Projet de programmation: analyseur statique par interprétation abstraite pour un langage numérique simple

Antoine Miné

11 septembre 2018

1 Introduction

Le but du projet est de se familiariser à la conception et à la programmation des analyseurs statiques par interprétation abstraite, en utilisant le langage OCaml. Pour cela, nous étendons un petit analyseur permettant l'analyse de valeurs sur un langage jouet impératif très simple. La syntaxe est inspirée de C, mais est extrêmement simplifiée. Le 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`. La langage ne comporte ni pointeur, ni fonction, ni tableau, ni allocation dynamique, ni objet.

Le projet suppose des connaissances préalables en interprétation abstraite et en programmation en OCaml. Concernant la partie interprétation abstraite, deux supports de cours possibles sont :

— le cours M2-6 du MPRI à Paris 7 (en anglais) ;

— le cours TAS du Master STL à Sorbonne Université (en français).

Le projet est inspiré d'un projet donné à l'École normale supérieure, puis dans le Master STL de Sorbonne Université.

Code de base. Vos trouverez dans l'archive du projet un squelette de base pour faciliter le développement de l'analyse :

— un analyseur syntaxique qui transforme le texte du programme en arbre syntaxique abstrait ;

— un interprète par induction sur la syntaxe, paramétré par le choix d'un domaine d'interprétation ;

— des signatures pour les domaines d'environnements et les domaines de valeurs ;

— le domaine concret, permettant de collecter l'ensemble précis des états de programme accessibles ;

— le domaine abstrait des constantes.

L'interprète et le domaine des constantes sont encore incomplets. Une première tâche sera donc de les compléter.

Dépendances. Les dépendances suivantes doivent être installées pour pouvoir compiler le projet :

— le langage OCaml ;

— Menhir : un générateur d'analyseurs syntaxiques pour OCaml ;

— GMP : une bibliothèque C d'entiers multiprécision (nécessaire pour Zarith et Apron) ;

— MPFR : une bibliothèque C de flottants multiprécision (nécessaire pour Apron) ;

— Zarith : une bibliothèque OCaml d'entiers multiprécision ;

— CamlIDL : une bibliothèque OCaml d'interfaçage avec le C ;

— Apron : une bibliothèque C/OCaml de domaines numériques.

Sous Ubuntu (et distributions dérivées), l'installation des dépendances peut se faire avec `apt-get` et Opam :

```
sudo apt-get update
sudo apt-get install -y m4 libgmp3-dev libmpfr-dev
sudo apt-get install -y ocaml ocaml-native-compilers ocaml-findlib opam
opam init -y
opam config -y env
opam install -y menhir zarith mlgmpidl apron
```

Compilation et test. Après installation des dépendances, faire `make` pour compiler. L'exécutable généré est `analyzer.byte`. En cas de succès de la compilation, vous pouvez tester le binaire :

- `./analyzer.bytetests/01_concrete/0111_rand.c` doit afficher sur la console le texte du programme `tests/01_concrete/0111_rand.c` (en réalité, le programme a été transformé en AST par le parseur et reconverti en texte) ;
- `./analyzer.bytetests/01_concrete/0111_rand.c-concrete` doit afficher sur la console le résultat de toutes les exécutions possibles du programme de test, ici, le fait que `x` vaut une valeur entre 1 et 5.

2 Architecture du projet

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 ;
- `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 représentant 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 ;
  - `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 votre analyseur.

3 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 :

```
1 {
2   int x;
3   x = 2+2;
4   print(x);
5 }
```

Pour plus d'informations sur la syntaxe, vous pouvez consulter le fichier d'analyse syntaxique `src/frontend/parser.mly`. Vous trouverez également des exemples de programmes dans le répertoire `tests`.

4 Travail demandé

4.1 Prise en main, domaine concret

L'option `-concrete` permet une exécution du programme dans la sémantique concrète collectrice.

Vous pouvez également utiliser l'option `-trace` pour observer le déroulement des calculs (affichage de l'environnement après chaque instruction).

4.1.1 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 ?

4.1.2 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).

4.1.3 Enrichissement

Implantez les 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).

4.2 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`

— `0100_if_true.c`

— `0101_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.

4.3 Domaine des intervalles

Dans cet exercice vous implanterez le domaine des intervalles. 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.

4.4 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).

4.5 Produit réduit

Implantez le domaine des parités, permettant d'inférer pour chaque variable si elle est paire ou impaire. Implantez ensuite 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.

5 Extensions

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

5.1 Analyse des entiers machine

En complément de la question 4.1.3, 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.

5.2 Analyse disjonctive

L'analyse des intervalles est imprécise car elle ne représente que des ensembles de valeurs convexes. Plusieurs constructions permettent 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.

5.3 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.

5.4 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 `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.

5.5 Analyse de pointeurs

Ajoutez le support dans votre langage et dans votre analyse pour les pointeurs.

Une variable pointeur sera déclarée avec `ptr p`. Si `x` est une variable entière, alors une référence sur `x` peut être stockée dans `p` par l'instruction `p = &x`. La variable référencée par `p` peut être lue par `*p`, utilisable au sein de toute expression (on peut par exemple écrire `x = *p + 1`). La variable référencée par `p` peut être modifiée par `*p = expression`. Enfin, si `q` est également une référence, il est possible de la copier dans `p` par `p = q` (ainsi, `*p` et `*q` dénotent la même variable).

Lire ou modifier la valeur référencée par un pointeur non initialisé (entre `ptr p` et la première affectation `p = ...`) provoque une erreur. Par ailleurs, si `p` référence une variable `x` déclarée dans un bloc, alors référencer ce pointeur après la sortie du bloc provoquera également une erreur. L'analyseur devra détecter ces erreurs et les afficher.

Le support pour les pointeurs peut être ajouté à l'analyseur par un domaine de pointeurs qui associe à chaque variable pointeur un ensemble de variables possibles référencées. Vous implanterez cette techniques et proposerez des exemples pour l'illustrer.

5.6 Analyse de chaînes de caractères

Ajoutez le support dans votre langage et dans votre analyse pour les chaînes de caractères.

Une variable chaîne sera déclarée par `string s`. Nous supposons les chaînes immuables (comme en Java, contrairement au C). Pour étendre nos expressions aux chaînes, nous ajoutons la syntaxe suivante :

— les chaînes littérales constantes, entre guillemets, comme : `"toto"` ;

— un opérateur de concaténation `.` (point) : `s = "un " . "mot"` ;

— l'extraction de la taille d'une chaîne : `i = length(s)`.

Le support pour les chaînes peut être ajouté à l'analyseur par un des domaines suivants (par ordre de complexité croissante) :

— un domaine gardant l'ensemble des lettres qui peuvent apparaître dans la chaîne, sans se souvenir de leur nombre d'occurrence ou ni de leur position ;

— un domaine associant à chaque chaîne sa taille, et pouvant abstraire ensuite cette information dans un domaine numérique (comme les intervalles), voir même dans un domaine numérique relationnel (pour découvrir des relations entre tailles de chaînes et variables numériques) ;

— un domaine approximant l'ensemble des chaînes possibles par un automate fini ;

— un produit réduit de deux des domaines (ou plus).