IFT3065 - RAPPORT PHASE 3

ALEXANDRE ST-LOUIS FORTIER AND EMMANUEL BENGIO

Résumé

Le rapport suivant fait un survol du compilateur rogue, et traite en profondeur de la troisième phase de l'écriture du compilateur. Cette phase visait à améliorer les performances du code généré, ainsi que de permettre un niveau avancé de gestion mémoire, notemment par l'addition d'un glaneur de cellules et de la gestion des appels terminaux. Le rapport tente d'expliquer le fonctionnement des différentes parties ajoutées lors de cette phase du projet et les différents choix adoptés et présente les problèmes rencontré lors du dévelopement.

Pipeline de compilation

La compilation du code scheme par rogue est divisée en plusieurs parties, combinées pour générer et compiler du code x86-64.

Analyse syntaxique et lexicale. À partir d'un fichier source lu, les expressions Scheme, qui sont des s-expressions, sont reconstruites dans des listes en suivant une grammaire qui se résume à peu près à ceci:

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

Cela inclut la lecture de symboles et de valeurs quotées ou quasiquotées.

Par la suite, l'expansion de macros effectue en passant une analyse syntaxique (par exemple pour vérifier que les formes spéciales `if` ou `let` sont bien formées)

Le résultat de la lecture est donc, dans le compilateur, une s-expression représentant le progamme dans son entièreté. Lors de la compilation d'un progamme, le code source de la libraire standard de rogue est aussi lu et mis avec le code source entré pour être compilé.

Expansion de macros. L'expansion de macros est ensuite effectuée sur le résultat de la lecture, où une certain analyse syntaxique est effectué, mais plus notamment, l'expansion des formes spéciales let, or, and, cond, etc. vers les formes primitives lambda et if.

C'est aussi dans cette phase que les quasiquotes sont traduites vers une forme compilable. C'est à dire que les valeurs non-unquotées à l'interieur d'une quasiquote le restent, et les autres sont laissées à être compilées comme expressions (et non comme constantes). Par exemple l'expression `` `((foo . ,bar) ,(+ 3 1)) `` sera traduite en `(cons (cons 'foo bar) (cons (+ 3 1) '()))`.

L'expansion de macros utilise abondamment l'associateur de patrons pour parvenir à ses fins, par exemple, l'expansion des defines globaux en deux séries de define et de set! match sur le patron du define suivi de code pour garder un liste des définitions et des valeurs définies.

Pliage de constantes. Un phase de constant folding est effectuée sur le code expansé. Il s'agit en fait d'une reconstruction par fouille en profondeur qui calcule en remontant les expressions ne comportant que des valeurs constantes.

Pour une forme qui est une liste, si on tombe sur un let ou un lambda, on étend l'environnement des variables déclarées par cette forme.

Sinon, après avoir plié récursivement chaque élément de la liste, si on détermine qu'il ne s'agit que de constantes (numériques par exemple), on effectue l'opération si elle est connue (par exemple + ou -) et si elle n'est pas redéfinie par l'environnment courant.

Analyse de variables libres. Dans cette phase on détecte l'ensemble des variables libres (free variables, FV) de chaque lambda-expression en suivant cette convention:

$$\begin{aligned} FV(c) &= \{\} \text{ pour } c\text{une constance} \\ FV(x) &= \{x\} \text{ pour } x\text{un symbole} \\ FV((\lambda\ (p_1 ... p_n)\ E)) &= FV(E) \backslash \{p_1, ..., p_n\} \\ FV(\ (E_0\ E_1 ...)) &= \bigcup_i FV(E_i) \end{aligned}$$

Lorsqu'on trouve une lambda expression qui doit être évaluée, on vérifie si cet ensemble est vide. S'il ne l'est pas, lors de la création de l'objet fermeture on rajoute la valeur des variables libres à l'objet fermetures. Ces valeurs de variable libres sont utilisées dans le code généré modifié par cette phases. En effet, si on détecte une variable libre dans une lambda expression, l'accès à ces variables se fait à partir de l'objet fermeture qui contient des références vers ces variables libres. De plus, ces références sont enfermées dans des paires lors de la création de la fermeture, donc chaque accès en lecture d'une variable libre x devient `(car x)` et les `set!` deviennent `(set-car! x ...)`.

Compilation vers une machine à pile virtuelle. Cette étape traverse récursivement la s-expression obtenue après toutes les transformations pour générer le code d'une machine à pile. Cette génération peut se décomposer en ceci:

- Les valeurs constantes (liste vide, nombre, chaine de caractère) génèrent une instruction qui les charge dans l'accumulateur. Par exemple 2 génère `(acc 2)`. De plus certaines valeurs, comme les chaînes de caractères, génèrent du code global (qu'on rajoute à la fin) qui définit.
- La forme let (qui est d'ailleurs seulement utilisée à l'interieur de cette phase quand on détecte une lambda expression immédiatement appellée) empile le résultat de chaque expression dans les liaisons et génère son corps avec ces variables liées.
- L'appel de fonction en général est fait en empilant le résultat de l'expression de chaque argument, en évaluant la fonction appellée en elle même (par exemple une variable locale doit être récupérée de la pile), puis en effectuant un `(call)` qui va brancher en empilant l'adresse de retour.
- Toutes les fonctions primitives (qui sont d'ailleurs généralement dénotées en commençant par %) sont inlinées ou sont des fonctions standard du runtime, par exemple `%+` est transformé directement vers l'instruction de machine à pile `(add)` (à la place de `(call)`), et par exemple l'appel de `%object-setq`

est transformé vers l'instruction `(std %object-setq)` qui sera transformé vers du code assembleur qui appelle directement une fonction écrite en assembleur dans la librairie standard.

- Les lambdas expressions génèrent des parties de code, une partie qu'on rajoute au code global qui est le corps de la lambda expression (qui entre autres choses vérifie le nombre d'arguments et crée une liste pour les paramètres reste avant d'exécuter le corps) et une autre partie qui est la création d'une fermeture. On crée tout le temps des fermetures pour manipuler des lambda expressions (ce n'est pas très efficace mais c'est plus simple), celles-ci font référence au variables libres et gardent un pointeur vers l'adresse du corps de la lambda expression.
- La forme spécial if génère le test, puis vérifie si'il est égal à Faux avant de brancher vers la branche "alors" ou "sinon".

Compilation vers une représentation de x86-64. Cette étape traverse linéairement le code de la machine à pile pour le transformer vers du x86-64. En effet chaque instruction de la machine virtuelle est en correspondance 1 à 1 avec une série d'instructions x86-64, encodées en s-expressions pour pouvoir mieux les manipuler par la suite. Par exemple l'instruction de machine à pile `(load x)` qui charge une valeur sur la pile à une distance x du dessus de la pile est traduite par une seule instruction:

```
‘((mov (& ,(* word-size (- x 1)) (% .  stack-pointer)) (% .  accumulator))
```

Compilation finale. L'étape finale consiste à générer le fichier assembleur à partir de la représentation en s-expression du x86-64. Puis ce fichier est compilé par gcc, accompagné de deux fichiers C de la librairie standard de rogue, respsonsables du GC et des entrées/sorties.

Gestion mémoire et objets

Allocation. Au début du déroulement d'un programme compilé, un espace mémoire est alloué pour la durée de l'exécution du programme.

Lors de l'allocation d'un objet, on commence par vérifier qu'il reste assez d'espace pour l'allocation. Si ce n'est pas le cas, on appelle la routine de GC. Par la suite, 16 octets sont tout de suite réservés, 8 pour la longueur de l'objet (en octets) et 8 pour le type de l'objet (un entier unique à chaque type). Puis un certain nombre de mots de 8 octets sont réservés pour les données selon la nature de l'objet. La manoeuvre pour réserver de l'espace se limite à incrémenter le pointeur d'allocation contenu dans un régistre.

Le pointeur vers le "millieu" de l'objet est ensuite retourné pour être utilisé, le millieu désignant le début de la plage de données de l'objet, incrémenté de 1, ou de 2 pour les paires.

Types et encodage d'objets. Toutes les valeurs manipulées dans les programmes compilés sont des mots de 8 octets, elles peuvent représenter de manière générale un entier, un objet, une paire ou une valeur spéciale, comme les booléens vrai et faux.

Pour différencier entre ces 4 catégories d'objet, les deux bits les moins significatifs des valeurs sont utilisées pour encoder chaque classe de valeur.

Plus particulièrement, les entiers sont encodés avec 00_2 (ce qui les multiplie par 4, et ne complique pas trop les calculs), les objets avec 01_2, les paires avec 10_2, et les valeurs spéciales par 11_2.

Paires. Lors de l'appel de la fonction `cons` un objet est alloué en mémoire de longueur 16 (donc 32 octets sont alloués), le premier mot de données contient la valeur du champ `car` de l'objet, et le deuxième mot la valeur du champ `cdr`.

Comme mentionné précédemment, la valeur d'une paire est "taguée" par les bits 10_2. Cela veut dire qu'à la place d'être le pointeur vers l'adresse mémoire alloué pour la paire, c'est un pointeur vers 2 octets plus loin. Ce choix implique de calculer les bons décalages, par exemple pour aller chercher le champ `car` il faut faire `mov -2(%R1),%R2` et non `mov (%R1),%R2` si R1 contient la valeur taguée de la paire.

Ce choix à été fait car la paire est un objet très utilisé en Scheme et donc il est pratique de pouvoir la détecter facilement et de pouvoir optimiser la lecteur et l'écriture de ses champs.

Objets. Comme pour les paires, les objets utilisent un tag pour reconnaitre la classe d'un valeur donnée comme étant un pointeur vers un objet. Les objets utilisent le tag 01_2, ce qui veut dire que le bon décalage est -1 lors d'accès aux données.

Il existe quelques types d'objets primitifs: les chaînes de caractères, les symboles, les caractères, les vecteurs, les fermetures et les blocs de code machine.

Les fonctions standards génériques `%object-new,%object-(set|get)[q,b,raw]` permettent d'écrire toutes les fonctions standards pour chaque type. (Nous donnerons l'exemple des vecteurs dans une section dédiée)

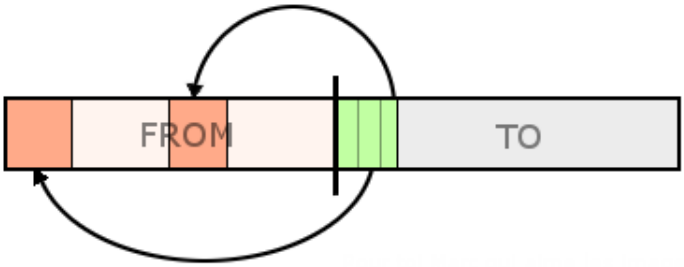
Valeurs spéciales. Il existe des valeurs spéciales pour les booléens faux (11_2) et vrai (111_2), pour la liste vide (1011_2) et pour la valeur `#eof` (11011_2).

Récupération de l'espace mémoire. La récupération de la mémoire allouée (par incrémentation d'un pointeur d'allocation) est effectuée par un GC de type Stop-and-Copy.

Stop and Copy. Cet algorithme de récupération mémoire consiste à séparer l'espace mémoire en deux, un from-space et un to-space. Les objets sont alloués dans le from-space. Lorsque la mémoire est pleine ou sur demande, les objets vivants sont copiés et compactés dans le to-space, et les pointeurs qui y pointent sont modifiés pour refleter cette copie. Les objets sont vivants si on peut les atteindre depuis une racine. Il y a deux espaces mémoires considérés comme des ensembles de racine, la pile et les variables globales.

Les variables globales sont délimitées par deux étiquettes, on peut donc considérer le tableau allant de la première à la deuxième comme un tableau de racines. Toutes la valeurs empilées sur la pile sont des objets scheme ou des addresses de retour et sont considérées comme des racines. Toutes les racines qui sont des objets et qui pointent vers le tas sont considérées par le GC.

Toutes les variables racines sont copiées dans le to-space et à leur ancienne place dans le from space on indique leur nouvelle adresse dans le to-space (ainsi que le fait qu'elles ont été copiées). Puis on répète ceci pour toutes les valeurs qui pointent encore dans le from-space et qui sont dans le to-space (chaque type d'objet possédant un tableau de ces pointeurs, qui est parfois vide comme pour les chaînes, il ne s'agit que d'une boucle).



Il y a un cas spécial, les blocs de code machine (voir section sur la compilation JIT pour leur origine). À l'intérieur de ces blocs il existe des références vers des objets externes au bloc qui peuvent être déplacés par le GC. Ces références sont un adressage relatif à la position de l'instruction (par exemple en x86-64, `movq 1234, $rax` signifie qu'on va chercher le mot 1234 octets après la position de l'instruction courante est qu'on la place dans `rax`. Il faut donc mettre à jour ces valeurs puisque la position entre le bloc de code et les objets référencés à l'interieur changer lors d'un GC. Un bloc de code machine maintient à sa fin une tableau de paires valeurs-position qui dit où dans le code changer la référence et vers quelle valeur cette référence pointe. On peut donc les mettre à jour aisément.

La routine de GC est informée de combien d'espace l'allocation aimerait disposer. S'il manque d'espace le GC affiche un message d'erreur et quitte le programme de par lui même. Une solution à explorer serait la gestion mémoire avec plusieurs (et surtout un nombre variable de) from-space.

Exemple d'objet: vecteurs

Un vecteur est un type d'objet standard. On les créés à l'interne avec la fonction `(%object-new type-uid-vector size)` où *size* est un multiple de la taille des mots, 8, et plus particulièrement 8 fois le nombre de cellules désiré pour le vecteur créé. Cela alloue un espace mémoire contigu de $16 + size$ octets, dont on peut se servir comme d'un vecteur dans ce cas particulier.

Pour mettre une valeur dans le vecteur on utilise la fonction `(%object-setraw vector index value)` qui met directement la valeur *value* à la bonne case, donc par exemple, la liste vide encodée par $1011_2 = 11_{10}$ sera placée à cet endroit en mémoire (donc contrairement à la fonction %object-setq qui prend un nombre en paramètre, setraw prend un objet). Pour aller chercher une valeur on utilise simplement `(%object-getraw vector index)`. On écrit ces fonctions en Scheme dans la librairie standard de rogue et elles sont compilées avec tout le reste, il est donc très simple d'écrire du nouveau code pour un type qu'on veut ajouter. Par exemple voici une autre fonction sur les vecteurs:

```
(define (vector-length v)
 (if (vector? v)
     (quotient (%object-length v) 8)
     (error "*** ERROR in vector-length: Argument not a vector" v)))
```

On peut aisément et similairement écrire `(vector?)`, `(list->vector)`, `(make-vector)`, ...

Associateur de Patrons

[UTILS/PATTERN-MATCHING.SCM]

Pour faciliter l'écriture future de plusieurs des composantes du compilateur, une macro d'association de patrons, basée sur du code de Marc Feeley, a été développée. La première phase du compilateur en bénéficiant est l'expanseur de macro. Cette macro accepte du code de la forme suivante:

```
(match <expression> [as <identifier>]
 {(<patron> [where <expression>] <expressions>)})
```

Un <pattern> est ici une une expression scheme. La macro tente simplement de faire concorder les symboles de l'expression avec ceux des patrons en partant avec celui du haut. Par défaut, les symboles d'un patron sont considérés comme des identifiants auxquels seront liés les symboles de l'expression afin de pouvoir les utiliser dans les expressions (car elles sont dans un begin implicite) de la branche. Toute identifiant valide peut être utilisé pour lier les variables d'une expression à un patron sauf le caractère sous-ligné qui a une fonctionnalité spéciale que nous verront un peux plus tard.

```
(match '(a b c d e)
 ((v w x)       "non") ;; pas assez d'arguments
 ((v w x y z ) "oui") ;; lie les symboles au variables
                      ;; v <= 'a, w <= 'b, ...
 ((v . w)       "non") ;; oui si le 2e patron avait été
                      ;; absent
```

Il est à noter que les patrons doivent êtres de la même longueur. La syntaxe avec un point peut être utilisé pour des patrons de liste de longueur inconnue.

Lorsqu'on veut comparer les symboles du patron à ceux de l'expression au lieu de les lier à des identifiants, il faut spécifer une telle comparaison par l'ajout d'une apostrophe devant le symbole à comparer.

```
(match '(a b c)
 (('a . rest) "oui")) ;; rest <= '(b c)
```

S'il a été mentionné précédemment que le caractère sous-ligné est un cas spécial, c'est parce qu'il agit comme une variable, mais il n'engendre pas le code nécessaire pour effectuer une liaison. Un patron ne contenant que le caractère sous-ligné agit effectivement comme un cas par défaut.

```
(match <n'importe quoi>
 (_ "oui")) ;; attrape tout
```

Une autre extension intéressante à la syntaxe de base de l'association de patron est la possibilité d'ajouter une clause where qui permet en fait d'ajouter une condition à l'utilisation d'une certaine branche. Il est à noter que les symboles liés à un identifiant du patron peuvent être utilisé à l'intérieur d'une clause where.

```
(match (list 1 2) as foo
 ((a . b)  where (> (length foo) 2) "est une liste")
 (_ where (number? a) "est numérique")
 (_                   "est autre chose"))
```

Stratégie lié à l'efficacité de l'associateur de patron. Un des problèmes rencontré lors de l'élaboration de l'associateur de patrons a été l'explosion des branches. Le code généré par l'associateur de patrons a une tendance à être une série de tests dont les branches «sinon» étaient répétées. Le problème d'explosion était si grand qu'il était rendu impossible de compiler le compilateur dû à une expression match un peut trop grande. Pour corriger cette déficiance, l'associateur de patrons crée maintenant des «thunks» référencés pour les branches «sinon» et les branches font seulement un appel au «thunk».

Compilation just-in-time - eval - bootstrap

Il existe un module (qu'il faut activer à la main) de compilation JIT dans rogue. Celui-ci permet de compiler du code x86-64 et de l'exécuter à l'interieur d'un programme. Il est même possible d'utiliser une forme simple de eval (qui ne fait référence qu'à des variables globales. En fait, pour faire fonctionner eval, le compilateur se compile presqu'en entier afin de pouvoir traduire du code sous forme d'une liste (d'une s-expression) vers du x86-64, mais au lieu d'écrire ce code assembleur dans un fichier il le compile en mémoire.

Encodage des instructions et exécution. Les instructions x86-64 sont encodées vers une représentation en tableau d'octets à partir de leur représentation en s-expression. Après avoir généré chaque octet, on génère un objet de type bloc de code machine qu'on remplit octet par octet avec le code.

Encodage des labels. Comme on ne connait pas d'avance leur adresse relative ou absolue, les instructions qui utilisent des labels comme opérandes sont encodées en deux temps. Quand on les recontre on laisse une place vide dans les instructions encodées et on y garde une référence dans une liste. À la fin de la première étape, on parcours cette liste pour venir chercher l'adresse relative à l'emplacement de l'instruction et la placer au bon endroit dans le code compilé.

Si on fait référence a une variable globale externe au code étant compilé (par exemple à la fonction `cons`) on va chercher l'objet global dans un table d'association de symboles vers les variables globales (générée par le compilateur) et on calcule la différence de position de l'objet (qui n'est par exemple qu'un objet fermeture dans le cas de *cons*) et la position dans le code relative à la position en mémoire du début de bloc de code machine.

À la fin de l'objet bloc, on rajoute chaque endroit dans le code machine où on fait référence à une valeur externe avec un tableau de paires valeur-position.

Eval. La fonction *eval* est presque fonctionnelle, en fait par manque de temps, le code machine généré lors du *eval* peut faire référence à des variables globales, mais pas à des routines de la partie de la librairie standard écrite en assembleur. Il faudrait soit mettre en place un mécanisme automatique pour y faire référence, ou bien recompiler à chaque eval les environ 700 instructions assembleur que représentent le runtime écrit en x86-64.

Bootstrapping. Il est possible de compiler rogue avec rogue compilé par Gambit, et le rogue (appellons le rogue2) généré sera totalement fonctionnel, seulement extrêmement plus lent. Pour cette raison, et parce qu'il manque eval, il n'est pas encore réellement envisageable de bootstrapper au delà de 2 générations gambit et de compiler un rogue3 avec rogue2.

Optimisation tail-call
stackvm.scm

(disponible dans la branche *tail-call*)

Dans la génération du code de machine virtuelle à pile, on vérifie à chaque appel de fonction si on est en position terminale et si on peut effectuer un tail-call. Si c'est le cas, on essaie d'empiler les arguments de la fonction appellée, puis de conserver l'adresse de retour de la fonction appellante, puis de faire glisser ces arguments vers le bas pour écraser le bloc d'activation courant, puis on rempile l'adresse de retour[1] et on branche vers la fonction appellée.

Malheureusement ça ne semble pas marcher, pour diverses raisons. La difficulté d'implantion est entre autre due au fait que le module stackvm fait trop de choses en même temps et qu'il devient très facile de le briser si on implémente une nouvelle fonctionnalité.

Optimisation constant folding

Comme mentionné précédemment, cette phase d'optimisation effectue une fouille en profondeur qui calcule en remontant les expressions ne comportant que des valeurs constantes et qui n'affecte pas le reste des instructions.

On gère les fonctions suivantes: `+,-,*,quotient,modulo,string-append`.

Dans le cas de quotient, un division par zéro pourrait entrainer une erreur à la compilation, ce qui est indésirable si le code n'est jamais executé. Il y a donc un cas spécial qui s'assure de ne pas diviser par zéro.

De plus, lors de la fouille, on s'assure de ne pas évaluer une expression si la fonction qu'on plie est assignée à une autre valeur (par exemple un let qui déciderait de réassigner + à une autre fonction).

Optimisation peephole

Lors de l'optimisation peephole, on regarde les instructions de la machine à pile virtuelle par fenêtres pour s'assurer qu'il n'y a pas d'opérations inutiles ou redondantes.

On gère ces cas:

- `(acc x) (acc y) => (acc y)`, la première instruction *acc* ne sert à rien.
- `(restore-state) (save-state) => ()`, ça ne sert à rien de récupérer l'état de la fermeture depuis la pile pour le rempiler tout de suite après
- `(pop 0) => ()`, il arrive parfois de générer un pop qui ne pop rien, il est facile de l'enlever ici.
- `(pop) (pop) => (pop 2)` , il arriver de générer deux pop de suite.
- De manière plus générale on gère: `(pop [x]) (pop [y]) => (pop (+ [x ou 1] [y ou 1]))`

L'algorithme est extrêmement simple a étendre, il suffit d'ajouter le pattern approprié ainsi que la séquence d'instructions qui devraient la remplacer, et d'appliquer l'optimisation peephole à la transformation du pattern associé au restant des instructions.

[1] Mais il semblerait que ce fut une erreur, lors de l'appel de fonction, d'empiler d'abord les arguments puis l'adresse de retour, voir rétrospection.

Instructions de la machine virtuelle à pile

acc: <valeur> : Met la valeur dans l'accumulateur.

load-from-offset: <int> : Va chercher la valeur décalé par int par rapport à la valeur pointé par l'acummulateur

store-at-offset: <int> : Écrit la valeur du dessus de la pile et l'écrit à la valeur pointée par l'accumulateur décalée par int. Conserve l'état de la pile.

push: [<n'importe quoi>] : Met la valeur de l'accumulateur sur le dessus de la pile. Le paramètre 'acc souvent passé à push ne sert à rien et semble être le vestige d'une ancienne décision de possiblement pouvoir donner une valeur littérale à l'instruction.

pop: [<int>] : Enlève <int> éléments du dessus de la pile. Si aucun paramètre n'est donné, un seul élément sera enlevé.

silde: <int1> <int2> : Fait glisser <int1> premier éléments de la pile par dessus les <int2> en dessous. Les valeur écrasées ne sont plus accessibles. Le compteur de pile n'est pas modifié.

load: <int> : Met la valeur se trouvant à un décalage de <int>+1 du dessus de la pile.

mul: : Multiplie la valeur dans l'accumulateur avec le dessus de la pile et le remet dans l'accumulateur.

mod: : Calcule le reste de la division de la valeur dans l'accumulateur avec le dessus de la pile et le remet dans l'accumulateur.

equal: : Regarde l'egalité entre la valeur dans l'accumulateur avec celle du dessus de la pile et le remet dans l'accumulateur.

less: : Regarde la relation d'ordre (<) entre la valeur dans l'accumulateur avec celle du dessus de la pile et le remet dans l'accumulateur.

lesseq: : Regarde la relation d'ordre (<=) entre la valeur dans l'accumulateur avec celle du dessus de la pile et le remet dans l'accumulateur.

std: <label> : Appelle une fontion écrite en assembleur dans runtime.scm dont l'étiquette est <label>.

label: <nom> : Insère une étiquette nommé <nom>.

jmpacc: : Déplace le pointeur d'instruction à l'adresse se trouvant dans l'accumulateur.

jump: <label> : Déplace le pointeur d'instruction à l'étiquette <label>.

jump-false: <label> : Déplace le pointeur d'instruction à l'étiquette <label>.

return: : Déplace le pointeur d'instruction à l'addresse sur le dessus de la pile.

load-address-of: <label> [<offset>] : Met l'adresse du label dans l'accumulateur et y ajoute l'offset.

call: <int> : appele (jump) la fonction pointé par l'accumulateur et met dans un registre le nombre de paramètres <int> empilés sur le dessus de la pile

procedure-header: <int> <booléen> : Génère l'initialisation d'une procédure qui reçoit <int> arguments. Gère entre autre les paramètre restse si <booléen> est vrai.

string-litt: <string> <symbol>: Réserve un espace dans le segment .data du programme généré. Cet espace peut être référencé par <symbol>.

symbol-litt: <string> <symbol> : Réserve un espace dans le segment .data du programme généré. Cet espace peut être référencé par <symbol>.

global: <symbol> : Réserve un mot dans le segment .data du programme généré.
call-extern-c-funciton: <string> : Appele la fonction C externe de nom string.
store-global: <label> : Met la valeur de l'accumulateur dans l'espace référencé par <label>.
load-global: <label> : Met la valeur de l'espace référencé par <label> dans l'accumulateur.
load-free: <int> : Met dans l'accumulateur la <int>-ième variable libre de la fermeture courante.
save-state: : Empile les registres qui doivent être conservé suite à l'exécution d'une fonction appellée qui pourrait y écrire.
restore-state: : Dépile les registre empilés par save-state.
gc-align: <int1> <int2> : Aligne le deuxième paramètre sur un mot dant le segment .data. Réserve un certain espace en fonction du premier paramètre.

Librarie standard

Il existe une librairie standard qui définit les opérations standard sur:

- les paires et les listes: cons, car, cdr, set-car!, set-cdr!, null?, list, list?, list-ref, list-set!, append, member, assoc, list-ref (lib/string.scm)
- les chaînes de caractères: char?, eof-object?, integer->char, char->integer, char[=|<=|>=]?, string, string?, string-length, string-ref, string-set!, string-append, string-equal, substring, string->number (lib/string.scm)
- les nombres: +, -, *, quotient, modulo, =, <, >, <=, >=, sum, ilog, pow, abs, integer->string (lib/arithmetic.scm)
- les vecteurs: vector, vector?, make-vector, list->vector, vector-ref, vector-set!, vector-length (lib/vector.scm)
- entrées/sorties: open-pipe, with-input-from-process, open-input-file, open-output-file, close-output-port, read-char, peek-char, write-string, read-line, shell-command, command-line (lib/io.scm, lib/io.c)
- des fonctions utilitaires et d'ordre supérieur: error, exit, print, display, write, write-char, newline, getchar, boolean? equal?, not, procedure?, closure?, apply, eq?, eqv?, symbol? symbol->string, string->symbol, map, for-each, gensym

De plus, avec un peu de manipulations, on peut avoir une fonction eval relativement limitée qui a accès seulement aux variables globales mais pas au runtime.

Rétrospective

L'élaboration du compilateur se faisant parallèlement à l'apprentissage de leur fonctionnement, plusieurs des choix fait tôt dans le dévoloppement peuvent maintenant être considéré avec du recul et de l'expérience.

Représentation intermédiaire. Ayant été développé en parallèle avec l'apparitions du besoin de nouvelle fonctionnalité, le langage intermédiaire s'est retrouvé à être un ensemble un trop gros d'instructions trop complexes. Cette lacune a posé des problèmes pour l'optimisation peephole pour laquelle il a été difficile de cerner des blocs constitués d'instructions simples à optimiser. De même, nous aurions dû éviter l'utilisation de fonction telles que call qui font abstraction de plusieurs détails

d'implantation. En utilisant seulement des instructions de base telles que jump, il aurait été peut-être plus facile de remarquer la présence de complexité inutile dans la gestion de mémoire (adresse de retour). Il semble maintenant beaucoup plus judicieux de planifier la représentation intermédiaire dès le début de l'élaboration du compilateur, et d'en faire un langage élégant aux instructions simples et de quantité limité mais suffisantes pour émuler une machine de turing.

Plusieurs passe simples. Une chose apprise lors de la traduction de scheme vers la représentation intermédiaire est la nécessite d'implanter des phases simples qui se composent facilement. Heureusement, la fonction translate-ast-inner est l'antithèse de ce principe. La fonction se charge à la fois de trouver les variables libres, d'identifier le appels terminaux (implantation partielle sur une la branche tail-call), traduire les expressions vers leur représentation intermédiaire en plus de de faire quelques optimisations. Certaines de ces fonctionnalités necessitent une approche descendante alors que pour d'autre une approche ascendante aurait été mieux indiquée. Une meilleure approche aurait été de créer un arbre de syntaxe abstraite qu'il aurait été facile d'annoter en plusieurs passe.

Adresse de retour. Un des problèmes rencontré lors de la mise en oeuvre de la gestion des appels terminaux, qui a été à l'origine de l'échec de leur implantation, fut le mauvais choix de la position de l'adresse de retour des fonctions dans la pile d'exécution. Nous avons fait l'erreur d'empiler l'adresse au dessus des arguments de la fonction, complexifiant du même coup la manipulation de la pile lors d'optimisation et de transformation du code. Un meilleur choix aurait été d'empiler les arguments par dessus l'adresse de retour. Dans un tel cas, il est beaucoup plus simple de calculer l'adresse de retour qui est simplement à la base du bloc d'activation plutôt qu'à un décalage variable selon la fonction.

Préparation et outils. Certains des outils développés au cours du projet se sont révélés particulièrement utiles lors du développement du compilateur. Avec une connaissance appropriée des problèmes qui seront rencontré, le choix et le développement d'outils et de librairies devrais se faire relativement tôt dans le projet afin d'en bénificier le plus rapidement possible.

Gestion des erreurs. Une bonne gestion des erreurs est primordiale autant pour l'usager que pour le développeur du compilateur. Cela permet de mieux identifier les problèmes de génération de code. Nous considérons la gestion des erreurs de Rogue comme étant plutôt faible et il nous semble qu'une meilleure gestion nous aurait sauvé beaucoup de temps. L'élaboration d'une infrastructure sophistiquée de gestion d'erreurs nous semble donc être un point à déterminer dès le début de l'écriture d'un compilateur.