Rapport étape 1

Emmanuel Bengio et Alexandre St-Louis

19 février 2013

1 Objectifs

Les objectifs sont relativements simples. On veut être capables de compiler un sous-ensemble réduit de Scheme vers du x86-64.
Cela comprend les `let`, les `if`, `print` et quelques opérations arithmétiques de base.
On veut aussi une représentation intermédiaire en code de machine à pile qui sera à un niveau intermédiaire entre le Scheme et l'implémentation en x86-64.

2 Structure du compilateur et méthodologie

Nous avons décidé de séparer le compilateur en 4 phases. La première s'occupe de lire le code, de l'analyser sémantiquement, lexicalement et syntaxiquement pour produire une ou des s-expression. La logique de cette phase se trouve dans le fichier read.scm, et le point d'entré pour l'analyse lexicale et syntaxique est la fonction `read-stream-input-stream`, alors que pour l'analyse sémantique, c'est la fonction `analyse-syntax` dont le nom serait à changer. La deuxième phase s'occupe de traduire une s-expression vers une représentation de code pour une machine à pile étendue d'un accumulateur. La logique de cette phase se trouve dans le fichier stackvm.scm et est principalement la· responsabilité de la fonction `translate-ast`. La troisème phase prend cette dernière représentation et la transforme vers une représentation des opérations de la pile pour une machine x86-64. C'est la fonction `stack-to-x86` qui se trouve dans le fichier register_ir.scm qui se charge de cette operation. La derniére phase s'occupe simplement de produire un fichier

d'assembleur pour x86-64 près à être compilé par as. C'est `x86-instruction->assembly` se trouvant aussi dans register_ir.scm qui se charge de convertir le tout vers une string.

2.1 Analyse syntaxique et lexicale

La structure récursive des s-expressions permet de recréer récursivement cette structure donnée à partir d'une chaîne de caractères. La grammaire des s-expressions peut se résumer à ceci si on ne se préoccupe pas de la validité des formes spéciale :

```
<expression> ::= <litteral> | ( <expression>* )
<litteral> ::= <number> | <boolean> | <symbol> | <string> | ...
```

Comme les valeurs littérales n'entrent pas en conflit avec l'organisation des expressions et que l'analyse est récursive, il est facile de mélanger l'analyse lexicale et syntaxique. De plus un seul caractère nous permet à tout moment de savoir quel type de token lire, ou quel chemin de la grammaire prendre. Il est donc facile d'écrire nous même la fonction read qui effectue l'analyse syntaxique et lexicale.
La fonction read se résume donc à, si l'on trouve une parenthèse, lire une série d'expressions récursivement, sinon à lire des valeurs littérales.
Chaque type de valeur littérale est analysé/lu linéairement, et relève de l'analyse lexicale. Par exemple les token de chaînes de caractères sont lues de gauche à droite jusqu'à ce qu'on trouve une double-quote qui n'est pas précédée d'un caractère d'échappement (d'un backslash).
À défaut de pouvoir terminer de lire un tel token, une erreur est lancée.

La fonction read permet présentement de lire un parenthésage présenté précédemment, des nombres, des booléens, des expressions quotées, des chaînes de caractères, des caractères seuls (`#\space`, `#\g`, ...) ainsi que des symboles. Il est à noter que la fonction read peut se lire elle même, ce qui pave la voie à un compilateur autogène.

On peut donc générer le code représenté par une chaîne de caractères dans une s-expression, sans passer par un arbre de dérivation. Par après on effectue une seconde analyse syntaxique sur la s-expression pour déterminer si elle est correctement construite (par exemple l'arité des `let` et des `if`).

2.2 Traduction vers une machine à pile

En Scheme, le code dans sa forme de s-expression a l'avantage d'être très proche, voir identique, de l'arbre de syntaxe abstrait. Cela nous sauve une étape de traduction lors de l'analyse syntaxique.
Dans cette étape, on va utiliser cet arbre/s-expression pour, encore une fois récursivement, générer du code pour une machine virtuelle à pile avec un accumulateur.

Chaque étape récursive consiste en une liste d'expressions (qui peuvent elles mêmes être des expressions), donc on n'a que quelque formes à tester : les opérations arithmétiques binaires, print, let et if. Donc si le premier symbole est une de ces formes, on peut la gérer :

opération arithmétique binaire on génère récursivement les deux opérandes de l'opération l'une à la suite de l'autre, de telle sorte que la première sera sur déplacé de l'accumulateur vers la pile (car le résultat d'une expression est dans l'accumulateur) et la seconde dans l'accumulateur suite à leur évaluation. Puis on rajoute au code généré l'opération en question qui "consomme" une valeur de la pile et remplace la valeur de l'accumulateur par le résultat de l'opération.

print On n'a qu'à générer l'instruction `print` qui s'occupe d'afficher ce qui est dans l'accumulateur

let Le let a 3 phases. Dans la première on génère le code qui calcule la valeur de chaque variable déclarée dans le let, qu'on empile sur le stack. Dans la seconde on génère le corps du let en tenant compte de la position sur le stack des variables du let (grâce à un environnement et à un suivi de la taille du frame d'éxécution). Dans la dernière, on enlève de la pile les variables qui ont servies dans le let.

if Le if a 5 phases. Dans la première on génère le code qui calcule la condition du if. Ensuite on génère une instruction de saut, qui saute vers la branche "else" quand l'accumulateur contient `#f`. Ensuite on génère le corps du "then" suivi d'un saut vers la fin du else (seulement s'il y en a un), puis un label qui indique la position du "else" (même s'il n'y en a pas) et le corps du "else" (s'il y en a un) suivi d'un label qui indique la fin du "else" (s'il y en a un).

Si le premier symbole n'est aucune de ces formes, il s'agit d'une valeur littérale qu'on met dans l'accumulateur.

2.2.1 Instructions de la machine virtuelle

Les instructions utilisées dans la machine à pile sont les suivantes :

`acc v` mets la valeur `v` dans l'accumulateur

`push` empile la valeur de l'accumulateur dans la pile

`[add|mul|sub|quotient|equal|less|mod]` prend une valeur de la pile comme opérande gauche et l'accumulateur comme opérande droit. L'opération est effectuée avec ces opérandes et le résultat est placé dans l'accumulateur.

`pop` incrémente le pointeur de pile de la taille d'un mot.

`load offset` met la valeur à l'adresse `sp - offset` dans l'accumulateur (où `sp` est le stack pointer.

`label L` indication que le label L correspond à cet endroit dans le code.

`jump L` saute vers l'instruction désignée par L.

`jump-false L` saute vers l'instruction désignée par L si `#f` est dans l'accumulateur.

`print` affiche la valeur contenue dans l'accumulateur (un nombre signé, `#f` ou `#t`)

2.3 Traduction vers x86-64

On utilise en x64 les valeurs de 64bit pour représenter 2 types en utilisant les deux premiers bits comme indicateur du type : les entiers (de 62 bits) sont encodés par 00, et les booléens par 01. On se réserve le deuxième bit pour représenter dans une version ultérieure les paires et autres objets.

On utilise aussi une représentation intermédiaire en s-expression pour représenter le code assembleur. La dernière étape étant soit d'avoir en sortie du texte assembleur ou d'exécuter du code machine, il est plus facile de travailler avec cette représentation intermédiaire puisqu'elle offre la possibilité d'abstraire les regsitres. Un de ces avantage est que nous avons pu créer un registre abstrait `accumulator` et un `temp` qui, au moment de la génération de l'assembleur, sont respectivement remplacé par `rax` et `rbx`.

Un des problême lié au passage de code pour une machine à pile avec un accumulateur vers du code pour l'architecture x86 a été d'assumer que les opérations binaires avaient comme source le dessus de la pile et comme destination l'accumulateur. L'architecture x86 nous a remit à l'ordre assez rapidement en nous donnant des résultats erronés pour des programmes com-

pilés. La solution à ce problème a été d'ajouter un registre pour garder la valeur de l'accumulateur et ensuite mettre le dessus de la pile dans l'accumulateur. Ce problème a été rencontré pour la première fois dans la traduction de l'instruction `sub` qui est faite comme suit :

```
mov %accumulator, %temp
mov 0(%stack-pointer) %accumulator
sub %temp %accumulator
shr $2 %accumulator
shl $2 %accumulator
```

Généralement, les opérations de la pile n'ont pas demandé autant de manipulation et elles sont relativement simple en implantation bien qu'il fut difficile de les traduires de façon juste vers une machine à registre.

2.3.1 Runtime x86-64

Pour éviter que le compilateur ne génère plusieurs fois la fonction print, nous avons créé un runtime qui est généré au moment de la compilation selon le système d'exploitation passé en paramètre. Le mecanisme utilisé pourra peut-être être réutilisé dans lors de la définition de fonction globale.

Le runtime pour ce compilateur est très limité, puisque la seule fonction necessaire est `print` (qui utilise la fonction `putchar`). Cette dernière fonctionne de la manière suivante : si la valeur dans l'accumulateur (`rax`) est 1 ou 5 on affiche `#f` ou `#t` respectivement. Sinon, si le nombre est négatif on affiche un caractère `-`, et on inverse le nombre. Puis on empile les digits du nombre en calculant le modulo 10 (en profitant de la division par 10 qui effectue les deux calculs) jusqu'à ce qu'on atteigne zéro. Puis on dépile la pile en affichant chaque chiffre sur la pile grâce à `putchar`

3 Problèmes rencontrés

Un problème de communication nous ayant fait croire que la date de remise était vendredi le 22 février, nous avons commencé à terminer le compilateur seulement dimanche soir, quand nous nous sommes rendus compte que la date de remise était mardi. Heureusement le café filtre au café Math-Info est une ressource à coût raisonnable.

La traduction vers une machine à pile s'est fait relativement bien, mais gérer les `let` nous a valu quelques conversations. Il faut s'assurer que l'on conserve (correctement) la position des variables sur la pile par rapport au fil d'exécution, et comme plusieurs implémentations étaient possibles, nous avons hésité sur le chemin à prendre.

Le fait que les valeurs entières signées soient encodées sur 62 bits nous ont forcé à penser correctement la manière dont les opérations arithmétiques ont lieu (quel shifts faut-il faire ou ne pas faire) ainsi que la manière dont il faut afficher les nombres négatifs.

Comme nous n'avions jamais travaillé autant avec x86-64, ce fut parfois un défi quand on s'attendait à un comportement donné mais que le vrai comportement en était un autre, ne serait-ce que parfois l'ordre des opérandes. Aussi, il a fallu apprendre à se servir de gdb en mode assembleur, ce qui n'est pas forcément trivial si on veut récupérer certaines informations mais pas trop (sinon ça peut porter à confusion et nous faire perdre le fil mental de l'éxécution).

4 Résultats et évaluation

Si on se fie aux tests effectués, ce qui devrait fonctionner fonctionne. Les calculs effectués sont les bons, et les nombres et valeurs affichées sont les bonnes. Par exemple on peut imbriquer plusieurs `let` avec des noms de variables pareils ou avec des références qu'il est important de choisir correctement, par exemple l'expression :
`(let ((x 2) (y 4)) (let ((x (+ y x)) (y (* y x))) (print (+ x y))))`
Affiche correctement 14.
Les tests qu'on effectue testent beaucoup de formes d'expressions valides.

Par contre les cas extrêmes ou d'exception ne sont pas forcément gérés, par exemple la division par zéro ou l'addition de deux booléens (qui va donner un résultat probablement invalide). D'un autre côté la plupart des erreurs de syntaxe auxquelles on pourrait s'attendre sont détectées.

Nous avons commencé à faire un JIT qui charge les instructions x86-64

en mémoire. Malheureusement la spécification de l'encodage des instructions est parfois complexe ou difficile à comprendre, et comme nous étions pressés par le temps, ce fut mis de côté avec seulement les instructions `mov push add` et `ret` supportés (et encore, seulement certaines formes de celles-ci).

5 À faire et à corriger

Comme nous avons écris le compilateur en relativement peu de temps suite à une erreur de notre part, une partie du code est probablement parsemé de mauvaises décisions et de code ayant des lacunes esthétiques. Une partie du travail au début de la deuxième phase sera de réusiner une partie du code afin de la rendre plus simple et plus modulaire. De même l'élaboration rapide du code fais que celui-ci contient quelques noms mal choisits qui devront être changés. De même, il nous faudra probablement revoir une partie de la structure du compilateur. Nous espérons qu'en entrant dans la deuxième phase de développement, il nous sera plus facile de faire ces modifications.

Nous n'avons pas non plus eu la possibilité de s'assurer que le code assembleur généré était efficace et optimisé. En fait il y a probablement beaucoup à faire en termes d'optimisation et de la machine virtuelle, et/ou de la génération de code (juste par exemple pour tout ce qui est allocation de régistres).