Grenoble INstitut Polytechnique

Ecole National Supérieur d'Informatique et de Mathématiques Appliquées de Grenoble

Travaux d'Etudes et de Recherche

Développement d'un mini Analyseur Statique de code intégré dans Eclipse

Loïc Crétin

<loic.cretin@ensimag.imag.fr>

Encadrants : Matthieu Moy et David Monniaux

Grenoble, le 28 janvier 2008

Remerciements

Je tiens à remercier tout particulièrement MM. David Monniaux et Matthieu Moy, mes tuteurs lors de ce projet, de m'avoir si bien encadré durant ce module « Travaux d'études et de Recherche », de m'avoir donné la chance de mener à bien ce projet qui me tenait à cœur.

Merci également au laboratoire Verimag et à son directeur de m'avoir accueilli très chaleureusement durant le second semestre.

Enfin merci à l'équipe pédagogique de mon école responsable de cet enseignement, en particulier Florence Maraninchi responsable qui est à l'origine de ce module, de m'avoir permis d'ouvrir mes horizons et de m'avoir fait découvrir de plus près un milieu qui m'attirait sans pour autant que je le connaisse.

Table des matières

Remerciements **i**

1 Introduction **3**
1.1 Contexte et motivation . . . 3
1.2 Techniques de Validation . . . 3
1.3 Plan . . . 4

2 Interprétation abstraite **5**
2.1 Définition . . . 5
2.2 Exemple . . . 6
2.3 Domaine Abstrait . . . 7
2.4 Opérateur d'élargissement, ou « widening » . . . 8

3 Mon Sujet **9**
3.1 Choix Techniques . . . 9
3.2 Contributions . . . 10

4 Pré-requis Techniques **11**
4.1 Eclipse . . . 11
4.2 Apron . . . 11
4.3 Java Native Access . . . 11

5 Théorie **13**
5.1 Produits existants . . . 13
5.2 Fonctionnement général . . . 13
5.3 Descente Syntaxique . . . 14
5.4 Exemple . . . 14
5.4.1 Expressions . . . 15
5.4.2 If (...) then ... else 15
5.4.3 while (...) 15

6 Pratique **17**

7 Conclusion **19**

A Extrait du code de l'analyseur **21**

B Résultat d'une exécution de notre outil **23**
Références . . . 24

Chapitre 1

Introduction

1.1 Contexte et motivation

Depuis la naissance de l'informatique et de la programmation, la complexité des logiciels informatiques croît à un rythme soutenu, il apparaît alors un problème de confiance en eux : est-ce-que les logiciels accomplissent bien la tâche que l'on attend d'eux avec toutes les garanties de sûreté nécessaires ?

Désormais les logiciels prennent de plus en plus de place dans notre vie (ordinateur personnel par exemple). On leur confie de plus en plus de responsabilités (système critique tel que le pilotage de véhicules, les centrales nucléaires...). Une petite erreur peut avoir de très lourdes conséquences, tant sur le plan financier que sur le plan humain. Ainsi le besoin de méthodes rigoureuses pour s'assurer du fonctionnement des logiciels se fait sentir dans l'industrie.

Prenons un exemple, le bogue d'Ariane 5 en 1996, sans doute un des bogues les plus chers de l'histoire, était dû à un problème de débordement d'entiers. Le problème majeur est que le calculateur pour cette fusée provenait directement d'Ariane 4, sur laquelle toute une batterie de tests avait été effectuée. Ce calculateur a été repris tel quel sur Ariane 5 sans reprise de spécifications, ni tests complémentaires. Le logiciel d'Ariane 4 était bogué pour des cas qui ne pouvait pas apparaître pour Ariane 4. L'erreur a d'abord été analysée manuellement, mais ensuite elle fut retrouvée automatiquement par l'outil *Polyspace*, ce qui démontrait qu'une analyse automatique pouvait retrouver des vrais problèmes dans des vrais programmes critiques.

En informatique, il est impossible d'analyser exhaustivement un programme, y compris la recherche d'erreurs à l'exécution. Plus précisément, il est impossible d'analyser :

1. de manière automatique.
2. sur des domaines infinis.
3. de manière exacte.
4. pour des exécutions non bornées.

Il n'existe pas de méthode « mécanique » permettant de toujours répondre sans se tromper au vu d'un programme si celui-ci peut ou non produire des erreurs lors de son exécution. Ceci est un résultat mathématique fondé sur des travaux d'Alonzo Church, Kurt Gödel et Alan Turing dans les années 1930 (problème de l'arrêt, théorème de Rice).

1.2 Techniques de Validation

De nos jours, il existe différentes solutions pour analyser les programmes mais chacune de ces solutions ne répondent qu'à deux des trois critères cités ci-dessus.

1. Preuve de théorème : technique qui permet de prouver certaines propriétés sur un programme donné, au besoin à l'aide d'un utilisateur qui donne des conseils au système (preuve interactive,

non automatique).

2. Model-checking : méthode algorithmique qui permet de vérifier qu'un système logiciel ou matériel satisfait à des exigences. Dans cette méthode, en général, on considère des systèmes à état fini. Joseph Sifakis, directeur de recherche au CNRS, au laboratoire VERIMAG, est l'un des inventeurs de cette méthode, ce qui lui donna le privilège d'obtenir le Prix Turing en 2007. [1]

3. Interprétation Abstraite : théorie de la résolution approchée d'équations de point fixe appliquée à l'analyse de programmes. Avec cette théorie nous obtenons une approximation des propriétés recherchées, dans une direction connue, entièrement automatiquement ; cependant, nous n'obtenons pas forcément le résultat exact. Cette théorie sera détaillée dans la suite de ce document.

1.3 Plan

Ce document explique le principe de notre travail qui est en quelques mots le développement d'un outil permettant de calculer les intervalles des valeurs pour les variables d'un programme à l'aide de l'interprétation abstraite, les choix effectués en particulier l'utilisation d'outils existants et les difficultés rencontrées telles que la cohabitation entre ces outils.

Dans un premier chapitre nous exposerons la théorie de l'interprétation abstraite qui consiste à extraire automatiquement des informations sur les exécutions possibles d'un programme Puis nous expliquerons plus précisément le sujet qui est « Développement d'un mini analyseur statique de code intégré dans Eclipse ».

Dans le chapitre suivant nous détaillerons les différents outils, c'est-à-dire Eclipse, APRON, JNA utilisés pour la réalisation du produit avant d'appliquer l'interprétation abstraite à notre problème.

Le point de vue théorique, produits existants, fonctionnement général, descente syntaxique de notre analyseur de même que quelques exemples seront introduits avant la partie pratique.

Enfin le dernier chapitre conclura le travail, énumérera les contributions et finira en présentant quelques perspectives .

Chapitre 2

Interprétation abstraite

2.1 Définition

L'interprétation abstraite est une théorie de l'approximation de sémantiques de langages de programmation ou de spécification.[2]

L'interprétation abstraite est une technique d'analyse de programme permettant de détecter des propriétés sur celui-la de manière automatique, tel que « le programme peut-il planter ? » ou « est ce que la valeur de la variable x ne dépasse pas 100 ? ».

Comme nous ne pouvons pas représenter l'ensemble des valeurs des variables d'un programme, *domaine concret*, nous nous fixons un domaine qui représente une sur-approximation du domaine de travail, appelé *domaine abstrait*, où l'on se fournit des opérations qui sont équivalentes aux opérations effectuées dans le domaine concret.

Pour bien comprendre le principe, nous allons commencer par un exemple non-informatique.

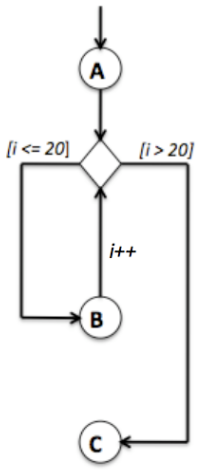
Considérons les élèves présents à un cours donné dans un amphithéâtre, auquel on associe le problème suivant : « prouver que certains élèves n'étaient pas présents au cours ». Nous pouvons noter uniquement le nom des élèves assistants au cours. Si le nom de l'élève n'est pas présent sur la liste, nous pouvons en conclure qu'il n'a pas assister au cours. Mais dans le cas contraire, nous ne pouvons pas être absolument certain que l'élève était pas en cours car deux élèves peuvent avoir le même nom. Ce résultat est donc approché dans une direction connue (on a une direction dans l'implication, mais pas forcément l'autre). En fait, comme la probabilité que deux élèves aient le même nom est plutôt faible, on a une bonne approximation de la réalité.

Dans la suite de ce chapitre nous considérons l'exemple suivant.

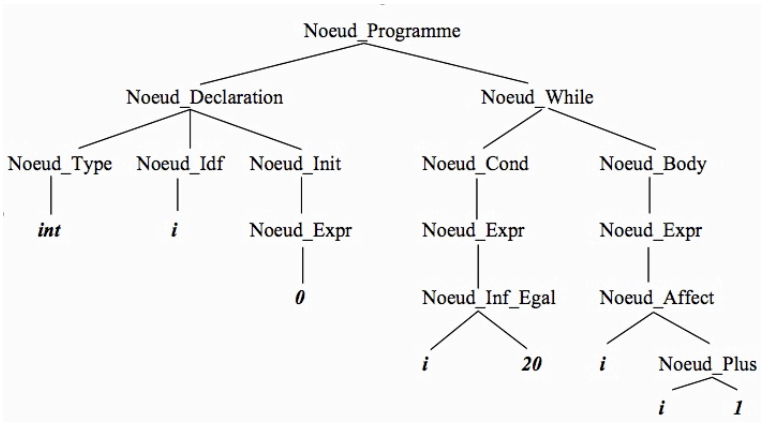
2.2 Exemple

```
void exemple() {

    int i = 0;
      /* A */
    while(i<=20) {
      /* B */
        i ++;
    }
      /* C */

}
```

(a) petit exemple



(b) Graphe de flot contrôle



(c) Arbre abstrait

Ci dessus nous fixons un petit programme (figure (a)) qui initialise la variable i à 0 et qui ajoute un à celle-ci tant qu'elle n'est pas strictement supérieure à 20. Sur la figure (b), nous décrivons le graphe de flot de contrôle correspondant au programme. Puis sur la figure (c) nous décrivons l'arbre abstrait de notre programme.

Dans un premier temps nous allons définir la notion de graphe de flot de contrôle et d'arbre abstrait d'un programme avant d'introduire l'interprétation abstraite.

Le graphe de flot de contrôle d'un programme est le un graphe dont les nœuds sont les suites indivisibles d'instructions, c'est-à-dire que lorsque l'on exécute la première instruction d'un nœud on doit obligatoirement exécuter successivement toutes les instructions qui sont contenues dans le bloc avant de passer à un autre bloc. De plus il y a un arc du bloc $\{B_i\}$ vers le bloc $\{B_j\}$ si et seulement si $\{B_j\}$ suit immédiatement $\{B_i\}$ dans l'ordre du programme sans branchement inconditionnel, ou si la dernière instruction de $\{B_i\}$ est un branchement (conditionnel ou inconditionnel) à la première instruction de $\{B_j\}$.

L'arbre abstrait d'un programme correspond à une représentation interne du programme sous forme d'arbre. Le nœud racine étant le début du programme, les nœud internes représentant les opérateurs et les nœuds feuilles correspondent aux opérandes de ces opérateurs.

Nous pouvons traiter le problème considéré par interprétation abstraite soit en analysant le graphe de flot de contrôle, comme c'est généralement le cas dans beaucoup de projets, ou bien effectuer le traitement directement sur l'arbre abstrait.

Ci dessous nous expliquons le principe général de l'interprétation abstraite.

2.3 Domaine Abstrait

Pour faire de l'interprétation abstraite, il faut d'abord trouver un domaine abstrait qui approxime par dessus le comportement du programme, c'est à dire que nous travaillons sur un sur-ensemble des comportements du programme. Nous définissons ensuite une équivalence entre les opérations sur le domaine concret et les opérations sur le domaine abstrait.

Puis nous effectuons les opérations sur le domaine abstrait en parcourant soit le graphe de flot de contrôle soit l'arbre abstrait, en démarrant avec le plus petit élément du domaine noté $\bot$.

On cherche un élément abstrait qui est vrai en tous points du programme, c'est à dire un invariant. Idéalement, on voudrait le plus petit. Ceci revient donc à chercher en tout point de notre programme le plus petit élément abstrait tel que ces éléments restent stables par exécution d'un pas de programme (un post-point-fixe).

Dans notre sujet nous voulons représenter les ensembles des valeurs des variables de notre programme. Nous définissons un élément abstrait comme étant une représentation approchée d'un ensemble. Notre approximation est conservatives du fait qu'à partir d'un ensemble concret on peut trouver un élément abstrait, mais l'élément abstrait peut correspondre à un ensemble plus grand.

Par exemple, si nous travaillons uniquement avec des entiers, une sur-approximation du comportement concret est de connaître l'intervalle des valeurs que prend les variables du programme. Supposons que nous avons une variable X qui peut avoir uniquement comme valeur $2, 4, 6$, alors nous approximons cet ensemble par l'intervalle $[2, 6]$ qui représente $X \in \{2, 3, 4, 5, 6\}$. En général, on ne peux pas calculer le plus petit intervalle qui contient la valeur des variables, mais seulement un intervalle qui les contient.

Maintenant nous rajoutons la structure définie ci-dessous *if (c) then ... else ....* Le choix de la branche s'effectue à l'exécution suivant la valeur de la condition c, c'est pourquoi une bonne approximation est de faire l'union (union ensembliste) des valeurs possibles en passant par les branches *then* et *else* du programme. Le calcul exact des valeurs abstraite après cette instruction est décrite dessous le schéma.

```
    ....
    /* A */
if (Condition) {
        ...
        /* B */
}else {
        ...
        /* C */
}
    /* D */
    ....
```

On note les point {A, B, C, D } comme étant la valeur du domaine abstrait en ces points du programme.

On calcul en deux temps le domaine au point de contrôle {D} :

1. on intersecte le domaine au point {A} avec la condition, puis on effectue les calculs dans la branche du *if* sur ce domaine. Ce qui nous donne le résultat au point {B}.

2. puis on intersecte le domaine au point {A} avec la négation de la condition, puis on effectue les calculs dans la branche du *else* sur le domaine. On obtient alors le résultat au point {C}.
3. enfin pour obtenir le domaine au point {D}, on fais l'union des deux domaines calculés précédemment, {B} et {C}.

Désormais, ajoutons une instruction supplémentaire *while (c) {...}*, qui correspond à effectuer une action tant que la condition n'est pas vérifiée. Ainsi il se pose le problème de la terminaison de notre calcul. C'est ce que nous allons expliquer dans la section suivante.

2.4 Opérateur d'élargissement, ou « widening »

Le problème qui se pose est que sur certain algorithme on peut avoir le soucis de la terminaison lors de la recherche du plus petit point fixe.

Si on reprend l'exemple vu précédemment et que l'on remplace la condition du *while* par la condition suivante $i \leq 100000$, nous devrions itérer 100000 fois. Si le corps de cette boucle était un tant soit peu long, le temps que nous passerions à itérer sur les instructions ne serait pas tolérable.

Si sur un petit programme comme le notre nous prenons un temps important pour effectuer les calculs alors il n'est pas imaginable de lancer ces mêmes calculs sur un programme contenant, par exemple les lignes de code de pilotage du métro parisien.

Dans le cas de domaines abstraits infinis, les calculs de point fixe ne convergent pas forcément en un temps fini. C'est pourquoi nous devons introduire une technique qui permet de trouver un point fixe rapidement, en quelques tours de boucle. C'est ce qui en fais en introduisant un opérateur d'élargissement.

Afin d'obtenir de bon résultat il existe différentes méthodes pour l'utilisation du widening. Intuitivement sur notre exemple d'intervalles, on regarde comment évoluent les valeurs au début puis nous extrapolons brutalement. Ensuite on s'arrange pour ne pas faire une infinité d'« extrapolations » qui pourrait nous conduire à trouver toutes les valeurs possibles pour les variables.

Ci-dessous nous donnons les résultats des invariants par interprétation abstraite sur notre programme précédent. Les détails des calculs seront expliqués dans la suite de ce document.

```
void exemple() {

    int i = 0;
    /* (A)   [0,0] */
    while(i<=20) {
    /* (B)   [0,20] */
        i ++;
    }
    /* (C) [21,21] */
}
```

calcul des invariants sans élargissement

```
void exemple() {

    int i = 0;
    /* (A)   [0,0] */
    while(i<=20) {
    /* (B)   [0,20] */
        i ++;
    }
    /* (C) [21, +OO] */
}
```

calcul des invariants avec élargissement.

Chapitre 3

Mon Sujet

Développement d'un mini analyseur statique de code intégré dans Eclipse.

Le but de ce sujet est d'écrire un analyseur statique pour un sous ensemble du langage Java, en s'appuyant sur l'environnement Eclipse. On se limitera à un sous ensemble du langage, seulement des variables de types « int », et à des propriétés simples, par exemple trouver des invariants du type « 12 $<$ x $<$ 42 ». Notre travail consiste à écrire un plugin Java permettant de trouver les invariants sur des variables entières, Eclipse étant un environnement de développement Java conçu autour de la notion de plugin. Eclipse nous permet de récupérer l'arbre abstrait d'un programme en quelques lignes de code.

L'analyse statique permet d'obtenir des résultats sur l'exécution du programme sans l'exécuter. Ce qui permet de prouver des propriétés, mais aussi de repérer des erreurs de programmations.

Ce travail consiste à effectuer de l'interprétation abstraite sur un « vrai » langage de programmation. C'est à dire que nous ne créons pas un pseudo langage, ni un formalisme bien particulier pour effectuer de l'interprétation abstraite, comme c'est souvent le cas, comme par exemple dans Nbac (Numerical and Boolean Automaton Checker), développé par Bertrand Jeannet (explicité chapitre suivant). [3]

Nous partons d'un langage de programmation, certes très restreint mais qui a le mérite de pouvoir être repris par la suite en ajoutant d'autres instructions, tel que les tableaux, afin de perfectionner notre analyseur.
Suite à différentes contraintes qui seront détaillées dans la partie suivante, nous choisissons que le programme à analyser contient une seule classe, une seule méthode qui est la fonction *main* et uniquement des variables de type *int* déclaré au début de la fonction. Par soucis de simplicité, on suppose que $int = \mathbb{Z}$ (dans la réalité, *int* calcule en arithmétique modulaire).

Le fait de travailler sur un vrai langage de programmation nous permet de ne pas nous occuper de la partie compilation du langage, mais aussi l'utilisateur peut entrer des programmes directement dans un langage qu'il maîtrise au lieu de devoir apprendre un langage propre à l'analyseur, ou au lieu de devoir utiliser un outil de conversion.

3.1 Choix Techniques

Cette étude se déroulant durant le second semestre à raison d'une demi journée par semaine, la principale contrainte est le temps qui est très limité. Suite à ceci nous avons du faire des choix entre :

1. Analyse sur le graphe de flot de contrôle ou de l'arbre abstrait :

   La plupart des outils existants parcours le graphe de flot de contrôle, d'ailleurs c'est l'approche habituelle au laboratoire Verimag. Cependant l'analyse peut très bien s'effectuer en parcourant l'arbre abstrait, à l'exception près que nous ne pouvons pas traiter l'instruction *goto* en toute généralité. Or, Java ne contient pas cette instruction.

Notre choix s'est orienté sur une analyse de l'arbre abstrait car ce dernier s'obtient gratuitement sous Eclipse. En revanche, pour construire le graphe de flot de contrôle, nous avons non seulement besoin de parcourir l'arbre abstrait mais aussi de construire une structure externe. De plus, nous avons besoin de fixer l'ordre de parcours du graphe et les points d'élargissement, ce qui n'est pas simple. Par ailleurs, certains ordres de parcours reconstruisent « à quelque chose près » les boucles présentes dans le programme avec points d'élargissement en tête de boucle, aussi le parcours du graphe peut revenir au parcours de l'arbre.

2. Analyse avant ou analyse arrière :

   Pour effectuer une analyse arrière, il faut au préalable effectuer une analyse avant et stocker les résultats en mémoire. Encore par soucis de temps, notre choix s'est orienté sur une unique analyse avant.

3. Utilisation d'un package contenant déjà le domaine abstrait et les opérations :

   Au lieu d'écrire un package correspondant au domaine abstrait avec toutes les opérations nécessaires qui nous prendraient un certain temps (développement et test), nous avons décidé d'utiliser le package APRON [4], contenant tous les outils nécessaires pour le domaine abstrait. Celui-ci étant développé en C, il nous a fallu introduire le package JNA (Java Native Access) permettant d'accéder à des fonctions développées en C dans un programme Java. JNA est moins lourd d'utilisation que JNI (Java Native Interface), qui impose la rédaction de fonctions d'interface entre C et Java.

3.2 Contributions

Les principaux résultats pratiques obtenus sont d'abord la réalisation d'un démonstrateur « agréable » permettant de trouver des propriétés non triviales sur les variables d'un programme java. C'est un outil convivial, au sens qu'il y a une petite interface graphique qui facilite l'affichage du résultat, ce n'est pas un outil en ligne de commande. Par ailleurs, le code est clair, facilement réutilisable et facile à compléter, de qui permettra éventuellement de l'étendre (autres domaines, langage plus riche).

Chapitre 4

Pré-requis Techniques

Dans cette partie, nous introduisons les différents outils utilisés pour créer notre analyseur afin de mieux de comprendre l'utilité de chacun.

4.1 Eclipse

Eclipse est à l'origine un IDE Java. Développé par IBM à partir de ses ancêtres Age et Visual Age For Java. Il a depuis été rendu Open Source et son évolution est maintenant gérée par la fondation Eclipse. La spécificité d'Eclipse vient de son architecture modulaire, totalement développée autour de la notion de plugins. Toutes les fonctionnalités de ce logiciel sont développées en tant que plugin. La licence d'Eclipse permet de fournir des plugins Open Source comme des plugins Closed-Source, des plugins gratuits ou payants. C'est pourquoi il est maintenant plus qu'un IDE Java, et gère un grand nombre de langages de programmation. [5]

4.2 Apron

Apron est une bibliothèque fournissant différents domaines abstraits, tel que les intervalles, paquetage BOX ; les octogones, paquetage OCT et les polyèdres convexes, paquetage NEWPOLKA, ainsi que les opérations utiles pour calculer des propriétés sur les variables numériques d'un programme.

Prenons par exemple les intervalles, il suffit de donner à Apron une expression et lui nous retourne la valeur correspondante dans le domaine abstrait. Cependant APRON nous offres pas le parcours du graphe de flot de contrôle ni de l'arbre abstrait. Mais aussi les points d'élargissement sont définis par le développeur et non par une fonction APRON.[4]

4.3 Java Native Access

JNA est une API qui permet d'adresser du code natif dans du code java. Le développement est assez simple en théorie, cependant nous avons quelques complications pour l'utilisation et pour faire cohabiter JNA avec eclipse. Il suffit de développer une interface Java décrivant le prototype des fonctions et structures contenues dans le code natif appelé.[6]

Chapitre 5

Théorie

Dans ce chapitre, nous donnons quelques exemples d'outils existants utilisant l'interprétation abstraite. Puis nous détaillons le fonctionnement général de notre outil. Et enfin nous donnons quelques exemples de descentes syntaxiques effectuées par notre outil.

5.1 Produits existants

1. ASTREE : analyseur statique de logiciels temps-réel embarqués, développé par l'équipe de cousot au laboratoire d'informatique de l'Ecole Normale Supérieur dont David Monniaux faisait partie. L'objectif est de démontrer statiquement et automatiquement l'absence d'erreur à l'exécution dans des logiciels critiques temps-réel embarqués. L'analyse statique est basée sur la théorie de l'interprétation abstraite. Il a été utilisé pour vérifier l'absence d'erreur à l'exécution sur les logiciels de commande de vol électrique primaire des avions Airbus A 340 et A 380.[7]

2. Nbac : outil d'analyse et de vérification de systèmes réactifs mis sous la forme d'équations de flot de données synchrones, dans lesquelles les variables sont de type booléen ou numérique (entiers, réels), développé par Bertrand Jeannet. Cet outil effectue de l'interprétation abstraite sur différents domaines abstraits, et retourne soit un verdict à un problème de vérification, soit une structure de contrôle réduite représentant un ensemble d'exécutions sélectionnées. Dans Nbac, il est très facile de changer un domaine abstrait par un autre, par exemple remplacer les polyèdres par les octogones.[3]

3. Polyspace : outil de tests et de validation de logiciels embarqués. Leur outil est principalement utilisé dans les secteurs de l'automobile, de l'aéronautique et de l'énergie.

5.2 Fonctionnement général

Pour ce sujet, comme nous l'avons cité précédemment, nous nous sommes restreins à un sous ensemble du langage Java :

1. une seule classe.
2. une seule méthode (fonction principale *main*).
3. uniquement des variables de type entière (*int*) déclarées au début du programme.

A la fin, le but étant d'avoir un outil qui prend un Programme Java, restreint aux propriétés ci dessus, et qui retourne l'ensemble des valeurs des variables en tous points du programme.

$$\text{Programme Java} \rightarrow \{\text{Invariant}\}$$

Une démonstration de notre outil est donnée en annexe B

5.3 Descente Syntaxique

Dans cette partie nous allons détailler plus précisément ce travail.

Le but de notre problème est de déterminer l'ensemble des valeurs dans lequel se trouve chacune de nos variables en tous points du programme. Nous devons donc déterminer un domaine qui permet de représenter ces valeurs. Choisissons le domaine des intervalles numériques.

Une approximation pour notre problème consiste à abstraire toutes les valeurs des variables par le plus petit intervalle qui les contiennent, et donc de choisir comme domaine abstrait le treillis τ des intervalles sur les entiers $\mathbb{Z}$:

$$\tau = \{\top\} \cup \{[l,u] | l \in \mathbb{Z} \cup \{-\infty\} \wedge u \in \mathbb{Z} \cup \{+\infty\} \wedge l \leq u\}$$

ordonné par la relation d'« inclusion » $\subseteq$ qui nous dis qu'un intervalle est contenu dans un autre si ces bornes sont comprises dans les bornes du deuxième intervalle. Par ailleurs pour deux intervalles donnés, il n'y a pas obligatoirement un sens d'inclusion. La relation d'inclusion est décrite mathématiquement ci-dessous :

$$\begin{cases} \forall X \top \subseteq X \\ [l_1,u_1] \subseteq [l_2,u_2] \Leftrightarrow l_1 \leq l_2 \leq u_2 \leq u_1 \end{cases}$$

Afin de trouver les différents intervalles, nous parcourrons de manière récursive l'arbre abstrait du programme en ajoutant toutes les nouvelles variables dans notre domaine en les initialisant à $\top$ qui correspond à l'intervalle $[-\infty, +\infty]$, ce qui signifie qu'à ce point du programme toutes les valeurs sont possibles pour cette variables. Puis nous effectuons les opérations en même temps sur le domaine abstrait afin d'obtenir les intervalles. Les différentes opérations effectuées sur le domaine abstrait sont mentionnées dans la section suivante.

Cependant comme explicité dans le Chapitre *Interprétation abstraite*, on cherche à calculer le plus petit point fixe en tous points du programme. Cette opération n'étant pas toujours possible nous introduisons un opérateur d'élargissement que nous utilisons en tête de boucle, car nous voulons que notre programme termine tous ces calculs, et ce, rapidement. Ainsi pour le cas des boucles, nous effectuons un tour de boucle puis nous élargissons ce qui nous permet d'obtenir un point fixe. Afin de se rapprocher au maximum du plus petit point fixe nous effectuons un tour de boucle supplémentaire (technique appelée *rétrécissement*).

Ci-dessous nous donnons le sens mathématique de notre opérateur d'élargissement ∇ pour le treillis τ proposé par Cousot (inventeur de l'interprétation abstraite) : [8]

$$\begin{aligned} \top \nabla X &= X \\ X \nabla \top &= X \\ [l_1,u_1] \nabla [l_2,u_2] &= [\text{ if } l_2 < l_1 \text{ then } -\infty \text{ else } l_1, \text{ if } u_2 > u_1 \text{ then } +\infty \text{ then } u_1] \end{aligned}$$

Il est très important de n'élargir qu'un minimum de fois car sinon nous perdons énormément de précision et parfois même nous pouvons obtenir pour une variable l'ensemble des entiers qui n'est pas très exploitable pour notre problème.

5.4 Exemple

Dans ce paragraphe nous donnons les calculs des intervalles après différentes instructions, notées *PostCondition* en fonction des intervalles avant les instructions, notée *PreCondition*. De plus on note par *INTER* la fonction d'intersection entre deux ensembles, par *UNION* la fonction d'union de deux ensembles, et par *INCLU* la fonction d'inclusion entre deux ensembles.

5.4.1 Expressions

Lorsqu'une expression est une déclaration de variables sans initialisation (*int x ;*), nous ajoutons cette variable dans le domaine en initialisant son intervalle à $\top \Leftrightarrow [-\infty; +\infty]$.

Lorsqu'une déclaration est suivit d'une initialisation (*int x = expr*), on décompose le travail. Dans un premier temps on ajoute la variable, puis on affecte la variable à la valeur de l'expression.

Lorsque l'on a une affectation composée d'une expression avec opérateur ($x = expr_1$ *opbin* $expr_2$), on calcule la valeur de $expr_1$ et de $expr_2$ puis on effectue l'opérateur opbin entre ces deux résultats. Et enfin on affect cette valeur à la variable x.

5.4.2 If (...) then ... else ...

Dans cette exemple, nous détaillons les appels pour l'instruction *if then else*. Nous notons le corps des branches par *Arbre_ corps_ if* et *Arbre_ corps_ else*. L'ensemble de l'arbre de cette instruction est noté *Arbre_ if*.

```
if (Condition) {
        Arbre_corps_if
} else {
        Arbre_corps_else
}
```

(b.1) Exemple

```
PostCondition Analyser_if(PreCondition pred, Arbre Arbre_if) {
      res_if =  Analyser_Arbre(pred INTER Condition, Arbre_corps_if);
      res_else = Analyser_Arbre(pred INTER not Condition, Arbre_corps_else);

      retourne res_if  UNION res_else;
}
```

(b.2) fonction Analyser_if

5.4.3 while (...) ...

Dans cette exemple, nous détaillons les appels pour l'instruction *while*. Nous notons le corps du while par *Arbre_ body*. L'ensemble de l'arbre de cette instruction est noté *Arbre_ while*.

Comme cité dans le chapitre *Interprétation abstraite*, le problème d'introduire l'instruction *while* est la terminaison du calcul, c'est pourquoi nous introduisons un opérateur d'élargissement. Dans notre exemple, nous effectuons un premier tour de boucle pour observer le comportement des variables. Puis nous élargissons afin d'obtenir un point fixe rapidement. Nous effectuons après un second tour de boucle pour rétrécir ce point fixe et ainsi se rapprocher du plus petit.

```
while (Condition) {
      Arbre_body;
}
```

(c.1) Exemple

```
PostCondition Analyser_while(PreCondition pred, Arbre Arbre_while) {
      res_zero_un = Analyser_Arbre(pred INTER Condition, Arbre_body) UNION Pred;
      /* res_zero_un correspond à l'intervalle après 0 ou 1 tour de boucle */
      res_elar = Elargissement( res_zero_un);
      faire {
      /* On cherche un point fixe */
         tmp_elar = res_elar;
         tmp = Analyser_Arbre(res_elar INTER Condition, Arbre_body);
         res_elar = Elargissement( tmp);
      }tant que(not ( tmp_elar INCLU res_elar));

      /* On rétréci l'intervalle en parcourant un tour de plus. */
      res = Analyser_Arbre(res_elar INTER Condition, Arbre_body);

      /* On sort du while, on filtre avec la négation de la condition */
      retourne res INTER not Condition;
}
```

(c.2) fonction Analyser_while

Le code complet de la structure *if then else* et *while* est donné dans l'annexe A.

Chapitre 6

Pratique

Notre outil a été développé sous Eclipse sous forme de plugin. Il fait appel aux fonctions C de la bibliothèque APRON et ainsi fait utilisation au package JNA.

Pour parcourir l'arbre abstrait du programme nous utilisons le patron Visiteur sur l'arbre abstrait, prédéfini dans Eclipse, qui parcourt récursivement les nœuds de l'arbre. Notre visiteur contient un domaine abstrait qui contient l'ensemble des variables du programme. Pour chaque nœud nous effectuons le traitement associé en modifiant les intervalles des variables.

Cependant nous avons eu quelques complications explicitées ci dessous. Après avoir passé un certain temps sur ces problèmes, David Monniaux a trouvé la plupart des solutions.

1. la plupart des fonctions développées sous APRON sont définies en *static inline* ce qui implique un problème lors de l'appel de ces fonctions sous Eclipse car celles-ci ne sont pas présentes dans la bibliothèque compilée. Il a donc fallu pour ces fonctions développer des fontions « patch » afin de pouvoir les appeler depuis Eclipse.

2. Il a aussi fallu rajouter des fonctions qui permettent de récupérer la sortie standard du C pour pouvoir afficher le résultat sous Eclipse (Apron ne possède curieusement pas de fonctions d'impression de résultats vers une chaîne de caractères).
3. De plus, nous avons eu un problème avec JNA lors du passage de paramètres par référence ou par valeur (le problème a été rapporté aux développeurs de JNA, qui l'ont confirmé).
4. JNA accède aux fonctions, pas aux variables, il a donc fallu ajouter des fonctions accesseurs.

Chapitre 7

Conclusion

Dans ce document nous avons résumé la théorie de l'interprétation abstraite ainsi que quelques outils existants. Puis nous avons expliqué le sujet, les choix effectués pour utiliser cette théorie sur une restriction du langage java.

Au final, nous avons montré qu'avec les outils d'aujourd'hui, il était possible d'écrire un petit interpréteur abstrait en moins de 1000 lignes de code (voir Annexe A)

Ce travail nous a permis d'obtenir un démonstrateur avec une petite interface graphique qui trouve des invariants en tous points d'un programme java contenant uniquement des variables entières (voir Annexe B). Nous avons donc conçu un petit analyseur qui peut être utile pour programmeur Java, mais aussi une base de code extensible pour le reste du langage Java.

Ainsi notre travail pourrait être un terrain d'expérimentation intéressant pour des techniques plus avancées d'interprétation abstraite sur le langage Java.

Par la suite nous pouvons faire évoluer notre analyseur afin qu'il prenne en compte d'autres instructions telles que les tableaux, les pointeurs, les appels de méthodes....

Mais aussi nous pourrons envisager dans un futur proche de modifier l'interface en la rendant plus conviviale. Par exemple en demandant à l'utilisateur de cliquer sur une variable du programme ainsi l'analyseur affichera l'intervalle dans lequel se trouve la variable à cet endroit du programme. Ceci étant plus ou moins similaire au package ASTView d'Eclipse qui permet d'afficher l'arbre abstrait du programme et de se diriger à l'intérieur en cliquant sur le programme.

Ce travail d'étude et de recherche m'a beaucoup apporté tant sur le plan découverte et développement qu'au niveau autonomie de travail.

Cette nouvelle méthode d'apprentissage par soi-même est très enrichissante : Travailler seul sur un projet, prendre conscience de la complexité de la recherche, apprendre à résoudre des bogues informatiques m'ont beaucoup plus.

De même que le sujet : Développement d'un analyseur statique de code intégré dans Eclipse est très intéressant et me donne envie de poursuivre ma recherche. De plus le fait de savoir que mon travail peut être utile par la suite m'a d'autant plus motivé.

En quelques mots je peux dire que ces premiers pas dans la recherche m'ont donné envie de continuer dans cette voie et de peut-être pouvoir créer un projet d'avenir sur le même modèle que mes tuteurs Matthieu Moy et David Monniaux

Annexe A

Extrait du code de l'analyseur

Dans cette annexe nous montrons un extrait du code de notre analyseur pour les instructions *if then else* et *while*.

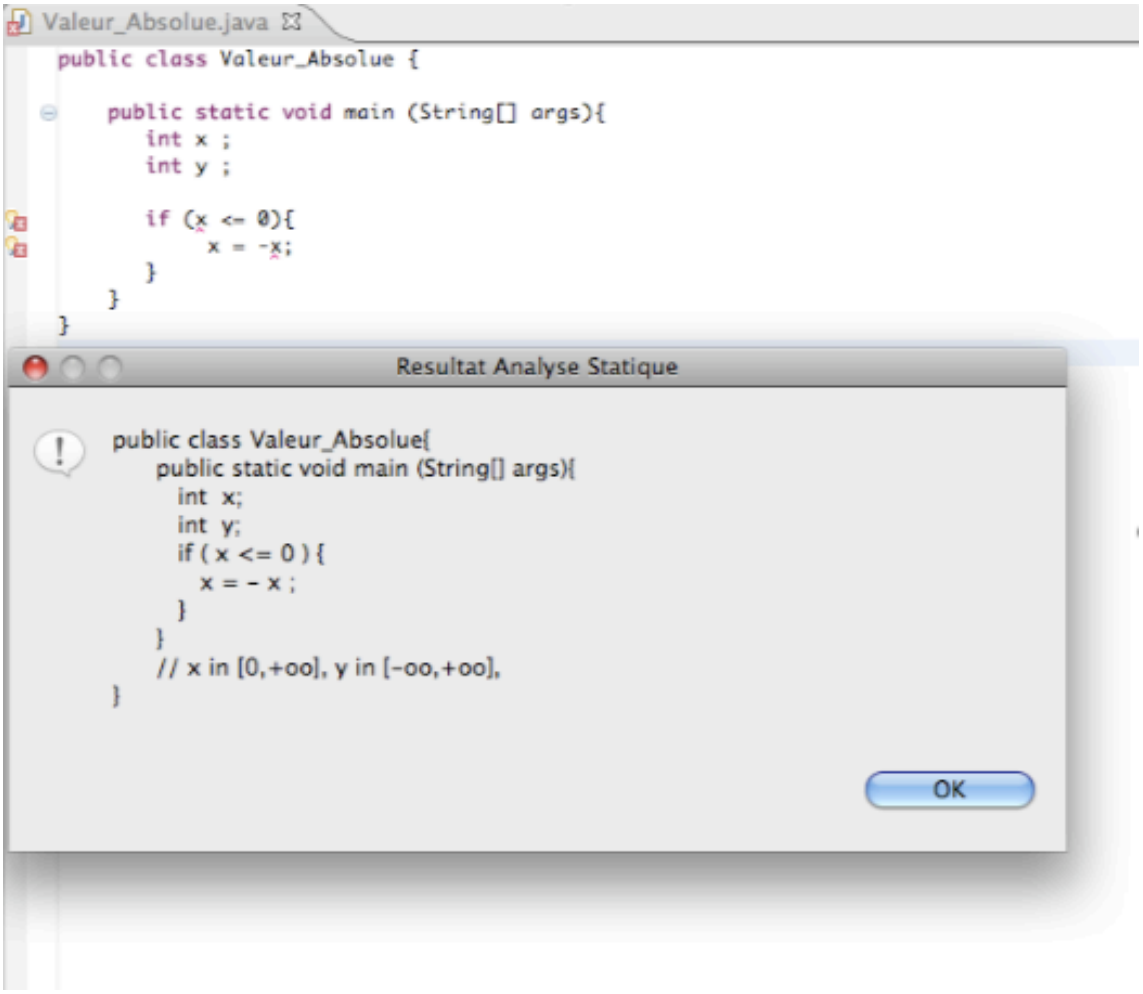
```
public ap_abstract1_t visit_if (IfStatement node, ap_abstract1_t abs,
                boolean modifTexte) throws IOException {
   if (modifTexte) this.ecr.add(indentTexte() + "if ( ");
   ap_abstract1_t copy_abs = ap_abstract1_copy(manager, abs);
   ap_tcons1_t cond = visit_condition((InfixExpression)node.getExpression(), abs,
        false, modifTexte);
   ap_tcons1_t not_cond = visit_condition((InfixExpression)node.getExpression(),
        copy_abs, true, false);
   if (modifTexte) {
      this.ecr.add(") {\n");
      this.indentTexte++;
   }
   // a_if = abs inter cond
   ap_abstract1_t a_if = ap_abstract1_meet_tcons1(false, abs, cond);

   // a_else = abs inter not_cond
   ap_abstract1_t a_else = ap_abstract1_meet_tcons1(false, copy_abs, not_cond);

   ap_abstract1_t if_abs = visit_statement(node.getThenStatement(), a_if,
        modifTexte);
   if (modifTexte)  {
      this.indentTexte--;
      if (node.getElseStatement() != null) {
         this.ecr.add(indentTexte()+ "}else{\n");
         this.indentTexte++;
      }else{
         this.ecr.add(indentTexte()+"}\n");
      }
   }

   ap_abstract1_t else_abs = visit_statement(node.getElseStatement(),
        a_else, modifTexte);

   ap_abstract1_t res = ap_abstract1_join(manager, false, if_abs, else_abs);
   if (modifTexte && node.getElseStatement() != null) {
      this.indentTexte--;
      this.ecr.add(indentTexte()+ "}\n");
   }
```

```
    // union de if et du else
    return res;
}


public Apron.ap_abstract1_t visit_while(WhileStatement node, Apron.ap_abstract1_t abs,
        boolean modifTexte) throws IOException {
    if (modifTexte) this.ecr.add(indentTexte() + "while (");
    ap_tcons1_t cond = visit_condition((InfixExpression)node.getExpression(),
        abs, false, modifTexte);
    ap_tcons1_t not_cond = visit_condition((InfixExpression)node.getExpression(),
        abs, true, false);

    if (modifTexte) {
        this.ecr.add(") {\n");
        this.indentTexte++;
    }

    // premier = visit(predCond inter Cond, body) U predCond + elargissement
    ap_abstract1_t tmp_premier = visit_statement(node.getBody(),
        ap_abstract1_meet_tcons1(false, abs, cond), modifTexte);
    ap_abstract1_t premier_whidening = ap_abstract1_widening(manager, abs,
        ap_abstract1_join(manager, false, tmp_premier, abs));
    boolean pointfixe = false;
    ap_abstract1_t tmp, tmp_widening = null;
    while (!pointfixe) {
    /* Tant qu'on a pas un point fixe on elargi */
        tmp_widening = premier_whidening;
        tmp = visit_statement(node.getBody(),
          ap_abstract1_meet_tcons1(false, premier_whidening, cond), false);
        premier_whidening = ap_abstract1_widening(manager, abs,
           ap_abstract1_join(manager, false, tmp, abs));
        pointfixe = ApronLibrary.INSTANCE.ap_abstract1_is_leq(manager,
          tmp_widening, premier_whidening);
    }

    ap_abstract1_t premier = ap_abstract1_meet_tcons1(false, tmp_widening, cond);

    // second = visit(premier inter Cond, body)
    ap_abstract1_t second = visit_statement(node.getBody(),
        ap_abstract1_meet_tcons1(false, premier, cond), false);

    if (modifTexte) {
        this.indentTexte--;
        this.ecr.add(indentTexte()+"}\n");
    }

    return ap_abstract1_meet_tcons1(false, second, not_cond);
}
```

Annexe B

Résultat d'une exécution de notre outil

Dans cet annexe, nous montrons le résultat de l'utilisation de notre outil sur un programme qui retourne la valeur absolue de la variable x.



Bibliographie

[1] L. Louis, Joseph sifakis, chercheur au cnrs, recoit le prix turing 2007, Disponible sur Internet a l'adresse http ://www2.cnrs.fr/presse/communique/1280.htm, 2008.

[2] P.Cousot and R. Cousot, Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints, 1977.

[3] B. Jeannet, The nbac verification/slicing tool, Disponible sur Internet a l'adresse http ://pop-art.inrialpes.fr/people/bjeannet/nbac/index.html, 2006.

[4] Apron numerical abstract domain library, Disponible sur Internet a l'adresse http ://apron.cri.ensmp.fr/library/, 2007.

[5] IBM, Eclipse, Disponible sur Internet a l'adresse http ://www.eclipse.org/.

[6] Sun, Java native access, Disponible sur Internet a l'adresse https ://jna.dev.java.net/.

[7] C. Team, The astree static analyzer, Disponible sur Internet a l'adresse http ://www.astree.ens.fr/.

[8] P.Cousot and R. Cousot, Static determination of dynamic properties of programs, Dunod, Paris, 1977.