Grenoble INP – ENSIMAG
École Nationale Supérieure d'Informatique et de Mathématiques Appliquées

Travaux d'Études et de Recherche

Analyse de programmes par interprétation abstraite

Marc PEGON
marc.pegon@ensimag.imag.fr

Encadrants

David MONNIAUX

Matthieu MOY

11 Mai 2011

Remerciements

Ce travail a été réalisé dans le cadre d'un TER (Travaux d'Etudes et de Recherche) au sein du laboratoire Verimag.

Je tiens à remercier tout particulièrement mes tuteurs Matthieu Moy et David Monniaux, chercheurs à Verimag, pour leurs précieux conseils et le temps qu'ils ont bien voulu m'accorder tout au long de ce semestre, ainsi que pour leur aide à la rédaction de cet article.

Résumé

L'interprétation abstraite est une méthode d'analyse statique utilisée dans plusieurs outils industriels d'analyse de programmes. Dans ce document, nous montrons comment il est possible de réaliser très rapidement un analyseur de programmes C par interprétation abstraite à l'aide d'outils biens choisis – les bibliothèques LLVM et Apron. Après un bref rappel sur l'interprétation abstraite, nous présentons ces outils, puis nous donnons les détails d'implémentation de notre analyseur. Enfin nous présentons les résultats obtenus sur des programmes simples.

1 Introduction

Les logiciels informatiques étant de plus en plus complexes, il est d'autant plus difficile d'en assurer la qualité. De plus, dans certains secteurs industriels comme l'avionique, certains logiciels sont intégrés dans des systèmes critiques, et doivent donc être particulièrement fiables.

Il existe différentes méthodes de vérification de programmes, dont fait partie l'interprétation abstraite. Introduite pour la première fois en 1977 par Patrick Cousot, l'interprétation abstraite [1][3] fait partie des méthodes d'analyse statique, qui permettent d'obtenir des informations sur un programme sans l'exécuter.

On cherche à prouver automatiquement des propriétés de sûreté sur les programmes, c'est-à-dire informellement des propriétés indiquant que « quelque chose de mauvais ne peut jamais se produire ». Par exemple « à la ligne 10 du programme, la variable x n'est jamais nulle » est une propriété de sûreté. Pour prouver qu'un programme ne viole pas cette propriété, il suffit de vérifier qu'aucune trace d'exécution ne permet d'atteindre l'état « à la ligne 10, x = 0 ».

Une possibilité est de calculer en chaque point du programme l'ensemble des valeurs prises par ses variables. Mais il a été montré que ce problème était dans le cas général trop complexe (indécidable). Pour contourner ce problème, l'interprétation abstraite se base sur une sur-approximation (abstraction) des états du programme.

Etant donné que cela engendre inévitablement une perte de précision, le point délicat est de garder suffisamment d'information pour permettre de déduire des propriétés de sûreté sur le programme. Par exemple, si en un point du programme on a pu déterminer qu'une variable x était strictement supérieure à 1, alors on sait qu'en ce même point on peut effectuer une division par x sans risquer une division par 0 : pour écarter ce cas d'erreur, il n'est pas nécessaire de connaître exactement la valeur de x.

Différents outils d'analyse de programmes utilisés dans l'industrie sont basés sur l'interprétation abstraite (mais combinent également d'autres méthodes d'analyse), comme Astrée [2] ou PolySpace.

Ma contribution est d'avoir écrit en C/C++ un analyseur simple de programmes C basé sur l'interprétation abstraite, à l'aide des outils de compilation de LLVM et de la bibliothèque numérique Apron.

LLVM [5] est un projet regroupant un certain nombre d'outils de compilation, basé sur une architecture très modulaire permettant facilement l'ajout d'étapes d'optimisation et, dans une certaine mesure, d'analyse. Typiquement, un compilateur écrit nativement avec LLVM, tel que Clang, procèdera en 3 étapes : d'abord, il convertira le code C/C++/Objective-C en une forme intermédiaire (IR) bien spécifiée, puis il modifiera cette représentation par des passes d'analyse et d'optimisation, et enfin il génèrera un exécutable. Dans le cas de notre analyseur, nous utilisons le compilateur Clang uniquement comme *front-end* pour obtenir la représentation intermédiaire d'un programme écrit en C (*bitcode* LLVM), et nous nous servons des bibliothèques fournies par LLVM

pour la manipuler et analyser le programme. Notre outil ne traite donc pas directement du C, mais du *bitcode* LLVM obtenu à partir de code source C.

Apron [4] est une bibliothèque écrite en C faisant l'interface entre différentes librairies de calcul numérique utilisées en particulier pour l'interprétation abstraite. Elle permet entre autres d'effectuer du calcul sur les intervalles et les polyèdres convexes, dont nous verrons l'utilité plus loin.

LLVM et Apron étant respectivement en C++ et en C, notre analyseur est écrit en C++.

Dans cet article, nous commencerons par faire un bref rappel sur la théorie de l'interprétation abstraite (section 2), puis nous évoquerons plus en détail la librairie LLVM (section 3), et en particulier la représentation interne qu'elle fournit. Nous reviendrons ensuite sur la librairie Apron (section 4). Enfin, nous exposerons avec plus de précision la conception de notre analyseur (section 5), et nous présenterons les résultats obtenus en illustrant par des exemples (section 6).

2 Interprétation abstraite

2.1 Calcul de point fixe

Dans un premier temps, nous voyons comment la recherche de bugs dans un programme peut se réduire à la résolution d'équations de point fixe. Pour cela, nous devons définir les notions d'état et de fonction de transition d'un programme.

Un programme peut être vu comme un triplet (Q, τ, Q_{init}), où Q est un ensemble d'états, Q_{init} l'ensemble des états initiaux, et τ une relation de transition définissant pour chaque état l'ensemble de ses successeurs.

A présent, supposons que notre programme compte un nombre fini, fixe de variables numériques entières $x_1, .., x_n$, ainsi qu'un nombre fini de points de contrôle (un point de contrôle est de manière informelle un « endroit » du code du programme). Alors on définit un état comme un $(n+1)-uplet$ $(P, v_1, .., v_n)$ où P désigne un point de contrôle et v_i la valeur de la variable x_i. Un état du programme est donc caractérisé par un point de contrôle du programme et la valeur de ses variables en ce point.

Exemple 1 Soit le programme C suivant

```
1  int main(int argc, char *argv[])
2  {
3     int i = 0;
4     int j = 0;
5     // (A)
6
7     i = 2;
8     // (B)
9     while (i+j <= 20) {
10       // (C)
11       if (i >= 10) {
12          i += 4;
13          // (D)
14       } else if (j <= 4) {
15          i += 2;
16          j++;
```

```
17            // (E)
18        }
19        // (F)
20    }
21    // (G)
22
23    assert(j-i != 0)
24
25    return EXIT_SUCESS;
26 }
```

On peut distinguer 7 points de contrôles $A, ..., G$, mais d'autres séparations sont possibles, on aurait pu définir un point de contrôle après chaque instruction. L'état initial serait ici $Q_{init} = (A, 0, 0, 0)$, et la relation de transition est donnée par la sémantique des instructions. Ici, Q_{init} a un seul successeur, l'état $(B, 2, 0, 0)$.

Si on veut montrer que le *assert* après le point de contrôle G est vrai, il suffit de montrer que quels que soient k, l entiers, l'état (G, k, k, l) n'est pas atteignable.

L'idéal serait de pouvoir calculer l'ensemble des états atteignables à partir des états initiaux : on peut ensuite vérifier que son intersection avec un ensemble d'états indésirables est vide.

On peut définir l'ensemble des états atteignables formellement. En notant C l'ensemble des parties de Q, on définit d'abord pour $I \in C$ l'ensemble de ses états successeurs

$$succ(\tau)(X) = \{s' \mid \exists s \in X \ tq \ (s, s') \in \tau\}$$

puis l'ensemble des états atteignables depuis X, $att(\tau)(X)$ par

$$\begin{aligned}
&att(\tau)(X) = \lim_{n \to +\infty} att_n(\tau)(X) \\
&att_0(\tau)(X) = X \\
&att_{n+1}(\tau)(X) = X \cup succ(\tau)(att_n(\tau)(X))
\end{aligned}$$

En fait, l'ensemble des états atteignable à partir d'un ensemble d'états I s'obtient par résolution d'une équation de point fixe. En effet, pour $I \in C$ fixé, en notant $F(X) = I \cup succ(\tau)(X)$, la relation de récurrence s'écrit $att_{n+1}(\tau)(I) = F(att_n(\tau)(I))$, et on a comme résultat que $att(\tau)(I)$ est le plus petit point fixe de F.

Exemple 2 Pour l'exemple en C donné plus haut, les équations de point fixe correspondantes sont

$$\begin{aligned}
&A = (0, 0, 0) \\
&B = succ(\texttt{i = 2;})(A) \\
&C = (B \cup E) \cap \{(i, j, h) \mid i + j \leq 20\} \\
&D = succ(\texttt{i += 4;})(C \cap \{(i, j, h) \mid i \geq 10\}) \\
&E = succ(\texttt{i += 2; j++;})(C \cap \{(i, j, h) \mid i < 10\} \cap \{(i, j, h) \mid j \leq 4\}) \\
&F = D \cup E \\
&G = (B \cup F) \cap \{(i, j, h) \mid i + j > 20\} \\
&H = succ(\texttt{h = 100 / (i-j);})(G)
\end{aligned}$$

2.2 Résolution dans un domaine abstrait

Intuitivement, l'ensemble des états accessibles à partir de X peut s'obtenir en appliquant successivement la fonction F jusqu'à ce que la suite d'ensembles d'états se statibilise. C'est une méthode itérative pour trouver le plus petit point fixe, mais en pratique, l'ensemble C des parties de Q est trop complexe pour que le problème puisse être résolu tel quel.

Le principe de l'interprétation abstraite est de remplacer le domaine C dit *concret* par un domaine *abstrait* A plus simple, et de résoudre une équation de point fixe sur ce nouveau domaine par une méthode itérative.

On suppose qu'on dispose d'une fonction d'abstraction α qui à un élément de C (un ensemble d'états concrets) associe un élément de A (un ensemble d'états abstraits). En pratique, le domaine abstrait A est un sous-ensemble de C (par exemple l'ensemble des polyèdres convexes de C, ou les "intervalles" en dimension n de C), et la fonction d'abstraction associe à un ensemble d'états X un ensemble d'états plus grand (au sens de l'inclusion).

Pour F la fonction de C dans C définie plus haut, on définit l'abstraction de F, $\alpha(F) : A \rightarrow A$, par $\alpha(F)(X) = \alpha(F(X))$. $\alpha(F)$ a alors pour bonne propriété que son plus petit point fixe contient le plus petit point fixe de F (l'ensemble des états atteignables).

En d'autres termes, cela signifie que l'ensemble des états abstraits accessibles à partir de l'abstraction de I contient l'ensemble des états concrets accessibles à partir de I.

On peut par une méthode itérative calculer une approximation supérieure de l'ensemble des états accessibles d'un programme : on applique successivement $\alpha(F$, jusqu'à stabilisation.

2.3 Opérateur d'élargissement

De même que la suite définie récursivement par application de la fonction F, la suite définie par application de son abstraction est convergente. Mais on aimerait que la convergence se fasse en un nombre fini d'itérations, ce qui n'est pas nécessairement le cas. Pour le garantir, on peut appliquer des opérateurs d'*élargissement* [6]. Ceux-ci finissent toujours par fournir un $\alpha(X)$ tel que $\alpha(F)(\alpha(X)) \sqsubseteq \alpha(X)$, c'est à dire un invariant inductif.

Toutefois, s'ils garantissent une convergence en un nombre fini d'itérations, c'est au prix d'une perte de précision : le point fixe obtenu sera en fait seulement une approximation supérieure du plus petit point fixe de l'abstraction de F. Il existe cependant des techniques pour essayer de réduire l'invariant obtenu par la suite.

2.4 Domaines abstraits

On peut choisir des domaines abstraits plus ou moins complexes, et en général, plus le domaine abstrait sera complexe (coût des calculs), moins on perdra en précision.

2.4.1 Domaine des intervalles

Le domaine abstrait le plus simple est sans doute la généralisation des intervalles en dimension n : pour un ensemble d'états, on associe pour chaque variable le plus petit intervalle contenant ses valeurs.

Exemple 3 Sur le code C donné section 2.1, sans élargissement à gauche, et avec élargissement au point F à droite.

```
int main(int argc, char *argv[])
{
   int i = 0;
   int j = 0;
   int h = 0;
   // (A)  [0,0] × [0,0] × [0,0]

   i = 2;
   // (B)  [2,2](B) × [0,0] × [0,0]
   while (i+j <= 20) {
      // (C)  [2,20] × [0,5] × [0,0]
      if (i >= 10) {
         i += 4;
         // (D)  [14,24] × [0,5] × [0,0]
      } else if (j <= 4) {
         i += 2;
         j++;
         // (E)  [4,11] × [1,5] × [0,0]
      }
      // (F)  [4,24] × [0,5] × [0,0]
   }
   // (G)  [16,24] × [0,5] × [0,0]

   // (i − j) ∈ [16,19]
   // donc peut conclure que i − j ≠ 0
   assert(i-j != 0)

   return EXIT_SUCESS;
}
```

```
int main(int argc, char *argv[])
{
   int i = 0;
   int j = 0;
   int h = 0;
   // (A)  [0,0] × [0,0] × [0,0]

   i = 2;
   // (B)  [2,2](B) × [0,0] × [0,0]
   while (i+j <= 20) {
      // (C)  [2,20] × [0,18] × [0,0]
      if (i >= 10) {
         i += 4;
         // (D)  [14,24] × [0,18] × [0,0]
      } else if (j <= 4) {
         i += 2;
         j++;
         // (E)  [4,11] × [1,5] × [0,0]
      }
      // (F)  [4,+∞[×[0,+∞[×[0,0]
   }
   // (G)  [4,+∞[×[0,+∞[×[0,0]

   // Ici on sait seulement
      (i − j) ∈] − ∞,+∞[
   // on ne peut pas conclure que
      (i − j) ≠ 0
   assert(i-j != 0)

   return EXIT_SUCESS;
}
```

On a simplement propagé par à pas les intervalles pour chaque variable, pour toutes les exécutions possibles du programme.

2.4.2 Domaine des polyèdres convexes

Le domaine des intervalles est trop simple en général pour une bonne analyse, car il ne permet pas d'exprimer de relations entre les variables d'un programme.

Exemple 4 Analyse d'intervalles sur le programme simple suivant

```
int x = 5 + (rand() % 6);
int y = 0;
int z = 0;

// [5,10] × [0,0] × [0,0]
y = x;
// [5,10] × [5,10] × [0,0]
z = x - y;
// [5,10] × [5,10] × [−5,5]
```

Une analyse sur le domaine des intervalles donne $z \in [-5, 5]$, alors qu'on voit bien que $x = y$, donc que $z = 0$.

Une solution pour obtenir de meilleurs résultats est de se placer dans le domaine des polyèdres convexes. Un polyèdre représentant un ensemble d'inégalités linéaires, on conserve bien dans une certaine mesure des relations qui lient les variables entre elles. Dans l'exemple ci-dessus, on obtient bien avec une analyse à l'aide de polyèdres le résultat $z = 0$.

Si le domaine des polyèdres convexes peut a priori paraître encore assez simpliste, car il ne permet de capturer que des relations *linéaires* entre les variables, d'un point de vue calculatoire il se situe à la limite de ce qu'on peut encore appliquer en pratique (en particulier si le programme contient beaucoup de variables). Le problème est qu'il y a essentiellement 2 façons de représenter les polyèdres en machine (soit par un ensemble d'arêtes, soit par un ensemble de points), et que certaines opérations se font beaucoup plus rapidement sous une forme, et d'autres sous l'autre forme. Or le coût pour passer d'une forme à l'autre est en pire cas exponentiel par rapport au nombre de variables. Néanmois, ce domaine est utilisable en pratique et permet d'obtenir de meilleurs résultats qu'avec le domaine des intervalles.

3 LLVM

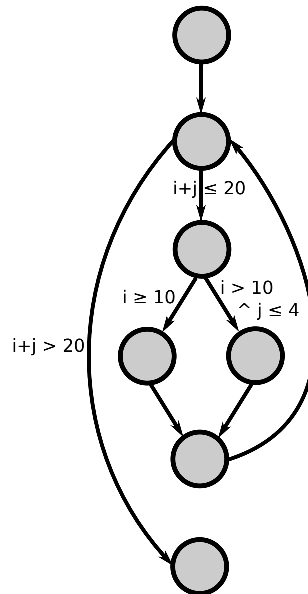
Dans cette section il s'agit de présenter la représentation intermédiaire utilisée par LLVM, à partir de laquelle fonctionne notre analyseur.

3.1 Graphe de flot de contrôle

Un programme peut être représenté par un graphe de flot de contrôle (CFG pour Control Flow Graph). Cette représentation est pratique car elle permet de visualiser directement tous les chemins d'exécution possibles d'un programme.

Dans un graphe de flot de contrôle, les noeuds sont des *basic blocks*, c'est-à-dire une suite d'instructions toujours exécutée d'un bloc. Un *basic block* ne peut pas contenir de saut (conditionnel ou non), sauf en dernière instruction. Typiquement, il s'agit en général d'une suite d'affectations, d'opérations arithmétiques, etc., terminée par un saut vers d'autres *basic blocks*.

Exemple 5 Graphe de flot de contrôle représentant le programme C donné section 2.1



Le fichier *bitcode* généré par LLVM a exactement la structure d'un graphe de flot de contrôle (CFG pour Control Flow Graph) : il est divisé en *basic blocks*, dont la dernière instruction est un branchement vers d'autres *basic blocks*.

3.2 Forme SSA

La représentation intermédiaire de LLVM est également sous forme SSA (Static Single Assignment Form). Un programme sous cette forme a de particulier qu'une variable n'est affectée qu'une unique fois syntaxiquement. Tout programme peut être mis sous forme SSA, en "indiçant" les variables du programme de départ.

Exemple 6
A gauche, un fragment de code C simple, et à droite une traduction possible sous forme SSA

```
int x, y;
x = 4;
x = x * 2;
y = x - 2;
```

```
int x1, x2, y;
x1 = 4;
x2 = x1 * 2;
y = x2 - 2;
```

Pour les programmes contenant des branchements conditionnels comme des *if*, la traduction n'est possible qu'en faisant intervenir une fonction particulière ϕ, permettant de choisir entre deux "versions" d'une variable, selon le chemin d'exécution emprunté.

Exemple 7
A gauche, un bout de code C comprenant un branchement conditionnel, et à droite une traduction possible sous forme SSA

```
int x, y;
x = rand() % 10;
if (x <= 5) {
   y = 5 - x;
} else {
   y = x - 5;
}
y = 2 * y;
```

```
int x, y1, y2, y3, y4;
x = rand() % 10;
if (x <= 5) {
   y1 = 5 - x;
} else {
   y2 = x - 5;
}
y3 = phi(y1,y2)
y4 = 2 * y3;
```

Exemple 8

A gauche, un programme C correspondant à l'exemple précédent, et à droite le *bitcode* généré par LLVM

```c
1  #include <stdlib.h>
2
3  int main(int argc, char *argv[]) {
4     int x, y;
5     x = rand() % 10;
6
7     if (x <= 5) {
8        y = 5 - x;
9     } else {
10       y = x - 5;
11    }
12    y = 2 * y;
13
14    return 0;
15 }
```

```llvm
1  define i32 @main(i32 %argc, i8** %
       argv) nounwind {
2  bb:
3    %tmp = call i32 @rand() nounwind
4    %tmp1 = srem i32 %tmp, 10
5    %tmp2 = icmp sle i32 %tmp1, 5
6    br i1 %tmp2, label %bb3, label %
        bb5
7  bb3: ; preds = %bb
8    %tmp4 = sub nsw i32 5, %tmp1
9    br label %bb7
10 bb5: ; preds = %bb
11   %tmp6 = sub nsw i32 %tmp1, 5
12   br label %bb7
13 bb7: ; preds = %bb5, %bb3
14   %y.0 = phi i32 [ %tmp4, %bb3 ], [
          %tmp6, %bb5 ]
15   %tmp8 = mul nsw i32 2, %y.0
16   ret i32 0
17 }
18
19 declare i32 @rand() nounwind
```

Le bloc bb correspond aux lignes 4 à 7 du programme C, le bloc bb3 à la 8, bb5 à la 11, et bb7 à la 13.

4 Apron

Apron est une librairie permettant d'effectuer des calculs dans différents domaines abstraits, comme les intervalles ou les polyèdres convexes. Elle fait en réalité office d'interface entre plusieurs librairies de calcul : elle fournit des fonctions génériques pour l'interprétation abstraite, indépendantes du domaine numérique.

Il suffit de préciser au départ sur quel domaine on décide de mener l'analyse, et Apron se chargera d'utiliser les fonctions de calcul appropriées. Par exemple, même si notre analyseur travaille à partir du domaine des polyèdres convexes, il suffit de changer une seule ligne de code dans le programme pour travailler à partir des intervalles, des octaèdres, etc.

L'avantage d'utiliser Apron est donc que si de nouveaux domaines abstraits, meilleurs que les polyèdres, sont mis au jour et intégrés à Apron, il n'y aura en pratique rien à changer pour que notre analyseur puisse les utiliser.

5 Conception d'un analyseur

Dans cette section, nous allons décrire plus en détail la conception de l'analyseur.

5.1 Repérer les boucles

L'analyseur effectue d'abord une première passe à la fois pour lister les variables du programme (cela permet de compter et nommer les dimensions du domaine numérique abstrait) et pour repérer les boucles, en particulier les blocs situés en tête de boucle. Il est nécessaire de repérer ces blocs pour garantir l'arrêt de l'analyseur. En effet c'est au niveau de ces blocs que nous appliquons l'opérateur d'élargissement (pour assurer la convergence en un nombre fini d'itérations), et que nous testons à chaque passage si nous avons obtenu un invariant (si la valeur abstraite est identique à celle calculée lors du dernier passage par ce bloc). Un simple parcours en profondeur permet de repérer les arcs à destination d'un noeud déjà visité, et ainsi les têtes de boucle.

En réalité, il existe d'autres ensembles de noeuds que ceux en tête de boucle sur lesquels on peut appliquer l'opérateur d'élargissement et garantir la converge en un nombre fini d'itérations. En pratique, on souhaite élargir sur le plus petit nombre de noeuds possible, car chaque élargissement réduit potentiellement la précision de l'invariant qu'on obtiendra.

5.2 Recherche des invariants

Concrètement, l'analyseur parcourt le graphe de flot de contrôle tant que les valeurs abstraites calculées à la fin de chaque bloc (juste avant le saut) ne sont pas stabilisées. Nous donnons dans un premier temps l'algorithme général de ce parcours, de manière informelle, puis nous détaillons comment nous avons traité concrètement les principales instructions du *bitcode* LLVM.

Algorithme général

On donne par étapes les opérations effectuées lors de la visite d'un *basicblock* `b`. On commencera bien sûr par visiter le bloc d'entrée de la fonction `main` du programme.

1. On récupère la liste des prédécesseurs de `b`.
2. Pour chaque prédécesseur `pred` de `b` on effectue les opérations suivantes :
   (a) Récupérer la valeur abstraite `abs_pred` de `pred` calculée lors de la dernière visite de `pred` (on prend la valeur vide s'il n'a pas encore été visité).
   (b) Intersecter `abs_pred` avec les conditions de saut associées à l'arc `pred` $\rightarrow$ `b`, dans le cas d'un saut conditionnel.
   (c) Appliquer la fonction de transition associée aux instructions du bloc `b` (typiquement suite d'affectations, voir détails des instructions plus bas).
3. Affecter à `abs` la plus petite valeur abstraite contenant les valeurs abstraites des blocs prédécesseurs tout juste calculées.
4. Si `b` est un point d'élargissement, i.e. est en tête de boucle alors :
   (a) Appliquer l'opérateur d'élargissement sur `abs` à partir de la valeur abstraite `old_abs` calculée lors de la dernière visite de `b` (ne rien faire si `b` n'a pas encore été visité).
5. Si `abs` est différente de `old_abs` alors :

(a) Visiter chacun des successeurs de `b` (il faut revisiter tous les successeurs de `b`, y compris ceux dont la valeur abstraite n'avait pas changé entre deux visites).

6. Sinon

   (a) Visiter les successeurs de `b` dont la valeur abstraite n'est pas stabilisée (la valeur abstraite de `b` étant stabilisée pour le moment, il suffit de visiter les successeurs dont ce n'est pas encore le cas).

Dans l'algorithme ci-dessus, nous n'avons pas détaillé comment la fonction de transition était calculée à partir des instructions du bloc, i.e. l'opération appliquée aux valeurs abstraites pour chaque instruction. C'est l'objet des paragraphes qui suivent.

Instruction d'opération binaire

Les opérations arithmétiques unaires et binaires ne sont pas différenciées par LLVM : elles sont toutes deux traduites par LLVM en des instructions binaires, de type `BinaryOperator`. Pour une instruction binaire, de la forme `x = y op z`, l'opération appliquée aux valeurs abstraites est très simple, car Apron fournit directement une fonction permettant d'affecter une expression à une variable. Celà revient en fait à ajouter la contrainte $x = y \ op \ z$ à la valeur abstraite. Concrètement, le traitement se résume à créer une expression Apron `exp` représentant `y op z` et à appliquer l'affectation `x` $\leftarrow$ `exp`.

Instruction de comparaison d'entiers

En *bitcode* LLVM, une instruction conditionnelle comme un if se décompose en deux instructions : d'abord une instruction pour calculer l'expression booléenne (l'instruction de comparaison, de type `ICmpInst`), puis un branchement conditionnel (de type `BranchInst`) portant sur une variable booléenne.

Exemple 9 A gauche le code C, et à droite la traduction par LLVM

```c
if (x > 10) {
   x += 2;
}
```

```
bb1:
  %tmp = icmp sgt i32 %x.0, 10
  br i1 %tmp, label %bb2, label %
      bb3
bb2:
  %x.1 = add nsw i32 %x.0, 2
bb3:
  %x.2 = phi i32 [ %x.0, %bb1 ], [
      %x.1, %bb2 ]
```

Lorsqu'on rencontre une instruction de branchement conditionnel, qui se fait par rapport à la valeur d'une variable booléenne (`tmp`), on veut pouvoir récupérer les contraintes correspondantes ($x.0 > 10$ vers `bb2` et $x.0 \leq 10$ vers `bb3`). Pour cela, nous traitons les instructions de comparaison arithmétiques en leur associant les contraintes correspondantes.

Instruction de branchement
Dans LLVM, une instruction de branchement est de type `BranchInst` ; Plusieurs cas sont possibles :

- Le branchement est non conditionnel, auquel cas il n'y a pas de traitement particulier à effectuer.
- Le branchement est conditionnel. Dans ce cas, si la variable booléenne concernée est résultat d'une instruction de comparaison d'entiers, on récupère les contraintes et on les ajoutes aux arcs correspondants. Si la variable booléenne n'est pas le résultat direct d'une instruction de comparaison (par exemple elle est le résultat d'opérations logiques sur des booléens), notre analyseur n'ajoute pour le moment pas de contrainte (les opérations logiques n'étant pas supportées) : l'analyseur reste néanmoins correct, il génèrera simplement des invariants trop grands.

Instruction ϕ
Dans le *bitcode* LLVM, une instruction ϕ est de type `PHINode` et a la forme `%x.2 = phi i32 [ %x.1, %bb1 ], [ %x.2, %bb2 ]` (le nombre de variables et de blocs est variable). L'instruction ϕ permet de choisir entre différentes versions d'une variable suivant le bloc prédécesseur visité dans le chemin d'exécution du programme. Notre analyseur n'exécute pas le programme, mais calcule des invariants en simulant toutes les exécutions possibles du programme.

Cela signifie qu'il parcourt les instructions d'un bloc pour chacun des prédécesseurs possibles (cf algorithme plus haut : pour chaque prédécesseur `pred` du bloc `b`, on calcule une valeur abstraite à partir de celle de `pred`, en parcourant les instructions de `b`). Lorsqu'on rencontre une instruction ϕ, suivant le prédécesseur considéré, on sait quelle affectation appliquer et sur quelle valeur abstraite. Par exemple, pour l'instruction ci-dessus, lorsqu'on vient du bloc `bb1`, il suffit d'appliquer l'affectation $x.2 \leftarrow x.1$ à la valeur abstraite de `bb1`.

Instructions non supportées
Nous avons détaillé le traitement de seulement 3 instructions, alors que le *bitcode* LLVM en comporte en réalité un certain nombre. Bien que nous en ayons traité d'autres partiellement (comme `SwitchInst`), la plupart des instructions restantes ne sont pas implémentées. En particulier, les instructions liées à la mémoire et aux pointeurs (`AllocaInst`, `LoadInst`, `StoreInst`, etc) sont ignorées dans l'analyse, de même que les appels de fonction et conversions de types.

Lorsqu'on rencontre une instruction non supportée, on décide d'oublier toutes les informations sur les variables qui interviennent dans celle-ci : elles ne sont plus contraintes. De cette manière, même si ces instructions interviennent dans le programme analysé, les invariants obtenus seront corrects.

5.3 Réduction des invariants

Dans l'algorithme général donné en section 5.2, on décide d'appliquer un opérateur d'élargissement au passage de chaque tête de boucle, pour assurer la convergence en temps fini, et donc l'arrêt de l'analyse. Or nous avons vu sur l'exemple 3 section 2.4.1 que dans certains cas, il n'est pas nécessaire d'appliquer d'élargissement pour que l'analyse termine, et qu'appliquer un opérateur d'élargissement fait perdre énormément en précision.

Il existe en réalité des méthodes pour réduire les invariants obtenus après coup. En particulier, il y en a une très simple que notre analyseur applique : il s'agit de parcourir une ou plusieurs fois les noeuds du graphe, en appliquant le même algorithme, mais sans faire d'élargissement. En effet, on peut montrer que les valeurs abstraites ainsi calculées sont encore des invariants, et en général, elles sont moins larges que les valeurs abstraites calculées précédemment.

5.4 Ajout d'hypothèses : *assume*

Parfois, on aimerait fournir à l'analyseur des hypothèses supplémentaires sur les variables pour le guider et obtenir de meilleurs résultats. Par exemple, on peut vouloir émettre des hypothèses sur les valeurs d'entrée des paramètres d'une fonction (à l'utilisateur ensuite, ou à un analyseur, de vérifier que ces conditions sont bien vérifiées à chaque appel), sur des valeurs entrées par l'utilisateur, ou encore sur les valeurs de retour de fonctions de librairies.

Typiquement, on souhaite que des instructions du type `assume(x >= 0);` soient comprises par l'analyseur : sans vérification, celui-ci doit continuer l'analyse en ajoutant comme contrainte $x \geq 0$. Une telle fonctionnalité est facilement implémentable. Il suffit de fixer comme convention que le code à analyser définit la macro `assume` et une fonction `____FAIL____` comme suit :

```
#define assume(X) if (X) {} else { ____FAIL____(); }

void ____FAIL____() {
   exit(EXIT_FAILURE);
}
```

Au moment de l'analyse, il suffit alors de repérer les appels à la fonction `____FAIL____`, et de forcer la valeur abstraite du bloc qui contient un tel appel à l'ensemble vide : de cette manière, on indique bien qu'un bloc dans lequel la condition `X` n'est pas vérifiée n'est jamais atteignable.

5.5 Vérification de propriétés : *check*

Un analyseur qui se contente de fournir en chaque point du programme un polyèdre contenant toutes les valeurs possibles des variables n'est pas directement utilisable. On veut plutôt pouvoir demander à l'analyseur de vérifier certaines propriétés, par exemple qu'en sortie de boucle, une variable est strictement positive.

Il s'agit donc que des instructions du type `check(x > 0)` soient comprises par l'analyseur : celui-ci ne doit cette fois pas supposer $x > 0$, mais le vérifier. Néanmoins, cette fonctionnalité s'implémente finalement de manière assez similaire à *assume*. De même, le code à analyser peut définir une macro `check` et une fonction `____CHECK_NOT_REACHABLE____` comme suit :

```
#define check(X) if (X) {} else { ____CHECK_NOT_REACHABLE____(); }

void ____CHECK_NOT_REACHABLE____() {
   exit(EXIT_FAILURE);
}
```

Dans l'analyse, il n'y a pas de traitement particulier à faire : il suffit d'ignorer les appels à `____CHECK_NOT_REACHABLE____`. En revanche, après avoir calculé et réduit les invariants, on

parcourt une dernière fois les blocs du programme, en vérifiant que chaque bloc contenant un appel à cette fonction a une valeur abstraite vide : cela signifie qu'en ce point de programme, `!X` est impossible.

Bien sûr, $x > 0$ peut être vrai en un point du programme sans que l'analyseur n'arrive pour autant à le garantir.

6 Résultats

Nous allons présenter les résultats qu'on peut obtenir en appliquant notre analyseur sur des exemples simples. Dans les bouts de code exhibés, on n'a pas écrit la définition des macros, mais on a indiqué par un commentaire l'endroit où elles devraient l'être.

Exemple 10 Soit le programme C suivant , qui contient une boucle simple

```
#include <stdlib.h>

// Définition des macros ici

int main(int argc, char *argv[]) {
   int i = 2;
   int j = 0;

   while (i <= 10) {
       i = i + 2;
       j = j + 1;
       check(i == 2*(j+1));
   }

   check(j == 5);

   return EXIT_SUCCESS;
}
```

Notre analyseur est bien capable de garantir que les conditions des deux *check* sont vraies. Ça n'aurait pas été le cas si on avait effectué une analyse basée sur les intervalles plutôt que sur les polyèdres, car on n'aurait pas pu conserver la relation linéaire liant les variables `i` et `j`.

Exemple 11 Soit le programme C suivant, proche de celui donné dans l'exemple 1 section 2.1

```
#include <stdlib.h>

// Définition des macros ici

int main(int argc, char *argv[]) {
   int i = 2;
   int j = 0;
```

```
    while (i+j <= 20) {
        if (i >= 10) {
            i = i + 4;
        } else {
            i = i + 2;
            j = j + 1;
        }
    }
    check(i+j > 20);
    check(i+j <= 24);

    return EXIT_SUCCESS;
}
```

Le programme présente une boucle contenant des branchements conditionnels *if () {} else {}* . Il n'y a pas directement d'égalité linéaire liant les variables `i` et `j`, mais plusieurs inégalités linéaires.

Ici encore, notre analyseur arrive à garantir les propriétés des deux *check*. La première n'est pas particulièrement difficile à obtenir (elle découle directement de la condition de sortie de la boucle), mais la deuxième peut être montrée grâce aux inégalités linéaires sur `i` et `j`. Celles-ci apparaissent directement dans le fichier de sortie de l'analyseur (voir en annexe), qui indique pour chaque *basicblock* l'invariant (le polyèdre) obtenu.

Exemple 12 Soit le programme C suivant

```
#include <stdlib.h>
#include <time.h>

// Définition des macros ici

int main(int argc, char *argv[]) {
    int i = 2;
    int n;

    srand(time(NULL));
    n = rand() % 20;

    for (i = 0; i < n; ++i) {
        // Faire quelque chose...
    }

    check(i == n);

    return EXIT_SUCCESS;
}
```

La condition du *check* est bien sûr vraie, mais notre analyseur affiche un avertissement car il n'arrive pas à le garantir. Cela est dû au fait que l'analyseur ne sait pas qu'après son initialisation, la variable `n` est positive (car `rand` retourne une valeur positive).

Pour résoudre ce problème, il suffit d'ajouter comme hypothèse $n \geq 0$, en utilisant la macro `assume`, comme suit :

Exemple 13 Soit le programme C suivant

```
#include <stdlib.h>
#include <time.h>

// Définition des macros ici

int main(int argc, char *argv[]) {
   int i = 2;
   int n;

   srand(time(NULL));
   n = rand() % 20;
   assume(n >= 0);

   for (i = 0; i < n; ++i) {
      // Faire quelque chose...
   }

   check(i == n);

   return EXIT_SUCCESS;
}
```

Dans ce cas, l'analyseur parvient à montrer que `i == n` en sortie de boucle.

Exemple 14a Soit le programme C suivant, avec deux boucles simples imbriquées

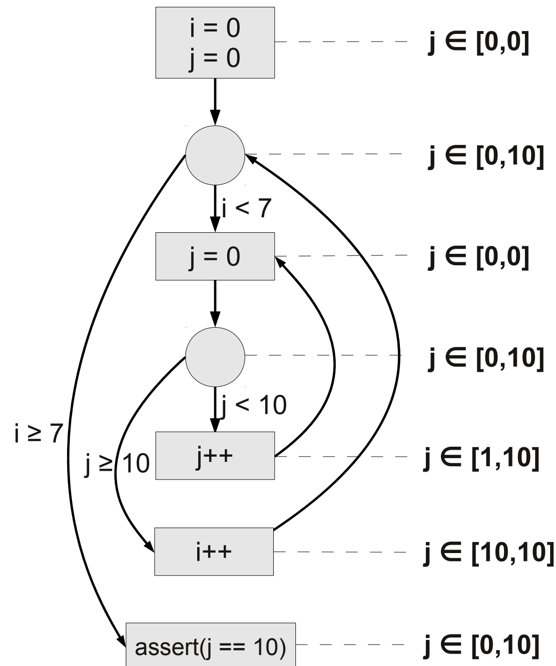
```
#include <stdlib.h>

// Définition des macros ici

int main(int argc, char *argv[]) {
   int i = 0;
   int j = 0;

   while (i < 7) {
      for (j = 0; j < 10; ++j) {
         // Faire quelque chose...
      }
      ++i;
   }

   check(j == 10);
```

```
    return 0;
}
```

Dans ce cas pourtant extrêmement simple, l'analyseur ne donne pas d'aussi bons résultats que ceux auxquels on pourrait s'attendre. En effet, il n'arrive pas à garantir que la propriété du *check* est vraie. Il y a deux raisons à cela.

D'abord, une fois les invariants obtenus, il faut réduire les réduire en parcourant au moins 2 fois le graphe (par défaut, l'analyseur n'effectue qu'un pas de calcul pour réduire les invariants, mais ce paramètre peut être passé en argument).

Ensuite, l'invariant en sortie de la boucle externe (celle sur `i`) est calculé à partir de l'invariant en tête de boucle, intersecté avec la condition de sortie $i \geq 7$: or ici, comme `j` est initialisée à 0 en début de programme, en tête de boucle on a $j = 0$ ou $j = 10$, c'est-à-dire $j \in [0, 10]$ (les valeurs abstraites sont convexes), et en intersectant avec la condition de sortie (qui ne porte que sur `i`), on obtient $j \in [0, 10]$.



Graphe de flot de contrôle de l'exemple 14 section 6, avec intervalles pour j

Une première solution pour résoudre ce problème est de modifier le programme et d'initialiser `j` à 10 au début. On aura donc bien $j = 10$ comme invariant en tête de la boucle sur `i`, donc en sortie de boucle.

Une autre solution aurait été de « dérouler »la boucle une fois (ce que certains analyseurs font automatiquement, ce qui n'est pas le cas du nôtre), de la façon suivante :

Exemple 14 bis En déroulant une fois la boucle externe de l'exemple 14 section 6

```
#include <stdlib.h>

// Définition des macros ici

int main(int argc, char *argv[]) {
   int i = 0;
   int j = 0;

   // On fait d'abord une itération de la boucle externe
   for (j = 0; j < 10; ++j) {
      // Faire quelque chose...
   }
   ++i;

   // Ici on a bien comme invariant j=10
   while (i < 7)
      for (j = 0; j < 10; ++j) {
         // Faire quelque chose...
      }
      ++i;
   }

   // D'où en sortie de boucle j=10
   check(j == 10);

   return 0;
}
```

Dans ce cas notre analyseur prouve la propriété $j = 10$.

7 Perspectives

L'analyseur produit est très basique ; il reste de nombreuses fonctionnalités possibles à implémenter, ainsi que des pistes à explorer pour affiner l'analyse. En particulier, on énonce les points suivants :

- L'analyseur ne supporte pour le moment pas les flottants. Néanmoins leur traitement devrait être très similaire à celui des entiers, grâce à Apron, qui permet d'associer un type à chacune des dimensions (donc des variables) du domaine numérique abstrait.
- Seule la fonction principale *main* du programme C est analysée. Il serait très facile de modifier l'analyseur pour qu'il traite toutes les fonctions d'un fichier source. Mais dans tous les cas, les appels de fonction ne sont pas supportés, c'est-à-dire que l'analyseur ne peut déduire aucune propriété sur les valeurs de retour de fonctions.
- Le programme ne fait aucune analyse sur les pointeurs, les tableaux, etc., mais uniquement sur les variables numériques entières.

- Les variables booléennes et opérateurs logiques qui s'y rapportent ne sont pas supportés. Dans le *bitcode* LLVM, les booléens sont des variables entières sur 1 bit : l'analyseur pourrait être modifié pour traiter les booléens comme des entiers valant 0 ou 1, mais il existe d'autres méthodes d'analyse plus adaptées au calcul sur les booléens.
- Dans l'algorithme général donné en section 5.2, on indique que l'analyseur profite d'une première passe pour lister toutes les variables du programme et fixer ainsi les dimensions du domaine abstrait. En pratique, cela signifie que les calculs sur les valeurs se feront toujours dans un espace avec beaucoup de dimensions, ce qui peut devenir très coûteux. Dans la plupart des cas, les instructions des *basicblock* n'agissent que sur un petit nombre de variables (par rapport à l'ensemble des variables du programme), il devrait donc être possible de faire des calculs plus rapides sur des espaces de plus faible dimension définis "localement".
- Dans le même ordre d'idées que le point précédent, de nombreuses variables n'ont pas besoin d'être rajoutées comme dimensions de l'espace : il suffit de remplacer chacune de leurs occurences dans le programme par leur expression. Par exemple, si une variable `temp` vaut 2, on peut remplacer toutes ses occurences dans le programme par sa valeur : cela éviterait d'augmenter inutilement le nombre de dimensions du domaine abstrait et accélèrerait les calculs.
- Enfin, on pourrait également ajouter une interface graphique à l'analyseur, ou l'intégrer dans un environnement de développement.

8 Conclusion/Bilan

Après quelques rappels sur l'interprétation abstraite, et avoir présenté les librairies LLVM et Apron, nous avons montré comment il était possible de concevoir un analyseur de programme C basé sur l'interprétation abstraite à l'aide de ces librairies. En très peu de lignes de code (la partie purement analyse est composée d'à peine plus de 1000 lignes), nous avons ainsi obtenu un analyseur capable de prouver des propriétés non triviales sur des programmes courts composés de boucles et de branchements conditionnels. De plus, l'analyseur travaillant à partir de la représentation intermédiaire de LLVM, il pourrait également analyser du code écrit dans d'autres langages que C, pour peu qu'il existe un front-end convertissant le code en *bitcode* LLVM.

Références

[1] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[2] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, April 2–10 2005. © Springer.

[3] B. Jeannet. *Partitionnement Dynamique dans l'Analyse de Relations Linéraires et Application à la Vérification de Programmes Synchrones.* PhD thesis, Institut National Polytechnique de Grenoble, September 2000.

[4] B. Jeannet and A. Miné. Apron : A library of numerical abstract domains for static analysis. In *Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV 2009)*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667, Grenoble, France, June 2009. Springer. http://www.di.ens.fr/~mine/publi/article-mine-jeannet-cav09.pdf.

[5] Chris Lattner and Vikram Adve. LLVM : A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[6] David Monniaux. A minimalistic look at widening operators. *Higher order and symbolic computation*, 22(2) :145–154, dec 2009.

Annexe

A Bitcode LLVM de l'exemple 11 section 6

```
define i32 @main(i32 %argc, i8** %argv) nounwind {
bb:
  br label %bb1

bb1:                                              ; preds = %bb10, %bb
  %i.1 = phi i32 [ 2, %bb ], [ %i.0, %bb10 ]
  %j.1 = phi i32 [ 0, %bb ], [ %j.0, %bb10 ]
  %tmp = add nsw i32 %i.1, %j.1
  %tmp2 = icmp sle i32 %tmp, 20
  br i1 %tmp2, label %bb3, label %bb11

bb3:                                              ; preds = %bb1
  %tmp4 = icmp sge i32 %i.1, 10
  br i1 %tmp4, label %bb5, label %bb7

bb5:                                              ; preds = %bb3
  %tmp6 = add nsw i32 %i.1, 4
  br label %bb10

bb7:                                              ; preds = %bb3
  %tmp8 = add nsw i32 %i.1, 2
  %tmp9 = add nsw i32 %j.1, 1
  br label %bb10

bb10:                                             ; preds = %bb7, %bb5
  %i.0 = phi i32 [ %tmp6, %bb5 ], [ %tmp8, %bb7 ]
  %j.0 = phi i32 [ %j.1, %bb5 ], [ %tmp9, %bb7 ]
  br label %bb1

bb11:                                             ; preds = %bb1
  %tmp12 = add nsw i32 %i.1, %j.1
  %tmp13 = icmp sgt i32 %tmp12, 20
  br i1 %tmp13, label %bb14, label %bb15

bb14:                                             ; preds = %bb11
  br label %bb16

bb15:                                             ; preds = %bb11
  call void @___CHECK_NOT_REACHABLE___()
  br label %bb16
```

```
bb16:                                            ; preds = %bb15, %
   bb14
  %tmp17 = add nsw i32 %i.1, %j.1
  %tmp18 = icmp sle i32 %tmp17, 24
  br i1 %tmp18, label %bb19, label %bb20

bb19:                                            ; preds = %bb16
  br label %bb21

bb20:                                            ; preds = %bb16
  call void @___CHECK_NOT_REACHABLE___()
  br label %bb21

bb21:                                            ; preds = %bb20, %
   bb19
  ret i32 0
}
```

B Sortie de l'analyseur pour l'exemple 11 section 6

```
bb:
polyhedron of dim (14,0)
empty array of constraints

bb1:
polyhedron of dim (14,0)
array of constraints of size 6
 0: -i.1 - j.1 + tmp = 0
 1: -i.1 - j.1 + 24 >= 0
 2: -tmp2 + 1 >= 0
 3: tmp2 >= 0
 4: i.1 - 4j.1 + 6 >= 0
 5: i.1 - 2j.1 - 2 >= 0

bb3:
polyhedron of dim (14,0)
array of constraints of size 7
 0: -i.1 - j.1 + tmp = 0
 1: tmp2 - 1 = 0
 2: -i.1 - j.1 + 20 >= 0
 3: -tmp4 + 1 >= 0
 4: tmp4 >= 0
 5: i.1 - 4j.1 + 6 >= 0
 6: i.1 - 2j.1 - 2 >= 0

bb5:
```

```
polyhedron of dim (14,0)
array of constraints of size 7
 0: -i.1 - j.1 + tmp = 0
 1: -i.1 + tmp6 - 4 = 0
 2: tmp4 - 1 = 0
 3: tmp2 - 1 = 0
 4: -i.1 - j.1 + 20 >= 0
 5: i.1 - 4j.1 + 6 >= 0
 6: i.1 - 10 >= 0

bb7:
polyhedron of dim (14,0)
array of constraints of size 8
 0: -i.1 - j.1 + tmp = 0
 1: -i.1 + tmp8 - 2 = 0
 2: -j.1 + tmp9 - 1 = 0
 3: tmp4 = 0
 4: tmp2 - 1 = 0
 5: -i.1 + 9 >= 0
 6: -j.1 + 3 >= 0
 7: i.1 - 2j.1 - 2 >= 0

bb10:
polyhedron of dim (14,0)
array of constraints of size 10
 0: -i.0 - i.1 - 2j.0 + 2tmp + 4 = 0
 1: -i.0 + i.1 - 2j.0 + 2j.1 + 4 = 0
 2: -i.0 + i.1 + 2tmp4 + 2 = 0
 3: tmp2 - 1 = 0
 4: -i.0 + i.1 + 4 >= 0
 5: -i.0 + 2i.1 - 4j.0 + 10 >= 0
 6: -i.0 + 2i.1 - 2j.0 + 2 >= 0
 7: i.0 - i.1 - 2 >= 0
 8: 3i.0 - 3i.1 - 5j.0 + 14 >= 0
 9: 7i.0 - 9i.1 - 2j.0 + 12 >= 0

bb11:
polyhedron of dim (14,0)
array of constraints of size 8
 0: -i.1 - j.1 + tmp12 = 0
 1: -i.1 - j.1 + tmp = 0
 2: tmp2 = 0
 3: -i.1 - j.1 + 24 >= 0
 4: -tmp13 + 1 >= 0
 5: tmp13 >= 0
 6: i.1 - 4j.1 + 6 >= 0
```

```
 7: i.1 + j.1 - 21 >= 0

bb14:
polyhedron of dim (14,0)
array of constraints of size 7
 0: -i.1 - j.1 + tmp12 = 0
 1: -i.1 - j.1 + tmp = 0
 2: tmp2 = 0
 3: tmp13 - 1 = 0
 4: -i.1 - j.1 + 24 >= 0
 5: i.1 - 4j.1 + 6 >= 0
 6: i.1 + j.1 - 21 >= 0

bb15:
empty polyhedron of dim (14,0)
Check OK

bb16:
polyhedron of dim (14,0)
array of constraints of size 10
 0: -i.1 - j.1 + tmp17 = 0
 1: -i.1 - j.1 + tmp12 = 0
 2: -i.1 - j.1 + tmp = 0
 3: tmp2 = 0
 4: tmp13 - 1 = 0
 5: -i.1 - j.1 + 24 >= 0
 6: -tmp18 + 1 >= 0
 7: tmp18 >= 0
 8: i.1 - 4j.1 + 6 >= 0
 9: i.1 + j.1 - 21 >= 0

bb19:
polyhedron of dim (14,0)
array of constraints of size 9
 0: -i.1 - j.1 + tmp17 = 0
 1: -i.1 - j.1 + tmp12 = 0
 2: -i.1 - j.1 + tmp = 0
 3: tmp2 = 0
 4: tmp18 - 1 = 0
 5: tmp13 - 1 = 0
 6: -i.1 - j.1 + 24 >= 0
 7: i.1 - 4j.1 + 6 >= 0
 8: i.1 + j.1 - 21 >= 0

bb20:
empty polyhedron of dim (14,0)
```

```
Check OK

bb21:
polyhedron of dim (14,0)
array of constraints of size 9
 0: -i.1 - j.1 + tmp17 = 0
 1: -i.1 - j.1 + tmp12 = 0
 2: -i.1 - j.1 + tmp = 0
 3: tmp2 = 0
 4: tmp18 - 1 = 0
 5: tmp13 - 1 = 0
 6: -i.1 - j.1 + 24 >= 0
 7: i.1 - 4j.1 + 6 >= 0
 8: i.1 + j.1 - 21 >= 0
```