Analyse par intervalles

Eric Goubault

`Eric.Goubault@cea.fr`

1 Préambule

L'objet de ce projet est d'implémenter une méthode classique [2] permettant d'analyser ce qui peut se passer à l'exécution d'un programme impératif simple. Ce que l'on entend par là est d'être capable de déterminer automatiquement, à la lecture du code source du programme analysé, un sur-ensemble des valeurs que peuvent prendre les variables, et ceci à chaque ligne du programme. Cela permet ensuite de vérifier le programme, en particulier de voir si les valeurs attendues du programme sont bien celles trouvées par l'analyseur, ou cela peut faire partie d'un optimiseur de code (en particulier pour la parallélisation) lors de la compilation du programme. L'idée est d'utiliser une représentation approximée mais correcte [3, 1] des valeurs possibles des variables, et d'inférer automatiquement celles-ci par des moyens calculatoires simples dans tout un programme, sans l'exécuter: on a alors une certitude sur le programme dans tous les cas d'exécution possible. De telles techniques d'«interprétation abstraite» sont utilisées dans des compilateurs ainsi que dans des outils de vérification automatique de logiciels (Polyspace Verifier par exemple, utilisé en particulier pour la vérification de certains programmes d'Ariane V, après la catastrophe du vol 501).

Dans la suite de ces notes, on présente d'abord la syntaxe des programmes à analyser (section 2). On continue en donnant une approximation possible des valeurs des variables par des intervalles de valeurs (section 3), détaillant les opérations élémentaires que l'on peut effectuer avec cette approximation. Dans la section 4, on donne les algorithmes permettant d'approximer l'effet de chaque instruction du langage sur les variables, exprimé en termes d'intervalles entre les valeurs de celles-ci. C'est ce que l'on appelle une sémantique abstraite du langage. Enfin, on passe à l'explication de la façon pratique de réaliser le projet dans la section 5.

2 Syntaxe du Langage

On décrit ici un langage impératif simple dont les instructions agissent en modifiant les valeurs de variables entières d'un ensemble de variables *Var*.

Les expressions arithmétiques sur ces variables sont,

$$\begin{array}{lll} \mathit{Expr} & = & \texttt{cste} & \text{expression constante entière} \\ & & X & \text{variable } X \in \mathit{Var} \\ & & \mathit{Expr} + \mathit{Expr} & \text{somme} \\ & & \mathit{Expr} * \mathit{Expr} & \text{produit} \\ & & \mathit{Expr} - \mathit{Expr} & \text{différence} \\ & & \mathit{Expr}\ /\ \mathit{Expr} & \text{division} \\ & & (\mathit{Expr}) & \text{parenthésage} \end{array}$$

Les tests sur les expressions arithmétiques sont,

$$\begin{array}{lcll} \mathit{Test} & = & \mathit{Expr} == 0 & \text{test de nullité} \\ & & \mathit{Expr} > 0 & \text{test de positivité stricte} \\ & & \mathit{Expr} \geq 0 & \text{test de positivité} \end{array}$$

Puis les instructions élémentaires du langage sont,

$$\begin{array}{lcll} \mathit{Instr} & = & X \texttt{ = } \mathit{Expr} & \text{affectation pour } X \in \mathit{Var} \\ & & \texttt{if } \mathit{Test} \texttt{ then } \mathit{Bloc} \texttt{ else } \mathit{Bloc} & \text{branchement conditionnel} \\ & & \texttt{while } (\mathit{Test})\ \mathit{Bloc} & \text{boucle while} \end{array}$$

où un bloc d'instruction est défini par,

$$\begin{array}{lcll} \mathit{Bloc} & = & \emptyset & \text{bloc vide} \\ & & \mathit{Instr}\,;\ \mathit{Bloc} & \text{concaténation} \end{array}$$

Enfin un programme *Prog* est une suite de blocs $\mathit{Bloc}\,;\ \mathit{Bloc}\,;\ \cdots\,;\ \mathit{Bloc}$.

On suppose par la suite que des étiquettes sont associées par défaut aux instructions dans un programme comme suit,

- une étiquette est un entier. Tout programme commence par l'étiquette (0).
- chaque instruction *Instr* de *Prog* qui n'est pas un `if` a une étiquette gauche (l) et une étiquette droite (r) et l'on note (l) *Instr* (r). Par défaut on pose $r = l + 1$.
- dans le cas d'un bloc, $\mathit{Instr}\,;\ \mathit{Bloc}$, l'étiquette gauche de la première instruction de *Bloc* (appelée étiquette gauche de *Bloc* en bref) est égale à l'étiquette droite de *Instr*. L'étiquette droite de la dernière instruction de *Bloc* est appelée en bref l'étiquette droite de *Bloc*.
- dans le cas de l'instruction composée $\mathit{Instr} \equiv \texttt{if } \mathit{Test} \texttt{ then } \mathit{Bloc}_1 \texttt{ else } \mathit{Bloc}_2$ l'étiquette gauche de Bloc_1 est l'étiquette gauche de l'instruction *Instr* plus un. L'étiquette gauche de Bloc_2 est l'étiquette droite de Bloc_1 plus un. L'étiquette droite de *Instr* est égale à l'étiquette droite de Bloc_2 plus un.
- dans le cas d'une boucle $\mathit{Instr} \equiv \texttt{while } (\mathit{Test})\ \mathit{Bloc}$: l'étiquette gauche de *Bloc* est l'étiquette gauche de *Instr* plus un. L'étiquette droite de *Instr* est l'étiquette droite de *Bloc* plus un.

On se reportera à la section 4.3 pour un exemple.

3 Domaine des intervalles

On suppose ici que $\mathit{Var} = \{x_1, \cdots, x_N\}$. On appelle domaine des intervalles

$$\mathcal{I} = \{[a_1, b_1] \times \ldots \times [a_N, b_N] \mid a_i, b_i \in \mathbb{Z} \cup \{-\infty, +\infty\}\}$$

Le domaine des intervalles abstrait les valeurs que peuvent prendre les variables de *Var* (considérées comme un élément de $\mathbb{Z}^N$), de manière évidente: soit

$$\alpha : \wp(\mathbb{Z}^N) \to \mathcal{I}$$

avec $\alpha(S) = [\min \pi_1(S), \max \pi_2(S)] \times \ldots \times [\min \pi_N(S), \max \pi_N(S)]$ où $\pi_i : \mathbb{Z}^N \to \mathbb{Z}$ est la $i$ème projection sur les entiers. Cette fonction est appelée fonction d'abstraction, elle décrit la perte d'information que l'on se permet, en représentant ici les ensembles de valeurs dans $\mathbb{Z}^N$ par des hypercubes de dimension N.

L'inclusion d'ensemble de n-uplets induit un ordre sur les éléments de $\mathcal{I}$. Il se trouve que cet ordre, noté simplement $\leq$, fait de $\mathcal{I}$ un treillis complet, c'est à dire que,

- étant donné deux élements X et Y de $\mathcal{I}$, il existe un plus petit majorant $x \cup Y$ et un plus grand minorant $X \cap Y$.

- le treillis est complet c'est à dire qu'il existe un plus petit majorant de tout sous-ensemble (fini ou pas) de $\mathcal{I}$.
- cela implique également que l'on a un plus petit élément dans le treillis des intervalles qui correspond à un ensemble vide de valeurs (on le note $\perp$) et que l'on a un plus grand élément (on le note $\top$) qui correspond à tout $\mathbb{R}^N$, c'est à dire que les variables peuvent prendre toutes les valeurs possibles.

4 Sémantique abstraite

4.1 Système d'équations sémantiques

On va écrire la sémantique d'un programme dans le domaine des intervalles, comme un système d'équations de point fixe (on indiquera comment le résoudre à la section 4.2) dans $\mathcal{I}$, une équation par étiquette.

Commençons par l'affectation $X = \mathit{Expr}$. On a une valeur abstraite $A \in \mathcal{I}$ et on cherche à trouver la plus petite valeur abstraite $B \in \mathcal{I}$ vraie après cette affectation, dans un contexte validant A. On note $B = [\![X = \mathit{Expr}]\!]A$: ce calcul peut se faire simplement par induction sur la syntaxe de Expr, en appliquant des règles de calcul simples sur les intervalles, à savoir, par exemple:

- $[a, b] + [c, d] = [a + b, c + d]$
- $[a, b] * [c, d] = [\min(a * c, a * d, b * c, b * d), \max(a * c, a * d, b * c, b * d)]$
  (on peut trouver des formules légèrement plus économiques)

On passe maintenant aux tests. On veut pouvoir interpréter Test comme une approximation supérieure de l'élément de $\wp(\mathbb{R}^n)$ tel que Test soit vrai sur ses points. On écrit cela $[\![\mathit{Test}]\!]A$ (où A l'environnement abstrait dans lequel on interprète le test). On interprète également Test de telle façon à avoir une approximation supérieure des valeurs pour lesquelles Test est faux dans l'environnement A. On représente cela par $[\![\neg \mathit{Test}]\!]A$. On peut calculer ces nouveaux intervalles (ou produits d'intervalles) par une «résolution de contraintes», s'effectuant également par induction sur la syntaxe de Test. Les étapes élémentaires sont:

- supposons $\mathit{Test} \equiv \mathit{Expr}$`==0`) (ou Expr`>0`, etc.), où l'arbre syntaxique représentant Expr commence par un noeud `+`, par exemple. On va résoudre récursivement des tests de la forme $\mathit{Expr} \in [a, b]$. Appelons les fils de Expr, X et Y respectivement. Alors résoudre $X + Y \in [a, b]$ dans l'environnement A revient à essayer de raffiner $[\![X]\!]A$ (l'intervalle représentant la valeur de X dans l'environnement A) par la connaissance de la contrainte $X + Y \in [a, b]$. On pose donc $X^1 = [\![X]\!]A \cap ([a, b] - [\![Y]\!]A)$ et $Y^1 = [\![Y]\!]A \cap ([a, b] - [\![X]\!]A)$ (on peut éventuellement itérer ce procédé et vérifier que l'on a atteint un point fixe), et le problème revient donc maintenant à résoudre les deux contraintes, dans l'environnement A: $X \in X^1$ et $Y \in Y^1$ (induction).
- le cas terminal est celui où l'expression est une constante ou une variable:
  - $[\![C \in [a, b]]\!]A$ vaut l'environnement abstrait $\perp$ si $C \notin [a, b]$ et A si $C \in [a, b]$.
  - $[\![x \in [a, b]]\!]A$ vaut A où l'intervalle représentant la variable x dans A est remplacé par l'intervalle $[a, b]$.
- Pour $[\![\neg \mathit{Expr} == 0]\!]A$, le principe est le même, si ce n'est que l'on résoud en même temps $[\![\mathit{Expr} \in [-\infty, -1]]\!]A$ et $[\![\mathit{Expr} \in [1, \infty]]\!]A$, dont on fait l'union.

On considère maintenant l'instruction composée

$$(l)\texttt{if } \mathit{Test} \texttt{ then } (l + 1)\mathit{Bloc}_1(l') \texttt{ else } (l' + 1)\mathit{Bloc}_2(r)(r + 1)$$

L'état abstrait en (l+1), que l'on écrit X_{l+1} est $X_{l+1} = [\![\mathit{Test}]\!]X_l$. Alors $X_{l'} = [\![\mathit{Bloc}_1]\!]X_{l+1}$, $X_{l'+1} = [\![\neg \mathit{Test}]\!]A$, $X_r = [\![\mathit{Bloc}_2]\!]X_{l'+1}$ and $X_{r+1} = X_{l'} \cup X_r$.

Les blocs d'instructions sont interprétés de façon itérative simple. Si $Bloc = (l)Instr;\ (l+1)Bloc'(r)$ alors on pose $X_{l+1} = [\![Instr]\!]X_l$ et $X_r = [\![Bloc']\!]X_{l+1}$.

Enfin, pour les boucles `while`, du style (l) `while` $(Test)$ $(l+1)Bloc(r)(r+1)$, on a les équations suivantes:

- $X_{l+1} = [\![Test]\!](X_l \cup X_r)$,
- $X_r = [\![Bloc]\!]X_{l+1}$,
- $X_{r+1} = [\![\neg Test]\!](X_l \cup X_r)$.

4.2 Résolution du système d'équations

Quand on a une fonction croissante f sur un treillis complet, on a au moins une solution à l'équation de point fixe $x = f(x)$, en particulier on a une plus petite (lfp f) une plus grande (gfp f).

Si f est de plus continue lfp $f = \bigcup_{i \in N} f^i(\perp)$, et c'est ce que l'on supposera ici. La fonction f ici est simplement la fonctionnelle de $\mathcal{I}^k$ vers $\mathcal{I}^k$ (k est le nombre d'étiquettes du programme) qui décrit le second membre du système d'équations construit à la section 4.1.

4.3 Exemple de calcul

Soit le programme,

```
(0) I=1;
 (1) while (I<=100)
  (2) I=I+1; (3)
(4)
```

La composante de fonctionelle associée à l'étiquette (2) est

$$f(X) = ([1,1] \cup (X \oplus [1,1])) \cap [-\infty, 100]$$

Donc les itérés sont:

$$\begin{array}{ccc} X^0 & = & \perp \\ X^1 & = & [1,1] \\ X^2 & = & [1,2] \\ \dots & & \\ X^{100} & = & [1,100] \\ X^{101} & = X^{100} & \end{array}$$

On a donc trouvé l'invariant de boucle $I \in [1,100]$, puis on trouve le résultat du programme $I = [101,101]$.

4.4 Elargissement/rétrécissement

Ce procédé de calcul de plus petit point fixe peut être extrèmement long; il peut même être infiniment long!

Il y a des moyens d'extrapoler les valeurs de l'union dans l'itération (ce que l'on fait généralement après un petit nombre d'itérations classiques, avec des unions), et ce de façon à ce que l'itération soit toujours finie. C'est ce que l'on appelle un élargissement. Par exemple, on a l'élargissement classique dans les intervalles:

$$\begin{array}{ll} [a_1,b_1]W[a_2,b_2] = [& \text{si } a_2 < a_1 \text{ alors } -\infty \text{ sinon } a_1, \\ & \text{si } b_2 > b_1 \text{ alors } +\infty \text{ sinon } b_1] \end{array}$$

Donc les itérés pour f de la section 4.3 sont:

$$\begin{array}{lcl} X^0 & = & \perp \\ X^1 & = & [1,1] \\ X^2 & = & [1,+\infty] \\ X^3 & = & X^2 \end{array}$$

On a atteint une surapproximation du plus petit point fixe. Il y a en fait souvent moyen de «redescendre» en un temps fini vers un point fixe, éventuellement plus grand que le plus petit point fixe. Cela peut se faire en faisant des calculs itératifs de la fonctionnelle f, cette fois-ci en les combinant avec un opérateur de rétrécissement. Un rétrécissement classique dans les intervalles est:

$$\begin{array}{ll} [a_1,b_1] M [a_2,b_2] = [& \text{si } a_1 = -\infty \text{ alors } a_2 \text{ sinon } \min(a_1,a_2), \\ & \text{si } b_1 = +\infty \text{ alors } b_2 \text{ sinon } \max(b_1,b_2)] \end{array}$$

Les itérées successives sont donc maintenant:

$$\begin{array}{lcl} Y^0 & = & X^2 \\ Y^1 & = & [1,100] \\ Y^2 & = & Y^1 \end{array}$$

On a donc trouvé le même point fixe, cette fois-ci beaucoup plus rapidement.

5 Réalisation du projet

On commencera par écrire un petit parser des expressions du langage considéré. Ensuite, on écrira la représentation des éléments de $\mathcal{I}$ ainsi que les opérations élémentaires de treillis. Pour ce faire, on pourra compléter la liste des opérations comprises par l'analyseur, et donner les règles sémantiques adéquates.

Enfin, on fera un simple calcul de point fixe en parcourant (dans l'ordre d'exécution dans le cadre de ce projet) l'arbre généré par la phase d'analyse syntaxique.

On essaiera ensuite de programmer les phases d'élargissement et de rétrécissement; on pourra également proposer d'autres opérateurs d'élargissement et de rétrécissement.

Pour aller plus loin, il y a de nombreux aspects qui peuvent être creusés (on se reportera à la bibliographie pour avoir plus de détails et on pourra contacter E. Goubault pour tout renseignement supplémentaire), par exemple:

- Comment améliorer la précision de l'analyse (avec ou sans assertions)? (par exemple en rajoutant une sémantique abstraite «arrière»)
- Comment faire pour stocker plus efficacement les valeurs abstraites (moins de consommation mémoire)?
- Comment analyser notre langage impératif, si on rajoute une notion de fonction, et d'appel de fonction?
- etc.

Références

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points. *Principles of Programming Languages 4*, pages 238–252, 1977.

[2] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[3] D. A. Schmidt. Trace-based abstract interpretation of operational semantics, 1998.