Présentation de la VM & Compilateur LISP

VIAL Sébastien
sebastien.vial@etu.umontpellier.fr

EL JAAFARI Samy
samy.el-jaafari@etu.umontpellier.fr

ALMALLOUHI Mohamad Satea
mohamad-satea.almallouhi@etu.umontpellier.fr

23 janvier 2024

Objectifs:

Tout d'abord en ce qui concerne la **machine virtuelle**:

1. Être capable d'exécuter les instructions simples
2. Gérer les instructions plus complexes (gestion des offsets dans les load, des constantes dans d'ordres instructions, ...)
3. Effectuer tout ça sans planter, en proposant des moyens de débuguer ainsi qu'en optimisant le plus possible les opérations.

Et ensuite pour le **compilateur**:

1. Compiler des expressions simples (arithmétiques, comparaisons)
2. Gérer les opérateurs de contrôle (if, cond, when, for, while)
3. Compiler des fonctions sans paramètres
4. Gérer les paramètres à l'aide de la pile
5. Implémenter les fonctions lambdas
6. Optimiser le tout si possible

La machine virtuelle

Attaquons-nous d'abord à la machine virtuelle. Nous allons dans un premier temps définir l'initialisation de la machine virtuelle:

```
(defun vm-reset(vm &optional (size 1000))
  (let ((size (max size 1000)) (variablesBasse 30) (tailleZones (- (max size 1000) 30)))
    (attr-set vm :R0 0)
    (attr-set vm :R1 0)
    (attr-set vm :R2 0)
    (attr-set vm :MAX_MEM size)
    (attr-array-init vm :MEM size)
    (var-basse-set vm +start-code-id+ (- size 1))
    (var-basse-set vm +etiq-id+ (make-hash-table))
    (pc-set vm (- size 1))
    (bp-set vm 30)
    (sp-set vm (bp-get vm))
    (fp-set vm (sp-get vm))
    (ms-set vm (+ variablesBasse (/ tailleZones 2)))
    (set-running vm 1)))
```

Chargement des programmes au sein de la VM

```
(defun vm-load (vm program)
    ;; Détermine l'adresse de départ pour charger le programme
  (let ((initial-pc (- (or (var-basse-get vm +last-code-id+) (+ (pc-get vm) 1)) 1)))
        (loop for insn in program do
            (if (is-label insn)
                ;; Si c'est un label, stocke son adresse dans la table des labels
                (etiq-set vm (string (second insn)) initial-pc)
                ;; Sinon, stocke l'instruction en mémoire et met à jour initial-pc
                (progn
                    (mem-set vm initial-pc insn)
                    (setq initial-pc (- initial-pc 1)))))
        ;; Met à jour :LAST_CODE
        (var-basse-set vm +last-code-id+ (+ initial-pc 1))
        ;; Mise à jour des adresses pour les sauts
        (update-labels-for-jumps vm)))
```

Execution des programmes

```
(defun vm-execute (vm)
  (loop while (and (>= (pc-get vm) (var-basse-get vm +last-code-id+)) (is-running vm)) do
    (let ((insn (mem-get vm (pc-get vm))))
      (if (is-debug vm) (format t "~A " insn))
      (cond
        ((equal (first insn) 'LOAD) (handle-load vm insn))
        ((equal (first insn) 'STORE) (handle-store vm insn))
        ; ... ;
        (t (format t "Instruction inconnue: ~A~%" insn)))
      (pc-decr vm)
      (if (is-debug vm)
        (format t "R0: ~A R1: ~A R2: ~A SP: ~A FP: ~A Stack: ~A~%"
                (attr-get vm :R0)
                (attr-get vm :R1)
                (attr-get vm :R2)
                (attr-get vm :SP)
                (attr-get vm :FP)
                (stack-get vm))))))
```

Gestion des constantes, des offsets, ...

Ici le load gère les cas:

- Si la source est un nombre: `(LOAD 0 R0)`
- Si la source est un registre: `(LOAD R1 R0)`
- Si la source est un offset: `(LOAD (+ R0 10) R0)`
- Si la source est une variable globale: `(LOAD (@ var) R0)`

```
(defun handle-load (vm insn)
  (let ((src (second insn)) (dst (third insn)))
    (cond
      ((numberp src) (attr-set vm dst (mem-get vm src)))
      ((keywordp src) (attr-set vm dst (mem-get vm (attr-get vm src))))
      ((is-offset src)
        (let ((offset (third src)) (attr (second src)))
          (attr-set vm dst (mem-get vm (+ (attr-get vm attr) offset)))))
      ((is-global-var src)
        (attr-set vm dst (etiq-get vm (second src))))
      (t (format t "La source doit être soit un nombre, soit un registre, soit un offset:
~A~%" insn)))))
```

Le compilateur

Nous compilerons *recursivement* les expressions. Nous prendons les expressions sous forme de liste afin d'assurer que le parenthèsage soit bon. Par exemple, il sera possible de charger un programme à l'aide du code suivant:

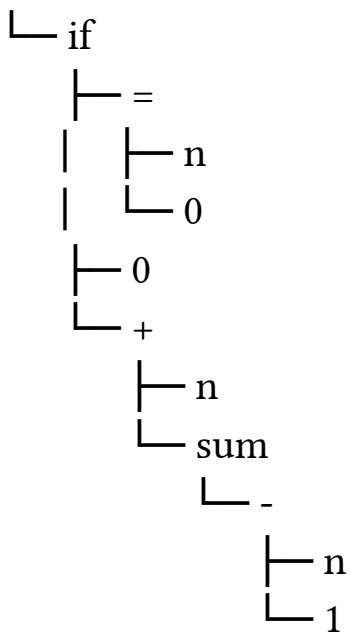
```
(let (
  (vm '())
  (func '(defun sum (n) (
    if (= n 0) 0 (+ n (sum (- n 1)))
  )))
  (call '(sum 10))
)
  (vm-init vm)
  (vm-load vm (comp func))
  (vm-load vm (comp call))
  (vm-execute vm)

  (format t "Somme n: ~A~%" (attr-get vm :R0))
)
```

Comprendre l'expression précédante

Afin de compiler l'expression:

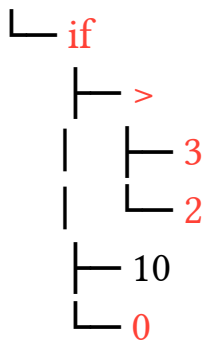
```
(if (= n 0) 0 (+ n (sum (- n 1))))
```



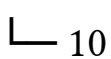
Optimisations possibles

Tout d'abord, on peut remarquer qu'il existe parfois dans les codes des tautologies. Par exemple, `(> 3 2) = t`, `(and nil expr1 expr2) = nil`, ...

On peut donc avant de compiler optimiser les expressions pour réduire le code, par exemple: `(if (> 3 2) 10 0)`



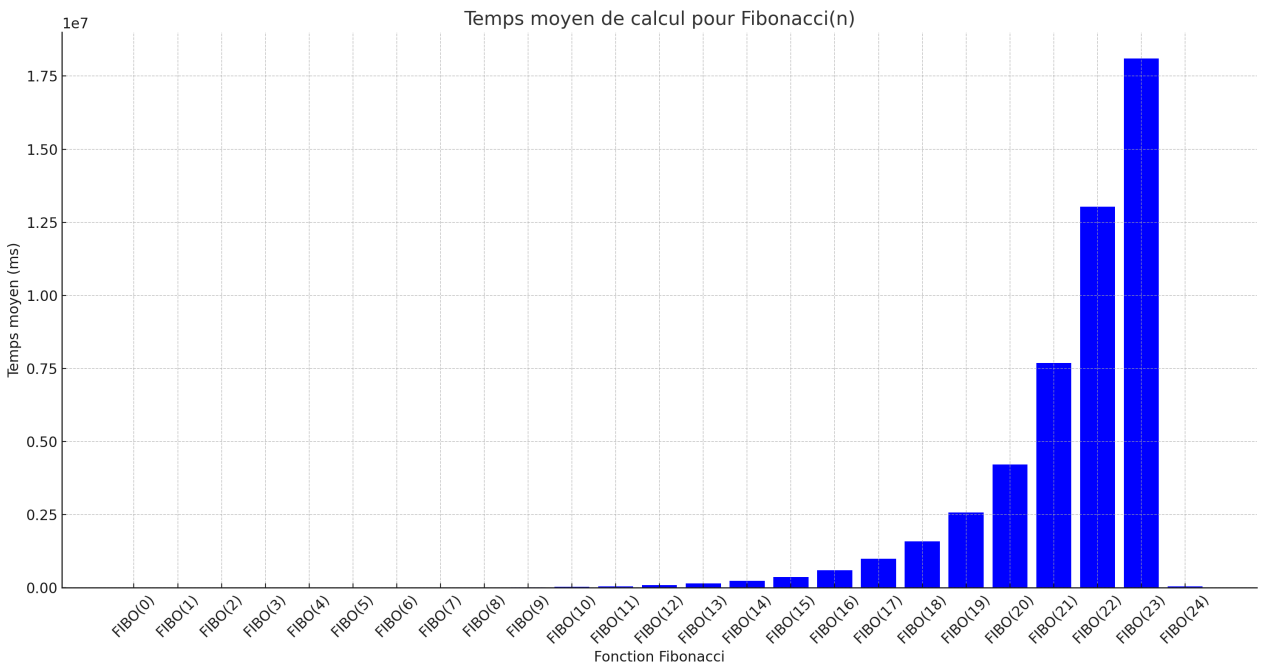
L'expression va donc devenir:



Benchmark

Comme vous pouvez l'imaginer ça fuse, ça va très vite

Voici par exemple un graphe du temps d'exécution de la fonction fibonacci:



Comparaison

Sachant que du côté de Pablo, ce code s'exécute en 27s en moyenne pour fibo(25), c'est $\frac{27}{47} \cdot 100 = 57\%$ plus lent que la machine virtuel de Pablo.

Démonstration

Passons maintenant à la démonstration de notre machine virtuel et compilateur.