Analyseur statique par interprétation abstraite
Projet du cours TAS — Master 2 STL

Antoine Miné & Sarah Zennou

2016–2017

1 Introduction

Le but du projet est de se familiariser à la conception et à la programmation des analyseurs statiques par interprétation abstraite. Pour cela, nous étendrons un petit analyseur permettant l'analyse de valeurs sur un langage jouet, à la syntaxe inspirée de C mais extrêmement simplifiée. Notre langage ne comporte, comme type de données, que les entiers mathématiques (dans $\mathbb{Z}$) et, comme structures de contrôle, le `if then else` et la boucle `while` (donc, ni pointeur, ni fonction, ni tableau, etc.).

Code de base. Afin de pouvoir nous concentrer sur la partie intéressante, l'analyse proprement dite, un code de base en OCaml est fourni. Ce code comporte en particulier :

- un analyseur syntaxique, qui transforme les programmes textes en arbres syntaxiques ;
- un interprète (presque complet) générique ; comme dans le cours, il s'agit d'un interprète par induction sur l'arbre syntaxique, et il est paramétré par le choix d'un domaine abstrait qui donne l'interprétation des expressions du langage ;
- quelques domaines d'analyse (un domaine concret, et une partie du domaine des constantes, vus en cours).

Technologie. L'analyseur utilise les technologies suivantes :

- le langage OCaml (`https://ocaml.org/index.fr.html`) ;
- Menhir, pour l'analyse syntaxique (`http://gallium.inria.fr/~fpottier/menhir`) ;
- Zarith, une bibliothèque de grands entiers, qui permettra d'éviter de se soucier des dépassements de taille maximale des entiers OCaml lors de l'analyse (`https://forge.ocamlcore.org/projects/zarith`) ;
- GMP, bibliothèque d'entiers longs en C, sur laquelle Zarith est adossée (`https://gmplib.org`).

Machine virtuelle. Le code est fourni dans une machine virtuelle VirtualBox (format portable Open Virtualization Format `.ova`) contenant une distribution Ubuntu 64-bit où ces dépendances sont pré-installées. Pour installer la machine virtuelle, choisissez le menu *Fichier* → *Importer une machine virtuelle* dans VirtualBox. Une fois la machine virtuelle lancée, connectez-vous au compte `tas` avec le mot de passe `tas` ; allez dans le répertoire `projet-tas/src` et tapez `make`.

Alternativement, vous pouvez télécharger sur la page du cours (en suivant les liens depuis le site du Master STL), une archive des sources, à utiliser de préférence sur une machine Linux (voir le fichier `INSTALL.txt` pour la liste précise de dépendances).

Travail demandé. Le projet est réalisé *par groupes de deux élèves*. La section 3 décrit les tâches que tous les groupes doivent obligatoirement réaliser. Par ailleurs, la section 4 décrit plusieurs extensions. Chaque groupe devra réaliser *une extension proposée en section 4*, en plus de *l'intégralité de la section 3*. Ces tâches sont initiées en TME et doivent être achevées chez soi. À la fin du projet, le jour de l'examen final, vous devrez rendre une archive contenant :

— les sources complètes de votre analyseur, compilables simplement avec la commande `make` ;

— un rapport en texte brut dans un fichier `README` qui liste les fonctionnalités de l'analyseur, la liste des options comprises, les extensions que vous avez apportées par rapport au sujet, et éventuellement les difficultés que vous auriez rencontrées ou les parties de votre analyseur qui ne fonctionnent pas de manière satisfaisante ;

— une banque de tests additionnels qui illustrent les réponses aux questions demandées, ainsi que les fichiers de sortie de votre analyseur sur ces tests.

2 Architecture du projet

Le projet est distribué principalement sous la forme d'une machine virtuelle Ubuntu 64-bit, utilisable directement avec VirtualBox, mais vous pouvez également installer les sources du projet sur une autre machine (Linux conseillé), à condition d'avoir installé les dépendances décrites plus haut (avec `apt-get` ou `opam`).

Dans le répertoire du projet, lancer `make` devrait compiler les sources et générer un fichier `analyzer.byte` (l'analyseur compilé en byte-code OCaml).

2.1 Fichiers OCaml

L'arborescence des sources est la suivante :

— `Makefile` : compilation de l'analyseur, à modifier au fur et à mesure que vous ajoutez des sources ;

— `main.ml` : point d'entrée, analyse des options en ligne de commande, lancement de l'analyse syntaxique puis sémantique ; à modifier pour brancher des nouvelles analyses et pour ajouter des options ;

— `libs/` : contient une version légèrement améliorée du module `Map` d'OCaml ; nous verrons en cours la justification de cette extension ;

— `frontend/` : transformation du source (texte) en arbre syntaxique :

  — `frontend/abstract_syntax_tree.ml` : type des arbres syntaxiques ;

  — `frontend/lexer.mll` : analyseur lexical OCamlLex ;

  — `frontend/parser.mly` : analyseur syntaxique Menhir ;

  — `frontend/file_parser.ml` : point d'entrée pour la transformation du source en arbre syntaxique ;

  — `frontend/abstract_syntax_printer.ml` : transformation inverse, affichage d'un arbre syntaxique sous forme de sources ;

— `domains/` : domaines d'interprétation de la sémantique ;

  — `domains/domain.ml` : signature des domaines représentant des ensembles d'environnements ;

  — `domains/concrete_domain.ml` : domaine concret, les environnements sont représentés comme des ensembles de tables, associant à chaque variable sa valeur ;

  — `domains/value_domain.ml` : signature des domaines représentants des ensembles d'entiers ;

  - `domains/constant_domain.ml` : un exemple de domaine d'ensembles d'entiers (donc obéissant à la signature `Value_domain.VALUE_DOMAIN`) : le domaine des constantes, vu en cours ;
  - `domains/non_relational_domain.ml` : un *foncteur* qui, étant donné un domaine représentant des ensembles d'entiers (`Value_domain.VALUE_DOMAIN`), construit un domaine représentant des ensembles d'environnements (`Domain.DOMAIN`), en associant à chaque variable un ensemble d'entiers abstrait ;
- `interpreter/interpreter.ml` : interprète générique des programmes, paramétré par un domaine d'environnements (`Domain.DOMAIN`) ;
- `tests/` : un ensemble de programmes dans le langage analysé, pour tester votre analyseur.
- `tests/tests-constant/`, `tests/result-interval/` : les résultats d'analyse, obtenus avec un analyseur de référence (non fourni !) et pouvant servir de point de comparaison avec à votre analyseur.

2.2 Syntaxe du langage

Le langage jouet obéit à une syntaxe décrite dans le fichier de grammaire `parser.mly`. Nous la décrivons succinctement, sachant que les exemples du répertoire `tests/` vous permettent également de vous familiariser avec la syntaxe. Par ailleurs, le fichier `abstract_syntax_tree.ml` donne une idée précise des constructions du langage.

Un programme est une suite d'instructions :

- tests : `if (bexpr) { block }` ou `if (bexpr) { block } else { block }` ;
- boucles : `while (bexpr) { block }` ;
- affectations : `var = expr` ;
- blocs : `{ decl1; ...; decln; inst1; ...; instn }` composés d'une suite de déclarations de variables `int var` et d'une suite d'instructions ; seul le type `int` est reconnu ; les déclarations n'ont pas d'initialisation (il faut faire suivre d'une affectation) ; on ne peut déclarer qu'une variable à la fois (`int a,b;` ne marche pas, il faut écrire `int a; int b;`) ; toutes les déclarations doivent précéder, dans le bloc, toutes les instructions ; il n'y a pas de variable globale, toute variable doit être déclarée dans un bloc ;
- les expressions entières, utilisées dans les affectations, sont composées des opérateurs classiques : `+`, `-`, `*`, `/`, des variables, des constantes, plus une opération particulière, `rand(l,h)`, où `l` et `h` sont deux entiers, et qui représente l'ensemble des entiers entre `l` et `h` ;
- les expressions booléennes, utilisées dans les tests et le boucles, sont composées des opérateurs `&&`, `||`, `!`, des constantes `true` et `false`, et de la comparaison de deux expressions entières grâce aux opérateurs `<`, `<=`, `>`, `>=`, `==`, `!=` ;
- `print(var1,...,varn)` permet d'afficher la valeur des variables `var1` à `varn` ;
- `halt` arrête le programme ;
- `assert(bexpr)` arrête le programme sur un message d'erreur si la condition booléenne n'est pas vérifiée, et continue l'exécution normalement sinon.

Un exemple simple de programme valide est : `{ int x; x = 2+2; print(x); }`

2.3 Utilisation

Le programme `analyzer.byte` compilé avec `make` prend en argument un fichier source (ou plusieurs). Le comportement par défaut est d'afficher le source du programme (après l'avoir converti d'abord en arbre syntaxique, puis en code source textuel). D'autre options sont introduites en cours de TME.

3 Travail demandé

3.1 Prise en main, domaine concret

L'option `-concrete` permet une exécution du programme dans la sémantique concrète collectrice. Note : vous pouvez également utiliser l'option `-trace` pour observer le déroulement des calculs (affichage de l'environnement après chaque instruction).

3.1.1 Observation

Lancez l'analyse concrète sur les exemples fournis, et créez vos exemples de tests. Le but est de répondre aux questions suivantes concernant la sémantique des programmes et leur adéquation avec le comportement de l'interprète concret, et de valider vos réponses en testant :

— quelles est la sémantique de l'instruction `rand(l,h)` dans un programme ? quel est le résultat attendu de l'interprète ?

— sous quelles conditions l'exécution d'un programme s'arrête-t-elle ? quel est alors le résultat de l'interprète ?

— si le programme comporte une boucle infinie, est-il possible que l'interprète termine tout de même ? dans quels cas ?

3.1.2 Assertions

Vous avez sans doute remarqué lors de vos tests que l'instruction `assert` se comporte comme une instruction `skip` : elle ne fait rien. Dans cette question, vous modifierez `interpreter.ml` pour corriger son interprétation, c'est à dire :

— afficher un message d'erreur si l'assertion n'est pas prouvée correcte ;

— et continuer l'analyse en la supposant correcte (ceci afin de ne pas indiquer à l'utilisateur plusieurs erreurs ayant la même cause).

3.1.3 Enrichissement

Implantez une des deux extensions suivantes :

— ajoutez une opération modulo `%` au langage (il sera nécessaire de modifier légèrement l'analyse lexicale, syntaxique, l'arbre syntaxique et le domaine d'interprétation ; conseil : partez d'une opération existante, comme la multiplication, pour suivre le fil des modifications à apporter) ;

— le type `int` du programme correspond à des entiers mathématiques parfaits ; modifiez cette interprétation dans `concrete_domain.ml` pour correspondre à des entiers 32-bit signés et illustrez avec des exemples de programmes où le comportement diffère (conseil : on peut voir une opération sur 32-bit comme une opération sur les entiers mathématiques, suivie d'une opération de correction qui ramène le résultat dans $[-2^{31}, 2^{31} - 1]$; il suffit donc d'ajouter cette étape après chaque calcul).

3.2 Domaine des constantes

L'analyse des constantes est accessible avec l'option `-constant`. Cependant, le domaine n'est pas complet. Le but de l'exercice est de le compléter. Vous vous intéresserez en particulier au résultat des tests suivants :

— `0024_mul_rand.c`

— `0200_if_true.c`

— `0201_if_false.c`

— `0209_cmp_eq_ne.c`

Dans chacun des cas, déterminez dans `constant_domain.ml` la source de l'imprécision et corrigez-là.

Par ailleurs, le traitement de la division n'est pas aussi précis qu'il pourrait l'être. Déterminez et corrigez cette imprécision. Proposez des tests mettant en valeur votre correction.

3.3 Domaine des intervalles

Dans cet exercice vous implanterez le domaine des intervalles vu en cours. Comme le domaine des constantes, il obéit à la signature `Value_domain.VALUE_DOMAIN` et sert de paramètre au foncteur `Non_relational_domain.NonRelational`. Faites attention à ce que l'on gère des entiers mathématiques arbitraires. Les bornes des intervalles ne sont donc pas forcément des entiers, mais peuvent être aussi $+\infty$ ou $-\infty$.

La signature `Value_domain.VALUE_DOMAIN` comporte de nombreuses fonctions. Vous implanterez au moins de la manière la plus précise possible les fonctions suivantes : `top`, `bottom`, `const`, `rand`, `meet`, `join`, `subset`, `is_bottom`, `print`, `unary`, `binary`, `compare`. Pour les fonctions `bwd_unary` et `bwd_binary`, une implantation approchée suffira. Néanmoins, il est **indispensable** que toutes les fonctions renvoient un résultat **sûr**, même si il est imprécis.

3.4 Analyse de boucles

Le traitement des boucles dans `interpreter.ml` suppose que le domaine abstrait n'a pas de chaîne infinie strictement croissante. Que se passe-t-il alors lors d'une analyse d'intervalles ?

Le but de la question est de corriger ce problème en ajoutant l'utilisation des élargissements. Nous procéderons par étapes :

1. assurez-vous que l'opération `widen` est bien implantée dans le domaine des intervalles ;
2. modifiez `interpreter.ml` pour que l'opération d'élargissement soit utilisée à tous les tours de boucle ;
3. ajoutez une option `-delay` n, permettant de remplacer les n premières applications de l'élargissement par une union (élargissement retardé) ;
4. ajoutez une option `-unroll` n, permettant de dérouler les n premiers tours de boucle avant le calcul avec élargissement ; quelle différence avec `-delay` n ? (illustrez à l'aide d'exemples) ;
5. ajoutez des itérations décroissantes pour raffiner le résultat (illustrez également le gain de précision par des exemples).

3.5 Produit réduit

Implantez le domaine de parité, mentionné en cours, puis le produit réduit des intervalles avec la parité. Proposez des exemples de programmes montrant l'intérêt de cette réduction. Essayez, autant que possible, de définir un foncteur générique "produit réduit" prenant en argument deux domaines abstraits de valeurs arbitraires.

4 Extensions

Cette section propose plusieurs améliorations que vous pourrez apporter à votre analyseur.

Il est demandé dans le projet de choisir et d'implanter une de ces extension, en plus de l'intégralité de la section 3. Vous pouvez alternativement proposer une extension non décrite ici et l'implanter, à condition que votre choix soit validé par le chargé de TME.

4.1 Analyse des entiers machine

En complément de la question 3.1.3, modifiez *tous* les domaines implantés (constantes, intervalles, parité) pour que la sémantique corresponde à un calcul dans des entiers signés 32-bit, et non dans les entiers mathématiques. Montrez sur des exemples la différence entre ces deux sémantiques, et en particulier l'impact en terme de précision de l'analyse.

4.2 Analyse disjonctive

L'analyse des intervalles est imprécise car elle ne représente que des ensembles de valeurs convexes. Nous verrons en cours plusieurs constructions permettant de corriger ce problème, en raisonnant sur des disjonctions d'intervalles : complétion disjonctive, partitionnement d'états, partitionnement de traces. Implantez une de ces techniques dans votre analyseur, et proposez des exemples illustrant l'amélioration de la précision qu'elle apporte.

4.3 Analyse relationnelle

Ajoutez le support pour les domaines numériques relationnels. Vous pourrez vous appuyer sur la bibliothèque Apron, qui propose des implantations toutes faites des octogones et des polyèdres, et possède une interface OCaml. Proposez des exemples illustrant l'amélioration de la précision.

Afin d'utiliser Apron, les dépendances suivantes seront nécessaires. Là encore, les dépendances sont pré-installées sur l'image virtuelle fournie lors du premier TME.

— CamlIDL : générateur d'interfaces OCaml pour bibliothèques C (`https://forge.ocamlcore.org/projects/camlidl`) ;

— MPFR : nombres flottants en précision arbitraire (`http://www.mpfr.org`) ;

— Apron : bibliothèque de domaines abstraits (`http://apron.cri.ensmp.fr/library`).

4.4 Analyse de tableaux

Ajoutez le support dans votre langage et dans votre analyse pour les tableaux. Chaque tableau sera déclaré avec une taille fixe, par exemple : `int tab[10]`. Lors d'un accès dans un tableau `tab[expr]`, nous nous intéressons à :

1. vérifier que l'expression `expr` représente bien un indice valide du tableau, c'est à dire s'évalue en une valeur entre 0 et $\texttt{n} - 1$ (sinon, une erreur est affichée, à la manière d'un échec d'assertion) ;
2. inférer des informations sur les valeurs contenues dans le tableau (par exemple, un intervalle de valeurs).

Pour le deuxième point, deux représentations abstraites d'un tableau sont possibles :

— traiter chaque case `tab[0], ..., tab[n-1]` comme une variable indépendante, et lui associer un intervalle ;

— ou utiliser une seule variable `tab[*]` et un unique intervalle par tableau qui représente l'ensemble de toutes les valeurs possibles de toutes les cases du tableau.

Vous implanterez ces deux techniques et proposerez des exemples pour illustrer la différence de précision et de coût entre les deux.