Projet de Compilation Avancée
Mini-ZAM : Interprète de bytecode fonctionnel

21 février 2019

Attention : *ce document ne constitue pas l'énoncé complet et final du projet. Il sera amené à évoluer jusqu'au 27/02/2019. Consultez bien régulièrement la page web sur laquelle est publié ce document.*

Présentation du sujet

Contexte : La machine virtuelle du langage OCaml, nommée *ZAM* (*ZINC Abstract Machine*) 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 différentes [1]. Chaque instruction bytecode modifie l'état interne de la machine virtuelle, et l'évolution de cet état représente l'exécution du programme OCaml associé.

Une grande partie des instructions de la ZAM correspond à des instructions qui ne sont que des versions spécialisées d'autres instructions (par exemple, l'instruction d'application d'une fonction avec 1 argument `APPLY1`, est une version spécialisée de l'application n-aire `APPLY n`), ou bien le regroupement en une instruction de plusieurs instructions distinctes (par exemple, l'instruction `PUSHACC n` est sémantiquement équivalente à l'instruction `PUSH` suivie de `ACC n`). D'autres part, un certain nombre d'instructions de la ZAM sert traiter les aspects impératifs ou orientés objet du langage, ainsi que plusieurs traits de haut niveau, comme la gestion des exceptions ou le filtrage par motif.

Il est alors possible, en ne s'intéressant qu'au noyau fonctionnel du langage, et en supprimant les instructions spécialisées, de réaliser une version simplifiée de la ZAM qui nécessite moins de 20 instructions bytecode différentes.

Objectif : Ce projet consiste en la réalisation (dans le langage de programmation de votre choix) d'un interprète de bytecode pour un langage *ML* restreint à un tel noyau fonctionnel simple. L'interprète créé permettra l'exécution de programmes fonctionnels qui réalisent des opérations simples (calculs arithmétiques et logiques, branchements,...), ainsi que l'application de fonctions à plusieurs arguments (potentiellement récursives [2]).

La traduction du code *ML* source vers le *bytecode* associé ne sera pas abordée : nous vous fournissons directement des fichiers contenant les instructions bytecode à interpréter, ainsi que les fichiers `.ml` qui leur sont associés.

Ces fichiers sont accessibles à l'adresse `https://www-apr.lip6.fr/~varoumas/`

Rendu : Vous devrez rendre, avant le **25/03/2019 à 23h59 (UTC+1)**, une archive au format `.tar.gz` contenant le code de votre implémentation de l'interprète *Mini-ZAM* (avec un *Makefile* et un fichier *README* précisant, si besoin, quelles sont les dépendances logicielles du programme), ainsi qu'un rapport (en français). Ce rapport (maximum 10 pages) devra décrire la structure générale du projet, vos choix d'implémentation (en particulier la représentation des valeurs fonctionnelles), et de nouveaux jeux d'essai. Le rendu du projet se fera par courrier électronique, aux adresses `steven.varoumas@lip6.fr` et `emmanuel.chailloux@lip6.fr`.

1. dans la version actuelle d'OCaml - la 4.07
2. cependant, nous ne traiterons pas dans ce sujet le cas des fonctions mutuellement récursives

Le sujet du mail aura la forme suivante :

`[4I504] Rendu Projet - <nom1> <nom2>`

(en remplaçant `<nom1> <nom2>` par les noms des membres de votre binôme)
Tout dépassement de la date limite entraînera des pénalités.

1 Description de la machine virtuelle

Valeurs : La machine virtuelle que vous réaliserez manipulera des valeurs d'un type que nous nommerons dans ce sujet `mlvalue`. Au départ, ces valeurs correspondront à deux catégories de valeurs distinctes :

1. Des entiers, qui représenteront toutes les valeurs immédiates manipulées par le programme (on représentera par conséquent `true` par l'entier 1, `false` par 0, et `()` par 0, tandis qu'une valeur entière quelconque sera représentée tel quel).
2. Des fermetures, qui sont des couples (*pointeur de code*, *environnement*) qui représentent les valeurs fonctionnelles manipulées par le langage. Un pointeur de code pourra être représenté par un entier, tandis qu'un environnement sera une collection de valeurs.
   Dans la suite, nous représenterons une fermeture associée à un pointeur de code `pc` et un environnement `e` de la façon suivante : { `pc , e` } .

Le type `mlvalue` pourra être plus tard étendu pour représenter d'autres types de valeurs. Vous ferez donc attention à ce que votre implémentation soit suffisamment souple pour gérer sans trop de difficulté ces ajouts.

Registres de la VM : La *Mini-Zam* est composée de 5 registres :

1. `prog`, un tableau de couples $(label, instruction)$ qui représente le programme en cours d'interprétation.
2. `stack`, la pile dans laquelle seront placées les paramètres des fonctions, des pointeurs de code, des fermetures, etc : c'est une structure LIFO (*Last In First Out*) qui contient des `mlvalue` (au début du programme, la pile est vide).
   Dans la suite, nous représenterons les éléments dans la pile entre crochets `[...]`, avec la tête de pile correspondant à la valeur la plus à gauche.
3. `env`, l'environnement de la fermeture courante : c'est un collection de `mlvalue` qui représente les valeurs accessibles pour la fonction en cours d'exécution (au début du programme, l'environnement est vide).
   Dans la suite, nous représenterons les éléments dans l'environnement entre chrevrons `<...>`.
4. `pc`, le pointeur de code vers l'instruction courante (au début du programme, `pc = 0`).
5. `accu`, un accumulateur utilisé pour stocker certaines valeurs (`mlvalue`) intermédiaires (au début du programme, l'accumulateur vaut `()`).

Le rôle de chaque instruction bytecode est de modifier tout ou partie des valeurs contenues dans ces registres. L'état de ces registres représentera ainsi à tout instant de l'exécution l'état de la machine virtuelle. Vous devrez fournir des fonctions permettant d'afficher lors de l'interprétation d'un fichier bytecode le contenu de ces registres.

2 Description des fichiers bytecode

Instructions : Le *bytecode* associé à un programme de notre langage « *mini-ML* » vous sera fourni sous la forme d'un fichier texte, dans lequel chaque ligne représente une instruction à exécuter. L'instruction est représentée par un caractère de tabulation (→), suivi par son nom, en lettres majuscules. Par exemple, la ligne suivante correspond à l'instruction `PUSH` qui ajoute une valeur en tête de la pile :

→`PUSH`

Arguments : Certaines instructions du bytecode doivent être paramétrées par une ou plusieurs valeurs. Ces arguments sont ajoutés à la suite du nom de l'instruction (après un espace), séparés par des virgules. Par exemple, la ligne suivante représente l'instruction `CLOSURE` (création de fermeture) avec comme argument un label L1 et une valeur 0 :

```
⟶CLOSURE␣L1,0
```

Labels : Enfin, certaines instructions sont identifiées par un label, afin d'y faire référence depuis un autre point du bytecode. Une instruction labellisée commence par une chaîne de caractères qui se termine par le symbole ':'. Ce label est suivi d'un caractère de tabulation, puis de l'instruction bytecode associée. Par exemple, la ligne suivante représente l'instruction d'application de l'opérateur `+`, labellisée par un label `N` :

```
N:⟶PRIM␣+
```

Exemple : L'exemple suivant est le contenu d'un fichier bytecode qui correspond au programme `(if true then 2 else 3)` :

```
⟶CONST␣1
⟶BRANCHIFNOT␣L
⟶CONST␣2
⟶BRANCH␣M
L:⟶CONST␣3
M:⟶STOP
```

3 Instructions de base et fonctions unaires

Nous vous demandons en premier lieu d'implémenter un interprète de bytecode Mini-ML qui puisse traiter des programmes faisant usage d'opérateurs de base (comme les opérateurs arithmétiques ou le `if then else`) et permettant l'application de fonctions non-récursives qui n'ont pour le moment qu'un argument. La gestion de tels programmes entraîne la définition d'une douzaine d'instructions distinctes.

Nous présentons dans la suite le détail du fonctionnement de ces dernières, ainsi que leur effet sur les registres de la machine virtuelle.

Attention : *Pour chaque instruction, seuls les registres ayant été modifiés (ou ayant un rôle dans l'instruction) seront mentionnés. En plus des modifications explicitement décrites, le pointeur de code (`pc`) sera incrémenté à la fin de chaque instruction (sauf dans les cas où il est précisé que `pc` prend une valeur différente).*

1. **CONST n** (Valeur constante)
   Description :
   — L'accumulateur (`accu`) prend pour valeur la constante n.

	Avant	Après
`accu`	_	n

2. **PRIM op** (Application de primitive)
   Description :
   — *op* est une primitive parmi :
   - des opérateurs arithmétiques à deux arguments : `+`,`-`,`/`,`*`

- des opérateurs logiques à deux arguments : `or`,`and`
- l'opérateur booléen `not`
- des opérateurs de comparaison `<>`,`=`,`<`,`<=`,`>`,`>=`
- une primitive d'écriture (d'un caractère représenté par sa valeur ASCII) sur *stdout* : `print`

— Pour les opérateurs à deux arguments, *op* est appliqué à l'accumulateur et à une valeur dépilée, et le résultat est mis dans `accu` :

	Avant	Après
`stack`	$[a_0; a_1; ...]$	$[a_1; ...]$
`accu`	x	$op(x, a_0)$

— Pour les opérateurs unaires, *op* est appliqué à l'accumulateur, et le résultat est mis dans `accu` :

	Avant	Après
`accu`	x	$op(x)$

3. **BRANCH L** (Branchement)

Description :

— `pc` prend pour valeur la position du label L.

— On utilisera une fonction *position* qui associe un label à une position dans le code (une valeur entière)

	Avant	Après
`pc`	_	$position(L)$

4. **BRANCHIFNOT L** (Branchement conditionnel)

Description :

— Si l'accumulateur vaut 0, alors `pc` prend pour valeur la position du label L :

	Avant	Après
`accu`	0	0
`pc`	_	$position(L)$

— Sinon, `pc` est incrémenté normalement :

	Avant	Après
`accu`	1	1
`pc`	n	$n+1$

5. **PUSH** (Empilement)

Description :

— Empile dans `stack` la valeur située dans `accu` :

	Avant	Après
`stack`	[_]	$[x;$ _]
`accu`	x	x

6. **POP** (Dépilement)

Description :

— Dépile la valeur située en tête de `stack` :

	Avant	Après
`stack`	$[x;$ _]	[_]

7. **ACC i** (Accès à la i-ième valeur de la pile)
   Description :
   — `stack[i]` est mis dans `accu` :

	Avant	Après
`stack`	$[a_0; a_1; a_2; ...; a_i; ...]$	$[a_0; a_1; a_2; ...; a_i; ...]$
`accu`	_	a_i

8. **ENVACC i** (Accès à la i-ième valeur de l'environnement)
   Description :
   — `env[i]` est mis dans `accu` :

	Avant	Après
`env`	$<e_0; e_1; e_2; ...; e_i; ...; e_n>$	$<e_0; e_1; e_2; ...; e_i; ...; e_n>$
`accu`	_	e_i

9. **CLOSURE L,n** (Création de fermeture)
   Description :
   — Si $n > 0$ alors l'accumulateur est empilé.
   — Puis, une fermeture dont le code correspond au label L et dont l'environnement est constitué de n valeurs dépilées de `stack` est créée et mise dans l'accumulateur.

   Par exemple, si $n > 0$:

	Avant	Après
`stack`	$[a_0; ...; a_{n-2}; a_{n-1}; ...]$	$[a_{n-1}; ...]$
`accu`	x	$\{\ position(L)\ ,\ <x; a_0; ...; a_{n-2}>\ \}$

10. **APPLY n** (Application de fonction)
    Description :
    — n arguments sont dépilés [3]
    — `env` puis `pc+1` sont empilés
    — les n arguments sont rempilés
    On se met dans le contexte de la fonction appelée :
    — `pc` reçoit le pointeur de code de la fermeture située dans `accu`
    — `env` reçoit l'environnement de la fermeture située dans `accu`

	Avant	Après
`stack`	$[a_0; ...; a_{n-1}; a_n; \_]$	$[a_0; ...; a_{n-1}; p+1; e; a_n; \_]$
`pc`	p	c
`env`	e	e'
`accu`	$\{\ c\ ,\ e'\ \}$	$\{\ c\ ,\ e'\ \}$

11. **RETURN n** (Sortie de fonction)
    Description :
    — n valeurs sont dépilées
    On retourne dans le contexte de l'appelant :
    — les valeurs associées à `pc` et `env` sont dépilées

	Avant	Après
`stack`	$[a_0; ...; a_{n-1}; p; e; a_n; \ ...]$	$[a_n; ...]$
`env`	_	e
`pc`	_	p

12. **STOP** (Fin de programme)
    Description :
    — Fin de l'exécution du programme
    — La valeur calculée par le programme est alors située dans `accu`

3. pour le moment, on ne traitera que les fonctions unaires

Fichiers de test

Vous disposez dans l'archive qui vous est fournie de fichiers vous permettant de tester votre implémentation. Pour cette partie, ces fichiers se trouvent dans le dossier `unary_funs`. Chaque fichier "bytecode" (extension .txt) est fourni avec une représentation en syntaxe OCaml du programme à tester [4].

Par exemple, l'interprétation du bytecode de `fun1.txt` produit la suite suivante d'états :

```
au début : pc=0 accu=0 stack=[] env=<>
BRANCH L2          pc=6 accu=0 stack=[] env=<>
L2: CLOSURE L1,0   pc=7 accu={ L1, <> } stack=[] env=<>
PUSH               pc=8 accu={ L1, <> } stack=[{ L1, <> }] env=<>
CONST 2            pc=9 accu=2 stack=[{ L1, <> }] env=<>
PUSH               pc=10 accu=2 stack=[2;{ L1, <> }] env=<>
CONST 4            pc=11 accu=4 stack=[2;{ L1, <> }] env=<>
PUSH               pc=12 accu=4 stack=[4;2;{ L1, <> }] env=<>
ACC 2              pc=13 accu={ L1, <> } stack=[4;2;{ L1, <> }] env=<>
APPLY 1            pc=1 accu={ L1, <> } stack=[4;14;<>;2;{ L1, <> }] env=<>
L1: ACC 0          pc=2 accu=4 stack=[4;14;<>;2;{ L1, <> }] env=<>
PUSH               pc=3 accu=4 stack=[4;4;14;<>;2;{ L1, <> }] env=<>
CONST 1            pc=4 accu=1 stack=[4;4;14;<>;2;{ L1, <> }] env=<>
PRIM +             pc=5 accu=5 stack=[4;14;<>;2;{ L1, <> }] env=<>
RETURN 1           pc=14 accu=5 stack=[2;{ L1, <> }] env=<>
PRIM *             pc=15 accu=10 stack=[{ L1, <> }] env=<>
STOP
```

4 Fonctions récursives

Ajoutons désormais la gestion des fonctions récursives. Pour qu'une fonction puisse faire appel à elle-même, elle doit avoir accès à un pointeur vers la première instruction de son code.

Vous devez ajouter alors deux nouvelles instructions :

1. L'instruction `CLOSUREC L,n` permet de créer une fermeture récursive. Elle est très semblable à l'instruction `CLOSURE L,n`, à ceci près qu'elle stocke également en tête de `env` le pointeur de code correspondant au label L (afin de pouvoir se rappeler elle-même). La fermeture créée devra également être empilée à la fin de l'instruction.
2. L'instruction `OFFSETCLOSURE` qui met dans `accu` une fermeture dont le code correspond au premier élément de l'environnement courant (`env[0]`) et l'environnement correspond à l'ensemble de valeurs contenues dans `env`.

Fichiers de test

Des fichiers de test associés aux fonctions récursives sont situés dans le sous-dossier `rec_funs`.

5 Fonctions n-aires et application partielle

Il est possible de compiler une fonction d'arité n grâce à un processus de curryfication [5] qui consiste à transformer une fonction à n arguments en une fonction à un argument qui retourne une fonction à $n-1$ arguments. Par exemple, la fonction `(fun x y z -> x + y + z)` peut être réécrite (en curryfiant toutes les fonctions) en :

4. À ce propos, les fichiers bytecode fournis ressemblent à ce qu'affiche le compilateur `ocamlc` lorsqu'on lui donne l'option `-dinstr`

5. `https://fr.wikipedia.org/wiki/Curryfication`

```
(fun x -> (fun y -> (fun z -> x + y + z)))
```

Cependant, ce procédé est assez coûteux, car avec notre mode d'application des fonctions, il serait alors créé n fermetures pour une fonction $n-$aire. Et ce surcoût semble encore plus inutile quand on réalise que, la plupart du temps, les fonctions sont appliquées avec tous leurs arguments.

Pour pallier à ce défaut, nous gérons alors l'application d'une fonction à n arguments en ajoutant à la machine *Mini-ZAM* un registre `extra_args` dont le rôle est de représenter le nombre d'arguments restant à appliquer à une fonction pour réaliser une application totale. Ce registre fera alors partie de l'état de la machine virtuelle, qui contiendra désormais 6 éléments.

Le contexte stocké lors de l'appel d'une fonction contiendra donc un élément supplémentaire. En effet, on aura besoin, en plus du `pc` et de l'environnement, de la valeur de `extra_args` pour enregistrer toutes les informations du contexte de l'appelant. Le mécanisme d'application de fonction est alors modifié en conséquence :

— L'instruction `APPLY n` empilera `extra_args` en plus de `pc` et `env`, et mettra ensuite la valeur $n-1$ dans `extra_args`.

— L'instruction `RETURN n` est modifiée comme suit :

- Dans tous les cas, on dépile d'abord n valeurs.
- Puis, si `extra_args` $= 0$, on conserve le fonctionnement précèdent (mais on fera attention à bien se remettre dans le contexte de l'appelant en tenant compte des modifications apportées à `APPLY`)
- Sinon :
  — `extra_args` est décrémenté de 1.
  — `pc` reçoit le pointeur de code de la fermeture située dans `accu`.
  — `env` reçoit l'environnement de la fermeture située dans `accu`.

Par exemple, si `extra_args` > 0 :

	Avant	Après
`stack`	$[a_0; a_1; ...; a_{n-1}; a_n; \ldots]$	$[a_n; \ldots]$
`extra_args`	m	$m-1$
`pc`	_	c
`env`	_	e
`accu`	$\{\ c\ ,\ e\ \}$	$\{\ c\ ,\ e\ \}$

De plus, deux nouvelles instructions seront à traiter :

1. **GRAB n** (Gestion de l'application partielle)
   Description :
   — Si `extra_args` $\geq n$, alors on a assez d'arguments pour appliquer la fonction : décrémenter `extra_args` de n et continuer à l'instruction suivante.

	Avant	Après
`extra_args`	$m+n$	m
`pc`	p	$p+1$

   — Sinon, on génère une nouvelle fermeture :
   - Dépiler n éléments
   - Mettre dans `accu` une nouvelle fermeture dont le code est situé à $pc-1$ (i.e une instruction `RESTART`) et dont l'environnement est constitué des n éléments qui ont été dépilés juste avant.
   - Trois valeurs sont dépilées et associées à `extra_args`, `env`, et `pc`.

	Avant	Après
`stack`	$[a_0; a_1; ...; a_{n-1}; a_n; a_{n+1}; a_{n+2}; a_{n+3}; \ldots]$	$[a_{n+3}; \ldots]$
`extra_args`	_	a_n
`pc`	p	a_{n+1}
`env`	_	a_{n+2}
`accu`	_	$\{\ p-1\ ,\ <a_0; a_1; ...; a_{n-1}>\ \}$

2. **RESTART**

   Description : Cette instruction précédera toujours une instruction `GRAB`
   - Soit n, la taille de l'environnement
   - Déplacer les éléments de env à partir du deuxième (donc de `env[1] à env[`$n-1$`]`) dans la pile
   - `env` prend pour valeur celle de son premier élément (`env[0]`)
   - `extra_args` est incrémenté de $n-1$.

	Avant	Après
`stack`	$[a_0; ...]$	$[e_1; ...; e_{n-1}; a_0; ..._-]$
`extra_args`	m	$m+n-1$
`pc`	p	a_{n+1}
`env`	$<e_0; e_1; ...; e_{n-1}>$	e_0

Fichiers de test

Des fichiers de test associés aux fonctions n-aires sont situés dans le sous-dossier `n-ary_funs`.

6 Optimisations et nouvelles fonctionnalités

suite de l'énoncé à venir ....