Travail d'étude et de recherche
Débogueur pour langage impératif

Erwan Lemattre

Janvier 2024 – Avril 2024

1 Introduction

Ce travail d'étude et de recherche a pour objectif de développer un débogueur pour langage impératif en utilisant le langage OCaml. Ce projet s'inscrit dans la continuation du cours de compilation dans lequel avait déjà été développé des interpréteurs pour langages impératifs et objets. Le langage utilisé pour ce débogueur se base en partie sur le langage l'un de ces langages. C'est un langage impératif simple appelé `Imp` dont nous décrirons les spécificités dans la première partie de ce rapport. Ce langage a été la base sur laquelle a été construit toutes les fonctionnalités nécessaires à la création du débogueur. Nous discuterons en détails des fonctionnalités de débogage et de leurs implémentations dans la seconde partie de ce rapport.

2 Le langage `Imp`

2.1 Fonctionnalités

Ce langage contient toutes les instructions de base qu'on peut attendre d'un langage impératif :

- Boucle `while`
- Condition `if`/`else`
- Fonctions
- Tableaux de taille fixe

2.2 Stockage des variables

Le langage `Imp` permet de créer des fonctions, des variables globales ainsi que des variables globales. L'exécution pas à pas ainsi et les fonctionnalités comme le retour en arrière ont nécessité une structuration différente du code. En effet, le fait de pouvoir retourner en arrière implique le stockage des états du programme à chaque nouvelle instruction ou suite d'instruction. On peut retrouver dans le fichier `global.ml` l'ensemble des variables globales à l'interpréteur dont la pile `undo_stack` qui est une liste des états du programme. À chaque nouvelle commande, l'état est ajouté à la pile. Pour revenir en arrière il suffit simplement de dépiler. On trouve également dans le fichier `gloabl.ml` les environnements locaux aux fonctions ainsi que l'environnement global. On utilise à présent des références qui peuvent facilement être modifiées. Cela permet lors d'un retour en arrière de récupérer l'ancien élément dans la pile et de le définir comme l'environnement courant.

2.3 Inférence de type

2.3.1 Un première réflexion

Le langage `Imp` n'a pas de définition de type explicite pour les variables et les fonctions. La première idée pour réaliser la vérification de type a été de vérifier seulement les variables et les fonctions dont on connait le type. Les éléments dont le type est inconnu sont accepté dans tous les cas car ils peuvent être du type souhaité (on saura à l'exécution si le type est le bon). Cette méthode implique plus d'erreur à l'exécution du programme.

2.3.2 Vérifier avec de l'inférence de type

La seconde idée, qui est celle utilisée dans ce projet, a été de réaliser un système d'inférence de type. L'idée de l'inférence de type est ici de générer des contraintes puis ensuite d'utiliser un algorithme d'unification pour vérifier que toutes les contraintes sont respectées. Les contraintes sont générées dans le fichier `typechecker.ml`. Pour chaque expression on génère un nouveau type qui représente le type de l'expression. Les noms de variables sont générés de manière unique par la fonction `get_var_name`. La fonction `type_expr` va retourner pour chaque expression la liste des contraintes sur cette expression ainsi que le type de retour de l'expression. Les instructions n'ayant quant à elles pas de valeur de retour, elles renvoient seulement les contraintes associées.

2.3.3 Inférer le type des fonctions

Une question intéressante à traiter a été comment inférer le type des fonctions. Il y a eu deux idées. D'abord, j'ai pensé à faire l'inférence sur les fonctions unes à unes et séparemment. Le problème est que cela implique une vérification des fonctions dans un ordre précis : il faut vérifier les fonctions qui sont appelées avant les fonctions dans lesquelles il y a les appels. Cela empêche également les fonctions récursives. Cette méthode étant trop restrictive, il a fallu faire autrement. La seconde idée utilisée pour ce langage est de regrouper toutes les contraintes ensembles et de les résoudres en une seule fois. Avant la vérification des types on commence par générer un ensemble de variables pour les fonctions. Lors de la vérification il suffit de récupérer le nom de ces variables pour nos contraintes. Cette méthode ne permet pas le polymorphisme des fonctions et implique que chaque fonction est d'un seul et unique type.

3 Débogage

3.1 Les fonctionnalités du débogueur

3.1.1 Commandes de débogage

Les commandes proposées sont :

- `next` : passer à l'instrution suivante
- `so` : step over – si appel de fonction, boucle ou condition passe au dessus
- `step` : s'arrête au prochain breakpoint. Si pas de breakpoint exécute tout le programme
- `break n` : ajoute un point d'arrêt à la ligne n
- `undo` : retourner en arrière
- `exit` : quitte le programme

Les commandes se basent principalement sur la fonction `step` qui permet d'exécuter une unique instruction et retourne le nouvelle environnement et les instructions suivantes. Le fait de retourner les instructions suivantes permet dans certains cas d'ajouter des instructions après l'exécution d'une instruction. Prenons par exemple le cas d'une instruction `while`. Si la condition d'entrée est vraie alors on ajoute le corps de la boucle dans la liste des prochaines instructions. Sinon rien n'est ajouté et les prochaines instructions sont celles qui suivent le `while`. On retrouve également ce fonctionnement avec les instructions conditionnelles. Nous verrons par la suite le cas particulier des appels de fonction. À partir de `step` on peut passer à l'instruction suivante (`next`), sauter une boucle ou condition (`so`) en répétant `step` tant qu'on est dans la boucle et aller au prochain point d'arrêt (`step`) en continuant `step` tant que l'instruction n'est pas un point d'arrêt. Les points d'arrêt sont stockés dans une table de hachage avec pour clé les numéros de ligne des points d'arrêt. Enfin comme expliqué en 2.2 le retour en arrière se fait en récupérant le dernier état du programme.

3.1.2 Les appels de fonctions

Les appels de fonctions ont été un point délicat à traiter. Les appels de fonctions sont des expressions qui comme les boucles ou les conditions vont ajouter un ensemble d'instructions. De plus, il faut créer un nouvel environnement local à l'appel. Ces premières

conditions n'ont pas vraiment posé de problème. La complexité vient du fait que des appels de fonctions peuvent se trouver par exemple en paramètre d'appels de fonctions, dans un tableau ou encore dans une opération arithmétique : il n'est plus possible d'exécuter une instruction en une seule fois. La solution a été de créer une fonction qui vérifie s'il y a un appel de fonction dans la prochaine instruction. Dans le cas où il n'y en a pas, on peut exécuter l'instruction et passer à la suivante. Dans le cas où on trouve un appel, on le remplace par une expression spéciale appelée `continuite` et on rajoute l'instruction modifiée à la séquence d'instruction. On ajoute ensuite les instructions de notre fonction. Ainsi les prochaines instructions seront celle de la fonction. Lorsqu'on trouve une instruction `return` la valeur retournée est ajouté dans une pile de retour. Un fois la fonction terminée on retrouve notre instruction dont l'appel a été remplacé par `continuite`. L'expression `continuite` est simplement remplacée par le retour de l'appel de fonction stocké dans la pile de retour. Ce mécanisme d'appel de fonction permet par exemple de gérer des cas comme `a = f(x + g(x), h(y) - g(x))` pour lesquels on doit rester à la même instruction après les quatres appels.

3.2 Suivi de la mémoire

Dans l'objectif de pouvoir suivre l'évolution de la mémoire, j'ai ajouté les informations *alive* et *free* sur les tableaux. Ces informations permettent de savoir si un tableau est toujours utilisé ou si dans le cas contraire il n'est plus accessible et a été libéré. Pour obtenir ces informations j'ai ajouté un identifiant unique à chaque tableau. À partir des variables de notre environnement il suffit ensuite de marquer les identifiants des tableaux qu'on trouve lors du suivi des variables. Les identifiants non marqués sont libres. On retrouve ici un fonctionnement similaire à celui des *Garbage Collector* avec le *mark and sweep*.

3.3 Affichage du code

L'affichage du code se fait avec un *pretty printer*. Ce *pretty printer* nous permet d'obtenir une indentation du code pour obtenir un code lisible. Le problème soulevé par l'affichage a été de colorer seulement une instruction pour indiquer notre position dans le code. De plus la première version de ce débogueur utilisant la librairie *ncurses*, il a fallu trouver une solution pour laisser la possibiliter d'ajouter de la couleur sans "casser" l'indentation. La solution a été d'ajouter des balises `<color>` avec le *pretty printer*. Les balises étant ajoutées avec la librairie *Format* d'OCaml, cela nous garantit que l'indentation sera conservée. J'ai ensuite utilisé dans expressions régulières afin d'extraire les balises de la chaine de caractères obtenue. On peut finalement facilement ajouter de la couleur sur la chaine extraite.

4 Conclusion