TD Feuille 5 — Analyse statique d'intervalles

Le but de cet exercice est de définir une analyse statique vérifiant l'absence de débordements arithmétiques dans les entier signés 32 bits. Dans l'exercice, toutes les variables sont implicitements de type **int** et donc, par invariant de typage, comprise entre $M_- = -2^{31}$ et $M_+ = 2^{31} - 1$. On considère ici un domaine abstrait non-relationnel $\mathbb{D}$ dérivé d'un domaine de valeurs abstraites $\mathbb{V}$, où chaque valeur abstraite est soit l'intervalle vide noté $\bot$, soit un couple de $\mathbb{Z} \times \mathbb{Z}$ noté $[a, b]$, avec $M_- \leq a \leq b \leq M_+$, qui représente l'ensemble des entiers v tels que $a \leq v \leq b$.
Pour simplifier les notations, on considère que $[a, b]$ avec $a > b$ se "simplifie" en $\bot$.

Exercice 1 (Mise en place du treillis borné de $\mathbb{V}$).

1. Définir $\sqcup$, $\sqcap$ et $\sqsubseteq$.
2. A quoi correspond $\top$? Le treillis est-il de hauteur finie ?

Formellement, il faut gérer l'intervalle vide : c'est sans difficulté ($\bot$ est neutre pour $\sqcup$, absorbant pour $\sqcap$ et minimum de $\sqsubseteq$). On se focalise sur le traitement des intervalles non vides.

1. $[a_1, b_1] \sqcup [a_2, b_2] \stackrel{def}{=} [\min(a_1, a_2), \max(b_1, b_2)]$;
   $[a_1, b_1] \sqcap [a_2, b_2] \stackrel{def}{=} [\max(a_1, a_2), \min(b_1, b_2)]$;
   $[a_1, b_1] \sqsubseteq [a_2, b_2] \stackrel{def}{=} (a_2 \leq a_1 \wedge b_1 \leq b_2)$.
2. $\top \stackrel{def}{=} [M_-, M_+]$. Le treillis est de hauteur finie égale à $M_+ - M_- + 1 = 2^{32}$ (c'est grand !)

Exercice 2 (Opérateurs arithmétiques dans $\mathbb{V}$). Pour calculer la valeur abstraite des termes (dans un certain état abstrait), on pense à définir les opérations suivantes sur les valeurs abstraites :

$$[a_1, b_1] + [a_2, b_2] \qquad -[a, b] \qquad [a_1, b_1] * [a_2, b_2] \qquad [a, b]/2$$

Ici $a/2$ représente la division entière par 2 de C. On rappelle qu'elle vérifie $(-a)/2 = -(a/2)$.

Pour simplifier, on va autoriser ces opérations à produire des intervalles qui débordent de $[M_-, M_+]$. L'idée est que l'analyse d'une affectation $x := T$ sera de toute façon précédée d'un **assert** générant une obligation de preuve que l'interprétation $T[A]$ de T dans l'état abstrait courant A satisfait $T[A] \sqsubseteq [M_-, M_+]$. Et *in fine*, on définira

$$A[x := T] \stackrel{def}{=} A \oplus \{x \mapsto T[A] \sqcap [M_-, M_+]\}$$

▹ **Question 1.** Définir les opérateurs ci-dessus. Comment se comporte $\bot$ vis-à-vis de ces opérateurs ?

1. $[a_1, b_1] + [a_2, b_2] \stackrel{def}{=} [a_1 + a_2, b_1 + b_2]$

2. $-[a,b] \stackrel{def}{=} [-b,-a]$
3. $[a_1,b_1] * [a_2,b_2] \stackrel{def}{=} [\min(E),\max(E)]$ où $E \stackrel{def}{=} \{a_1*a_2, a_1*b_2, b_1*a_1, b_2*a_2\}$.
4. $[a,b]/2 = [a/2, b/2]$

$\bot$ est absorbant pour tout ces opérateurs.

▹ **Question 2.** On suppose deux variables x et y dans un état abstrait tel que $A(x) = [10,100]$ et $A(y) = [-10,5]$. Que vaut $A[x:=((x+y)*(-y))/2]$?

$A(x) + A(y) = [0,105]$; $-A(y) = [-5,10]$; $[0,105] * [-5,10] = [-525,1050]$.
Aucun calcule ne déborde ici.
In fine, $A[x:=((x+y)*(-y))/2] = A \oplus \{x \mapsto [-262,525]\}$.

▹ **Question 3.** Que vaut $A[x:=42]$?

$A[x:=42] = A \oplus \{x \mapsto [42,42]\}$.

Exercice 3 (Interprétation abstraite des gardes). Définir les gardes suivantes dans un contexte où x_1 et x_2 sont variables entières, et c est un littéral entier.

1. $A \sqcap (x_1 < c)$
2. $A \sqcap (x_1 < x_2)$
3. $A \sqcap (x_1 == x_2)$
4. $A \sqcap (x_1 \texttt{ != } x_2)$

On suppose $A(x_1) = [a_1,b_1]$ et $A(x_2) = [a_2,b_2]$.

1. $A \sqcap (x_1 < c) \stackrel{def}{=} A \sqcap \{x_1 \mapsto [M_-, c-1]\}$ (rem : si $c \leq a_1$, ici on retourne bien $\bot$).
2. $A \sqcap (x_1 < x_2) \stackrel{def}{=} A \sqcap \{x_1 \mapsto [M_-, b_2 - 1]; x_2 \mapsto [a_1 + 1, M_+]\}$
3. $A \sqcap (x_1 == x_2) \stackrel{def}{=} A \sqcap \{x_1 \mapsto A(x_2); x_2 \mapsto A(x_1)\}$
4. $A \sqcap (x_1 \texttt{ != } x_2) \stackrel{def}{=} (A \sqcap (x_1 < x_2)) \sqcup (A \sqcap (x_2 < x_1))$. rem : si x_1 et x_2 sont la même variable, ça retourne bien $\bot$.

Exercice 4 (Vérification d'une recherche dichotomique). On considère maintenant le code ci-dessous dans un contexte où `v` a une valeur dans **int** inconnue et "**int** `f(int x);`" est une fonction externe qu'on suppose sans effet de bord observable, comme EVA le fait en l'absence de spécification ACSL explicite.

```
1  l = 0;
2  u = 10;
3  while (l < u) {
4    m = (u + l) / 2;
5    r = f(m);
6    if (r < v) l = m + 1;
7    else if (r > v) u = m - 1;
8    else l=u=m;
9  }
10 if (l==u && v==f(l)) return l;
11 else return -1;
```

▹ **Question 1.** Insérer dans le programme les assertions RTE qui garantissent l'absence de débordement arithmétique. Puis, colorier ces assertions avec une analyse du programme qui prend $\sqcup$ comme opérateur d'élargissement (*widening* en anglais).

```
l = 0;   // l: [0,0]
u = 10 ; // u: [10,10]
// état initial d'avant la boucle l:[0,0]; u:[10,10]; m,r,v:⊤
// **1-ière itération**
if (l < u) {  // u, l inchangé
  assert (M_ <= u+l <= M+); // u+l: [10,10] ok (vert)
  m = (u + l) / 2; // m: [5,5]
  r = f(m); // r: ⊤ (on n'écrit plus les variables à ⊤)
  if (r < v) {
    assert (m+1 <= M+); // m+1: [6,6] ok
    l = m + 1; // l: [6,6]
  } else {
    if (r > v) {
      assert (M_ <= m-1); // m-1: [4,4] ok
      u = m - 1; // u: [4,4]
   } else {
      l=u=m; // l,u: [5,5]
   }
   // l: [0,5], u: [4, 5]
} // l: [0,6], u: [4, 10], m: [5,5]
// l'union avec l'état initial remet juste m à ⊤
// **2-ième itération**
if (l < u) {  // u, l inchangé
  assert (M_ <= u+l <= M+); // u+l: [4,16] ok
  m = (u + l) / 2; // m: [2,8]
  r = f(m);
  if (r < v) {
    assert (m+1 <= M+); // m+1: [3,9] ok
    l = m + 1; // l: [3,9]
  } else {
    if (r > v) {
      assert (M_ <= m-1); // m-1: [1,7] ok
      u = m - 1; // u: [1,7]
   } else {
```

```
      l=u=m; // l,u: [2,8]
   }
} // l: [0, 9], u: [1, 10]
// union avec état initial
// **3-ième itération**
if (l < u) {  // u, l inchangé
  assert (M- <= u+l <= M+); // u+l: [1,19] ok
  m = (u + l) / 2; // m: [0,9]
  r = f(m);
  if (r < v) {
    assert (m+1 <= M+); // m+1: [1,10] ok
    l = m + 1; // l: [1,10]
  } else {
    if (r > v) {
      assert (M- <= m-1); // m-1: [-1,8] ok
      u = m - 1; // u: [-1,8]
   } else {
      l=u=m; // l,u: [0,9]
   }
} // l: [0, 10], u: [-1, 10]
// union avec état initial
// 4-ième itération
if (l < u) {  // l: [0,9]; u: [1,10]
  ... // même calculs qu'au-dessus !
} // l: [0, 10], u: [-1, 10]
// on a trouvé un invariant de boucle
// on sort avec
assume (l >= u); // qui ne change rien.
// plus d'assert ensuite => les assert sont ok.
```

Conclusion : tous les asserts sont au vert !

▹ **Question 2.** Pour éviter un fameux bogue le jour où u sera initialisé à une valeur supérieure à 2^{30}, on change la ligne 4 en

```
m = l + (u - l) / 2;
```

Qu'est-ce que ça change pour notre analyse par intervalle (en conservant ici la valeur initiale `u==10`) ? Expliquer le phénomène. Comment améliorer la situation ?

Le assert devient

```
assert (M- <= u-l <= M+);
assert (M- <= l+(u-l)/2 <= M+);
```

A la première itération :

```
u-l: [10,10]
l+(u-l)/2: [5,5]
```

Donc inchangée.

A la 2-ième itération :

```
l: [0,6], u: [4, 10], -l:[-6, 0]
u-l: [-2,10]
l+(u-l)/2: [-1,11]
```

En sortie :

```
l: [0,12], u:[-2,10]
```

A la 3-ième itération (après `assume (l < u)`) :

```
l: [0,9], u: [1, 10], -l:[-9, 0]
u-l: [-9,10]
l+(u-l)/2: [-4,14]
```

En sortie :

```
l: [-3,15], u:[-5,13]
```

A la 4-ième itération (après `assume (l < u)`) :

```
l: [-3,12], u: [-2, 13], -l: [-12,3]
u-l: [-14,16]
l+(u-l)/2: [-10,20]
```

En sortie :

```
l: [-9,21], u:[-11,19]
```

Bref, ça diverge (ou plus exactement converge vers $\top$)... Et in fine, on colore les assert en orange ! Fausses alarmes... EVA fait la même chose !

Une première approximation vient du fait qu'on fait le calcul de `u-l` en intervalle sans prendre en compte de façon fine que `u > l` (et donc que c'est forcément strictement positif). L'autre problème c'est qu'à `u` fixé, quand `l` prend la valeur maximale dans son intervalle, `(u-l)/2` prend sa valeur minimale, ce que l'addition d'intervalle ne prend pas en compte (elle additionne les deux valeurs maximales qui, en fait, ne peuvent pas arriver en même temps). C'est une motivation pour prendre les domaines numériques plus précis que les intervalles, comme les polyèdres/polytopes (mais plus coûteux), ou alternativement incorporer des simplifications symboliques avant le calcul d'intervalle (cf. question suivante).

▹ **Question 3.** (Question optionnelle : à sauter si manque de temps). Pour corriger le manque de précision à la question précédente, on procède à des transformations symboliques sur les expressions contenant `/2` (car c'est un pattern qui apparaît dans les algos "diviser-pour-régner" qu'on aimerait supporter autant que possible). Ce genre de transformation symbolique est par exemple appliqué dans l'analyseur Astrée sur les patterns qu'on trouve en programmation embarquée (e.g. calculs flottants de traitement du signal).

Ici, nos transformation symboliques consistent d'abord à remplacer `/2` avec une fraction exacte, avec `x/2`$= \frac{\texttt{x}-\texttt{x\%2}}{2}$. On "simplifie" ensuite les polynômes multivariés sur le corps $\mathbb{Q}$. Dans notre exemple, cela conduit à "simplifier" `l+(u-l)/2` en $\frac{1}{2}\texttt{l} + \frac{1}{2}\texttt{u} + \frac{-1}{2}(\texttt{u} - \texttt{l})\%2$.

On peut alors sur-approximer le calcul d'un tel polynôme dans les intervalles de $\mathbb{Q} \times \mathbb{Q}$. Typiquement, on prend $[a,b]\%2 = [-1,1]$ quand $a < 0$ et $b > 0$. Finalement, au moment de l'affectation, on restreint l'intervalle de rationnels $[q_1, q_2]$ en intervalles d'entiers $[\lceil q_1 \rceil, \lfloor q_2 \rfloor]$ (puisqu'on sait que la valeur exacte est un entier de $[q_1, q_2]$).
Refaites l'analyse du programme ainsi transformé.

A la 1°itération, `l:[0,0]; u:[10,10]`, ici, on prend $[10,10]\%2 = [0,0]$ - on retrouve la 1° itération de la 1° question.

A la 2°itération. Pour `l: [0,6], u: [4, 10]`, on obtient : m : $\frac{[4,16]+[-1,1]}{2} = [2,8]$ - on retrouve la 2° itération de la 1° question.

A la 3°itération. Pour `l: [0,9], u: [1, 10]`, on obtient : m : $\frac{[1,19]+[-1,1]}{2} = [0,10]$.

A la 4°itération, l'état d'entrée de la boucle est donc `l: [0,11], u: [-1, 10]`. A l'entrée du corps de boucle, on retombe donc sur : `l: [0,9], u: [1, 10]`. Au final, on converge sur l'invariant `l: [0,11], u: [-1, 10]`. Et les assert sont coloriés en vert.

Exercice 5 (Opérateurs d'élargissements et rétrécissement). On considère le code ci-dessous, dans un contexte où `N` est un littéral entier fixé.

```
1 i = 0;
2 while (i < N) i++;
3 assert (i==N);
```

▹ **Question 1.** Faire l'analyse quand `N==4` avec $\sqcup$ comme opérateur d'élargissement.

```
i = 0; // i: [0,0]
if (i < N) i++; //i: [1,1]
// i: [0,1]
if (i < N) i++; //i: [1,2]
// i: [0,2]
if (i < N) i++; //i: [1,3]
// i: [0,3]
if (i < N) i++; //i: [1,4]
// i: [0,4]
if (i < N) // i: [0,3]
  i++; // i: [1,4]
// i: [0,4] // <- invariant de boucle
// sortie de la boucle
if (N <= i) // i:[4,4]
  assert (i==N); // vert !
```

▹ **Question 2.** Faire la même question avec `N==10000`. Pour accélérer la convergence, on prend

comme opérateur d'élargissement

$$[a_1,b_1]\nabla[a_2,b_2] \stackrel{def}{=} \begin{cases} [a_1,b_1] & \text{si } a_1 \leq a_2 \text{ et } b_2 \leq b_1 \\ [a_1,M_+] & \text{si } a_1 \leq a_2 \text{ et } b_1 \leq b_2 \\ [M_-,b_1] & \text{si } a_2 \leq a_1 \text{ et } b_2 \leq b_1 \\ \top & \text{sinon} \end{cases}$$

```
i = 0; // i: [0,0]
if (i < N) i++; //i: [1,1]
// i: [0,M+] // <- élargissement
if (i < N) i++; //i: [1,N]
// i: [0,M+] // <- invariant de boucle
// sortie de la boucle
if (N <= i) // i:[N, M+]
  assert (i==N); // orange !
```

On a donc une fausse alarme

▹ **Question 3.** Pour gagner en précision, on ne s'arrête plus au premier invariant de boucle J trouvé par élargissement : on essaye d'abord de voir si $A_0 \sqcup A_1$ est un invariant de boucle où A_0 est l'état initial de la boucle et A_1 l'état obtenu après un tour de boucle à partir de J. En effet, comme J est un invariant, on sait que $A_0 \sqsubseteq J$ et $A_1 \sqsubseteq J$. Donc, dans un treillis, $A_0 \sqcup A_1 \sqsubseteq J$. Par conséquent, $A_0 \sqcup A_1$ s'appelle un *rétrécissement* de J. Reste à savoir si c'est lui aussi un invariant de boucle...

Commencez par ajouter au système d'inférence ${}^{\#}\{A\}\,S\,\{A'\}$ une règle qui vérifie qu'un état abstrait I est un invariant de boucle, avant de le propager pour la suite du calcul. Puis appliquer votre règle pour mettre en œuvre la suggestion ci-dessus.

La règle s'obtient directement en adaptant celle du strong-post.

$$\frac{{}^{\#}\{I\}\,S\,\{A_1\} \quad A_0 \sqsubseteq I}{{}^{\#}\{A_0\}\,S^*\,\{I\} \quad A_1 \sqsubseteq I}$$

Sur l'exemple précédent :

```
i = 0; // i: [0,0]
if (i < N) i++; //i: [1,1]
// i: [0,M+] // <- élargissement
if (i < N) i++; //i: [1,N]
// i: [0, N] // <- rétrécissement
if (i < N) i++; //i: [1,N]
// i: [0, N] // <- invariant de boucle
// sortie de la boucle
if (N <= i) // i:[N, N]
  assert (i==N); // vert !
```

On a donc fait 3 itérations en tout !