester le plus précis possible.

L'interprétation abstraite se base sur la théorie des connexions de Galois, qui va être introduite ici.

1.2.2 Les connexions de Galois

Définition : un *treillis* est un quadruplet $(E; \sqsubseteq; \sqcup; \sqcap)$ où
- E est un ensemble ;
- $\sqsubseteq$ est une relation d'ordre partielle sur E ;
- $\sqcup : E \times E \to E$ est une borne supérieure binaire :
  - $\forall (x; y) \in E^2, \quad x \sqsubseteq x \sqcup y \quad et \quad y \sqsubseteq x \sqcup y$
  - $\forall (x; y; m) \in E^3, \quad (x \sqsubseteq m \wedge y \sqsubseteq m) \Rightarrow x \sqcup y \sqsubseteq m$
- $\sqcap : E \times E \to E$ est une borne inférieure binaire :
  - $\forall (x; y) \in E^2, \quad x \sqcap y \sqsubseteq x \quad et \quad x \sqcap y \sqsubseteq y$
  - $\forall (x; y; m) \in E^3, \quad (m \sqsubseteq x \wedge m \sqsubseteq y) \Rightarrow m \sqsubseteq x \sqcap y$

Exemple : $\wp(\mathbf{Z})$ peut être munis d'une structure de treillis. La relation d'ordre partielle sera l'inclusion ; on la voit comme l'implication (au sens de propriétés) ; elle est partielle car deux propriétés ne sont pas toujours comparables. La borne supérieure sera l'union (de deux ensembles) ; on la voit comme la disjonction de deux hypothèses. La borne inférieure sera l'intersection (de deux ensembles) ; on la voit comme la conjonction de deux hypothèses.

Définition : étant donnés $(E; \sqsubseteq; \sqcup; \sqcap)$ et $(E^\#; \sqsubseteq^\#; \sqcup^\#; \sqcap^\#)$ deux treillis et une paire de fonctions $\alpha : E \to E^\#$ et $\gamma : E^\# \to E$. La paire $(\alpha; \gamma)$ forme une connexion de Galois lorsque :

$$\forall \left(x; x^\#\right) \in E \times E^\#, \quad \alpha(x) \sqsubseteq^\# x^\# \Leftrightarrow x \sqsubseteq \gamma(x^\#)$$

Informellement, on interprète E comme étant le monde concret et $E^\#$ le monde abstrait. α est la fonction d'abstraction : elle associe à chaque propriété concrète P sa meilleure abstraction $P^\#$. γ est la fonction de concrétisation : elle associe à chaque propriété abstraite $P^\#$ la propriété concrète la plus générale parmi celles qu'elle approxime correctement. On essaie de comprendre l'équivalence. Soit x un élément de E, une propriété donc. $\alpha(x)$ en est son abstraction, une propriété abstraite donc. Soit $x^\#$ une autre propriété abstraite qui approxime aussi x ; le fait

que α soit une meilleure abstraction s'exprime par : $\alpha(x) \sqsubseteq^{\#} x^{\#}$. Un autre façon d'exprimer cette optimalité se fait par le biais de la fonction de concrétisation : le concrétisé de $x^{\#}$ est plus général que x (puisque $x^{\#}$ en est une approximation) : $x \sqsubseteq \gamma(x^{\#})$.

Propriétés : étant donnés une connexion de Galois $(\alpha; \gamma)$ entre deux treillis $(E; \sqsubseteq; \sqcup; \sqcap)$ et $(E^{\#}; \sqsubseteq^{\#}; \sqcup^{\#}; \sqcap^{\#})$, on a :

1. α et γ sont croissantes
2. $\forall x \in E, \quad x \sqsubseteq \gamma(\alpha(x))$
3. $\forall x^{\#} \in E^{\#}, \alpha(\gamma(x^{\#})) \sqsubseteq^{\#} x^{\#}$

Informellement, il faut comprendre ces trois propriétés ainsi :

1. α et γ préservent l'ordre, i.e l'implication entre les informations.
   On reprend l'exemple de l'abstraction par signe : dans le monde concret, on a par exemple $(x \; premier \wedge x \neq 2) \Rightarrow (x \; impair)$, soit $\{x | x \; premier \wedge x \neq 2\} \sqsubseteq \{x | x \; impair\}$.
   On a annoncé que l'abstraction d'une propriété serait le plus petit élément qui le contient (on admet qu'on a une connexion de Galois). Donc : $\alpha(\{x | x \; premier \wedge x \neq 2\}) = \mathbf{Z}_+$ et $\alpha(\{x | x \; impair\}) = \mathbf{Z}$.
   On a bien $\mathbf{Z}_+ \sqsubseteq^{\#} \mathbf{Z}$: α conserve l'implication entre les informations.
   A l'inverse : $\gamma(\mathbf{Z}_+) = \{x | 0 \sqsubseteq x\}$ et $\gamma(\mathbf{Z}) = \{x | x \in \mathbf{Z}\}$.
   On a bien $\{x | 0 \sqsubseteq x\} \sqsubseteq \{x | x \in \mathbf{Z}\}$: γ conserve l'implication entre les informations.
2. α sur-approxime (si on passe vers le monde abstrait puis qu'on revient au concret, on a un élément plus grand qu'au départ).
3. γ n'introduit pas de perte d'information.

Les propriétés 2. et 3. traduisent ce qu'on cherchait en terme de perte d'information : la première fonction sur-approxime pour être sûre de ne manquer aucune exécution, aucun état, aucun calcul possible du programme, et la deuxième assure qu'on ne perd pas d'information sur les calculs effectués en revenant dans le monde concret.

Le théorie de l'interprétation abstraite fait normalement intervenir des treillis complet. Dans un treillis complet, union et intersection ne sont pas binaires, mais opèrent sur des parties quelconques de l'ensemble de base. De telles opérations permettent de manipuler des notions de point fixe. Dans le cadre que nous avons retenu, nous n'étudions pas les points fixes ; nous nous sommes donc contentés de la notion de treillis, qui est suffisante.

2 Formalisation de l'interprétation abstraite en Coq

La construction d'un analyseur statique, utilisant la théorie de l'interprétation abstraite, certifié en Coq a déjà été entreprise [3]. Mais la théorie de l'interprétation abstraite n'y est pas entièrement exploitée. Les fonctions d'abstraction et de concrétisation α et γ permettent d'exprimer des critères d'optimalité sur les approximations effectuées. Pour un opérateur concret f, l'opérateur abstrait $f^{\#}$ qui l'approxime le mieux se caractérise par $f^{\#} = \alpha \circ f \circ \gamma$. Par une suite de dérivation d'une telle spécification, on peut obtenir un opérateur abstrait explicite. Tenter d'implémenter de telles dérivations en Coq a été l'objectif de ce stage.

Nous allons présenter comment nous avons implémenté la théorie de l'interprétation abstraite en Coq. Cette théorie fait intervenir deux structures principales : les treillis et les connexions de Galois. Nous allons montrer comment nous les avons formalisés en Coq. Nous avons utilisés pour cela les *Type Classes*. Les *Type Classes* permettent de définir des structures, et de prouver des théorèmes sur toutes les instances de ces structures. Les Type Classes permettent de surcharger des fonctions ou des théorèmes, en tirant parti du système de typage de Coq pour inférer les arguments qui manqueraient et aller chercher les bons objets. Un exemple d'une telle surcharge se trouve à la fin de cette section, avec la preuve d'un théorème.

2.1 Les treillis

L'implémentation est la suivante :

```
Class Lattice (A : Type) : Type :=
  {
    equiv : A → A → Prop ;
    le : A → A → Prop ;
    union : A → A → A ;
    inter : A → A → A
  }.
Class Wf_Lattice {A : Type} (L : Lattice A) : Prop :=
  {
    le_refl : forall x, le x x ;
    le_antisymm : forall x1 x2, le x1 x2 → le x2 x1 → equiv x1 x2 ;
    le_trans : forall x1 x2 x3, le x1 x2 → le x2 x3 → le x1 x3 ;
    union_le_l : forall x1 x2, le x1 (union x1 x2) ;
    union_le_r : forall x1 x2, le x2 (union x1 x2) ;
    union_opt : forall x x1 x2, le x1 x → le x2 x → le (union x1 x2) x ;
    inter_le_l : forall x1 x2, le (inter x1 x2) x1 ;
    inter_le_r : forall x1 x2, le (inter x1 x2) x2 ;
    inter_opt : forall x x1 x2, le x x1 → le x x2 → le x (inter x1 x2)
  }.
```

Quelques explications sont de rigueur.

Un treillis est un type, fait d'un ensemble partiellement ordonné munis d'une union et d'une intersection. Ici, `A` est l'ensemble considéré, `le` est un ordre sur `A`, `equiv` une relation d'équivalence, `union` et `inter` sont deux lois de composition interne.

`Wf_Lattice` est une propriété sur les treillis. Dans son corps 3 théorèmes caractérisent le fait que `equiv` est une relation d'équivalence, 3 théorèmes caractérisent le fait que `le` est un ordre partiel, 3 théorèmes caractérisent l'union (le plus petit des majorants) et 3 théorèmes caractérisent l'intersection (le plus grand des minorants).

Remarque : Nous avons associé à A sa propre relation d'égalité car celle proposée naturellement en Coq est parfois trop précise. Prenons un exemple avec les fonctions sur les entiers naturels :

$$f_1 : x \to 0$$

$$f_2 : x \to x - x$$

$$\forall\, (g_1; g_2) \in \mathbf{N}^{\mathbf{N}},\ g_1 \leq g_2 \quad ssi \quad (\forall n \in \mathbf{N}, g_1(n) \leq g_2(n))$$

Les deux fonctions f_1 et f_2, bien que égales en tous points, n'en sont pas égales pour autant. Cela a pour conséquence que l'ordre (point à point) sur les fonctions ne peut pas être antisymétrique.

Utilité : Nous pouvons désormais prouver des théorèmes très généraux sur les treillis : plus besoin de savoir si on considère les entiers naturels, relatifs, les fonctions, etc. Par exemple :

```
Theorem union_commutative :
  forall a b, equiv (union a b) (union b a).
```

2.2 Les connexions de Galois

Nous considérons l'implémentation suivante des connexions de Galois : une connexion de Galois est une paire de treillis qui vérifie la propriété donnée en 1.2.

```
Class Galois
  (* Types *)
  {X Y : Type}
  (* Structures sur X et Y *)
  (X_L : Lattice X)
  (Y_L : Lattice Y)
  (* Les treillis sur X et Y verifient bien les proprietes voulues *)
  (X_L_P : Wf_Lattice X_L)
  (Y_L_P : Wf_Lattice Y_L)
  (* Fonctions alpha et gamma *)
  (a : X → Y)
  (g : Y → X)
: Type :=
  {
    (* Propriete fondamentale *)
    galois : forall (x : X) (y : Y), le x (g y) ↔ le (a x) y
  }.
```

Là encore, cela permet démontrer des théorèmes sur les connections de Galois qui ignorent les treillis particuliers sur lesquels nous travaillerons. Un exemple : $\alpha \circ \gamma$ est réductive (propriété (3)), i.e :

$$\forall\, y \in E^{\#}, \quad \alpha \circ \gamma(y)\ \leq^{\#}\ y$$

Ce théorème, et sa preuve, se formalisent ainsi en Coq :

```
Theorem ag_reductive `{G : Galois} :
  forall (y : Y), le (a (g y)) y.
Proof.
  intros.       (* But : le (a (g y)) y *)
  apply galois. (* But : le (g y) (g y) *)
  apply le_refl. (* But resolu par reflexivite *)
Qed.
```

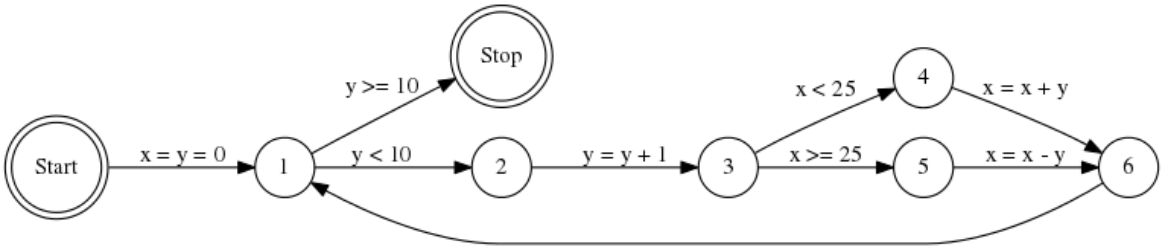
Avec cette preuve, on voit le fonctionnement de la surcharge des opérateurs : le terme `le` revient plusieurs fois, en s'appliquant à des objets différents ; les types des arguments permettent de savoir à quel ordre il est fait référence.

3 Construction d'un analyseur statique en Coq : interprétation avant des expressions

Pour tous ce qui se réfère à la construction d'un analyseur statique, nous nous référons à un document de Patrick Cousot, où toutes les étapes de la construction sont données [4]. Nous détaillons dans cette section certains des points de ce document que nous avons implémenté (l'implémentation est dans la section suivante).

Un analyseur statique se basant sur l'interprétation abstraite est fait de plusieurs parties. En premier lieu, il y a le choix de l'abstraction. On va détailler le reste ici.

Reprenons le code C donné en exemple précédemment, représenté cette fois sous la forme d'un graphe de flot de contrôle :



Les nœuds représentent différents points du programme ; les arrêtes sont de deux types : des instructions ou des gardes (des conditions). Il faut donc trouver un équivalent abstrait de ces deux points : instructions et gardes.

Prenons un exemple pour illustrer le type de raisonnements faits avec les instructions, toujours avec l'abstraction par signe : si au point *2* on sait que y est positif, alors après avec effectué l'instruction $y = y+1$ on sait qu'au point *3* y sera aussi positif ; de plus, il s'agit de l'information la plus précise qu'on puisse donner avec ce treillis abstrait. Pas contre, si on avait comme instruction $y = y - 1$, la seule chose qu'on aurait pu déduire aurait été que y est dans $\mathbf{Z}$

L'analyseur statique doit pouvoir automatiser ces raisonnements. Le choix de l'abstraction va permettre de régler la précision des raisonnements effectués.

3.1 Analyse par intervalles

Nous avons implémenté une analyse des entiers par les intervalles. On suppose que les entiers sont non bornés. Le domaine concret est donc $\wp(\mathbf{Z})$. Le domaine abstrait est l'ensemble des intervalles. Les intervalles ont leurs bornes finies ou infinies :

$$E^{\#} = \{\emptyset\} \cup \{[\![a; b]\!] \mid a \in \mathbf{Z},\, b \in \mathbf{Z},\, a \leq b\} \cup \{[\![-\infty; b \mid b \in \mathbf{Z}]\!]\} \cup \{[\![a; \infty \mid a \in \mathbf{Z}]\!]\} \cup \{[\![-\infty; \infty]\!]\}$$

3.2 Interprétation avant concrète des expressions

Examinons l'instruction suivante du code C présenté :

```
x = x + y ;
```

Connaissant les intervalles dans lesquels "vivent" x et y, que peut-on en déduire sur le nouvel intervalle où vit x ? Peut-on généraliser à des expressions plus complexes ? Est-on toujours optimal dans la détermination du nouvel intervalle, i.e peut-on calculer exactement l'intervalle d'arrivée ?

Répondons à ces question une par une.

L'interprétation avant d'une expression est une relation qui, étant donnée un ensemble de valuations des variables (des "environnements"), indique si une expression peut s'évaluer en un entier donné.

Mathématiquement, on écrira $\rho \vdash A \Rightarrow v$ si l'expression arithmétique A peut s'évaluer en v sous l'environnement ρ.

Pour une expression arithmétique A et un ensemble d'environnements R, l'interprétation avant est définie ainsi :

$$Faexp[\![A]\!]R = \{v \mid \exists \rho \in R,\ \rho \vdash A \Rightarrow v\}$$

L'interprétation avant est une notion concrète : il faut donc définir la notion abstraite associée, celle qui est calculable.

3.3 Interprétation avant abstraite des expressions

Nous voulons définir une interprétation "abstraite" des expressions : étant donnée une expression et un environnement abstrait (c'est-à-dire une valuation de l'ensemble des variables dans l'ensemble des intervalles), nous voulons obtenir l'intervalle dans lequel va s'évaluer l'expression. Ou au moins une sur-approximation : un intervalle trop grand, qui contient celui où peut s'évaluer l'expression.

L'interprétation avant d'une expression est un opérateur sur les ensembles d'environnements (les valuations). On cherche l'opérateur abstrait associé. La théorie de l'interprétation abstraite donne une spécification d'un tel opérateur. Si f est un opérateur concret et (α, γ) une connexion de Galois, l'opérateur abstrait qui l'approxime le mieux s'exprime par $f^{\#} = \alpha \circ f \circ \gamma$.

On note γ' la fonction qui applique γ point à point (et qui transforme donc une valuation abstraite en un ensemble de valuations concrètes). On cherche à définir $Faexp^{\#}$ tel que, pour A une expression, pour R une valuation abstraite, c'est-à-dire une fonction des variables dans les intervalles (c'est l'information abstraite qu'on a sur les données du programme), on veut :

$$\forall A,\ \forall R, \quad \alpha\left(Faexp[\![A]\!](\gamma'(R))\right) \sqsubseteq Faexp^{\#}[\![A]\!](R)$$

Cette expression traduit le fait que l'interprétation avant abstraite d'une expression est une sur-approximation de la notion concrète associée.

Pour pouvoir définir une telle fonction, il faut procéder par induction sur la forme de l'expression A. Voyons un exemple avec la somme, puisque dans l'exemple donné plus tôt est effectuée la somme de deux variables. En interprétation abstraite, on déporte les calculs dans un environnement abstrait. Il faut donc ici définir un opérateur $+^{\#}$ qui mime ce qui se passerait dans le monde concret : si $x_1 \in I_1$, si $x_2 \in I_2$, alors $x_1 + x_2 \in I_1 +^{\#} I2$. Ici, c'est-à-dire avec les intervalles, nous posons :

$$\forall (a_1; a_2; b_1; b_2) \in \overline{\mathbf{Z}}^4, \quad [\![a_1; b_1]\!] +^{\#} [\![a_2; b_2]\!] = [\![a_1 + a_2; b_1 + b_2]\!]$$

Remarque : Il faut définir les opérations sur $\overline{\mathbf{Z}}$. Cela pose quelques problèmes, comme : "que vaut $-\infty + \infty$? ". On peut évacuer de tels problèmes en disant qu'ils n'apparaissent pas dans le cas d'intervalles où les bornes sont "bien formées".

Pour pouvoir analyser des expressions plus poussées, il suffit de définir les approximations de tout ce qui peut apparaître dans une expression : des soustractions, des constantes, des multiplications, des nombres aléatoires, etc.

3.4 Problèmes de précision

Le choix de la façon dont on abstrait les composants d'une expression est crucial, au sens où il régit la précision des informations obtenues. Par exemple, la façon dont nous avons défini l'addition est optimale ; en notant $\mathbf{I}$ l'ensemble des intervalles, on a :

$$\begin{cases} (1) \quad \forall (i_1; i_2) \in \mathbf{I}^2,\ \forall (x_1; x_2) \in \mathbf{Z}^2, \quad x_1 \in i_1 \ \wedge x_2 \in i_2 \Rightarrow x_1 + x_2 \in i_1 +^{\#} i_2 \\ \qquad \text{(l'intervalle somme respecte la somme)} \\ (2) \quad \forall (i_1; i_2) \in \mathbf{I}^2,\ \forall (x) \in \mathbf{Z}^2, \\ \qquad x \in i_1 +^{\#} i_2 \Rightarrow \exists (x_1; x_2) \in \mathbf{Z}^2, \quad x = x_1 + x_2 \wedge x_1 \in i_1 \wedge x_2 \in i_2 \\ \qquad \text{(l'intervalle somme correspond à la somme ensembliste)} \end{cases}$$

Mais rien ne nous a empêché de définir la somme de deux intervalles comme étant toujours $[\![-\infty; \infty]\!]$: un tel intervalle est bel et bien compatible avec la somme, il est juste moins précis.

Dans le cas de la somme, il est possible d'en avoir une abstraction optimale. Cependant, pour certaines combinaisons de monde abstrait de d'opérateurs, il peut ne pas en exister : par exemple, l'abstraction par intervalles et la multiplication. C'est un des sources de perte de précision.

Remarque : le mot optimal est vague. Le sens que nous lui accordons est "qui n'entraîne pas de perte de précision". Il se trouve que la somme définie comme nous l'avons proposé n'est pas exactement "optimale", puisqu'elle ne tient pas compte des relations entre les variables. Par exemple, si on sait que X est dans $[\![0; 1]\!]$, on a $-X$ dans $[\![-1; 0]\!]$. La façon dont nous avons défini la somme nous dit alors que $X + (-X)$ est dans $[\![-1; 1]\!]$, ce qui est vrai, mais insuffisant. Il existe de domaines abstraits qui permettent de résoudre partiellement ce genre de problèmes. Ce n'est cependant pas de cela dont nous allons discuter dans la suite.

4 Formalisation en Coq de l'interprétation abstraite d'expressions

Nous détaillons ici la façon dont nous avons implémenté ce qui a été décrit dans la section précédente.

4.1 Les intervalles

Puisque les bornes des intervalles peuvent être infinies, nous avons commencé par étendre $\mathbf{Z}$ en $\overline{\mathbf{Z}}$ en y ajoutant les infinis. En Coq cela donne :

```
Inductive Zf : Type :=
  | Zf_min
  | Zf_Z (z : Z)
  | Zf_max
```

Nous avons décidé de représenter les intervalles par des couples d'éléments de $\overline{\mathbf{Z}}$. C'est une définition simple, mais elle apporte un souci : il y a plusieurs représentants de l'intervalle vide. Par exemple, $[\![1; -1]\!]$ et $[\![-\infty; -\infty]\!]$ représentent tous deux l'intervalle vide. En réponse, il a fallu créer une fonction prenant deux éléments de $\overline{\mathbf{Z}}$ et nous disant si ils peuvent correspondre aux bornes d'un intervalle non vide. Cette fonction est :

```
Definition represent_bottom (z1 z2 : Zf) : Prop :=
  match z1 with
  | Zf_min ⇒ match z2 with
             | Zf_min ⇒ True
             | _ ⇒ False
             end
  | Zf_Z x1 ⇒ match z2 with
              | Zf_min ⇒ True
              | Zf_Z x2 ⇒ (x2 < x1)
              | Zf_max ⇒ False
              end
  | Zf_max ⇒ True
  end.
```

Ainsi, un intervalle sera une simple paire d'éléments de $\overline{\mathbf{Z}}$; cependant, à chaque fois qu'on travaillera dessus, on regardera si les bornes sont bien agencées pour agir en conséquence. Voilà comment nous avons implémenté les intervalles, leur ordre (inclusion) et leur égalité :

```
(* Definition des intervalles. *)
Inductive int : Type :=
  | int_pair (z1 z2 : Zf).

(* Relation d'ordre sur les intervalles (inclusion). *)
Inductive int_le : int → int → Prop :=
  | int_le_bottom : forall (a b : Zf),
                    represent_bottom a b →
                    forall (i : int), int_le (int_pair a b) i
  | int_le_pair : forall (a1 a2 b1 b2 : Zf),
                  ¬ represent_bottom a1 b1 →
                  ¬ represent_bottom a2 b2 →
                  a2 <= a1 → b1 <= b2 →
                  int_le (int_pair a1 b1) (int_pair a2 b2).

(* Egalite sur les intervalles. *)
Inductive int_eq : int → int → Prop :=
  | int_eq_bottom : forall (a1 a2 b1 b2 : Zf),
                    represent_bottom a1 b1 →
```

```
                represent_bottom a2 b2 →
                int_eq (int_pair a1 b1) (int_pair a2 b2)
| int_eq_pair : forall (a b : Zf),
              int_eq (int_pair a b) (int_pair a b).
```

On définit aussi une union et une intersection sur les intervalles, et on montre qu'on a bien une structure de treillis (on ne détaille pas pour des raisons de place).

4.2 Les expressions

Nous définissons les expressions arithmétiques de la façon suivante en Coq :

```
Inductive expr : Type :=
  | c_expr (z : Z)
  | var_expr (n : nat)
  | sum_expr (exp1 exp2 : expr)
  | rand_expr.
```

Autrement dit, une expression est soit une constante, soit une variable (chaque variable est associée à un élément de type *nat*), soit la somme de deux expressions, soit un nombre aléatoire (ce cas particulier sert à gérer des entrées non déterministes, telles des nombres aléatoires ou des entrées d'un utilisateur). Bien sûr, il sera possible de rajouter d'autres opérateurs par la suite.

4.3 L'interprétation avant concrète

Informellement, on a :

- une expression constante peut s'évaluer en la constante qu'elle représente ;
- une expression de type variable peut s'évaluer en ce que l'environnement lui dicte ;
- la somme de deux expressions peut s'évaluer en n'importe quelle somme d'évaluation des deux expressions qui la compose ;
- une expression aléatoire peut s'évaluer en n'importe quel entier.

Nous avons défini l'opérateur ternaire $. \vdash . \Rightarrow .$ ainsi :

```
Fixpoint interp (v : env) (e:expr) : Z_ens :=
  match e with
  | rand_expr ⇒ fun _ ⇒ True
  | c_expr c ⇒ fun z ⇒ z = c
  | sum_expr exp1 exp2 ⇒ fun z ⇒ exists z1, exists z2,
        (interp v exp1 z1) ∧ (interp v exp2 z2) ∧ (z = z1 + z2)
  | var_expr n ⇒ fun z ⇒ z = v n
  end.
```

Ensuite nous définissons l'interprétation avant d'une expression :

```
Definition Faexp (exp : expr) (A : env_ens) : Z_ens :=
  fun (x : Z) ⇒ exists e, (A e ∧ interp e exp x).
```

4.4 L'interprétation avant abstraite

On rappelle l'inégalité que dois vérifier l'interprétation avant abstraite :

$$\forall A, \forall R, \quad \alpha\left(Faexp[\![A]\!](\gamma'(R))\right) \sqsubseteq Faexp^{\#}[\![A]\!](R)$$

Pour définir $Faexp^{\#}$, on procède par induction sur la forme de l'expression A. Pour chaque cas, on part du membre de gauche de l'inégalité, $\alpha\left(Faexp[\![A]\!](\gamma'(R))\right)$, et par une suite de dérivations on essaie de faire apparaître un intervalle explicite I : on pose alors $Faexp^{\#}[\![A]\!](R) = I$ pour la

forme de l'expression donnée. Par "suite de dérivations", on entend une suite de transformations de l'objet sur lequel on travaille ; il peut s'agir de déplier des définitions ou bien de sur-approximer l'objet en question.

L'objectif est de pouvoir effectuer cette suite de dérivations à l'intérieur même de Coq pour en vérifier toutes les étapes, puis d'en extraire le code OCaml correspondant. On aura alors construit (une partie de) l'analyseur et vérifié en même temps.

Exemple : avec une expression aléatoire, notée " ?", et toujours l'abstraction par signes : On suppose que R est non vide. Alors :

$$
\begin{array}{ll}
 & \alpha\left(Faexp[\![?]\!](\gamma'(R))\right) \\
= & \alpha(\{v \mid \exists \rho \in \gamma'(R), \quad \rho \vdash ? \Rightarrow v\}) \\
= & \text{(par définition de } . \vdash . \Rightarrow .) \\
 & \alpha(\{v \mid v \in \mathbf{Z}\}) \\
= & \text{(ce terme est calculable)} \\
 & \mathbf{Z} \\
= & \text{(en définissant } Faexp^{\#}[\![?]\!](R) = \mathbf{Z}) \\
 & Faexp^{\#}[\![?]\!](R)
\end{array}
$$

Pour construire l'analyseur, on reproduit ces raisonnements en Coq (et le logiciel s'assure alors qu'ils sont corrects). Ayant défini l'interprétation avant abstraite en Coq, on utilise le mécanisme d'extraction pour obtenir son code Ocaml. Le code que nous obtenons est le suivant :

```
let interpa exp v =
  match enva_bottom_dec v with
  | Left ->
    let rec f = function
    | Rand_expr -> Int_pair (Zf_min, Zf_max)
    | C_expr z0 -> Int_pair ((Zf_Z z0), (Zf_Z z0))
    | Sum_expr (e0, e1) -> int_sum (f e0) (f e1)
    | Var_expr n0 -> er_env v n0
    in f exp
  | Right -> Int_pair (Zf_max, Zf_min)
```

Remarques :

- `interpa` est le nom qu'on a donné à $Faexp^{\#}$ à l'intérieur du code ;
- `enva_bottom_dec` décide si l'environnement abstrait est sain.

5 Conclusion

Dans ce travail, nous avons abordé la construction d'un analyseur statique en Coq. Nous nous sommes concentrés sur une abstraction particulière (les intervalles), et n'avons étudié qu'une seule partie de l'analyseur (l'interprétation avant des expressions). L'objectif final est de pouvoir construire un analyseur correct.

Faire de l'abstraction abstraite au sein d'un assistant de preuve pour pouvoir construire un analyseur correct est une idée qui a déjà été avancée et partiellement mise en œuvre [3]. Mais jusqu'ici, la théorie des connexions de Galois n'était pas entièrement utilisée : seules des concrétisations étaient prises en compte.

Pour l'instant, nous avons exploré un exemple d'abstraction et une partie de l'analyseur (l'interprétation avant des expressions). L'objectif à plus long terme est d'automatiser, au moins partiellement, la construction de l'analyseur pour pouvoir le rendre plus générique (utilisation d'autres abstractions).

Références

[1] URL : https://coq.inria.fr/.

[2] URL : https://isabelle.in.tum.de/.

[3] David Cachera et David Pichardie. « A certified denotational abstract interpreter ». In : *Interactive Theorem Proving*. Springer, 2010, p. 9–24. URL : http://www.irisa.fr/celtique/pichardie/papers/itp10.pdf.

[4] Patrick Cousot. « The calculational design of a generic abstract interpreter ». In : *NATO ASI SERIES F COMPUTER AND SYSTEMS SCIENCES* 173 (1999), p. 421–506.

[5] Georges Gonthier et al. « A Machine-Checked Proof of the Odd Order Theorem ». In : *ITP 2013, 4th Conference on Interactive Theorem Proving*. Sous la dir. de Sandrine Blazy, Christine Paulin et David Pichardie. T. 7998. LNCS. Rennes, France : Springer, juil. 2013, p. 163–179. DOI : 10.1007/978-3-642-39634-2_14. URL : https://hal.inria.fr/hal-00816699.

[6] Benjamin Werner. « Une Théorie des Constructions Inductives ». Theses. Université Paris-Diderot - Paris VII, mai 1994. URL : https://tel.archives-ouvertes.fr/tel-00196524.