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

3.2 Suivi de la mémoire

3.3 Affichage du code