IFT3065 - RAPPORT PHASE 2

ALEXANDRE ST-LOUIS FORTIER AND EMMANUEL BENGIO

Résumé

Le rapport suivant traite de la deuxième phase de l'écriture du compilateur rogue. Cette phase visait à obtenir un coeur fonctionnel pour le langage scheme incluant l'élaboration d'une librairie standard et d'un expanseur de macro, la gestion correcte de la mémoire dans le contexte d'un langage fonctionnel ainsi de générer une phase de compilation JIT. 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.

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>
 {(<patron> [where <expression>] <expression>)})
```

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 l'expression de la branche. Toute idetifiant 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 a
 (_ 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».

Expanseur de Macro [expand.scm]

Suite à la deuxième phase de développement du compilateur, l'expanseur de macro bénéficie d'un rôle primordial dans la transformation du code ainsi que dans sont analyse.

Expanseur de macro comme mécannisme de génération de représentation intermédiaire. Un des premier but de l'expanseur de macros est de générer une représentation intermédiaire simple qu'il sera facile d'analyser. Cette représentation intermédiaire est caractérisée par l'utilisation du symbole «pour cent» devant les formes primitives.

Pour faire cette transformation, la fonction expand remplace les expressions non-primitives à l'aide d'un environnement contenant des macros prédéfinies. Cette fonction définit comment se fait le remplacement des formes non-primitives et comment se propage l'expansion sur les formes primitives.

Expanseur de macro comme mécannisme d'analyse suntaxique.

Propriétés liés à l'expansion de macros.

(1) Une macro qui test si le code qu'elle reçoit est correct génèrera du code correct.

(2) Tout code passe passe par l'expanseur de macro.

(3) L'expanseur de macro doit conserver des données sur les environnements

Ces trois propriétés sont à la base de l'idée d'utiliser les macros pour analyser la syntaxe du programme. Comme ont a vu précédement, on utilise l'expanseur de macro pour générer la représentation intermédiaire. Lorsque cette transformation est faite, toutes les informations nécessaire à l'analyse syntaxique son présentes.

Pour effectuer l'analyse on met à profit l'associateur de patron définit comme la forme match utilisé à l'intérieur de macros définies dans la variable environement.

```
(cons 'let
 (make-env-record 'macro
  (lambda (expression)
   (match expression
    ((_ bindings . expressions)
     where (or (bindings? bindings)
               (error "ill-formed bindings in let"))
     (let ((binding-names (map car bindings))
           (binding-expressions (map cadr bindings)))
       `((lambda ,binding-names %body ,expressions)
         ,@binding-expressions)))
    (_ (error "ill-formed let"))))))
```

L'associateur de patron nous permet de facilement tester la structure d'une expression et on utilise la clause where du patron pour effectuer des tests sur la forme des différentes expressions constituantes et ainsi rapporter des erreurs plus précises. Un cas par défaut nous permet facilement de rapporter une expression mal formée.

Les macros définies de la sorte ne prennent pas en compte l'environnement. S'il se trouve à être nécessaire, il serait facile de l'ajouter en paramètre et d'ajouter des test supplémentaires. La fonction qui s'occupe de la propagation de l'expansion semble pour l'instant suffisante pour repérer les erreurs nécessitant une analyse de l'environnement.

Problèmes rencontré dans l'analyse syntaxique. Comme l'analyseur de syntaxe de la phase précédente a été abandonné au profit de l'expanseur de macros, certains tests sont peut-être absents. Cette lacune est dû à une priorité mise sur la génération de programme correct pour du code correct. Il reste neanmoins trivial avec l'architecture actuelle d'ajouter les tests qui rejeterons les mauvais programmes.

Gestion de define et define macro. Un problème rencontré par l'expanseur de macro à été la gestion des formes define et define-macro. Ces formes sont problématiques dû au fait que leur expansion dépend de l'expression où elles sont définies.

Selon la spécification du langage scheme r6rs, les formes define et define macro peuvent être entrelacé par des expressions, alors que les mêmes formes à l'intérieur de lambda et de let (let* et letrec inclus) doivent être à la tête de leur corps.

Pour différencier ces deux cas lors de l'expansion, le compilateur génère deux instructions. Un programme se trouve généralement dans une expression %toplevel.

```
(%toplevel
  {<expression> | <definition> | <definition de macro>})
```

L'autre instruction, %body, est généré lors de l'expansion des formes lambda et let (let* et letrec non-inclus puisqu'il sont expansé vers let).

```
(%lambda <liste-arg> (%body <liste-expressions>))
(%let <liste-associations>
  (%body <liste-expressions>))
```

Lorsque c'est formes sont rencontrées, l'expanseur appelle respectivement les fonctions expand-toplevel et expand-body. En pratique, l'expansion de ces formes pourrait se faire par la variables environnment, mais la complexité de ces cas fait qu'il est peut-être mieux de le garder à l'extérieur.

Modèle mémoire et objets

Allocation. Au début du déroulement d'un programme compilé, 4 mégaoctets de mémoire sont alloués pour la durée de l'exécution du programme. Comme le compilateur ne possède pas encore de garbage collector on ne peut se limiter qu'à remplir progessivement cet espace, en espérant que les programmes soient assez petits pour ne pas dépasser cette capacité.

Lors de l'allocation d'un objet, 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. Par exemple pour une chaîne de caractères, on alloue plus de mémoire que nécessaire pour que la fin de l'objet soit alignée sur un mot.

La manoeuvre pour réserver de l'espace se limite à incrémenter le pointeur d'allocation, usuellement dans un régistre. Cela laisse place à l'implémentation d'un garbage collector du genre stop-and-copy, qui parcours la mémoire à la recherche d'objets inutilisés une fois que le pointeur d'allocation dépasse une certaine borne.

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 (voir section sur l'encodage des objets), ou de 2 pour les paires (idem).

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. Un des premiers objets utiles à implanter sont les 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.

Il aurait été possible de tout simplement allouer des paires alignées sur des adresses qui se terminent par 10_2, mais cela aurait probablement été inefficace car les processeurs modernes sont concus pour manipulées des valeurs alignées sur les mots, soit 8 octets dans le cas de x86-64.

Objets en général. 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.

Chaînes de caractères. Comme mentionnée précédemment elles sont allouées dans la plage de données d'un objet, et se terminent par un octet nul (car l'espace alloué est généralement plus grand que la chaîne afin de respecter l'alignement sur les mots)

Symboles. Les symboles sont des objets dont la plage de données est une chaîne de caractères, mais dont le type est différent de celui de ces dernières. De plus ils sont gardés dans un liste d'association (au runtime) afin de conserver leur unicité.

Caractères. Les caractères uniques sont aussi encodés en mémoire, mais il aurait fallu les encoder comme valeur spéciale pour gagner en temps et en vitesse.

Liste vide. La liste vide est représentée par un symbole non-interné, mais il aurait aussi fallu l'encoder comme une valeur spéciale puisque c'en est un autre qui revient souvent.

Procédures et fermetures. Les procédures sont représentées par des objets, ayant comme premier mot de donnée l'adresse de la lambda expression compilée. Les fermetures contiennent en plus la valeur des variables libres lors de la création de la fermeture dans les champs suivants (voir section sur les fermetures).

Fermetures

Encodage des fermetures. Au runtime, les fermetures sont encodées telles que les variables libres auquelles elles font référence sont stockées dans l'objet fermeture, qui est passé dans un registre donné au moment de l'appel de la lambda expression, pour que celle-ci puisse aller les chercher convenablement.

Compilation des fermetures. À la compilation on détecte l'ensemble des variables libres (free variables, FV) 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 au moment de la création dans les données de l'objet.

Tentative de conversion source-à-source. Le problème de la méthode précédente est qu'elle ne permet réellement la modification des valeurs. Pour pallier a ce problème nous avons essayé de faire la transformation source-à-source qui converti les variables libres vers des paires, et leur accès à des car/set-car !. Malheureusement nous n'avons pu implémenter cette solution correctement et avons du la laisser tomber momentanément.

Compilation just-in-time

Bien que non fonctionnelle pour le moment, une bonne partie du travail a été effectué pour permettre la compilation et l'exécution "just-in-time" du code Scheme.

Encodage des instructions. Les instructions sont encodées vers une représentation en tableau d'octets à partir de leur représentation en s-expression (qui est une des représentation intermédiaires du compilateur normal).

Encodage des sauts. Comme on ne connait pas d'avance l'adresse relative ou absolue de leur destination, les diverses instructions jumps 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 au saut et la placer au bon endroit dans le code compilé.

Encodage des instructions avec immédiats. En général, on prends de manière paresseuse l'encodage des instructions où il y a une valeur immédiate avec la plus grande taille d'immédiat possible. Cela simplifie le code et évite plusieurs problèmes, notemment le fait que (par exemple dans le cas d'un label) on ne connait pas forcément la valeur d'avance.

Exécution. Pour exécuter le tableau d'octets résultant de la compilation JIT, on alloue un espace de mémoire exécutable dans lequel on copie les instructions.

Idéalement, et dans le futur, un index des adresses réelles des variables globales déclarées dans chaque bloc compilé seraient gardés, pour pouvoir facilement y faire référence dans d'autres compilations JIT lors de la même exécution (par exemple pour faire un eval qui compile en utilisant le compilateur compilé).

Librairie standard

Un librairie standard existe maintenant et fournit les fonctions et les macros les plus communes de Scheme. Dans cette librairie, certaines fonctions sont écrites en assembleur soit pour des raisons de performances, soit parce que ce soit des fonctions de base qu'on doit écrire de cette manière, notemment les fonctions pour manipuler les objets (par exemple %object-new, %object-getq, etc).

La librairie standard permet entres autres de manipuler les paires (avec cons, car, cdr, caa..ddr), les listes (length, list-ref, ...) , les chaînes de caractères et les symboles (string->symbol, symbol->string), et permet aussi l'affichage de ces valeurs (display, print, ...), en plus des valeurs telles que les entiers et les booléens.