Implantation des fonctions

Le problème

Question:

Comment implanter les fonctions comme objets de première classe en C?

Hypothèses de travail:

- On suppose qu'on dispose d'un type C '`DATA`' capable de représenter n'importe quelle donnée Scheme (même une fonction, le cas échéant).
- On ignore les questions reliées à la gestion mémoire.
- On se contente de faire une traduction assez directe.

Tentative infructueuse

Créer une fonction C pour chaque λ-expression et utiliser son pointeur comme représentant. Ex:

En Scheme:

```
> (define inc (lambda (x) (+ x 1)))
> (inc 5)
```

En C:

```
DATA LAM1(DATA x)                        /* Fonctions equivalentes en C */
{
  return invoke2(plus, x, make_int(1));
}
...
inc = make_clos(LAM1);                          /* Programme principal */
invoke1(inc, make_int(5));
...
DATA invoke1(DATA f, DATA x)                  /* Fonctions utilitaires */
{
  return (*clos_ptr(f))(x);
}
DATA invoke2(DATA f, DATA x, DATA y)
{
  return (*clos_ptr(f))(x, y);
}
```

Fonctionne!

Tentative infructueuse

Ex:

En Scheme:

```
> (define comp
    (lambda (f g)
      (lambda (x)
        (f (g x)))))
```

En C:

```
DATA LAM2(DATA f, DATA g)              /* Fonctions equivalentes en C */
{
  return make_clos(LAM3);
}
DATA LAM3(DATA x)
{
  return invoke1(f, invoke1(g, x));
}
...
comp = make_clos(LAM2);                          /* Programme principal */
```

Note: d'où viennent 'f' et 'g' dans 'LAM3'? !

Tentative infructueuse

Explications:

- ‘`f`’ et ‘`g`’ ne sont pas accessibles depuis ‘`LAM3`’.
- ‘`LAM2`’ a nécessairement fini d’exécuter lorsque ‘`LAM3`’ est appelée, donc ‘`f`’ et ‘`g`’ ont cessé d’exister (en C) lorsqu’il serait temps de les relire.
- Rendre ‘`f`’ et ‘`g`’ globales ne suffirait pas à cause de multiples appels à ‘`comp`’, éventuellement.

Tentative infructueuse

Même lorsque les instances de variables nécessaires à une fonction existent, notre représentation ne fournit pas toujours un accès à celles-ci.

Ex: En Scheme:

```
> (define map1 (lambda (f lst) ...))
> (define add2all
    (lambda (n lst)
      (map1 (lambda (x) (+ x n)) lst)))
```

Note: La fonction interne et la variable 'n' doivent exister tant que l'appel à 'map1' est actif. Or, 'n' existe réellement pendant ce temps.

En C:

```
DATA LAM4(DATA f, DATA lst) { ... }   /* Fonctions equivalentes en C */
DATA LAM5(DATA n, DATA lst)
{
  return invoke2(map1, make_clos(LAM6), lst);
}
DATA LAM6(DATA x)
{
  return invoke2(plus, x, n);
}
...
map1 = make_clos(LAM4);                    /* Programme principal */
...
```

Problème! Pas de 'n'!

Tentative infructueuse

Reprenons le même exemple en supposant que C fournit les fonctions imbriquées comme en Pascal.

En C:

```
DATA LAM4(DATA f, DATA lst) { ... }   /* Fonctions equivalentes en C */
DATA LAM5(DATA n, DATA lst)
{
  DATA LAM6(DATA x)
    {
      return invoke2(plus, x, n);
    }
  return invoke2(map1, make_clos(LAM6), lst);
}
...
map1 = make_clos(LAM4);                    /* Programme principal */
...
```

OK!

Tentative infructueuse

Revue des 3 types de fonctions:

- sans variables à mémoriser (comme '`inc`');
- avec variables à mémoriser mais seulement durant la période où la fonction qui les a créées est encore en activation (comme '`add2all`');
- avec variables à mémoriser au-delà de l'activation de la fonction qui les a créées.

On peut identifier les fonctions du premier type. Il est impossible de distinguer les fonctions des deux derniers types, en général.

Solution générale: une fonction doit se *souvenir* des variables de son environnement.

Fermetures

La capture explique le nom de *fermetures* que l'on donne souvent aux fonctions ayant mémorisé certaines variables (et, par abus de langage, même à celles n'ayant rien mémorisé).

On peut implanter la représentation des fonctions grâce à une *transformation source à source* que l'on appelle *défonctionnalisation*.

Il s'agit de transformer les programmes comportant des fonctions qui ont des variables libres en des programmes qui utilisent des structures de données simples et des fonctions *sans* variables libres.

Note: les fermetures ne capturent que les variables "locales"; par opposition aux variables globales. Ces dernières peuvent devenir des variables globales en C.

Une *transformation source à source* consiste à transformer le code source en un code cible écrit dans le même langage. Le code cible est habituellement plus simple ou plus régulier.

Représentation chaînée

Nous pouvons transformer un programme de la manière suivante afin d'utiliser explicitement la *représentation chaînée* pour les fermetures.

Les formes suivantes sont transformées: les λ-expressions, les appels, les lectures de variables locales et les affectations aux variables locales.

Dans le code résultant, la variable locale `env` est toujours disponible et contient la liste des variables locales englobantes.

- `(lambda (x1 ... xN) <exp>)` devient `(make-closN (lambda (env) <exp'>) env)`
- `(<f> <exp1> ... <expN>)` devient `(clos-applyN <f'> <exp1'> ... <expN'>)`
- `x` devient `(list-ref env i)` où `x` est la $j$ème variable locale la plus proche et où `i` $= j - 1$
- `(set! x <exp>)` devient `(list-set! env i <exp'>)` où ...
- `x` devient `x` où `x` est une variable globale (similaire pour `(set! x <exp>)`)

On dit que l'environnement est chaîné car une fermeture provenant d'une λ-expression imbriquée capture un environnement (une liste) qui est une extension de l'environnement capturé par la fermeture immédiatement englobante et ainsi de suite.

Représentation chaînée

Le code transformé nécessite quelques services auxiliaires.

- On suppose qu'on dispose d'une famille de nouveaux types d'objets à deux champs: `clos0`, `clos1`, `clos2`, ...
- Le numéro associé à ces objets représente l'*arité* de la fermeture ainsi créée, i.e. le nombre d'arguments que peut recevoir la fermeture.
- On construit une fermeture avec `(make-closN <lambda-exp> <env-exp>)`.
- On teste si on a une fermeture avec `(clos? <exp>)` ou, plus spécifiquement, avec `(closN? <exp>)`.
- On extrait le contenu d'une fermeture avec `(clos-code <exp>)` et `(clos-env <exp>)`.
- Une fonction d'application, disons `clos-applyN`, peut être définie ainsi:

```
(lambda (f x1 ... xN)
  (if (closN? f)
    ((clos-code f)
     (cons x1 (... (cons xN (clos-env f))...)))
    (error <exp>)))
```

Représentation chaînée

Regardons le code de nos exemples suite à la transformation source à source.

```
(define inc (lambda (x) (+ x 1)))
(inc 5)
```

devient:

```
(define inc
  (make-clos1 (lambda (env)
                (clos-apply2 +
                  (list-ref env 0) 1))
              env))
(clos-apply1 inc 5)
```

Représentation chaînée

Suite...

```
(define comp
  (lambda (f g)
    (lambda (x)
      (f (g x)))))
```

devient:

```
(define comp
  (make-clos2 (lambda (env)
                (make-clos1 (lambda (env)
                              (clos-apply1 (list-ref env 1)
                                (clos-apply1 (list-ref env 2)
                                  (list-ref env 0))))
                            env))
              env))
```

Représentation chaînée

Suite...

```
(define map1 (lambda (f lst) ...))
(define add2all
  (lambda (n lst)
    (map1 (lambda (x) (+ x n)) lst)))
```

devient:

```
(define map1 (make-clos2 (lambda (env) ...) env))
(define add2all
  (make-clos2 (lambda (env)
                (clos-apply2 map1
                  (make-clos1 (lambda (env)
                                (clos-apply2 +
                                  (list-ref env 0)
                                  (list-ref env 1)))
                              env)
                  (list-ref env 1)))
              env))
```

Représentation chaînée

La représentation chaînée est une solution générale à notre problème de représentation des fonctions.

Avantages et inconvénients:

+ Fermetures faciles à construire.
+ Nécessite des fonctions comme celles qui sont disponibles en C, i.e. sans variables libres.
– Accès inefficace aux variables de l'environnement.
– Rétention de variables inutiles dans l'environnement, donc rétention de données inutiles, donc surcharge de travail pour le GC.

Représentation plate

Une représentation alternative est celle des **fermetures plates**.

Au lieu d'employer des objets de taille fixe qui contiennent (la liste de) l'environnement, on emploie des objets de taille variable qui contiennent directement les variables.

On utilise à nouveau une transformation source à source et quelques services auxiliaires.

Représentation plate

La transformation source à source est décrite comme suit.

À nouveau, les formes suivantes sont transformées: les λ-expressions, les appels, les lectures de variables locales. La question des affectations aux variables locales est plus délicate.

- `(lambda (x1 ... xN) <exp>)` devient
  `(make-closN-M (lambda (clos x1 ... xN) <exp'>)`
  `<exp1> ... <expM>)`
  où les expressions `<exp1>` à `<expM>` servent à récupérer les variables libres locales de la λ-expression originale
- `(<f> <exp1> ... <expN>)` devient `(clos-applyN <f'> <exp1'> ... <expN'>)`
- `x` devient `(clos-varI clos)` où `x` est la `I`ème variable libre capturée
- `x` devient $\underline{\texttt{x}}$ si c'est un paramètre de la λ-expression immédiatement englobante
- `x` devient $\underline{\texttt{x}}$ si c'est une variable globale

On dit que l'environnement est plat car la représentation d'une fermeture contient immédiatement toutes les variables d'environnement qui sont capturées.

Représentation plate

Voici les services auxiliaires nécessaires:

- On suppose à nouveau qu'on dispose d'une famille de nouveaux types d'objets. Ces types sont les `closN-M` où `N` indique l'arité et `M`, le nombre de variables capturées.
- On construit une fermeture avec `(make-closN-M <lambda-exp> <exp1> ... <expM>)`.
- On teste si on a une fermeture avec `(clos? <exp>)` ou, plus spécifiquement, avec `(closN? <exp>)` ou `(closN-M? <exp>)`.
- On extrait le contenu d'une fermeture avec `(clos-code <exp>)` et `(clos-varI <exp>)`.
- Une fonction d'application, disons `clos-applyN`, peut être définie ainsi:

```
(lambda (f x1 ... xN)
  (if (closN? f)
    ((clos-code f) f x1 ... xN)
    (error <exp>)))
```

Représentation plate

Exemple de 'add2all':

```
(define map1 (lambda (f lst) ...))
(define add2all
  (lambda (n lst)
    (map1 (lambda (x) (+ x n)) lst)))
```

devient:

```
(define map1 (make-clos2-0 (lambda (clos f lst)
                             ...)))
(define add2all
  (make-clos2-0 (lambda (clos n lst)
                  (clos-apply2 map1
                    (make-clos1-1 (lambda (clos x)
                                    (clos-apply2 +
                                      x
                                      (clos-var1 clos)))
                                  n)
                    lst))))
```

Représentation plate

La représentation par fermetures plates est aussi une solution générale.

Avantages et inconvénients:

+ Accès rapide aux variables de l'environnement.
+ Ne capture que les variables utiles (les variables libres).
– Construction plus coûteuse lorsqu'il y a plusieurs variables à capturer.

Représentation plate

La gestion des variables locales mutables est un cas spécial.

Supposons que nous ayons `(lambda (x y z) <exp>)` et que `y` soit mutable, i.e. qu'il y a une expression `(set! y <exp2>)` dans `<exp>`.

Il faut alors rendre la variable `y` non-mutable en plaçant la valeur normalement contenue par `y` dans une "boîte".

On remplace l'expression originale par:

```
(lambda (x y z)
  (let ((y (make-box y)))
    <exp'>))
```

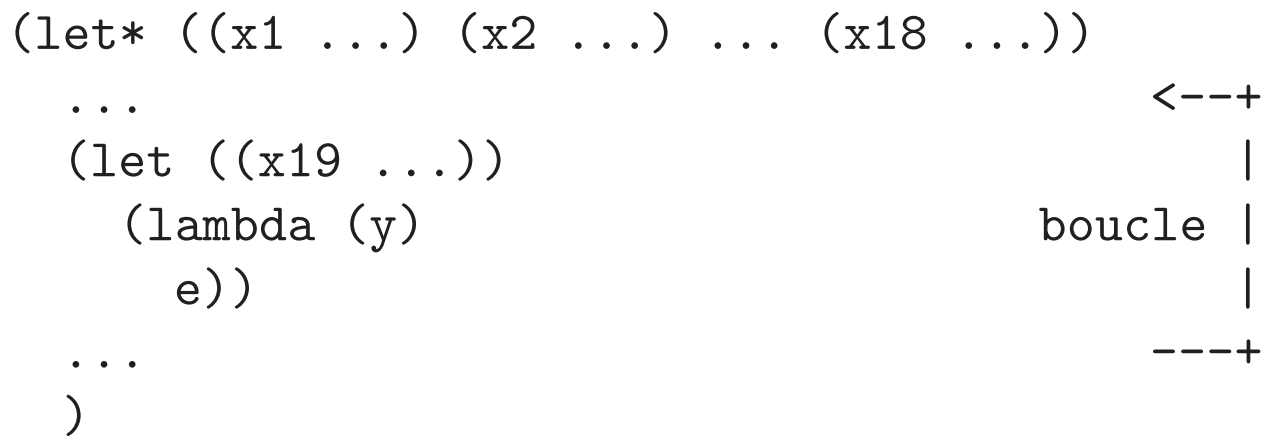
En remplaçant `<exp>` par `<exp'>`, toute lecture de `y` devient `(box-ref y)` et toute écriture `(set! y <exp2>)` devient `(box-set y <exp2>)`.

Cette élimination des variables mutables doit être faite avant la défonctionnalisation.

Exemple: `get-inc`

Comparaison entre les représentations

Soit le code suivant où ‘e’ référence les variables ‘x1’, ‘x2’, . . . ‘x19’:



1. La représentation plate est clairement plus rapide si la boucle se répète peu et que ‘e’ effectue de nombreuses références aux variables mémorisées.

2. La représentation chaînée est clairement plus rapide si la boucle se répète intensivement et que ‘e’ lit peu les variables mémorisées.

3. Il n’est pas clair de départager les deux représentations dans des situations intermédiaires.

Considérations diverses

Il existe une multitude de représentations pour les fermetures. Par exemple, des hybrides entre les deux représentations décrites.

Quels critères sont importants pour nous?

- Une représentation optimisant la vitesse?
- Une représentation optimisant l'espace?
- Une représentation qui évite la capture de variables inutiles?

Effets de la capture de variables inutiles

Ex:

```
(let loop ((sum 0) (f (lambda (x) x)))
  (let ((obj (read)))
    (cond ((eof-object? obj)
           (f sum))
          ((< obj 0)
           (loop sum (lambda (x) (- x))))
          ((= obj 0)
           (loop (+ sum 1) f))
          (else
           (loop sum (lambda (x) x))))))
```

Cette boucle consomme apparemment un espace constant. Mais ce n'est pas le cas si les fonctions capturent des variables inutilement. Auquel cas cette boucle pourrait consommer un espace qui croît linéairement avec le nombre d'itérations.

On dit qu'une implantation est *safe for space* si les fermetures (et les continuations) ne capturent pas de variables inutilement.

