Université de Nice Sophia Antipolis
DEA d'Informatique

Rapport de stage

présenté en Juillet 2002

par
Damien CIABRINI

Scheme et JVM : un environnement de programmation

Responsables de stage : Manuel SERRANO et Bernard SERPETTE
Rapporteur : Jacques FARRÉ

Remerciements

Je tiens à remercier très chaleureusement mes deux encadrants Manuel SERRANO et Bernard SERPETTE, qui m'ont beaucoup aidé durant ce stage et qui ont toujours été disponibles lorsque j'avais besoin d'eux.

Je remercie aussi l'ensemble des membres du projet MIMOSA, notamment Frédéric BOUSSINOT et Christian BRUNETTE, pour m'avoir accueilli à l'INRIA dans de si bonnes conditions.

Enfin je tire mon chapeau à mes parents qui ont eu le courage de lire ce rapport !

Table des matières

1 Introduction

Le débogage est l'action de détecter, localiser et corriger des erreurs dans un programme. Un débogueur est un outil qui permet d'assister le programmeur dans cette tâche et qui l'aide à comprendre le comportement de son programme. Les bogues pouvant se produire dans un programme sont tous de nature très différente. Ainsi, pour être efficaces, les débogueurs existants sont conçus pour permettre d'analyser et de résoudre un type particulier d'erreur. Les différentes approches de débogage peuvent être regroupées en deux grandes catégories :

- *Les analyses statiques*, qui cherchent à déterminer statiquement si certaines opérations d'un programme peuvent provoquer des erreurs à l'exécution. Cela permet de détecter des erreurs subtiles ou de prouver, en absence d'erreur, qu'un programme fonctionne correctement.
- *Le débogage au moment de l'exécution*, qui permet d'instrumenter l'exécution d'un programme, et d'obtenir des informations sur la valeur de ses différentes variables durant son exécution.

Ces deux techniques sont complémentaires, en ce sens qu'elles permettent de tirer des conclusions différentes sur le programme analysé.

Tous les programmeurs sont un jour amenés à déboguer un programme, il y a donc un réel besoin d'outils d'aide au débogage. Mais curieusement, ce domaine de l'informatique ne connaît plus d'évolution majeure : les programmeurs doivent se contenter des outils existants. Très souvent dans la pratique, ils se passent complètement de ces outils pour déboguer. Ils préfèrent mettre en œuvre des solutions *ad hoc*, qui consistent généralement à annoter le "code source avec des `prints`. Ils sont alors obligés de recompiler leur programme et de le relancer autant de fois que nécessaire pour localiser le bogue. Dans bien des cas il serait nécessaire de disposer d'outils de débogage plus expressifs ou plus agréable à utiliser que ce qui existe actuellement. Ce désintérêt pour les débogueurs au profit de solutions très primitives en est bien le reflet.

Nos travaux se focalisent sur le domaine du débogage dynamique. Pour tenter de répondre aux besoins précédemment évoqués nous avons développé le débogueur Bugloo, un outil permettant de faire du débogage à l'exécution. Son modèle de fonctionnement est le même que celui des débogueurs dynamiques classiques. Nous essayons d'améliorer ce modèle en lui ajoutant des fonctionnalités qui rendent Bugloo plus pratique à utiliser que les débogueurs actuels.

La section 2 est un aperçu des travaux antérieurs réalisés dans le domaine du débogage statique et dynamique. La section 3 précise le contexte de développement dans lequel nous avons réalisé Bugloo, et quels étaient nos objectifs. La section 4 présente le débogueur Bugloo et décrit les éléments fondamentaux de son fonctionnement. La section 5 expose en détail les nouvelles fonctionnalités de débogage implémentées et présente des exemples de leurs utilisations. La section 6 compare Bugloo aux outils existant dans les différentes approches de débogage. La section 7 expose les résultats obtenus, propose des orientations futures et conclut le rapport.

2 État de l'Art

Il existe différentes approches de déboguage selon le type de problème qu'un programmeur veut résoudre. Chacune d'elles permet de corriger des catégories de bogues différentes. Dans cette section nous présentons les principales approches de déboguage ainsi qu'un aperçu des travaux réalisés dans chacun de ces domaines.

2.1 Déboguage par analyses statiques

Le déboguage statique permet d'analyser les opérations d'un programme et de déterminer si certaines d'entre elles peuvent éventuellement causer des erreurs à l'exécution. Grâce à cette méthodologie, il est possible de détecter automatiquement des erreurs directes comme des dé-référencement de pointeur nul ou des erreurs de typage. Certains langages de programmation sont par conception pourvus de mécanismes de vérification[1] qui permettent de notifier qu'une erreur de ce type s'est produite durant l'exécution du programme. Cependant, s'ils assurent que le programme ne retournera pas de résultat aléatoire dû à une erreur non signalée, ils ne permettent de découvrir les erreurs qu'au moment de l'exécution. De plus, une seule exécution ne suffit pas pour détecter toutes les erreurs exhibées par les analyses statiques.

Un débogueur statique est un outil qui réalise des exécutions symboliques [6] d'un programme. Lors de ces exécutions abstraites, une variable est représentée par l'ensemble des valeurs qu'elle pourra prendre pour toutes les exécutions possibles. Ce genre de débogueur calcule une approximation de l'ensemble des valeurs que peuvent prendre une variable ou une expression d'un programme. Cette approximation est *conservative* : toutes les valuations possibles seront effectivement contenues dans l'ensemble calculé. En propageant ces informations à l'intérieur du programme, le débogueur statique peut découvrir des valuations de variables qui entraîneront des erreurs lors d'une exécution réelle.

2.2 Déboguage par assertions

Bien souvent, le typage statique d'un programme est correct, mais ce dernier ne fait pas ce que l'utilisateur attend de lui à l'exécution. Les assertions sont des expressions logiques qui permettent d'exprimer des propriétés sur le comportement d'un programme. Grâce aux assertions, le programmeur peut spécifier des propriétés sur le sens de son programme qui doivent être respectées. Syntox [4, 3] est un débogueur qui utilise une interprétation abstraite d'un sous ensemble du langage Pascal pour déterminer statiquement certains comportements d'un programme à l'exécution. Dans Syntox, les assertions sont essentiellement utilisées pour décrire des invariants de programmes de petite taille.

En général, les assertions sont plutôt employées pour faire du déboguage à l'exécution. L'approche la plus courante d'utilisation des assertions est celle que l'on retrouve dans le langage Eiffel, avec sa notion de *contrats*. Cette approche consiste a évaluer une assertion (*pré-condition*) avant l'exécution d'une fonction, puis une autre (*post-condition*) après son exécution. Cela permet de contrôler qu'une fonction peut être appelée et de vérifier qu'elle s'est exécutée correctement. De plus, ce mécanisme peut servir à déterminer la portion du

[1] Par exemple en Java, on effectue une vérification de bornes quand on accède à un élément d'un tableau.

programme qui est fautive lorsqu'une violation de contrat a eu lieu.

Des techniques de contrats ont aussi été développées pour les langages d'ordre supérieur. Le *contrat-calcul* [8] définit formellement une sémantique de contrat qui permet d'exprimer des assertions sur les *lambda-expressions*. Les pré-conditions et post-conditions enveloppent des lambda-expressions, et il est possible d'exprimer des contraintes sur leurs arguments, même s'il s'agit d'objets d'ordre supérieur.

2.3 Déboguage pendant l'exécution

Le déboguage dynamique permet d'obtenir des informations précises sur l'état d'un programme au cours de son exécution, ce que ne peuvent pas faire les analyses statiques ou les assertions parce qu'elles ne disposent pas de la vraie valeur des variables. Ce genre de pratique se généralise dans les langages de programmation. En Java par exemple, lorsqu'une exception est levée, il est possible d'afficher la pile d'exécution du programme. Les débogueurs dynamiques sont des outils d'exploration d'états et de contrôle d'exécution qui s'utilisent « après coup » : lorsqu'un bogue s'est produit durant une exécution, on tente de le reproduire et de l'analyser en lançant une nouvelle exécution qui cette fois ci sera instrumentée par le débogueur.

2.3.1 Déboguage dynamique classique

Le modèle de fonctionnement qui s'est imposé dans tous les débogueurs dynamiques est celui que l'on retrouve dans GDB [25]. Conceptuellement, ce type de débogueur est capable de lancer un programme et d'instrumenter son exécution. Les opérations de déboguage qu'il permet d'effectuer peuvent être regroupées en deux grandes catégories :

- **Il peut contrôler le déroulement de l'exécution du programme**. L'utilisateur peut suspendre et reprendre l'exécution du programme quand il le souhaite. Il peut visualiser la progression de l'exécution dans le code source à chaque fois que le programme est suspendu. L'utilisateur peut placer des *breakpoints* n'importe où dans le code source. Cela permet de suspendre l'exécution du programme lorsqu'un de ces points symboliques est atteint. Il est aussi possible de placer des *watchpoints* sur n'importe quelle variable. Le programme n'est plus suspendu lorsque l'exécution atteint un endroit précis du code source, mais quand le contenu d'une variable est lu ou modifié. Pour contrôler plus finement le déroulement du programme, il est possible de faire du *stepping*. L'exécution se déroule alors « pas à pas », et chaque fin de pas suspend le programme. On peut par exemple tracer ligne par ligne l'exécution d'une fonction, décider de tracer les sous-fonctions qu'elle est susceptible d'appeler, ou tout simplement exécuter la fonction en un seul pas.
- **Il peut connaître l'état du programme au cours de son exécution**. Lorsque le programme est suspendu, il est possible de « l'interroger » sur son état courant. L'utilisateur peut demander des informations sur la pile d'exécution[2]. Il peut ainsi connaître la fonction dans laquelle le fil d'exécution s'est arrêté et l'ensemble des fonctions appelantes. L'utilisateur peut demander la valeur des différentes variables du programme, qu'elles soient locales (dans la pile), ou globales (dans le tas). Il peut aussi les modifier, ce qui permet d'influer sur le déroulement « normal » du programme[3].

[2] Ou sur les piles d'exécution, dans le cas de programmes *multi-threads*.

[3] Cette fonctionnalité existe, mais n'est pas forcément un atout.

Il peut enfin obtenir des informations de plus bas niveau, comme par exemple la quantité de mémoire encore disponible ou l'état des différents registres de l'architecture matérielle sur laquelle s'exécute le programme débogué.

2.3.2 Déboguage d'allocation mémoire

GDB permet de contrôler de manière plus ou moins générale l'exécution d'un ensemble de langages impératifs classiques. Certains outils proposent des analyses qui tiennent compte des spécificités d'un langage particulier. Dans le cadre des langages à gestion automatique de la mémoire (*automatic garbage collection* [2]), KBDB [22] est un outil capable de faire du déboguage d'allocation mémoire. Les auteurs modifient l'allocateur mémoire pour qu'il maintienne en permanence les *références inverses* des objets vivants dans le tas du programme. Pour un objet donné, KBDB peut retourner un pointeur sur un autre objet qui le référence. Cette propriété permet de comprendre les causes d'éventuelles *fuites mémoires*, qui se produisent lorsque des objets que l'on croyait morts (c'est à dire plus référencés par personne) ne sont pas ramassés par l'allocateur mémoire (car en réalité ils sont encore référencés).

2.3.3 Analyses « post-mortem »

Certains débogueurs proposent à l'utilisateur de déboguer leur programme une fois qu'il s'est exécuté. Ces outils fonctionnent en analysant des enregistrements des différents événements qui se sont produits durant l'exécution. Jinsight [20, 7] fait partie de ces outils, il fonctionne pour le langage Java. Son but est de présenter correctement les informations accumulées lors de la trace (l'enregistrement de ce qui s'est passé durant l'exécution d'un programme), en proposant des patrons d'affichage qui simplifient l'étude de l'exécution. Il utilise une version modifiée de la JVM de Sun qui enregistre une trace de tous les événements produits par le programme débogué. Les mécanismes de débogage visuel de Jinsight permettent d'analyser plusieurs types de problèmes. Certains de ses patrons d'affichage permettent à l'utilisateur de visualiser les différents chemins d'exécution d'un programme. D'autres permettent de filtrer les événements produits pour ne présenter que ceux qui sont responsables de fuites mémoires.

Jinsight EX [24] est une variante de Jinsight. Il introduit le concept de *tranches d'exécution*. Lors d'une exécution de programme, Jinsight EX enregistre tous les appels de méthode et le temps qu'elles prennent pour s'exécuter. Il est ensuite possible de grouper graphiquement les appels de méthodes en entités logiques : les tranches. Cela permet de faire du débogage de performance de programmes.

Le gros inconvénient de ces techniques d'enregistrement de traces est qu'elles manquent la plupart du temps de souplesse. Les traces enregistrées sont souvent très importantes et la quantité de mémoire nécessaire à leur interprétation devient prohibitive pour des programmes de taille conséquente.

3 Contexte de développement

Le but du stage est de mettre au point un débogueur pour le langage Scheme [15], utilisant le modèle de fonctionnement de GDB et proposant des fonctionnalités qui le rendent plus agréable à utiliser que ce dernier. Le langage fonctionnel Scheme a des qualités pour la conception d'un débogueur :

- C'est un langage de haut niveau qui n'a pas été conçu avec des contraintes d'architecture matérielle en tête. Il ne manipule donc pas de concept trop primitif comme une adresse mémoire ou un registre machine.
- C'est un langage qui dispose d'une sémantique claire, suffisamment forte pour éviter les problèmes trop « bas niveau » que l'on peut rencontrer dans des langages trop laxistes comme C[4]. Cela nous permet d'implémenter aisément certaines fonctionnalités de déboguage sans que le langage nous pose de problème.

L'outil que nous avons développé permet de déboguer des programmes écrits par le compilateur optimisant Bigloo [23].

3.1 Bigloo et environnement de programmation

Bigloo est un compilateur de Scheme étendu par quelques commodités linguistiques, comme par exemple la notion de module ou un système objet à la CLOS [1]. Bigloo propose deux générateurs de code différents : il est capable de générer du code C[5] pour la rapidité, ou directement du bytecode JVM [17] pour la portabilité. Il fournit aussi un environnement complet de programmation. En particulier, il proposait déjà un débogueur fonctionnant pour le générateur de code C qui s'appelait BDB. Mais ce dernier n'est plus maintenu.

Le débogueur BDB a été abandonné, principalement parce que c'était une interface à GDB. À chaque nouvelle version de ce dernier, la connexion entre les deux outils était à revoir. Cela impliquait des coûts de maintenance bien trop prohibitifs. Le fait de dépendre de GDB et du langage C pour débogueur du Scheme entraînait aussi des problèmes. En effet, la compilation du C produit un code exécutable en langage machine. Ce langage machine est différent pour chaque architecture matérielle. GDB est ainsi obligé d'instrumenter l'exécution du programme de manière différente d'un langage machine à l'autre. BDB n'ayant pas le moyen de contrôler ces différences de comportement, cela se traduisait par des imprécisions lors du déboguage.

3.2 Scheme et JVM

Pour la première fois dans le domaine du déboguage dynamique, on dispose d'un moyen standardisé et multi-plateformes de réaliser un débogueur grâce aux différentes APIs fournies avec la JVM. Ces APIs permettent de faire communiquer le débogueur avec le programme débogué. Elles réifient des concepts primitifs de l'exécution (comme une machine virtuelle, une requête de breakpoint, une valeur de retour de fonction…) pour les rendre accessibles à partir d'un programme Java.

Ces APIs nous permettent de réaliser à moindre coût un débogueur pour le générateur de code JVM de Bigloo. Cette approche a un certain nombre d'avantages par rapport à l'approche de BDB :

- Les APIs utilisées sont standards, notre débogueur est donc portable sur toutes les JVMs qui les implémentent.

[4] Par exemple avec les pointeurs, où le type réel d'un objet référencé n'est jamais connu.
[5] Qui sera ensuite passé à un compilateur comme GCC pour créer l'exécutable.

- Le code compilé s'exécute dans une machine virtuelle Java, il n'y a donc théoriquement plus de problème lié au fait d'instrumenter directement le processeur d'une architecture matérielle, ni de problème de portabilité du code généré par Bigloo.
- Il n'y a plus d'outil qui vienne s'interposer entre le débogueur et le programme à déboguer, comme cela était le cas avec BDB. Ainsi nous évitons les problèmes ponctuels de compatibilité dus à l'utilisation d'un outil intermédiaire.
- Il n'est pas nécessaire de disposer d'une version spéciale de la bibliothèque d'exécution de Bigloo compilée pour le déboguage.

4 Le débogueur Bugloo

Pour rendre le déboguage plus pratique à utiliser, nous avons développé le débogueur Bugloo. C'est un débogueur qui permet d'instrumenter l'exécution d'un programme. Le contrôle de l'exécution qu'offre Bugloo est quasiment équivalent à celui des débogueurs traditionnels comme GDB. Le code de Bugloo est un mélange de 4000 lignes de Java, de 900 lignes de Scheme et de 100 lignes de C. Dans cette section nous décrivons les mécanismes qui sont le cœur du fonctionnement du débogueur, nous expliquons les principes d'utilisation de Bugloo et nous présentons une analyse de ses performances.

4.1 La JVM et le déboguage

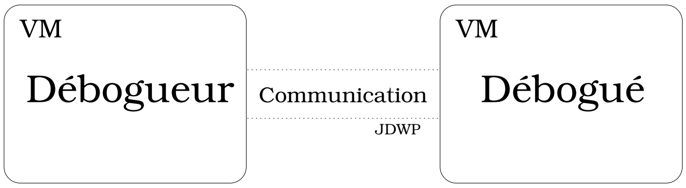

Fig. 1 – Modèle de déboguage de la JVM

La figure 1 est une vision schématique du modèle de déboguage de la JVM. Lorsqu'on débogue un programme on est toujours en présence de deux machines virtuelles :
- Le débogueur s'exécute dans la première JVM. Elle est créée lorsque l'utilisateur lance Bugloo ; c'est le processus classique de chargement de tout programme Java en mémoire.
- La seconde JVM est créée par le débogueur. C'est dans celle-ci que sera exécuté le programme à déboguer. C'est une JVM classique lancée dans un mode particulier qui permet au débogueur de l'instrumenter en temps-réel.

Le principe général du déboguage avec la JVM consiste à suspendre ponctuellement l'exécution du programme débogué, de recueillir des informations sur son état ainsi que celui de la JVM dans laquelle il s'exécute, puis de reprendre l'exécution.

La communication entre le débogueur et le débogué se fait à l'aide de deux APIs de déboguage : JVMDI [18] et JVMPI [19]. Chacune de ces APIs permet d'obtenir un certain type d'information sur le changement d'état du débogué au cours de son exécution. La première est une API pour le langage Java ; c'est celle qui permet à Bugloo de communiquer avec la JVM déboguée. La seconde est une API d'un peu plus bas niveau, elle fonctionne pour le langage C et les bouts de programmes qui l'utilisent doivent résider dans la JVM déboguée. Elle ne permet pas de faire transiter de l'information entre le débogueur et le débogué, c'est pourquoi Bugloo l'utilise indirectement.

Le protocole JDWP définit la manière dont sont transportées les informations qui transitent entre le débogueur et le débogué. Ce protocole permet d'abstraire le type de liaison qui relie les deux JVMs entre elles. Deux types de liaison standards sont fournis avec la JVM, pour pouvoir déboguer un programme sur une machine locale ou sur une machine distante.

4.2 Récupération des événements

Les informations que reçoit le débogueur sur l'état du débogué sont appelées des *événements*. Ces événements correspondent à des changements d'état du programme débogué :
- Exécution pas à pas
- Chargement d'une classe
- Déclenchement d'une exception
- Création et destruction d'un *thread*
- Arrêt sur un breakpoint, un watchpoint
- Entrée dans une méthode, sortie d'une méthode
- Démarrage et terminaison de la JVM déboguée

Ces événements peuvent avoir pour effet de suspendre l'exécution du programme débogué, ou peuvent simplement être émis à titre indicatif (comme le démarrage de la JVM par exemple). La figure 2 présente la manière dont Bugloo traite ces événements.

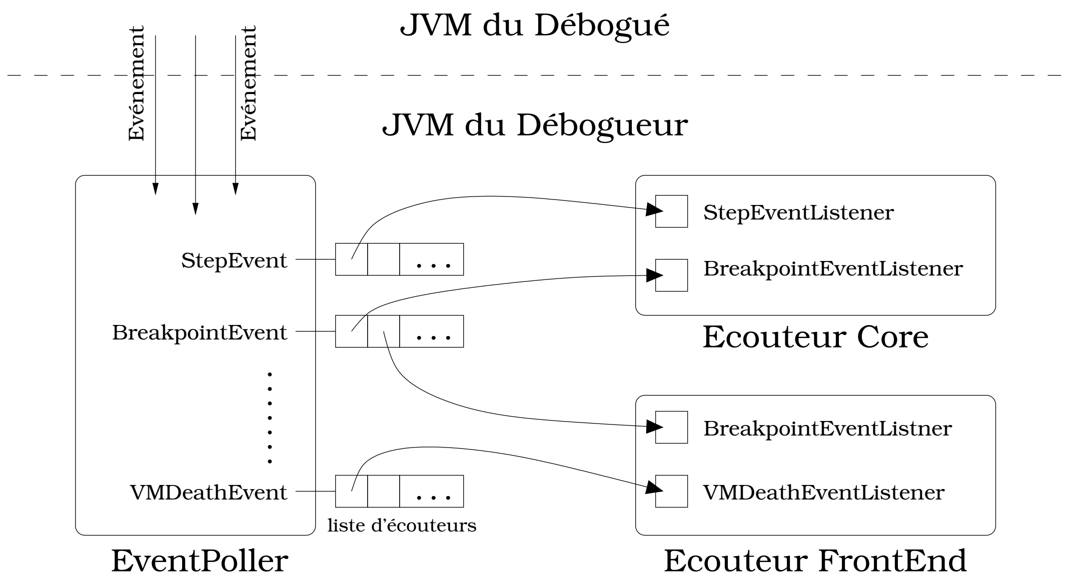

Fig. 2 – Gestion des événements

L'objet `EventPoller` se charge de récolter les événements émis par la JVM déboguée. Au démarrage du débogueur, des écouteurs s'enregistrent auprès d'`EventPoller` pour recevoir des types d'événement qui les intéressent :

– L'ecouteur `Core` s'enregistre en premier. C'est lui qui implémente tous les mécanismes internes fondamentaux du débogueur.
– L'ecouteur `FrontEnd` s'enregistre en second. C'est lui qui s'occupe des traitements « non vitaux » (comme afficher un message à l'écran lorsqu'un événement est reçu).

`Core` est capable de court-circuiter la propagation d'un événement. Par exemple, lorsqu'un breakpoint est atteint, un événement *Breakpoint* est envoyé par la JVM déboguée. Si la durée de vie de ce breakpoint a expiré, `FrontEnd` n'est pas informé qu'un événement a été émis.

`EventPoller` est un objet qui possède son propre fil d'exécution. Pendant tout le processus de déboguage, il effectue en boucle les opérations suivantes :

1. Il attend qu'un événement soit émis par la JVM déboguée
2. Il réveille les écouteurs qui sont intéressés par le type d'événement reçu
3. Si l'événement reçu a suspendu l'exécution du débogué, il notifie l'utilisateur et se met en attente.
4. Il recommence ce processus quand l'utilisateur reprend l'exécution du débogué.

L'algorithme complet est donné en annexe.

4.3 Mécanismes de contrôle du débogué

Le fait que le débogué s'exécute dans sa propre JVM permet au débogueur d'être très peu invasif. Il est donc acceptable de considérer que le programme débogué se comporte de la même manière pendant son déboguage que lors d'une exécution normale. Mais devoir raisonner avec deux JVMs entraîne aussi un certain nombre de problèmes. Par exemple, il est possible de demander à la JVM dans laquelle on s'exécute une liste de tous les objets vivants dans le tas à un instant donné. Pour obtenir des informations sur l'état de la JVM déboguée, BUGLOO doit donc forcer l'exécution d'une méthode dans cette JVM et attendre que le résultat lui soit automatiquement retransmis grâce à JDWP.

Ainsi, il est nécessaire que le code de certaines fonctionnalités du débogueur soit présent dans la JVM du débogué, et qu'il soit directement exécuté à l'intérieur de cette JVM. Pour implémenter ce genre de fonctionnalités, BUGLOO charge dans la JVM déboguée une *classe de contrôle*, juste avant que le programme débogué ne commence. Cette classe contient le code des fonctionnalités qui doivent être exécutées dans la JVM déboguée. Le programme n'est pas conscient de l'existence de cette classe, il est donc raisonnable de considérer que son exécution ne sera pas influencée par cette intrusion.

La section 5 explique plus en détail la manière dont les fonctionnalités avancées de BUGLOO se servent de cette classe de contrôle pour instrumenter le débogué.

4.4 Utilisation et performances

La figure 3 est une capture d'écran du débogueur BUGLOO. Au démarrage, le programme à déboguer est chargé dans sa JVM, et cette dernière est suspendue avant que le programme ne commence son exécution. L'utilisateur peut ensuite contrôler le débogueur par la ligne de commande, à la manière d'un débogueur classique comme GDB. La documentation des commandes de BUGLOO est donnée en annexe.

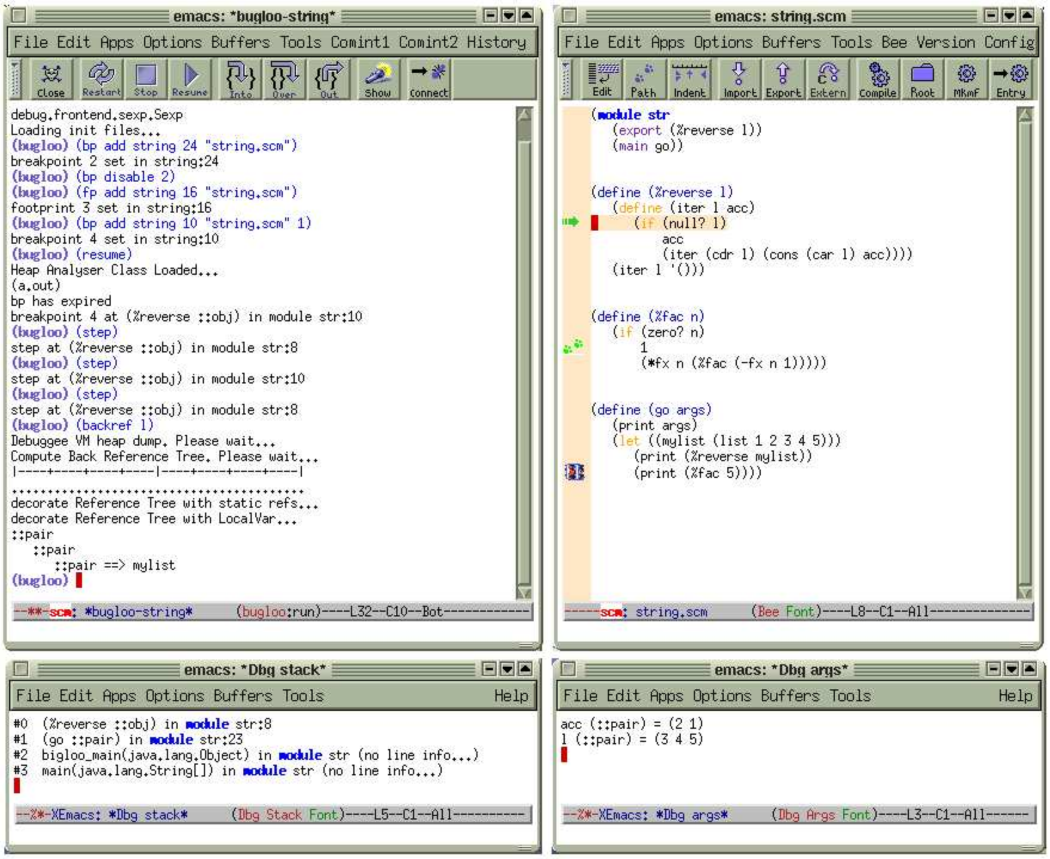

FIG. 3 – Capture d'écran du débogueur BUGLOO

BUGLOO fournit aussi un environnement de déboguage pour les éditeurs Emacs et XEmacs. Les fonctionnalités principales pour contrôler l'exécution du débogué sont alors accessibles depuis la barre d'outils ou depuis les menus. Il est toujours possible d'accéder à la ligne de commande à l'intérieur de l'éditeur.

Il est important de vérifier que BUGLOO est bien utilisable dans la pratique. Par exemple, Psd [14] est un débogueur de Scheme qui fonctionne en transformant le code source original en un code qui, lorsqu'il s'exécutera, fournira les propriétés classiques de déboguage. Mais cette approche transformationnelle entraîne une exécution 200 fois plus lente !

Pour tenter d'évaluer l'impact de BUGLOO sur les performances d'exécution, trois séries de tests ont été effectuées. Le test FIB calcule le résultat de la fonction Fibonacci(300). Le test COMP utilise Bigloo pour compiler un programme Scheme de 3500 lignes. Le test RGC est un analyseur syntaxique qui lit 2000 fois un fichier Scheme. Pour chacun de ces programmes nous avons mesuré le temps d'exécution (TIME) et la mémoire totale allouée (MEM) quand ils étaient lancés « normalement », puis quand leur exécution était instrumentée par BUGLOO. La plateforme utilisée pour les tests est un biprocesseur Pentium III 1GHz avec 512 Mo de RAM (Linux 2.2.18). Le tableau 1 présente les résultats.

D'une manière générale, les résultats des tests révèlent que le comportement de l'alloca-

	(vide) $\mathcal{T}_0$	_	FIB TIME	FIB MEM	COMP TIME	COMP MEM	RGC TIME	RGC MEM
NORMAL	_	_	4.12s	85.22 Mo	21.61s	45.62 Mo	44.50s	110.26 Mo
BUGLOO	2.30s	_	6.60s	84.72 Mo	29.40s	46.48 Mo	48.49s	110.25 Mo

TAB. 1 – Impact de l'utilisation de BUGLOO sur les performances

teur mémoire n'est pas modifié par le fait que la JVM déboguée soit instrumentée durant son exécution. Par contre, ces résultats indiquent que le temps d'exécution est accru lorsqu'on utilise le débogueur. Le temps $\mathcal{T}_0$ représente le temps que met BUGLOO à se charger en mémoire. Nous pouvons donc considérer que le rapport

$$\frac{\mathsf{TIME}_{\text{Bugloo}} - \mathcal{T}_0}{\mathsf{TIME}_{\text{Normal}}}$$

est une approximation raisonnable du coût réel du déboguage d'un programme en terme de vitesse d'exécution. Il en ressort que le fait d'utiliser BUGLOO entraîne au minimum un ralentissement de l'exécution allant de 5 à 20%. Ce faible coût provient sûrement du fait que le JIT n'est pas désactivé pendant le déboguage, cela permet de limiter la chute de performance. A titre de comparaison, des tests équivalents effectués avec GDB montrent un ralentissement allant de 2 à 20%. Nous considérons donc que le coût qu'entraîne l'utilisation de BUGLOO est acceptable car l'ordre de grandeur reste inchangé : un très gros programme reste convenablement déboguable.

Lorsque l'utilisateur se sert réellement des fonctionnalités de GDB (comme les breakpoints ou les watchpoints), les performances de déboguage varient beaucoup selon les moyens que lui fournit l'architecture matérielle pour instrumenter l'exécution du débogué. BUGLOO est moins sensible au fait de s'exécuter sur une plateforme particulière, puisque la JVM instrumente toujours de la même manière l'exécution des programmes.

5 Fonctionnalités avancées

Pour que l'utilisateur puisse déboguer ses programmes de manière plus agréable et plus pratique, nous avons implémenté dans BUGLOO un certain nombre de fonctionnalités qui ne sont pas disponibles dans les débogueurs dynamiques classiques comme GDB. Dans cette section nous décrivons les trois grands types de fonctionnalités avancées que propose BUGLOO : la possibilité d'enregistrer des traces des événements qui surviennent pendant l'exécution du programme, la possibilité de faire du déboguage d'allocation mémoire et la possibilité de pouvoir déboguer un programme de manière interactive avec un interprète Scheme.

5.1 Enregistrement d'actions pendant l'exécution

5.1.1 Session de déboguage

Souvent l'utilisateur aimerait pouvoir se rappeler de la configuration[6] dans laquelle se trouve le débogueur et de l'endroit dans le code source où est suspendue l'exécution du

[6] L'ensemble des breakpoints, des watchpoints, etc...

débogué. Cela lui permettrait de continuer une session de déboguage ultérieurement, ou de la rejouer plusieurs fois sans avoir à tout retaper manuellement.

Tout au long de l'exécution, BUGLOO maintient une liste des commandes tapées par l'utilisateur. Cette liste peut être consultée, sauvegardée et rechargée ultérieurement. Ces opérations permettent de se rappeler de l'état dans lequel se trouvait le débogueur à un moment donné du processus de déboguage. Conserver cet historique permet de « sauvegarder » de manière rudimentaire l'état de la session de déboguage.

Lorsque l'utilisateur commence une nouvelle session de déboguage, il peut ré-exécuter automatiquement toutes les commandes contenues dans l'historique en le rechargeant. Cette action permet de ramener le programme dans un état de déboguage équivalent à celui de la session originale, en supposant que ce programme soit parfaitement déterministe. Il ne s'agit donc pas de faire une sauvegarde complète de l'état d'une exécution, mais plutôt de proposer une fonctionnalité qui offre un service à peu près équivalent dans un grand nombre de cas.

BUGLOO utilise aussi cette fonctionnalité pour implémenter un mécanisme de configuration. Au démarrage, il charge un fichier de commande qui définit le comportement par défaut du débogueur. En adaptant ce fichier, l'utilisateur peut par exemple changer la manière d'afficher la valeur des variables du programme, ou définir des breakpoints à poser avant le démarrage de n'importe quel programme débogué.

5.1.2 Trace des affectations de variables

Pendant le déboguage, un problème de typage[7] ou une valeur erronée affectée à une variable entraînent souvent une erreur à l'exécution. Ce genre d'erreur est automatiquement détecté par BUGLOO qui suspend l'exécution du programme débogué. Mais souvent, la simple valeur de la variable incriminée ne suffit pas à comprendre le bogue.

BUGLOO permet d'enregistrer les affectations de variables globales qui ont lieu tout au long de l'exécution du programme débogué. L'utilisateur choisit les variables qu'il souhaite surveiller, et peut déclencher ou débrayer l'enregistrement à chaque fois que le programme débogué est suspendu. Dès qu'une variable surveillée est affectée, le débogueur sauvegarde l'événement dans une liste en relevant l'endroit dans le code où cela s'est produit et la valeur de la variable après affectation. Si plusieurs variables sont surveillées, les événements sont ajoutés à la trace dans l'ordre où ils se produisent.

La manière dont cette fonctionnalité a été implémentée dans BUGLOO est inefficace. Les performances chutent fortement en fonction du nombre d'affectations qui ont lieu pendant l'enregistrement. Cela est dû à la manière dont le débogueur instrumente la JVM déboguée : à chaque événement d'affectation, cette JVM se suspend et BUGLOO doit la forcer à reprendre son exécution. Nous n'avons pas eu le temps d'expérimenter l'autre politique de notification présentée dans la section 4.2, qui permet d'être informé de la présence d'un événement sans suspendre la JVM.

[7]Par exemple l'application d'un objet non fonctionnel.

5.1.3 Historique des appels

Quand le programme débogué est suspendu, l'utilisateur peut demander à BUGLOO des informations sur l'état de la pile d'exécution, c'est-à-dire les fonctions qui sont en attente d'un résultat. Mais avec ce genre de requête, il peut être difficile de comprendre comment le programme débogué est arrivé à l'endroit dans le code source où il est actuellement suspendu.

Comme pour les événements d'affectation de variable, BUGLOO peut enregistrer les événements d'entrée dans une méthode qui se produisent pendant l'exécution du programme débogué. Grâce à cette trace, l'utilisateur a la possibilité de connaître un chemin complet d'exécution plutôt qu'une information limitée comme les calculs suspendus dans le programme. Les débogueurs dynamiques traditionnels comme GDB n'offrent pas ce genre de fonctionnalités.

Cet mécanisme d'enregistrement des événements d'entrée dans une méthode souffre du même problème de performance que la trace des affectations de variables. Le dialogue inutile entre le débogueur et la JVM déboguée pour constamment reprendre son exécution entraîne des pénalités qui pourraient être évitées à l'aide d'une meilleure implémentation.

Plus généralement, le problème de ces deux traces est qu'elles ne sont pertinentes que lorsque l'utilisateur a besoin d'analyser « localement » le comportement d'un programme ou lorsque la quantité d'information enregistrée n'est pas trop importante. En effet, BUGLOO ne dispose pas de mécanismes évolués d'analyse de trace, qui sont plutôt présents dans des outils spécialisés comme Jinsight.

5.2 Déboguage d'allocation mémoire

5.2.1 Objets vivants dans le tas

Lorsque le programme débogué est suspendu, l'utilisateur peut demander à BUGLOO une liste de tous les objets vivants dans le tas de la JVM du programme débogué, c'est-à-dire ceux qui sont encore utilisés par l'application et qui ne seront donc pas ramassés par l'allocateur mémoire (dans la suite GC).

JVMPI permet de renvoyer ce genre d'information. Pour implémenter cette fonctionnalité, BUGLOO se sert donc de la classe de contrôle présentée dans la section 4.3 pour réaliser tous les calculs dans la bonne JVM (celle du débogué) et récupérer automatiquement le résultat. L'utilisateur peut ainsi connaître le nombre d'objets vivants pour chacune des classes présentes dans la JVM déboguée. Il peut aussi filtrer l'affichage des classes à l'aide d'une expression régulière. Cette fonctionnalité permet de vérifier grossièrement que le GC libère bien la mémoire allouée par le programme durant son exécution.

5.2.2 Graphe de références inverses

Scheme est un langage qui utilise un GC pour l'allocation mémoire ; la mémoire est donc libérée automatiquement. Il arrive parfois que le GC se déclenche mais ne puisse plus libérer de mémoire car tous les objets présents dans le tas sont encore *vivants*. Dans la terminologie du GC de la JVM, un objet est vivant s'il est référencé par d'autres objets ou si c'est une *racine* (c'est à dire, s'il est contenu dans une variable statique d'une classe ou dans une variable locale). Une racine n'a pas besoin d'être référencée pour rester vivante.

L'incapacité pour un GC de libérer de la mémoire est souvent le signe d'une *fuite mémoire*. Ce type de problème se produit généralement dans un programme lorsque la taille d'une structure de donnée croît beaucoup plus que prévu.

A partir de la liste d'objets vivants retournée par la classe de contrôle du débogué, Bugloo est capable de reconstruire un *graphe de références inverses*. Dans ce graphe orienté, les nœuds représentent les objets vivants dans le tas de la JVM déboguée et les arrêtes orientées représentent la relation « est référencé par ». Pour un objet donné de la JVM, ce graphe permet de retrouver tous les objets qui le référencent. Le parcours du graphe se poursuit récursivement et s'arrête lorsqu'une racine est atteinte.

Bugloo est ainsi capable de retrouver tous les chemins de références inverses allant d'un objet quelconque du tas jusqu'à une racine du GC. Cela est un moyen de comprendre pourquoi un objet est toujours vivant dans la JVM. Cette fonctionnalité permet donc à l'utilisateur de faire du déboguage d'allocation mémoire, ce qui n'est pas le cas des débogueurs traditionnels à la GDB. L'algorithme de construction du graphe est donné en annexe.

```
 1: (module str (main go))
 2:
 3: (define (myreverse l)
 4:    (define (iter l acc)
 5:       (if (null? l)
 6:           acc
 7:           (iter (cdr l) (cons (car l) acc))))
 8:    (iter l '()))
 9:
10: (define (go args)
11:    (let ((mylist (list 1 2 3 4 5)))
12:       (print (myreverse mylist))))
```

Fig. 4 – Programme d'exemple 1

Pour illustrer le fonctionnement du graphe de références inverses, supposons que l'utilisateur soit en train de déboguer le programme d'exemple de la figure 4 et qu'il décide de suspendre l'exécution du programme débogué à la ligne 7. Supposons qu'il se soit arrêté à un moment de l'itération où la variable locale `acc` vaut `(2 1)`, et la variable locale `l` vaut `(3 4 5)`. La figure 5 représente le résultat qu'affiche Bugloo après avoir calculé le graphe de références inverses.

- La ligne 7 représente le type de la variable locale `l` (c'est une paire).
- La ligne 8 indique que cette variable est référencée par un objet de type `::pair`. Il s'agit de la paire `(2 3 4 5)`.
- La ligne 9 indique enfin qu'une autre paire référence l'objet de la ligne 8. Bugloo reconnaît l'objet en question et indique à l'utilisateur qu'il s'agit de la variable locale `mylist`, définie dans le code source à la ligne 11 de la figure 4.

```
Debuggee VM heap dump. Please wait...
Compute Back Reference Tree. Please wait...
|----+----+----+----|----+----+----+----|
.......................................
decorate Reference Tree with static refs...
decorate Reference Tree with LocalVar...
::pair
   ::pair
      ::pair ==> mylist
```

FIG. 5 – Résultat du calcul du graphe

Le calcul du graphe de références inverses se fait dans le débogueur, cela évite d'utiliser des ressources de la JVM déboguée et permet de conserver le comportement normal du programme débogué.

La complexité de l'algorithme implémenté pour construire le graphe est mauvaise. En effet, l'algorithme construit le graphe de manière à conserver tous les chemins possibles de références inverses. A chaque fois qu'un objet référence un autre objet, une arrête est créée dans le graphe. Ainsi, il faut sensiblement la même quantité de mémoire pour stocker l'arbre de références inverses que pour stocker les vrais objets dans la JVM déboguée. Cette contrainte limite l'utilisation de cette fonctionnalité dans la pratique.

Dans KBDB, les auteurs modifient l'allocateur mémoire de manière à conserver pour chaque objet une seule référence inverse. Leur algorithme ne construit pas toutes les arrêtes du graphe, mais cela lui permet d'avoir des performances raisonnables. De plus, un seul chemin est suffisant pour détecter une fuite mémoire. Une implémentation similaire pour la construction de notre graphe nous aurait permis de conserver les avantages des deux approches : une fonctionnalité peu gourmande en mémoire et non intrusive.

5.3 Interprétation de code rajouté à l'exécution

5.3.1 Interprète Scheme embarqué

La manière classique de déboguer un programme est de suspendre ponctuellement son exécution puis d'inspecter la valeur de certaines de ses variables. Cette interaction avec le programme débogué reste d'une manière générale très limitée. Souvent dans la pratique, l'utilisateur aimerait pouvoir demander au débogué la valeur de véritables expressions écrites dans le même langage que celui du programme débogué.

Lorsque l'exécution du programme est suspendue, BUGLOO est capable de lancer un interprète Scheme dans la JVM déboguée dans le but de rendre le déboguage plus interactif. L'environnement lexical de cet interprète est étendu par celui du programme débogué à l'endroit où il a été arrêté. L'utilisateur peut ainsi dialoguer avec le programme débogué en utilisant toute l'expressivité d'un langage. En particulier il est capable de définir des nouvelles fonctions à la volée.

Pour implémenter cette fonctionnalité, il faut que la classe contenant le code de l'interprète Scheme puisse être rajoutée dynamiquement dans la JVM du programme débogué. Le chargement de l'interprète sera automatiquement effectué par la classe de contrôle du débogué au moment où l'utilisateur aura besoin de cette fonctionnalité.

5.3.2 Breakpoints conditionnels évolués

Pendant une session de déboguage, l'utilisateur se sert de breakpoints pour suspendre son programme à des endroits précis du code source. Parfois, il souhaiterait que l'exécution soit suspendue seulement « à une condition ». BUGLOO fournit des breakpoints conditionnels qui permettent de réaliser ce genre d'action. Une condition est une expression logique qui exprime une propriété sur le programme. Lorsqu'un breakpoint conditionnel est atteint, sa condition est évaluée. L'exécution du programme est suspendue si le résultat de l'évaluation vaut la valeur logique *vrai*.

Les débogueurs traditionnels comme GDB permettent déjà de poser des breakpoints conditionnels, mais le langage de condition qu'ils utilisent est toujours un sous-ensemble très primitif du langage dans lequel est écrit le programme à déboguer. Dans le cas du langage C, le langage utilisé pour les conditions permet d'effectuer des opérations arithmétiques sur la valeur d'une variable, ou éventuellement d'appeler des fonctions du programme débogué. Ce sous ensemble de langage n'est jamais défini formellement, mais il est suffisamment restreint pour empêcher l'utilisateur d'écrire des conditions évoluées contenant par exemple des fonctions locales ou des variables statiques.

BUGLOO propose d'utiliser Scheme comme langage de condition. La condition est une *fermeture* qui prend en paramètre l'environnement lexical du programme débogué et qui retourne la valeur booléenne *vrai* pour suspendre l'exécution du programme. Cela permet d'obtenir une expressivité strictement équivalente à celle du langage du programme débogué. Il est ainsi possible d'utiliser des variables ou des fonctions locales et de faire des affectations. L'utilisation d'une fermeture rajoute une « mémoire » à la condition. Toutes ces fonctionnalités permettent à l'utilisateur d'écrire des conditions très évoluées.

```
(module ex2 (main go))

(define (gcd x y)
   (cond
      ((> x y) (gcd y x))
      ((< x y) (gcd x (- y x)))
      (else x)))

(define (go args)
  (print (gcd 25 453)))
```

FIG. 6 – Programme d'exemple 2

Pour illustrer le fonctionnement des breakpoints conditionnels, supposons que l'utilisateur soit en train de déboguer le programme d'exemple de la figure 6 et qu'il ait posé un breakpoint conditionnel à la ligne 4. L'utilisateur pourrait suspendre l'exécution du programme quand la valeur de y serait un multiple de 7 pour la troisième fois, en écrivant la lambda donnée dans la figure 7.

```
(let ((n 1))
  (lambda (env)
    (if (= 0 (modulo (env-ref env 'y) 7))
      (if (= 3 n)
        #t
        (set! n (+ n 1))))))
```

FIG. 7 – Condition sous forme de lambda-expression

Comme pour l'interprète Scheme embarqué, l'environnement lexical de la condition est étendu par celui du programme débogué à l'endroit où il a été arrêté, mais il n'est plus possible de le faire de manière transparente pour l'utilisateur. En effet, une condition est une fermeture et la seule manière pour BUGLOO de transmettre cet environnement lexical est de le passer en paramètre à la lambda de la condition.

BUGLOO étend la fermeture de l'utilisateur avec la définition de la fonction locale `env-ref`. Pour accéder à l'environnement lexical du programme à travers l'argument opaque `env`, il est nécessaire d'appeler cette forme `env-ref` avec le symbole représentant la variable à accéder.

La syntaxe des conditions n'est pas exactement identique à celle du programme débogué puisque l'utilisateur est obligé de programmer le corps de la lambda de manière « indirecte » en utilisant la forme `env-ref`. Toutefois, même si cette approche n'est sans doute pas la meilleure, elle a le mérite d'avoir une sémantique claire : le fait de rendre explicite le paramètre `env` permet à l'utilisateur de créer une fermeture, et définir la fonction locale `env-ref` revient à envelopper cette fermeture par l'instruction Scheme `let`.

Lorsque l'utilisateur crée un breakpoint conditionnel, une fermeture est automatiquement créée par la classe de contrôle dans la JVM déboguée. Lorsque le breakpoint conditionnel est atteint, il faut toujours évaluer la même fermeture pour que la « mémoire » de la condition fonctionne. Il est donc nécessaire de conserver une référence sur ces fermetures dans la JVM débogué, sous peine que le GC ne les collecte malencontreusement. C'est encore une fois la classe de contrôle du débogué qui maintient une liste des fermetures de tous les breakpoints conditionnels de BUGLOO. Quand l'un d'eux est supprimé, sa fermeture est automatiquement retirée de la liste.

6 Comparaison avec les travaux antérieurs

Bugloo est un débogueur qui permet à l'utilisateur de détecter des comportements anormaux dans son programme au moment de l'exécution. C'est un outil qui aborde le problème du débogage de manière dynamique. En réalité, toutes les approches de débogage sont complémentaires car chacune d'entre elles se spécialise dans la résolution d'un certain type de problème. Dans cette section, nous comparons la manière dont Bugloo traite les problèmes susceptibles d'apparaître dans un programme par rapport aux approches antérieures mises au point dans chacun des grands domaines du débogage.

6.1 Déboguage par analyses statiques

Durant une session de déboguage, Bugloo suspend automatiquement l'exécution du programme débogué lorsqu'un bogue direct survient (par exemple une erreur de typage). Souvent, pour comprendre comment une erreur a pu se produire, le seul moyen dont dispose l'utilisateur est de rejouer l'exécution en enregistrant une trace de la variable incriminée.

Pour résoudre ce genre d'erreurs directes, les analyses statiques sont les outils les plus puissants. Ce sont en général des analyses ensemblistes, ou *set based analysis* (SBA [11]). Conceptuellement, ce genre d'analyse essaie de calculer un ensemble approché des valeurs que peuvent prendre toutes les expressions d'un programme. Il est ainsi facile de déterminer si certaines valuations de ces expressions pourront déclencher des erreurs, et si c'est le cas de localiser précisément la cause du bogue.

Mr Spidey [10] est un débogueur statique pour le langage Scheme. Ses auteurs ont développé la *componential SBA* [9] qui permet de calculer l'ensemble des valeurs que peuvent prendre les variables d'un programme. Leur algorithme analyse chacun des modules du programme séparément, cette relaxation de la SBA permet de traiter des programmes de plusieurs milliers de lignes. Pessaux et Leroy introduisent une analyse de type [21] pour les exceptions non rattrapées dans le langage fonctionnel Objective CAML[8]. Le principe de fonctionnement est identique à celui des analyses de valeur : à chaque point du programme, l'analyse calcule l'ensemble des exceptions pouvant être levées. Cela permet d'exhiber les exceptions non rattrapées dans le programme ainsi que les valeurs possibles de leurs arguments, ce que les compilateurs classiques de ML ne détectent pas.

Des approches d'analyses « locales » ont été développées, comme l'analyse CLA [13, 12] de Heintze et Tardieu, qui permet de connaître l'ensemble des éléments que peuvent référencer les pointeurs d'un programme. Les auteurs proposent une analyse interactive dirigée par l'utilisateur qui repose sur l'idée que souvent cet utilisateur a uniquement besoin d'informations sur une partie du programme.

Il serait intéressant de pouvoir utiliser ce genre d'analyses statiques dans Bugloo. On pourrait imaginer de lancer dynamiquement des analyses qui tiendraient compte de l'état dans lequel se trouvait le programme débogué au moment où son exécution a été suspendue. Cela permettrait peut-être d'aider l'utilisateur à comprendre le comportement de son programme, tout en limitant la complexité des analyses.

[8]Objective CAML [5] est une extension de ML, langage qui fonctionne par inférence de type.

6.2 Déboguage par assertions

La notion de contrat permet de préciser le sens d'un programme en exprimant dans un langage particulier des propriétés logiques sur les fonctions du programme supposées être vraies pendant l'exécution. Ces assertions sont en général intégrées au programme source et donc compilées en même temps que lui. Le fonctionnement des assertions est similaire aux breakpoints conditionnels de BUGLOO : quand une assertion est fausse, le programme déclenche une erreur.

BUGLOO pourrait s'inspirer de ce paradigme de programmation en permettant à l'utilisateur de rajouter dynamiquement des contrats sur les fonctions de son choix pendant la session de déboguage. Il suffirait d'utiliser une technique similaire aux conditions présentées dans la section 5.3.2. Le principe consisterait à exécuter une fermeture (la pré-condition) lorsqu'on entre dans une fonction du programme débogué, puis à exécuter une autre fermeture (la post-condition) lorsqu'on sort de la fonction.

JML [16] est un langage de spécification comportemental pour Java. Il permet d'écrire des propriétés logiques très évoluées sur les méthodes d'un programme. JML utilise le langage Java étendu par des quantificateurs pour l'expressivité de la logique. Il permet entre autres de déclarer des variables locales ou d'utiliser la notation `old` d'Eiffel[9]. Certaines de ses fonctionnalités sont là aussi facilement implémentables pour BUGLOO grâce à l'utilisation de fermetures pour les conditions.

6.3 Déboguage pendant l'exécution

Le fonctionnement de BUGLOO se base sur le modèle classique de déboguage à l'exécution. Il permet de contrôler l'exécution du programme débogué de manière quasiment équivalente à celle de GDB. L'API de déboguage JVMDI propose plusieurs scénarios de connexion entre le débogueur et le débogué. Mais actuellement, BUGLOO n'est pas encore capable de « s'accrocher » à un programme en cours d'exécution. D'autres fonctionnalités sont encore absentes, comme les watchpoints conditionnels ou la possibilité de modifier la valeur d'une variable pendant l'exécution.

Il serait très important que BUGLOO puisse être utilisé de manière complètement asynchrone, comme c'est le cas pour les débogueurs concurrents. En effet, lorsque l'utilisateur demande au programme débogué de continuer son exécution, il est obligé d'attendre que la JVM s'interrompe de nouveau pour reprendre la main. Cette limitation provient du fait que BUGLOO se commande depuis la console et qu'il est très difficile avec cette interface d'afficher en même temps la ligne de commande de l'utilisateur et les sorties du programme débogué. Il serait ainsi intéressant de développer une interface plus « visuelle ».

Un autre point intéressant à explorer est la possibilité pour BUGLOO defaire du déboguage de programme sur des machines distantes. JDWP permet au débogueur de ne pas se préoccuper de l'endroit où s'exécute la JVM déboguée, BUGLOO est donc capable de déboguer un programme à distance automatiquement. Cependant, dans cette hypothèse de déboguage, il faut veiller à limiter au maximum le dialogue entre le débogueur et le débogué.

[9]Cette notation fait référence à la valeur qu'avait une variable la dernière fois qu'une méthode avait été appelée.

A l'heure actuelle, Bugloo n'est pas suffisamment optimisé pour déboguer à une vitesse convenable un programme à distance.

7 Conclusion

Durant ce stage nous avons développé Bugloo, un débogueur dynamique qui permet d'instrumenter l'exécution d'un programme écrit en Scheme pour le compilateur Bigloo ou en Java. Bugloo offre un contrôle quasiment équivalent à celui de GDB. Dans le but de le rendre plus agréable et plus pratique à utiliser que les débogueurs dynamiques traditionnels, nous avons rajouté des fonctionnalités supplémentaires comme l'enregistrement de traces durant l'exécution, le déboguage d'allocation mémoire ou l'interprète Scheme embarqué dans le programme débogué.

Un certain nombre de fonctionnalités n'a pas pu être implémenté. Par exemple, il serait utile de disposer d'un véritable interprète Scheme pour la ligne de commande du débogueur. Cela permettrait à l'utilisateur d'exécuter du code dans le débogueur lorsqu'un breakpoint ou un watchpoint serait atteint. D'autres fonctionnalités ont été implémentées mais de manière encore inefficace ou pas assez pratique. Ainsi, le graphe de références inverses n'est pas utilisable pour des programmes consommant beaucoup de mémoire. Il faudrait aussi pouvoir recouper les informations provenant des différents types de trace que propose Bugloo. Dans cette optique, il serait intéressant de disposer d'une interface graphique plus aboutie.

Des techniques implémentées dans Bugloo comme les breakpoints conditionnels semblent relativement proches de concepts utilisés dans d'autres branches du déboguage, notamment les assertions et la notion de contrat. D'une manière générale, il serait très intéressant de poursuivre le développement de Bugloo en y incorporant des techniques issues du domaine de l'analyse statique ou du déboguage par assertions. Cela permettrait d'obtenir un débogueur dynamique complet intégrant des fonctionnalités qui sont en général uniquement présentes dans les outils des autres types d'approche de déboguage.

Références

[1] D. Bobrow, L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. Common lisp object system specification. In *special issue*, number 23 in SIGPLAN Notices, September 1988.

[2] H.J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software — Practice and Experience*, 18(9) :807–820, September 1988.

[3] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, pages 46–55, 1993.

[4] F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 501–516. Springer-Verlag, 1993.

[5] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective CAML*. O'Reilly France, 2000.

[6] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, CA, USA, January 1977.

[7] W. de Pauw and G. Sevitski. Visualizing reference patterns for solving memory leaks in Java. pages 116–134, 1999.

[8] Robert Bruce Findler and Matthias Felleisen. Contract for higher-order functions. In *To appear in International Conference on Functional Programming*, 2002.

[9] C. Flanagan and M. Felleisen. Modular and polymorphic set-based analysis : Theory and practice, 1996.

[10] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. *ACM SIGPLAN Notices*, 31(5) :23–32, 1996.

[11] Nevin Heintze and Joxan Jaffar. Set constraints and set-based analysis. In *Principles and Practice of Constraint Programming*, pages 281–298, 1994.

[12] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, 2001.

[13] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA : A million lines of c code in a second. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, 2001.

[14] P. Kellomaki. Psd — a portable scheme debugger, 1995.

[15] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9) :26–76, 1998.

[16] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML : A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[17] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, 1997.

[18] Sun Microsystems. *The Java Virtual Machine Debugger Interface Specification*. http ://java.sun.com/j2se/1.4/docs/guide/jpda/jvmdi-spec.html, 2002.

[19] Sun Microsystems. *The Java Virtual Machine Profiler Interface Specification*. http ://java.sun.com/j2se/1.4/docs/guide/jvmdi, 2002.

[20] Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. *Lecture Notes in Computer Science*, 821 :163– ? ?, 1994.

[21] Francois Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *Symposium on Principles of Programming Languages*, pages 276–290, 1999.

[22] Manuel Serrano and Hans-J. Boehm. Understanding memory allocation of scheme programs. *ACM SIGPLAN Notices*, 35(9) :245–256, 2000.

[23] Manuel Serrano and Pierre Weis. Bigloo : A portable and optimizing compiler for strict functional languages. In *Static Analysis Symposium*, pages 366–381, 1995.

[24] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. 2001.

[25] Richard Stallman, Roland H. Pesch, and Richard Stallman. *Debugging with GDB : the GNU source-level debugger*. Free Software Foundation, 4.09 for GDB version 4.9 edition, 1993. Previous edition published under title : The GDB manual. August 1993.

A Annexes

A.1 Création du graphe de références inverses

{*création des nœuds et des arrêtes*}
$H_{node} \leftarrow$ nouvelle table de hachage pour les nœuds du graphe
$O_{\text{JVM}} \leftarrow$ récupérer les objets de la JVM déboguée
pour tout $o \in O_{\text{JVM}}$ **faire**
  $node_{parent} \leftarrow$ nœud_associe(H_{node}, o)
  si (o est un tableau) **alors**
    pour tout e élément du tableau o **faire**
      $node_{fils} \leftarrow$ nœud_associe(H_{node}, e)
      ajouter $node_{parent}$ comme parent de $node_{fils}$
    fin pour
  sinon si (o est un object) **alors**
    pour tout c champ de o qui contient un objet **faire**
      $node_{fils} \leftarrow$ nœud_associe(H_{node}, valeur(c))
      ajouter $node_{parent}$ comme parent de $node_{fils}$
    fin pour
  fin si
fin pour

{*marquer les nœuds qui sont des racines du GC*}
$C_{\text{JVM}} \leftarrow$ toutes les classes chargées dans la JVM déboguée
pour tout $c \in C_{\text{JVM}}$ **faire**
  pour tout s champ statique de la classe c qui contient un objet **faire**
    $node \leftarrow$ nœud_associe(H_{node}, valeur(s))
    marquer $node$ comme étant une racine
  fin pour
fin pour
$T_{\text{JVM}} \leftarrow$ tous les Threads de la JVM déboguée
pour tout $t \in T_{\text{JVM}}$ **faire**
  $B \leftarrow$ tous les blocs d'activation de la pile d'appels de t
  pour tout $b \in B$ **faire**
    pour tout v variable locale du bloc d'activation b **faire**
      si (v contient un objet) **alors**
        $node \leftarrow$ nœud_associe(H_{node}, valeur(v))
        marquer $node$ comme étant une racine
      fin si
    fin pour
  fin pour
fin pour

A.2 Récupération des événements de la JVM débogué

répéter
  $suspendu \leftarrow faux$
  $doit\_reprendre \leftarrow faux$
  attendre un événement dans la queue d'événements Q_{event}
  tantque Q_{event} non vide **faire**
    $e \leftarrow$ dépiler(Q_{event})
    si e a suspendu la JVM **alors**
      $suspendu \leftarrow vrai$
    fin si
    $L \leftarrow$ liste des écouteurs intéressés par l'événement e
    pour tout $l \in L$ **faire**
      l.traite_evenement(e)
    fin pour
  fin tantque
  {*un écouteur a pu demander la reprise de l'exécution de la JVM*}
  si ($doit\_reprendre = vrai$) **alors**
    reprendre l'exécution de la JVM
  sinon si ($suspendu = vrai$) **alors**
    donner la main à l'utilisateur et attendre
  fin si
tant que la JVM est vivante