Master STL - M1
Module CA (4I504)

Compilation Avancée
Documents autorisés

Sorbonne Université
Année 2021–2022

Examen du 17 mai 2022

1ère partie (12 points) - à traiter sur une copie séparée

1ère partie : implantation de la machine virtuelle *TVM*

Ce sujet s'inspire fortement du système Ribbit[1] : implantation légère d'un compilateur Scheme (langage fonctionnel dynamiquement typé) vers une machine virtuelle nommée *RVM* (Ribbit Virtual Machine) et possédant une boucle REPL (Read-Eval-Print-Loop) pour les systèmes embarqués à faibles ressources comme les micro-contrôleurs pouvant tenir sur 4Ko. On s'intéressera à une version simplifiée de la machine virtuelle *RVM*, appelée *TVM* (Triplet Virtual Machine) et à son *runtime* que l'on implantera.

En *TVM*, les valeurs sont soit des valeurs immédiates, soit des valeurs allouées. Les seules valeurs immédiates sont les entiers. Les valeurs allouées sont représentées par des triplets de taille fixe appelés *trip*. *TVM* interprète un flot de code (graphe de code) en utilisant une pile (liste chaînée de valeurs). La pile et le code sont représentés par des *trip* : la variable `sp` fait référence au trip du sommet de pile, tandis que la variable `pc` fait référence au trip contenant l'instruction TVM en cours d'exécution. Chaque valeur allouée est un triplet (champ 1, champ 2, champ 3) dont le dernier contient son type, et les deux premiers champs les éléments de la structure : *fst* (premier composant) et *snd* (second composant) pour les paires; *code* et *env* pour les fermetures ; *name* et *value* pour les symboles; *chars* (caractères) et *length* (longueur) pour les chaînes de caractères; *elems* et *length* pour les vecteurs; et sans importance pour les valeurs spéciales `#f` et `#t` pour les booléens, et `()` pour unit car elles seront allouées chacune une seule fois.

valeurs (*trip*)	type
(*fst*, *snd*, **0**)	paire
(*code*, *env*, **1**)	procédure (primitive ou fermeture)
(*valeur*, *nom*, **2**)	symbole, valeur globale
(*chars*, *length*, **3**)	chaîne de caractères
(*elems*, *length*, **4**)	vecteur
(−, −, **5**)	valeur spéciale : `#f`, `#t`, `()`

La machine *TVM* est définie par un petit ensemble d'instructions, 6 correspondant aux "formes spéciales" de Scheme à l'exception de *lambda* qui est traitée spécifiquement à la compilation, avec un ensemble plus ou moins important de primitives. Pour le sujet il sera minimaliste. Les représentations des valeurs allouées en *trip* et des instructions sont décrites dans le tableau suivant. Chaque instruction est un *trip* dont le premier champ est le code de l'instruction, le deuxième est l'environnement ou la valeur, et le troisième est l'instruction suivante (pointeur vers le triplet de celle-ci), sauf pour `jump` qui n'en n'a pas besoin et `if` qui a deux instructions de branchement (branche *then* et branche *else*) qui seront suivies selon la valeur du sommet de pile. Les instructions `get` et `set` utilisent le 1er champ du *trip* (slot de la pile ou variable globale) pour lire une valeur ou écrire une nouvelle valeur.

instructions (*trip*)	action
(**0**, *slot de pile/variable globale*, **0**)	`jump` : appel terminal
(**0**, *slot de pile/variable globale*, *next*)	`call` : appel non terminal
(**1**, *slot de pile/variable global*, *next*)	`set` : slot de pile/variable globale ← pop()
(**2**, *slot/global*, *next*)	`get` : push(slot/global)
(**3**, *valeur*, *next*)	`const` : push(valeur)
(**4**, *then*, *else*)	`if` : if pop() ≠ #f goto then)

Ces instructions sont complétées par les primitives définies dans le tableau suivant, et opèrent sur des valeurs. Une primitive est aussi un *trip* dont le premier champ est son *opcode* représenté par un entier, le deuxième est un *env* (qui n'est pas utilisé par la primitive), et le troisième contient l'entier 1 (marqueur du type des primitives ou fermetures). Ces numéros de primitives peuvent correspondre à une localisation du code de celles-ci dans un tableau de primitives.

[1] Samuel Yon et Marc Feely. " A Small Scheme VM, Compiler, and REPL in 4K ". In Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages, VMIL, 2021. `https://github.com/udem-dlteam/ribbit`

Dans le tableau, on utilise la notation : "$bool(x)$ = `#t` si x est vrai, et `#f` sinon". On note x,y,z des variables temporaires et r le résultat d'un appel de primitive, $x[0]$ correspond au champ 1 du *trip*, $x[1]$ au champ 2, $x[2]$ au 3.

primitive	code	action	primitive	code	action
trip	0	$z \leftarrow pop(); y \leftarrow pop(); x \leftarrow pop();$ $r \leftarrow trip(x, y, z)$	*field2*	8	$x \leftarrow pop(); r \leftarrow x[2]$
			field0-set!	9	$y \leftarrow pop(); x \leftarrow pop(); x[0] \leftarrow y; r \leftarrow y$
id	1	$x \leftarrow pop(); r \leftarrow x$	*field1-set!*	10	$y \leftarrow pop(); x \leftarrow pop(); x[1] \leftarrow y; r \leftarrow y$
arg1	2	$y \leftarrow pop(); x \leftarrow pop(); r \leftarrow x$	*field2-set!*	11	$y \leftarrow pop(); x \leftarrow pop(); x[2] \leftarrow y; r \leftarrow y$
arg2	3	$y \leftarrow pop(); x \leftarrow pop(); r \leftarrow y$	$==$	12	$y \leftarrow pop(); x \leftarrow pop(); r \leftarrow bool(x == y)$
close	4	$x \leftarrow pop(); r \leftarrow trip(x[0], sp, 1)$	$<$	13	$y \leftarrow pop(); x \leftarrow pop(); r \leftarrow bool(x < y)$
trip?	5	$x \leftarrow pop(); r \leftarrow bool(x \; est \; un \; trip)$	$+$	14	$y \leftarrow pop(); x \leftarrow pop(); r \leftarrow x + y$
field0	6	$x \leftarrow pop(); r \leftarrow x[0]$	$-$	15	$y \leftarrow pop(); x \leftarrow pop(); r \leftarrow x - y$
field1	7	$x \leftarrow pop(); r \leftarrow x[1]$	$*$	16	$y \leftarrow pop(); x \leftarrow pop(); r \leftarrow x * y$

Question 1 Choisir le langage d'implantation puis définir les types et/ou les structures de données pour représenter les valeurs immédiates, les valeurs allouées (les *trip* ou triplets) dont les 3 valeurs spéciales `#f`, `#t` et `()`, la pile (de valeurs) avec un des champs du *trip* est utilisé pour le chaînage et un autre pour la valeur empilée, et l'environnement global, aussi une liste chaînée de "symboles".

Question 2 Définir les types et/ou les structures de données pour représenter les instructions et le flot d'instructions (vu comme un graphe) ainsi que la structure permettant d'associer un code (extérieur à la VM) à un entier (à la manière d'un tableau des primitives). Cet entier correspondra à la définition ensuite de la primitive de la manière suivante : `(define id (trip 1 0 1))` qui indique la création de la valeur primitive dont le code vaut 1, l'environnement vide (entier 0) et le type `1` indiquant une procédure. Le champ *code* des fermetures est aussi un *trip* dont le 1er champ est le nombre d'arguments attendus, et le 3ème l'instruction d'entrée dans le code de la fonction.

Question 3 Donner la structure générale de l'interprète *TVM*.

Question 4 Définir le traitement des 4 instructions suivantes : `set`, `get`, `const` et `if`

Question 5 Définir le traitement des primitives suivantes du tableau précédent,

(a) *trip*, *trip?*, *field1* et *field-set2!*

(b) $+$, $==$, *arg2* et *close*

On s'intéresse maintenant aux appels de primitives et de valeurs fonctionnelles. On va implanter les instructions `jump` pour les appels terminaux et `call` pour les appels non terminaux. Les primitives ont un environnement vide (champ 2 du *trip*) et leur code correspond à leur numéro (entre 0 et 16) dans la structure d'association *numéro* $\mapsto$ *code* de la primitive. Tant les appels classiques (`call`) que les appels optimisés pour les appels récursifs terminaux (`jump` peuvent s'appliquer à des fermetures ou des primitives. Pour les appels terminaux, le compilateur utilise l'instruction `jump` qui, n'ayant pas d'instruction suivante, nécessite un traitement spécial.

- Quand une primitive *prim* (sur le sommet de pile) est appelée par un `call`, elle dépile le nombre d'arguments attendus, et empile le résultat de l'opération sur la pile, puis passe à l'instruction de son champ *next*. C'est la primitive qui vérifie le bon nombre d'arguments. Par contre quand cette opération est exécutée via un `jump`, les mêmes opérations sont effectuées mais avant l'empilement du résultat, la pile `sp` et le compteur ordinal `pc` sont mis à jour en fonction de la continuation *cont* dans le cadre courant d'appel qui contient l'état de ces variables au moment de l'appel du `call`. Cette continuation *cont* est la première de ce type rencontrée dans la pile.
- Quand une fermeture *proc* (sur le sommet de pile) est appelée par un `call` : 1/ allouer un *trip* de continuation appelé *cont*, 2/ dépiler les n valeurs en sommet de pile et construire une liste chaînée *args* avec, 3/ remplir *cont* avec : `sp` au premier champs, la procédure appelée *proc* en second champ, et le *next* du `call`, 4/ empiler *cont*, 5/ copier en les empilant les éléments de *args* dans `sp`, 6/ mettre à jour `pc` en prenant la valeur du code de *proc*

Quand une fermeture `proc` est appelée par un `jump`, les étapes ne concernant pas *cont* sont effectuées, à la fin du `jump` le triplet `cont` de la continuation est trouvé en explorant la pile.

Question 6 Implanter les instructions `call` et `jump` selon ce protocole.

Question 7 (**bonus**) : comment compileriez-vous une définition classique d'une fonction récursive (ou récursive terminale) comme `factorielle` et son appel sur `(fact 3)`. Donner le byte-code en instructions *TVM*.