

Projet de Compilation Avancée

Garbage Collector pour la Mini-ZAM

v0.1a

Antonin Reitz, en s'appuyant sur le travail de Darius Mercadier

19 février 2024

Présentation du sujet

Contexte : Une année, le projet de Compilation Avancée portait sur la réalisation d'une machine virtuelle (VM) pour le langage OCaml, nommée Mini-ZAM. Il s'agissait d'une version simplifiée de la machine virtuelle utilisée en pratique par le langage OCaml, la ZAM (*ZINC Abstract Machine*). La Mini-ZAM, tout comme la ZAM, est une machine à pile qui peut être vue, dans son noyau fonctionnel, comme une machine de Krivine avec une stratégie d'évaluation par appel par valeur (*call-by-value*). Pour exécuter tout programme OCaml, la ZAM interprète du *bytecode* OCaml représenté par 149 instructions différentes¹, tandis que la Mini-ZAM n'utilise que 27 instructions. Chaque instruction bytecode modifie l'état interne de la machine virtuelle, et l'évolution de cet état représente l'exécution du programme OCaml associé.

Si un langage contenant un garbage collector (GC) est utilisé pour implémenter la Mini-ZAM, il n'est pas nécessaire de doter celle-ci d'un GC, puisqu'elle peut utiliser celui du langage hôte. Cependant, une telle implémentation ne serait probablement pas très performante : il s'agirait d'une VM OCaml tournant au dessus de la VM du langage hôte tournant sur le CPU. Afin de supprimer la couche intermédiaire de la VM du langage hôte, il est nécessaire d'implémenter la Mini-ZAM en C, ou tout du moins dans un langage compilé vers de l'assembleur, et offrant un contrôle de bas niveau sur la machine. Il devient alors nécessaire d'implémenter un garbage collector pour cette machine virtuelle.

Objectif : En partant d'une implémentation de la Mini-ZAM en C ne contenant pas de garbage collector, nous allons y rajouter un GC Mark & Compact.

Rendu : Vous devrez rendre, avant le **17/03/2024 à 23h59 (UTC+1)**, une archive au format `.tar.gz` contenant le code de votre implémentation de l'interprète *Mini-ZAM* (incluant au moins les sources ainsi que les tests), ainsi qu'un rapport (en français ou anglais selon votre préférence). Ce rapport (maximum 10 pages) devra décrire la structure générale du projet, vos choix d'implémentation, ainsi qu'une section détaillant quelles améliorations potentielles pourraient être faites à votre GC (vous pourrez vous inspirer de la section "Pour aller plus loin"). Tout warning durant la compilation devra être solidement justifié dans ce rapport. Le rendu du projet se fera sur Moodle.

1 Introduction

Cette introduction vise à vous familiariser avec la Mini-ZAM. Commencez par lire la description de la Mini-ZAM est disponible sur Moodle (il s'agit du sujet du projet CA de 2019). Toutes les extensions mentionnées sur ce document (appels terminaux, blocs de valeurs et exceptions) sont déjà implémentées dans le code source de la Mini-ZAM qui vous est fourni sur Moodle.

L'implémentation de la Mini-ZAM vise à être au plus proche de l'implémentation de la ZAM complète, tout en gardant le plus de simplicité possible.

1. dans la version « d'époque » d'OCaml - la 4.07

bytecode. La Mini-ZAM prend en entrée un fichier de bytecode au format spécifié par le projet de l'an dernier. Lors du parsing, les primitives (+, -, >=, *etc.*) sont remplacées par des entiers (pour ne pas avoir à manipuler de chaînes de caractères lors de l'interprétation), les labels sont supprimés et les adresses des sauts sont calculées : `BRANCH L1` sera remplacé par `BRANCH addr` où `addr` est l'adresse absolue correspondant au label `L1`. Le bytecode manipulé par la VM en interne est donc un simple tableau d'entiers 64-bits. Par exemple, le bytecode suivant :

```
CONST 1
BRANCHIFNOT L2
CONST 40
PUSH
CONST 2
PRIM +
BRANCH L1
L2: CONST 42
L1: STOP
```

Qui correspond au code OCaml `if true then 40 + 2 else 42`, sera convertis en le tableau suivant :

```
[CONST, 1, BRANCHIFNOT, 13, CONST, 40, PUSH, CONST, 2,
PRIM, PLUS, BRANCH, 15, CONST, 42, STOP]
```

Où `CONST`, `BRANCHIFNOT`, `PLUS`, *etc.* sont définis dans des `enums` et sont donc des entiers.

mlvalues. Toutes les données manipulées par la Mini-ZAM sont de type `mlvalue` (défini dans `mlvalues.h`). Une `mlvalue` est soit un pointeur soit un entier sur 63 bits. Le bit de poids faible différencie les deux cas : si une valeur se finit par un 1, alors il s'agit d'un entier, tandis que si elle se finit par 0, alors il s'agit d'un pointeur. Les macros `Val_long`, `Long_Val`, `Val_ptr` et `Ptr_val` permettent d'obtenir un entier ou un pointeur à partir d'une `mlvalue` et inversement.

blocks. Une `mlvalue` qui n'est pas entier est nécessairement un pointeur sur un bloc. Un bloc est un tableau de `mlvalues` ainsi qu'un header de type `header_t` contenant sa taille, son tag, ainsi que 2 bits réservés pour le garbage collector (inutilisés présentement, mais qui vous serviront plus tard dans le sujet). Les tags possibles sont `ENV_T`, représentant un environnement, `CLOSURE_T` représentant une fermeture, et `BLOCK_T` représentant n'importe quel autre bloc. Un pointeur vers un bloc pointe toujours vers le premier élément du bloc, et non vers le header : le header d'un bloc `b` est à l'adresse `b-1` (cela implique que qu'un header (`header_t`) doit fait la même taille qu'une `mlvalue`). Vous pouvez constater qu'en effet, lorsqu'un bloc est alloué (`make_block` dans `mlvalues.c`), un pointeur vers son premier élément est renvoyé et non vers le header. Les macros `Size` et `Tag` permettent d'accéder à la taille et au tag d'un bloc sans passer explicitement par son header. La macro `Field` permet d'accéder à un élément d'un bloc.

Allocations. Dans la version de la Mini-ZAM qui vous est fournie, toutes les allocations mémoires sont faites par les fonctions `make_empty_block`, `make_block` et `make_closure`; qui appellent toutes `caml_alloc`. La mémoire n'est jamais libérée.

Variables globales. Un singleton de type `caml_domain_state` nommé `Caml_state` est déclaré dans le fichier `domain_state.h` et initialisé dans `domain.c` au lancement de la VM. Cette variable contient les données globales du programme. Dans la version qui vous est fournie, il ne contient que la pile. Lorsque vous aurez besoin de variables globales (par exemple, semi-spaces pour le Stop & Copy, freelists et pages pour le Mark & Sweep), vous pourrez les ajouter à cette structure. Similairement, la configuration de votre programme (taille de la pile, taille du tas, *etc.*) sera définie dans le fichier `config.h`.

Compilation et testing. Un makefile vous est fournis pour compiler la Mini-ZAM. Additionnellement, le script `run_tests.pl` (qui est également lançable par la commande `make test`) lance la Mini-ZAM sur un ensemble de fichiers de tests présents dans le dossier `tests`. Pour lancer manuellement la minizam, la syntaxe est `minizam <fichier_de_bytecode> [-res]`. L'argument `-res` est optionnel. Si il est fournis, il doit être après le fichier de bytecode dans la liste des arguments, et il indique à la Mini-ZAM d'afficher la dernière valeur calculée par le programme. Parmi les fichiers de tests fournis, le dossiers `tests/bench` contient des tests manipulant beaucoup de mémoire et qui mettront vos garbage collectors à l'épreuve.

Bytecode de la Mini-ZAM


Afin de prendre en main le bytecode de la Mini-ZAM, **traduisez manuellement le programme suivant en bytecode** :

```
let rec f n =  
  if n > 1000 then 0  
  else  
    (1 + (f (n + 3)))  
let _ = f 3
```

Ce programme calcule la somme de tous les multiples de 3 inférieurs à 1000.

Vous nommerez ce fichier `sum.3.txt`, et vous le placerez dans un dossier nommé `test/perso`. Vous pourrez placer dans ce dossier tous les fichiers de bytecode que vous écrirez durant ce projet.

Implémentation de la Mini-ZAM

La Mini-ZAM qui vous est fournie souffre de quelques défauts d'implémentation. En particulier, les implémentations des bytecodes `APPLY` et `APPTERM` (dans `interp.c`) ont deux défauts : premièrement, elles font appel à `malloc`, et deuxièmement, elles copient deux fois les arguments de la fonction appelée.  Ces deux facteurs impactent fortement les performances des programmes faisant beaucoup d'appels de fonctions ; ce qui représente la quasi-totalité des programmes OCaml. **Réécrivez `APPLY` et `APPTERM` sans utiliser de `malloc` ni de mémoire temporaire.**

Fuites de mémoire & Valgrind

Lors de ce projet, vous vous retrouverez probablement plusieurs fois confrontés à des “segmentation faults”, ou autres problèmes de corruption mémoire, ainsi que des fuites mémoire. Un outil essentiel afin de traquer l'origine de ces erreurs est `valgrind`.

La Mini-ZAM qui vous est fournis contient deux fuites mémoire principales. La première est liée à l'absence de GC, et peut être ignorée pour le moment. La seconde vient du parseur, qui a été codé avec peu d'attention. **Traquez et corrigez cette fuite mémoire.**

2 Implantation d'un GC Mark & Compact pour la Mini-ZAM

Le Mark & Compact peut sembler similaire au Mark & Sweep : la phase de marquage est effectivement identique. En revanche, le Mark & Compact corrige le principal défaut du Mark & Sweep, à savoir la fragmentation de la mémoire, particulièrement pour des programmes à longue durée d'exécution.

En première approximation, un Mark & Compact fonctionne en trois étapes :

- la première consiste à parcourir et “marquer” toute la mémoire accessible à partir des racines ;
- la seconde consiste à parcourir l'intégralité de la mémoire pour libérer chaque bloc qui n'est pas marqué, car inaccessible (depuis le programme) ;
- la troisième consiste à compacter les blocs marqués, de sorte à ce qu'ils forment une zone mémoire contigüe à la fin de cette étape.

Il existe différentes manières d'implémenter cette troisième étape : nous choisissons la méthode du *sliding* (ou LISP2). Cette troisième étape consiste alors en trois parcours successifs de tout ou partie du tas.

- Un premier parcours, de tout le tas, où on maintient à jour la somme des tailles des objets rencontrés dans une variable, ce qui permet de calculer pour chaque objet marqué rencontré sa future adresse après compaction. En effet, chaque objet marqué va être déplacé de manière à être contigu à l'objet marqué précédemment traité.
- Un second parcours, sur tous les objets marqués, où pour chaque objet marqué on met à jour les pointeurs qu'il contient vers les futures adresses après compaction.
- Une troisième parcours, sur tous les objets marqués, où chaque objet marqué est déplacé à sa nouvelle adresse.

Comme vu en TD4, en pratique, les blocs libres sont ajoutés à une liste chaînée appelée *freelist*. Lorsqu'une allocation est effectuée dans le programme, la *freelist* est parcourue à la recherche d'un bloc libre, qui sera retiré de la *freelist* et donné au programme.

Implémentez un GC Mark & Compact pour la Mini-ZAM.