Conception d'un analyseur statique simple

Jonathan Laurent

2 juin 2014

On décrit dans ce document les fonctionnalités et les caractéristiques techniques de l'analyseur statique `lisa` ainsi que les choix d'implémentation qui ont été réalisés.

1 Présentation des fonctionnalités

1.1 Lancement du programme et options

Pour compiler `lisa`, il suffit d'utiliser la commande `make`. La commande `make test` permet de générer trois fichiers `tests.`*`domain`*`.txt` dans le répertoire `tests` comparant pour chacun des tests fournis la sortie de `lisa` ainsi que la sortie de référence. `lisa` se lance avec la commande :

```
./lisa [--DOMAIN] [OPTIONS] INPUT_FILES
```

où `INPUT_FILES` est une liste de fichiers sources à analyser et où `DOMAIN` peut prendre les valeurs `polyhedra` (par défault), `consts` et `intervals`. Les options principales incluent les paramètres d'analyse des boucles (cf. deuxième cours d'interprétation abstraite)

Option	Valeur de n par défault
`--widening-delay` *n*	3
`--unrolling-factor` *n*	6
`--decreasing-iterations` *n*	3

ainsi que les options `auto-phantom-vars` et `--no-verbose-unrolled-statements` dont nous détaillerons ultérieurement l'utilisation.

1.2 Caractéristiques générales et fonctionnalités

`lisa` implémente toutes les fonctionnalités minimales imposées dans les consignes du projet, ainsi que :

— La gestion minimale des **variables locales**.
Toutes les variables doivent cependant avoir un nom distinct, ce dont il est possible de s'assurer lors de la construction de l'arbre syntaxique.
— L'implémentation d'un mécanisme personnel permettant de trouver de **meilleurs invariants de boucle** avec le domaine `polyhedra`, en générant automatiquement des variables fantômes.

Les opérations de division et de modulo sur les entiers ne sont pas implémentées pour les domaines `consts` et `intervals`.

1.2.1 Déroulement de l'analyse et sortie du programme

La sortie de `lisa` est très proche des sorties de références proposées avec le jeu de test : une ligne est affichée pour chaque instruction `print` ou chaque assertion qui échoue, dans l'ordre de parcours de l'arbre de syntaxe abstraite. L'instruction `print` utilisée sans arguments affiche tout le contexte local. L'état des variables globales à la fin du programme est également indiqué. Si la vérification d'une assertion échoue, une erreur est signalée mais l'assertion est supposée vraie pour la suite de l'analyse. C'est ainsi que le programme suivant provoque une seule erreur :

```
int x;
assert(X > 0);
assert(x >= 0);
```

Les instructions `print` et `assert` qui se trouvent dans le corps d'une boucle sont éxecutées pour les n premières itérations, où n est l'argument de l'option `unrolling-factor`, puis dans l'environnement constitué de l'invariant de boucle à partir de l'itération $n+1$. Il est possible de désactiver les n premiers affichages avec l'option `no-verbose-unrolled-statements`, au risque d'ignorer des violations d'assertions sur les premières itérations d'une boucle. Par exemple, `./lisa --no-verbose-unrolled-statements --unrolling-factor 1` ne signale aucune erreur sur le programme suivant :

```
int x = rand(1, 100);
while(x <= 100) {
  x = x + 1;
  assert(x >= 3);
 }
```

En effet, l'erreur levée lors du premier passage dans le corps de la boucle n'est pas signalée, et l'assertion est supposée vraie pour le reste de l'éxecution.

1.2.2 L'option `--auto-phantom-vars`

Reprenons un programme donné comme exemple dans le deuxième cours d'interprétation abstraite :

```
int v = 1;
while(v <= 50) {
  v = v + 2;
 }
```

Sur cet exemple, on trouve $v \in [51; 52]$ en utilisant le domaine `intervals` et en calculant au moins une itération décroissante. On ne fait pas mieux avec le domaine des polyhèdres, ce qui est assez décevant. Cependant, on peut obtenir avec ce dernier domaine un résultat bien plus satisfaisant en modifiant légèrement notre programme :

```
int v = 1;
int i = 0
while(v <= 50) {
  v = v + 2;
  i = i + 1;
 }
```

Ici, on trouve bien $v = 51$. En effet, l'invariant de boucle $v - 2i = 1$ est inféré ; de surcroît, i est facile à estimer exactement à l'aide d'une itération décroissante car il augmente par pas de 1. Plus généralement, l'option `--auto-phantom-vars` permet d'insérer automatiquement de tels compteurs de boucle, afin de donner la possibilité à la bibliothèque Apron de trouver de meilleurs invariants. Cela est réalisé de manière transparente, en modifiant directement l'arbre de syntaxte abstraite (cf. `ast_transformations.ml`).

2 Caractéristiques techniques et détails d'implémentation

2.1 Les interfaces `Value_domain` et `Environment_domain`

Les interfaces proposées lors du deuxième cours d'interprétation abstraite ont été utilisées, et elles aident à écrire facilement un code élégant et générique. L'interface `Value_domain.S` comprend en particulier deux fonctions `assume_leq` et `assume_gt` de signature

```
t -> t -> t * t
```

Toute implémentation doit vérifier

$$\texttt{assume\_leq}\ a_1\ a_2 = (a_1', a_2')$$

où

$$a_1' = \alpha\left(\{x \in \gamma(a_1) \mid \exists\, y \in \gamma(a_2),\ x \leq y\}\right)$$
$$a_2' = \alpha\left(\{y \in \gamma(a_2) \mid \exists\, x \in \gamma(a_1),\ x \leq y\}\right)$$

et il en est de même pour `assume_gt` avec l'inégalité stricte $>$. Intuitivement, ces deux fonctions servent à *raffiner* deux valeurs abstraites en tenant compte d'une inégalité sur leurs représentants. L'interface `Environment_domain.S` exporte également deux fonctions du même nom, mais dont la signature est différente :

```
t -> expr -> expr -> t
```

Ici, toute implémentation doit garantir

$$\mathbb{C}[e_1 \leq e_2\ ?]\,\rho \ \sqsubseteq\ \texttt{assume\_leq}\,\rho\ e_1\ e_2$$

et le membre de droite doit être de préférence le plus petit possible. Intuitivement, ces deux fonctions permettent de *raffiner* un domaine d'environnement à partir d'une inégalité.

2.2 Le domaine `Non_relational`

Le foncteur `Non_relational` prend un module de type `Value_domain` et l'étend point par point. Dans le cas de l'opération `widen`, on vérifie facilement que cette extension préserve l'impossibilité de construire des chaînes d'élargissement infinies strictement croissantes. Les implémentations de `assume_leq` et `assume_gt` sont les seules qui méritent d'être détaillées. `assume_leq` $\rho\ e_1\ e_2$ ne raffine la valeur d'une variable que si elle est seule d'un côté de l'inégalité. Si aucun des deux côtés n'est occupé par une variable, `assume_leq` $\rho\ e_1\ e_2$ retourne ρ ou $\bot$. Ce comportement n'est pas optimal pour gérer des situations du type

```
if(x + y <= z + t) {...}
```

Une extension possible serait d'appliquer des transformations linéaires à l'inégalité afin d'isoler le plus de variables possibles et d'*apprendre* en conséquence.

2.3 Le domaine `Intervals`

`Intervals.Make` est un foncteur qui attend un module de type `Intervals.PARAMS`. Ce motif est utilisé à plusieurs reprises dans le code de `lisa`. Le paramètre le plus significatif est la liste des *seuils d'élargissement*. Si cette liste est non vide, cela permet par exemple d'élargir à droite un intervalle par le plus petit entier de cette liste strictement supérieur à sa borne droite, plutôt que de faire sauter directement cette dernière à $+\infty$. La version actuelle de `lisa` n'utilise pas cette possibilité, qui peut cependant être activée en changeant une ligne dans le code source.

L'implémentation du domaine `Intervals` se fait avec des bornes rationnelles, ce qui permet de l'étendre pour raisonner sur des programmes dont les variables sont à valeurs dans $\mathbb{Q}$ (paramètre `integer_mode = false` ou option `-r` de `lisa`). Sans parler de la division et du modulo, la différence majeure entre la gestion du cas des variables rationnelles et celui des variables entières réside dans l'implémentation de `assume_gt`.

Si les variables sont à valeurs entières, on peut utiliser la propriété

$$\forall x, y \in \mathbb{Z} : x < y \iff x + 1 \leq y$$

pour déduire `assume_gt` de `assume_leq`. Dans le cas, des variables rationnelles, on pourrait vouloir se contenter de faire une sur-approximation de $>$ avec $\geq$, mais cela conduit à une perte de précision qui pose problème notamment dès lors que l'on veut gérer correctement le problème de l'égalité dans les conditions. La meilleure solution serait d'étendre notre domaine à des intervalles dont les bornes peuvent être ouvertes. Dans le cas où on refuse ce changement, il est au moins nécessaire de traiter des cas particuliers du type

$$a < b \implies \texttt{assume\_gt}\ [b; b]\ [a; b] = (\bot, [a; b])$$

2.4 Le domaine `Polyhedra`

Le module `Polyhedra` est une encapsulation directe de la bibliothèque Apron. Par chance, le binding OCaml d'Apron n'exporte que des structures de données persistantes, ce qui facilite considérablement le travail.

2.5 Le module `Analysis`

L'analyse proprement dite du programme source est réalisée par le foncteur `Analysis`, qui prend notamment pour paramètre le domaine d'environnement à utiliser. Les deux problèmes majeurs que posent ce module sont la gestion des boucles et des conditions. Toutes les optimisations suggérées dans le cours pour le traitement des boucles sont implémentées.

La fonction `process_cond` prend pour argument un environnement et une expression c. Elle retourne deux environnements obtenus respectivement en ajoutant les contraintes c et $\neg c$. Cette signature se prête très bien à une définition par récurrence sur c, les cas de base correspondant à une utilisation des fonctions `assume_leq` et `assume_gt`.

3 Résultats expérimentaux

Comme il est possible de le vérifier en exécutant `make test`, `lisa` donne les même sorties que le programme de référence sur tous les tests avec les domaines `intervals` et `consts`. Avec le domaine `polyhedra`, on trouve deux différences :

— Le test *0602_rate_limiter.c* réussit

— Le test *0601_heap_sort.c* échoue

On note parfois des écarts de temps d'exécution étonnants entre deux programmes très proches. L'analyse du programme *heap_sort* est ainsi plus lente d'un ordre de grandeur 10 avec l'option `--auto-phantom-vars`. L'utilisation du profiler `gprof` semble indiquer que la bibliothèque Apron est en cause.