Interprétation abstraite

Projet INF441

L'objectif de ce sujet est d'implémenter un outil de vérification de programmes utilisant un type d'analyse appelé *interprétation abstraite*. Lors de l'exécution d'un programme, certaines instructions peuvent produire des erreurs : par exemple, si l'on essaye de diviser un nombre par zéro (c'est la seule source d'erreur que l'on considérera ici). On voudrait s'assurer que de telles erreurs n'arriveront jamais, y compris quand les valeurs de certaines variables ne sont pas connues précisément (on sait simplement qu'elles résident dans un intervalle donné). Un cas typique d'utilisation est un système embarqué : à partir de la valeur de capteurs (donnant par exemple la vitesse dont on sait qu'elle est comprise entre 0 et 300 m/s), le programme de commande d'un train règle la puissance des moteurs ; il serait dommage qu'un tel programme contienne un bug.

On considère un langage de programmation simplifié, constitué à partir des constructions suivantes. Les seules valeurs manipulées sont des flottants. Une *expression* e est soit une variable (`x`, `y`, ...), soit un nombre flottant, soit de la forme

$$e_1\texttt{+}e_2 \qquad e_1\texttt{-}e_2 \qquad e_1\texttt{*}e_2 \qquad e_1\texttt{/}e_2 \qquad \texttt{rand(}a\texttt{,}b\texttt{)}$$

où e_1 et e_2 sont des expressions et a et b sont des flottants (les quatre premières expressions ont leur sens arithmétique habituel et `rand(`a`,`b`)` produit aléatoirement un flottant compris, au sens large, entre a et b). Un *programme* p correspond à l'une des actions suivantes :

— on met le résultat de l'évaluation de e dans la variable `x`

$$\texttt{x = } e$$

— on exécute le programme p_1 puis le programme p_2 :

$$p_1 \texttt{ ; } p_2$$

— on exécute p_1 si e est non nul, sinon on exécute p_2 :

$$\texttt{if } e \texttt{ then } p_1 \texttt{ else } p_2$$

On considérera qu'une variable non initialisée peut contenir n'importe quelle valeur et on négligera les erreurs d'arrondi dues aux calculs flottants (on assimilera les flottants à l'ensemble $\mathbb{R}$ des réels, et les opérations aux opérations usuelles).

On supposera que le résultat du programme est toujours stocké dans la variable `x`. Votre analyseur devra déterminer deux choses :

1. une sur-approximation de l'ensemble des valeurs que peut prendre `x` à la fin de l'exécution du programme,
2. si une erreur (c'est-à-dire une division par 0) peut survenir lors de l'exécution de votre programme.

Les conventions d'affichage sont précisées à la Section 5. On s'efforcera de produire un code modulaire et de fournir des programmes d'exemple montrant le bon fonctionnement de votre projet, ainsi que ses limites (théoriques ou pratiques).

1 Simulateur

La première tâche consistera à écrire un simulateur pour le langage : étant donné un programme, votre programme devra soit afficher la valeur de la variable `x` à la fin de l'exécution, soit `ERROR` si une erreur est survenue lors de l'exécution. On s'efforcera de placer l'évaluation des fonctions dans un module (ou une classe) qui pourra être remplacé par un autre par la suite. Notons que du fait de la présence de l'opération `rand` l'exécution n'est pas déterministe, par exemple le programme

```
y = rand(1,2);
z = rand(0,1);
x = (y/z)-1
```

pourra donner lieu aux valeurs `36.78`, `1` ou `ERROR`. De même,

```
y = rand(-1,1);
if y then
  x = 5
else
  x = 1/y
```

pourra afficher `5.` ou `ERROR`.

2 Évaluation par signes

En pratique, dans l'exemple précédent on n'observe quasiment jamais l'erreur, alors qu'elle peut se produire. On souhaiterait donc mettre au point une méthode *correcte* de vérification de programmes : il se peut qu'elle déclare qu'un programme peut mener à une erreur (`MAYBE ERROR`) alors que ça n'est pas le cas (ce qu'on appelle un *faux positif*), mais si elle déclare que le programme est correct (`NO ERROR`) alors il le sera quelle que soit l'exécution (en particulier quelle que soit la valeur produite par les opérations `rand`).

Plutôt que de faire un nombre infini de simulations correspondant à tous les cas possibles, l'idée va être de ne considérer qu'une information grossière sur les nombres. On considérera donc que les valeurs possibles (ce que l'on appelle le *domaine*) peuvent être les suivantes :

— `B` : non-défini
— `N` : strictement négatif
— `NZ` : négatif ou nul

— `Z` : nul
— `ZP` : nul ou positif
— `P` : strictement positif
— `NZP` : n'importe quel flottant

On évaluera les différentes opérations du programme pour ces valeurs, en cherchant le domaine le plus précis possible auquel le résultat va appartenir. Par exemple, si x et y sont `P` alors x+y sera `P` et x-y sera `NZP`. Dans le programme suivant

```
y = rand(-1,1);
z = y*y+1;
x = 1/z
```

on pourra déterminer rapidement que le résultat pour `x` est toujours `P` et qu'il n'y aura jamais d'erreur, quelle que soit la valeur de `y`. Votre analyseur devra donc afficher :

```
P
NO ERROR
```

De même, le programme

```
a = rand(-2,-1)
b = rand(1,2)
if c then
  y = a
else
  y = b
x = 1/y
```

affichera

```
NZP
MAYBE ERROR
```

En effet, la variable `c` étant non initialisée, elle peut prendre n'importe quelle valeur (elle vaut donc `NZP`) et à la fin de la boucle `y` peut valoir soit `N`, soit `P` suivant la branche prise. On considérera donc qu'il vaut `NZP`. Le calcul de `x` divise par `y` qui peut donc a priori être nul et donc a priori lever une erreur. Notez que votre analyseur sur-approxime l'ensemble des valeurs que peut prendre une variable et donc peut lever de fausses alertes (dans l'exemple précédent il considère que `y` est potentiellement nul, alors qu'il ne le sera jamais en réalité).

3 Évaluation par intervalles

La méthode précédente est relativement simple et efficace mais n'est pas très précise (elle détecte souvent des erreurs potentielles inexistantes). Par exemple, dans

```
y = rand(1,2);
z = 2*y - 1;
x = 1/z
```

votre programme déterminera que `z` vaut `NZP` et donc qu'il y a potentiellement une erreur lors du calcul de `x`. Afin d'améliorer la précision de l'analyse, on interprétera chaque expression par un intervalle $[a, b]$ de sorte que l'on soit sûr qu'il contienne la valeur pour chaque exécution : on utilisera pour cela l'arithmétique d'intervalles [1]. Par exemple, dans le programme ci-dessus, on déterminera successivement $\mathtt{y} \in [1, 2]$, $\mathtt{z} \in [1, 3]$ et $\mathtt{x} \in [1/3, 1]$, et on s'apercevra donc qu'il ne contient pas d'erreur : la sortie de votre analyseur sera

```
[0.333333333333,1.]
NO ERROR
```

Les formes d'intervalles possibles considérés dans ce domaine sont :
- $\emptyset$ (l'ensemble vide),
- $[a, b]$ avec $a, b \in \mathbb{R}$ tels que $a \leq b$,
- $]-\infty, a]$ ou $[a, \infty[$ avec $a \in \mathbb{R}$,
- $]-\infty, \infty[$.

4 Extensions

Il est demandé de réaliser au moins l'une des extensions proposées ci-dessous (au choix des candidats). La réalisation de plusieurs extensions sera bien sûr vue favorablement.

4.1 Erreurs certaines

Dans les analyses précédentes (signes et intervalles), votre analyseur détermine soit que le programme ne contient pas d'erreur (`NO ERROR`), soit qu'il contient peut-être une erreur (`MAYBE ERROR`). L'objectif de cette extension est, pour les deux analyses précédentes, de déterminer quand on peut être sûr qu'une erreur peut se produire (qu'il existe une erreur qui n'est pas un faux positif), et dans ce cas d'afficher `ERROR` au lieu de `MAYBE ERROR`.

4.2 Ensembles finis

L'objectif est d'implémenter un nouveau domaine d'analyse qui permet parfois d'obtenir des résultats plus précis que les précédents. On interprétera une variable, soit par un ensemble fini de valeurs flottantes, soit par $\mathbb{R}$ (signifiant n'importe quel flottant). Ainsi le programme

```
a = rand(-1,1)
if a then
  b = -2
else
  b = 2
x = 1/b
```

affichera

```
{-0.5,0.5}
NO ERROR
```

1. Voir par exemple `https://en.wikipedia.org/wiki/Interval_arithmetic`.

Comme `a` peut prendre un nombre infini de valeurs il sera interprété par $\mathbb{R}$, à la fin du branchement `b` sera interprété par `{-2,2}` et donc le calcul de `x` ne donnera pas lieu à une erreur. Notez que l'analyse par signe ou par intervalles déterminera `MAYBE ERROR` sur ce programme.

4.3 Boucles

Ajoutez un support pour les programmes contenant des boucles : on étendra la syntaxe des programmes par

`while` e `do` p `done`

dont l'effet est d'exécuter p tant que e est non nul. Par exemple

```
n = 5
y = rand(1,2)
while n do
  n = n-1
  y = y+1
done
x = 1/y
```

Dans les modes d'évaluation par signes et par intervalles (mais pas simulateur), on s'efforcera que le programme renvoie toujours un résultat au bout d'un temps fini (et court) : votre programme devra déterminer que le programme

```
n = 1
y = 0
while n do
  y = y + 1
done
x = 1 / n
```

ne contient pas d'erreur.

5 Conventions

Un squelette d'application vous est fourni en OCaml ou en Java. Celui-ci effectue une analyse grammaticale d'un programme, puis l'affiche. La version Java nécessite CUP [2].

Votre programme devra suivre les conventions suivantes. Il est important de les respecter car elles seront utilisées par les tests qui permettent d'évaluer votre projet. Le programme devra s'appeler `analyzer` et prendra un flag optionnel ainsi que le nom du fichier en paramètre. Par défaut,

```
analyzer test.prog
```

lancera la simulation du programme contenu dans le fichier `test.prog`. Les "flags" permettront de changer le mode, par exemple

2. Voir `https://www.cs.princeton.edu/~appel/modern/java/CUP/`, disponible dans le paquet `cup` sous Ubuntu ou Debian.

```
analyzer --sign test.prog
```

lancera l'analyse en utilisant le domaine des signes. Les flags qui devront être supportés sont

- `--sign` : analyse du signe (Section 2),
- `--interval` : analyse par intervalles (Section 3),
- `--set` : analyse ensembliste (Section 4.2).

La sortie de votre programme devra consister en exactement deux lignes :

1. les valeurs possibles pour `x` à la fin du programmes affichées sous les formes illustrées par les exemples suivants :
   - Section 1 : `5.23` ou rien (si une erreur s'est produite),
   - Section 2 : `B`, `N`, `NZ`, `Z`, `ZP`, `P` ou `NZP`,
   - Section 3 : `O/` (notation pour $\emptyset$), `[-2.,5.]`, `]-oo,4.]`, `[8,oo[` ou `]-oo,oo[`,
   - Section 4.2 : `{}` (ensemble vide), `{2.,6.,7.265}` ou `R` (notation pour $\mathbb{R}$).
2. l'une des chaînes suivantes :
   - `NO ERROR` si le programme ne contient pas d'erreur,
   - `MAYBE ERROR` si le programme pourrait donner lieu a une erreur,
   - `ERROR` si il existe une exécution dans la quelle le programme lève une erreur (extension de la Section 4.1).

On rappelle enfin l'importance

- d'écrire un code modulaire,
- de fournir des programmes de test simples.