IFT3065/IFT6232 Compilation

Environnements

Environnement lexical (1)

- Un **environnement** associe des valeurs à des noms
- Étant donné un nom et un environnement on peut retrouver la valeur associée (**“lookup”**)
- Nous allons étudier les environnements dans le contexte de l’interprétation et compilation de ce sous-ensemble de Scheme:

$$\begin{array}{lcl} \langle \textit{expr} \rangle & ::= & C \\ & | & V \\ & | & \texttt{(lambda (}V\texttt{...)} \quad \langle \textit{expr} \rangle\texttt{)} \\ & | & \texttt{(}\langle \textit{expr} \rangle \ \langle \textit{expr} \rangle\texttt{...)} \end{array}$$

`cons`, `+`, etc sont des variables prédéfinies

Environnement lexical (2)

- Dans un premier temps, simplifions encore plus le sous-ensemble pour retirer les lambda-expressions:

$$\begin{array}{lll} \langle expr \rangle & ::= & C \\ & | & V \\ & | & (\langle expr \rangle \ \ \langle expr \rangle \ldots) \end{array}$$

Environnement lexical (3)

- En Scheme, la **liste d'association** est une représentation simple pour les environnements:

```
(define genv                    ;; environnement global
  (list (cons 'cons cons)
        (cons 'car car)
        (cons 'cdr cdr)
        (cons '+ +)))

(define (env-lookup env var)
  (let ((x (assq var env)))
    (if x (cdr x) (error "unbound" var))))



(env-lookup genv 'cons) => #<procedure cons>

(env-lookup genv 'x)    => erreur: unbound x
```

Environnement lexical (4)


```
(define (ev expr)  ;; interprète sans lambda
  (match expr

    (,const when (constant? const)
     const)

    (,var when (variable? var)
     (env-lookup genv var))

    ((,fun . ,exprs)
     (apply (ev fun)
            (map (lambda (x) (ev x))
                 exprs)))))

(define (constant? x)
  (or (number? x) (string? x) (boolean? x)))

(define (variable? x)
  (symbol? x))

(define (eval expr)
  (ev expr))


(eval '(cons 1 2)) => (1 . 2)
```

Environnement lexical (5)

- Pour interpréter les lambda-expressions, il faut une fonction pour **étendre** un environnement avec des nouvelles associations de noms et valeurs:

```
(define (env-extend env vars vals)
  (append (map cons vars vals) env))


(env-lookup genv 'cons) => #<procedure cons>

(env-lookup genv 'j) => erreur: unbound j

(define e (env-extend genv '(j cons) '(1 2)))

(env-lookup e 'j) => 1

(env-lookup e 'cons) => 2
```

Environnement lexical (6)


```
(define (ev expr env)  ;; interprète avec env
  (match expr

    (,const when (constant? const)
     const)

    (,var when (variable? var)
     (env-lookup env var))

    ((lambda ,params ,body)
     (lambda args (ev body (env-extend env params args))))

    ((,fun . ,exprs)
     (apply (ev fun env)
            (map (lambda (x) (ev x env))
                 exprs)))))

(define (eval expr)
  (ev expr genv))


(eval '((lambda (x y) (cons y x)) 1 2)) => (2 . 1)
```

Environnement lexical (7)

- Alternative: maintenir séparément un **environnement de compilation** et un **environnement d'exécution**

```
(define gcte (list 'cons 'car 'cdr '+))
(define grte (list  cons  car  cdr  +))

(define (cte-extend cte vars) (append vars cte))
(define (rte-extend rte vals) (append vals rte))

(define (cte-lookup cte var)
  (let ((x (memq var cte)))
    (if x
        (- (length cte) (length x))
        (error "unbound" var))))

(define (rte-lookup rte pos)
  (list-ref rte pos))


(define pos (cte-lookup gcte '+))
(rte-lookup grte pos) => #<procedure +>
```

Environnement lexical (8)


```
(define (ev expr cte rte)  ;; interprète avec cte/rte
  (match expr

    (,const when (constant? const)
     const)

    (,var when (variable? var)
     (let ((pos (cte-lookup cte var)))
       (rte-lookup rte pos)))

    ((lambda ,params ,body)
     (lambda args (ev body (cte-extend cte params)
                           (rte-extend rte args))))

    ((,fun . ,exprs)
     (apply (ev fun cte rte)
            (map (lambda (x) (ev x cte rte))
                 exprs)))))

(define (eval expr)
  (ev expr gcte grte))
```

Environnement lexical (9)


```
(define (ev expr cte)  ;; interprète currifié
  (lambda (rte)
    (match expr

      (,const when (constant? const)
       const)

      (,var when (variable? var)
       (let ((pos (cte-lookup cte var)))
         (rte-lookup rte pos)))

      ((lambda ,params ,body)
       (lambda args ((ev body (cte-extend cte params))
                     (rte-extend rte args))))

      ((,fun . ,exprs)
       (apply ((ev fun cte) rte)
              (map (lambda (x) ((ev x cte) rte))
                   exprs))))))

(define (eval expr)
  ((ev expr gcte) grte))
```

Environnement lexical (10)


```
(define (ev expr cte)  ;; interprète rapide
  (match expr

    (,const when (constant? const)
     (lambda (rte) const))

    (,var when (variable? var)
     (let ((pos (cte-lookup cte var)))
       (lambda (rte) (rte-lookup rte pos))))

    ((lambda ,params ,body)
     (let ((b (ev body (cte-extend cte params))))
       (lambda (rte)
         (lambda args (b (rte-extend rte args))))))

    ((,fun . ,exprs)
     (let ((f (ev fun cte))
           (es (map (lambda (x) (ev x cte)) exprs)))
       (lambda (rte)
         (apply (f rte)
                (map (lambda (e) (e rte)) es))))))

(define (eval expr)
  ((ev expr gcte) grte))
```

Environnement lexical (11)


```
(define const1 (lambda (rte) 1))
(define const2 (lambda (rte) 2))
(define pos0 (lambda (rte) (car rte)))
(define pos1 (lambda (rte) (cadr rte)))

(define (ev expr cte)  ;; interprète rapide optimisé
  (match expr

    (,const when (constant? const)
     (case const
       ((1) const1)
       ((2) const2)
       (else
        (lambda (rte) const))))

    (,var when (variable? var)
     (let ((pos (cte-lookup cte var)))
       (case pos
         ((0) pos0)
         ((1) pos1)
         (else
          (lambda (rte) (rte-lookup rte pos)))))))

    ...))
```

Environnement lexical (12)


```
;; Environnements chaines:

(define gcte (list (list  'cons 'car 'cdr '+)))
(define grte (list (vector cons  car  cdr  +)))

(define (cte-extend cte vars) (cons vars cte))
(define (rte-extend rte vals) (cons (list->vector vals) rte

(define (cte-lookup cte var)
  (let loop ((cte cte) (up 0))
    (if (null? cte)
        (error "unbound" var)
        (let ((x (memq var (car cte))))
          (if x
              (cons up (- (length (car cte)) (length x)))
              (loop (cdr cte) (+ up 1)))))))

(define (rte-lookup rte pos)
  (vector-ref (list-ref rte (car pos)) (cdr pos)))
```

Conversion de fermeture (1)

- Caractéristiques pertinentes de Scheme :
  - Portée lexicale
  - Étendue indéfinie des fonctions (une instance de variable peut survivre l'activation de fonction qui l'a créée)

- Solutions :
  - **Fermetures** : mémorisent l'environnement du contexte de création de la fonction
  - **Garbage collector** : pour récupérer les environnements qui ne sont plus utiles au calcul

Conversion de fermeture (2)

- Exemple de fermetures :

```
(define (keep f lst)
  (cond ((null? lst)
         '())
        ((f (car lst))
         (cons (car lst) (keep f (cdr lst))))
        (else
         (keep f (cdr lst)))))

(define curry2
  (lambda (f)
    (lambda (x)
      (lambda (y) (f x y)))))

(define swap2
  (lambda (g)
    (lambda (x y) (g y x))))

(define gt (curry2 (swap2 >)))

(keep (gt 3) '(3 1 4 1 5)) => (4 5)
```

Conversion de fermeture (3)

- Dans une expression E, une référence à une variable V est :
  - **liée** (“bound”) si V est déclarée dans E
  - **libre** (“free”) si V n’est pas déclarée dans E

- Exemple : dans `(lambda (y) (+ x y))`
  - `x` et `+` sont libres
  - `y` est liée

- Les variables libres de E sont donc les variables dans l’environnement d’évaluation de E qui “paramétrisent” l’expression E

Conversion de fermeture (4)

- Analyse des variables libres pour le noyaux de Scheme :

$$FV(\ c\ ) = \{\}$$

$$FV(\ v\ ) = \{v\}$$

$$FV(\ \texttt{(lambda (}p_1 \ldots\texttt{)}\ E\texttt{)}\ ) = FV(E) \setminus \{p_1, \ldots\}$$

$$FV(\ \texttt{(}E_0\ E_1 \ldots\texttt{)}\ ) = FV(E_0) \cup FV(E_1) \cup \ldots$$

- Exemple :

$$FV(\ \texttt{(lambda (y) (+ x y))}\ ) =$$

$$FV(\ \texttt{(+ x y)}\ ) \setminus \{\texttt{y}\} =$$

$$FV(\ \texttt{+}\ ) \cup FV(\ \texttt{x}\ ) \cup FV(\ \texttt{y}\ ) \setminus \{\texttt{y}\} =$$

$$\{\texttt{+}, \texttt{x}, \texttt{y}\} \setminus \{\texttt{y}\} = \{\texttt{+}, \texttt{x}\}$$

Conversion de fermeture (5)


```
(define (fv expr)
  (match expr

    (,const when (constant? const)
     '())

    (,var when (variable? var)
     (list var))

    ((set! ,var ,E1)
     (union (list var) (fv E1)))

    ((if ,E1 ,E2)
     (union (fv E1) (fv E2)))
    ((if ,E1 ,E2 ,E3)
     (union (fv E1) (union (fv E2) (fv E3))))

    ((lambda ,params ,E)
     (difference (fv E) params))

    ...))
```

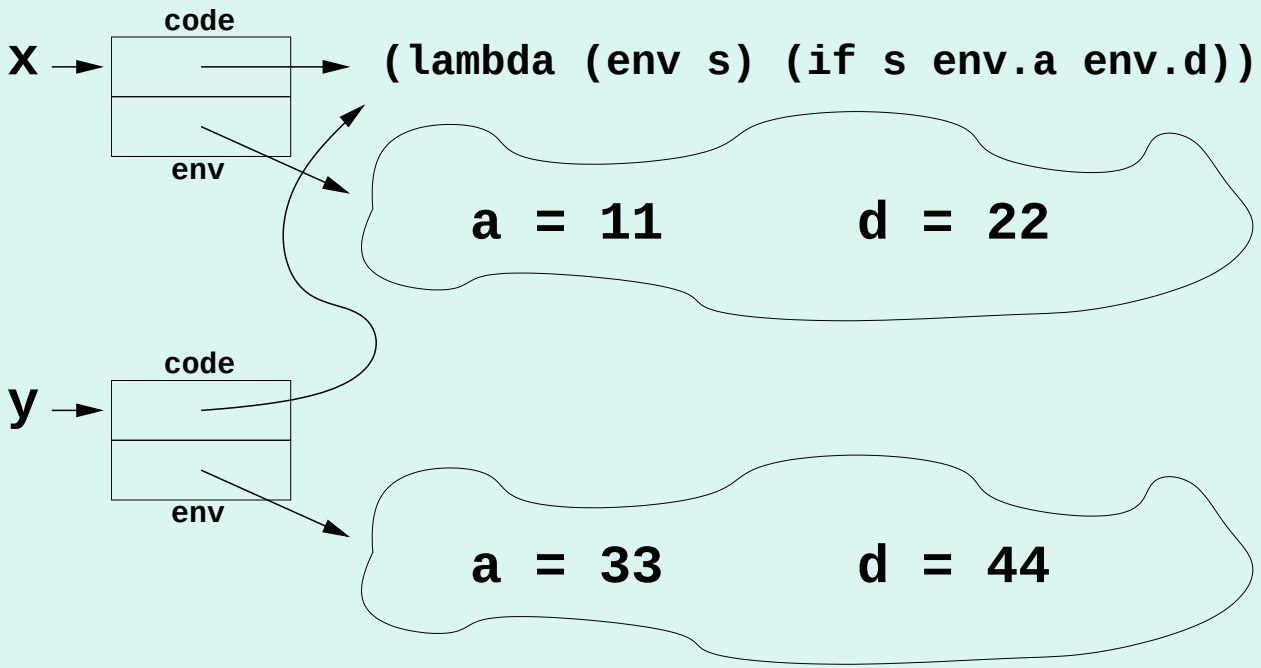
Conversion de fermeture (6)

- Soit une lambda-expression E dont l'évaluation donne une fermeture F
- $FV(E)$ sont les variables de l'environnement d'évaluation de E qui sont (possiblement) utilisées pour éxécuter le corps de F
- Exemple :

```
(define (kons a d) (lambda (s) (if s a d)))
(define (kar p) (p #t))
(define (kdr p) (p #f))

; FV( (lambda (s) (if s a d)) ) = {a,d}

(define x (kons 11 22))
(define y (kons 33 44))

(kar x) => 11
(kar y) => 33
(kdr y) => 44
```

Conversion de fermeture (7)

- Représentation typique des fermetures : objet qui contient un **pointeur vers le code exécutable** de la fermeture et l'**environnement** qui contient au moins les variables libres




```
(p #t) -> (p.code p.env #t)
```

Conversion de fermeture (8)

- Représentation typique des environnements :
  - **Chaîne lexicale** : environnement = liste des blocs d'activation englobants
  - **"Display"** : environnement = vecteur des blocs d'activation englobants
  - **Fermeture plate** : stocker les variables libres directement dans la fermeture

Conversion de fermeture (9)

- Pour expliquer l'implantation tout en restant en Scheme, on va supposer une transformation de code qui **crée explicitement les blocs d'activation** des fonctions

  1. `(f X Y Z)` $\rightarrow$ `(f (vector X Y Z))`
  2. `(lambda (...`p_i`...) ...`p_i`...)` $\rightarrow$
     `(lambda (`a`) ...(vector-ref `a $i-1$`)...)`

- Le compilateur doit se souvenir dans quel bloc d'activation (a) une variable p_i est déclarée en plus de sa position dans le bloc (i)

Conversion de fermeture (10)


```
(lambda (a0 a1 a2)
  ...
    (lambda (b0 b1)
      ...
        (lambda (c0 c1 c2)
          ...
           (lambda (x) (+ x c0 b1 a2))
          ...)
      ...)
  ...)
```

devient

```
(lambda (a)
  ...
    (lambda (b)
      ...
        (lambda (c)
          ...
           (lambda (d) (+ (vector-ref d 0)
                          (vector-ref c 0)
                          (vector-ref b 1)
                          (vector-ref a 2)))
          ...)
      ...)
  ...)
```

Conversion de fermeture (11)

- Cette façon de faire nous rapproche du langage machine
- En effet, puisque toutes les fonctions ont toujours un seul paramètre (le bloc d'activation courant) on peut utiliser **un registre machine** pour implanter le protocole d'appel de fonction

Conversion de fermeture (12)

- L'approche par chaîne lexicale représente l'environnement comme une **liste de blocs d'activation**
- Chaque fermeture mémorise l'environnement du **contexte d'évaluation** de la lambda-expression

Conversion de fermeture (13)

- Le code de la fermeture reçoit l'environnement en paramètre

```
(define (make-closure code env) (vector code env))
(define (closure-code clo) (vector-ref clo 0))
(define (closure-env clo) (vector-ref clo 1))

(lambda (a) ...) -> (make-closure
                     (lambda (env a) ...)
                     ENV)

(f (vector X Y)) -> ((closure-code f)
                     (closure-env f)  ;; -> env
                     (vector X Y))    ;; -> a
```

Conversion de fermeture (14)

- À noter que maintenant les fonctions ont toujours 2 paramètres
- On peut donc maintenant utiliser 2 registres machine pour implanter le protocole d'appel de fonction (un pour l'**environnement** et l'autre pour le **bloc d'activation courant**)

Conversion de fermeture (15)


```
(make-closure
 (lambda (env a)
   ...
     (make-closure
      (lambda (env b)
        ...
          (make-closure
           (lambda (env c)
             ...
             (make-closure
              (lambda (env d)
                (+ (vector-ref d 0)
                   (vector-ref (car env) 0)
                   (vector-ref (cadr env) 1)
                   (vector-ref (caddr env) 2)))
              (cons c env))
             ...)
           (cons b env))
        ...)
      (cons a env))
   ...)
 env-global)
```

Conversion de fermeture (16)

- Le temps d'accès à une variable dépend de sa profondeur dans la chaîne (pire cas = variables les plus globales)
- L'approche par "display" représente l'environnement comme un **vecteur des blocs d'activation**
- Le temps d'accès à toute variable est constant
- Cependant le temps de création des environnements n'est plus constant (pire cas = lambda-expression très imbriquée)

Conversion de fermeture (17)


```
(make-closure
 (lambda (env a)
   ...
    (make-closure
     (lambda (env b)
       ...
        (make-closure
         (lambda (env c)
           ...
           (make-closure
            (lambda (env d)
              (+ (vector-ref d 0)
                 (vector-ref (vector-ref env 0) 0)
                 (vector-ref (vector-ref env 1) 1)
                 (vector-ref (vector-ref env 2) 2)))
            (vector c
                    (vector-ref env 0)
                    (vector-ref env 1)
                    (vector-ref env 2)))
           ...)
         (vector b (vector-ref env 0) (vector-ref env 1)))
       ...)
     (vector a (vector-ref env 0)))
   ...)
 (vector env-global))
```

Conversion de fermeture (18)

- Exemple de la fonction `curry2` :

```
(define curry2
  (lambda (f)
    (lambda (x)
      (lambda (y) (f x y)))))

devient...
```

Conversion de fermeture (19)


```
(define curry2
  (make-closure
   (lambda (env a)
     (make-closure
      (lambda (env b)
        (make-closure
         (lambda (env c)
           (let ((f (vector-ref (vector-ref env 1) 0)))
             ((closure-code f)
              (closure-env f)
              (vector (vector-ref (vector-ref env 0) 0)
                      (vector-ref c 0)))))
         (vector b (vector-ref env 0) (vector-ref env 1))))
      (vector a (vector-ref env 0))))
   (vector env-global)))
```

Conversion de fermeture (20)

- Les environnements **chainés** et avec **“display”** sont similaires : l’environnement contient l’ensemble des blocs d’activation des lambda-expressions englobantes

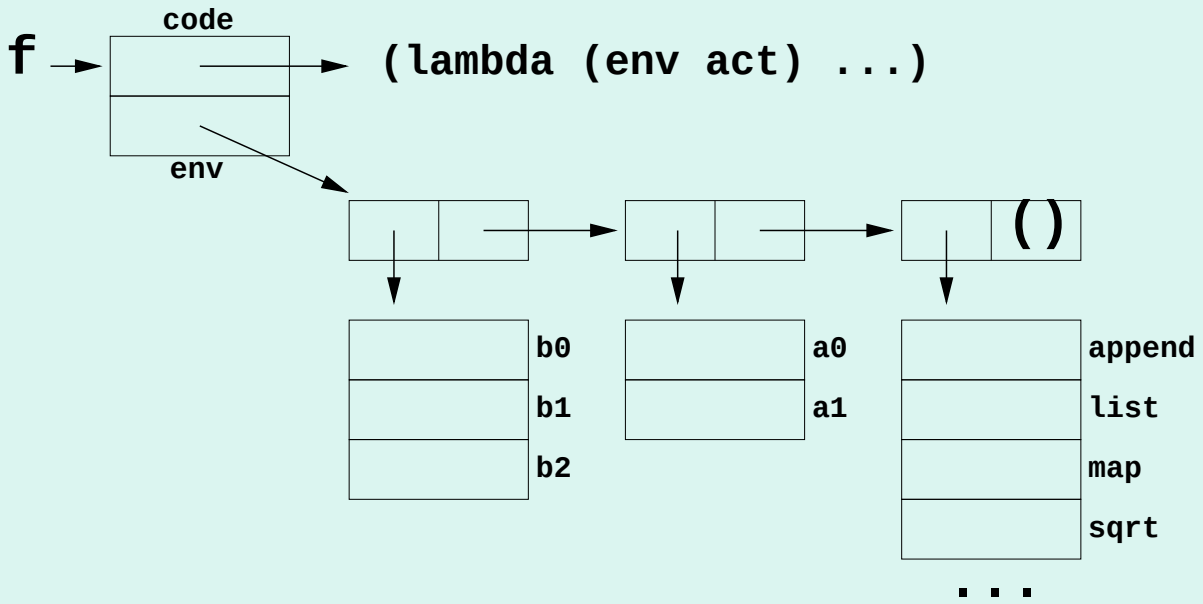
- Soit le code source suivant :

```
(lambda (a0 a1)
  ...
    (lambda (b0 b1 b2)
      ...
        (let ((f (lambda (c0)
                   (list c0 b1 b2))))
          ...)
      ...)
  ...)
```

- Quelle est la valeur de `f` avec chaque représentation?

Conversion de fermeture (21)

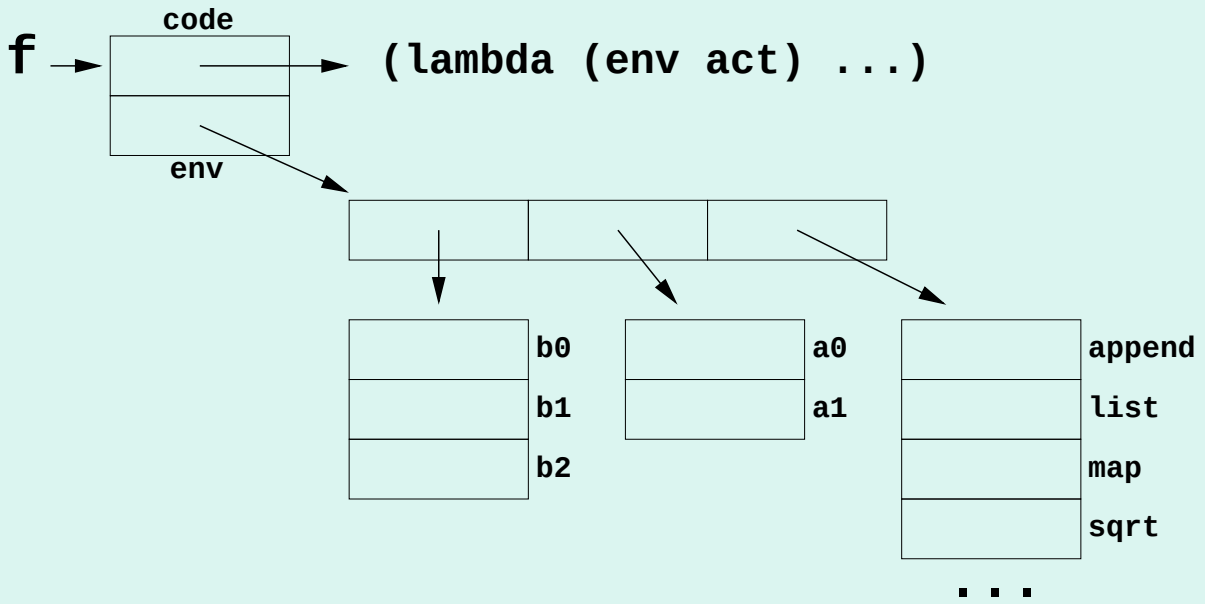
- Fermeture avec environnement chaîné :




```
(lambda ...) -> (make-closure
                  (lambda (env act) ...)
                  (cons act env))

(F X Y Z)    -> (let ((clo F))
                  ((closure-code clo)
                   (closure-env clo)
                   (vector X Y Z)))

(define env '())
(define act (vector append list ...))
```

Conversion de fermeture (22)

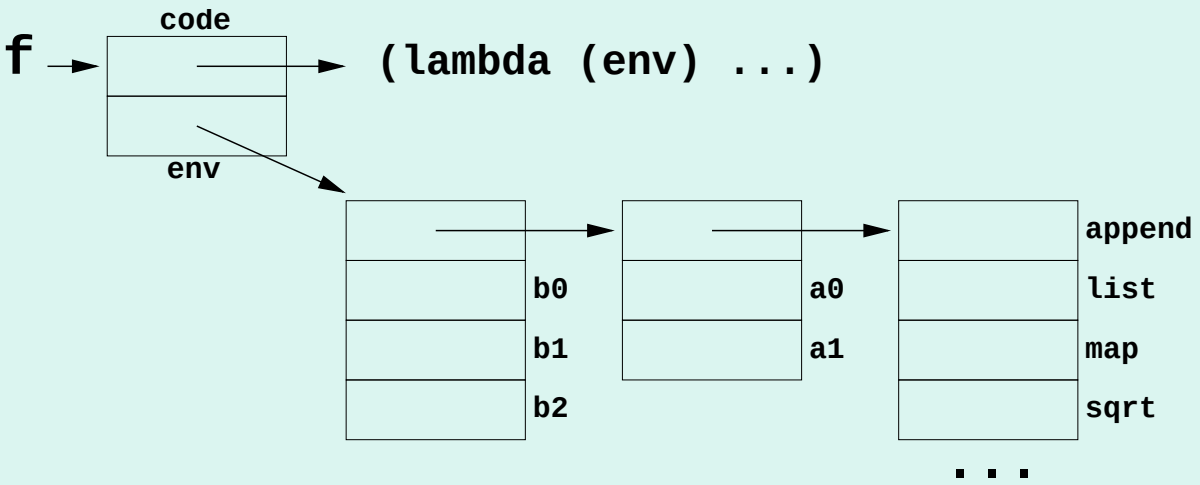
- Fermeture avec environnement “display” :




```
(lambda ...) -> (make-closure
                  (lambda (env act) ...)
                  (vector act
                          (vector-ref env 0)
                          (vector-ref env 1)
                          ...))

(define env '#())
(define act (vector append list ...))
```

Conversion de fermeture (23)

- Peut-on améliorer la représentation chaînée?
- Oui : faire le chaînage **à même les blocs d'activation**



- L'**appelant** construit l'environnement de la fonction appelée en allouant un bloc d'activation ("frame") qui contient les **paramètres** et l'**environnement de la fermeture**

Conversion de fermeture (24)

- L'appel `(F X Y Z)` se traduit par

```
(let ((clo F))
  ((closure-code clo)
   (vector (closure-env clo) X Y Z)))
```

- La lambda-expression `(lambda ...)` se traduit par

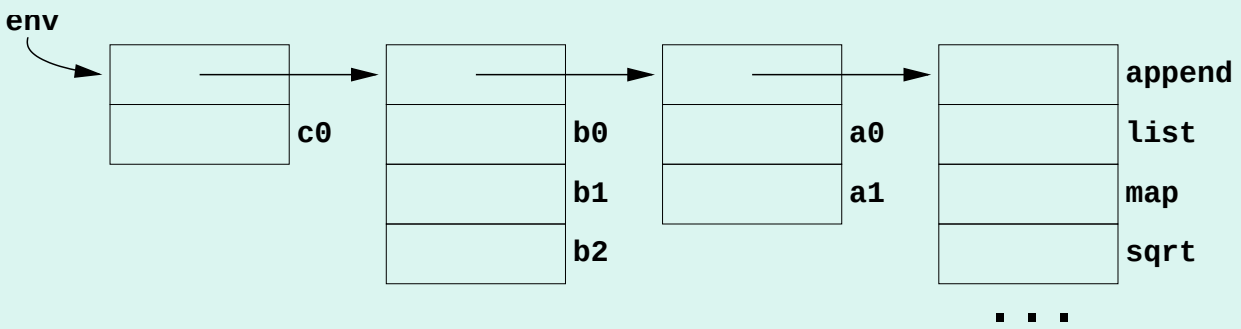
```
(make-closure (lambda (env) ...)
              env)
```

- Globalement on a

```
(define env (vector append list ...))
```

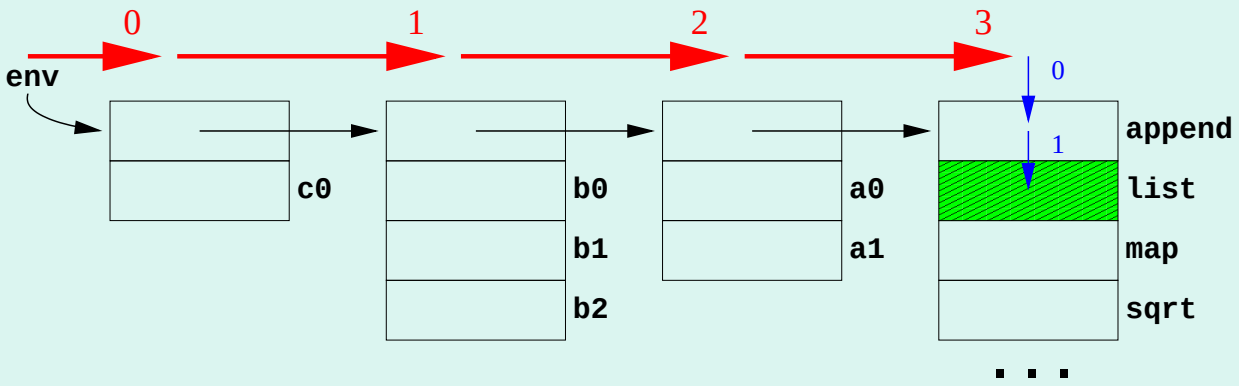
Conversion de fermeture (25)

- Dans le corps de la fonction appelée `env` réfère au début de la chaîne d'environnement, donc le bloc d'activation de la fonction
- Par exemple, dans le corps de la fonction "code" de `f` :



Conversion de fermeture (26)

- Chaque variable dans l'environnement a une coordonnée *up* et *over*
- *up* indique le nombre de liens de chaînage qu'il faut traverser pour se rendre au bloc d'activation qui contient la variable
- *over* c'est l'index de la variable dans le bloc d'activation
- Par exemple, pour `list` : *up*=**3**, *over*=**1**



Conversion de fermeture (27)

- Lors de la traduction il faut maintenir un **environnement de compilation** qui décrit la forme qu'aura l'environnement à l'exécution

- Cet environnement de compilation permet d'obtenir la coordonnée *up/over* pour toute variable lexicale

- Nous utiliserons des listes de listes d'identificateurs (et `#f` = lien) :

```
((#f c0)
 (#f b0 b1 b2)
 (#f a0 a1)
 (append list map ...))
```

Conversion de fermeture (28)

- La **conversion de fermeture** (“closure conversion”) c’est la procédure de traduction d’une expression E en une expression E' équivalente où
  - E contient (possiblement) des lambda-expressions avec des variables libres
  - E' contient seulement des lambda-expressions sans variables libres

- Donc, dans E' les lambda-expressions peuvent s’implanter simplement comme un **pointeur vers du code machine** (ou un **index dans une table de fonctions**)

Conversion de fermeture (29)


```
(define gcte
  '((append list map ...)))

(define (cte-lookup cte var)
  (let loop ((cte cte) (up 0))
    (if (null? cte)
        (error "unbound" var)
        (let ((x (memq var (car cte))))
          (if x
              (cons up
                    (- (length (car cte))
                       (length x)))
              (loop (cdr cte)
                    (+ up 1)))))))

(define (closure-conv expr)
  (closurec expr gcte))
```

Conversion de fermeture (30)


```
(define (closurec expr cte)

  (define (cc expr)
    (closurec expr cte))

  (define (get-frame up)
    (if (= up 0)
        `env
        `(vector-ref ,(get-frame (- up 1)) 0)))

  (match expr

    (,const when (constant? const)
     expr)

    (,var when (variable? var)
     (let* ((p (cte-lookup cte var))
            (up (car p))
            (over (cdr p)))
       `(vector-ref ,(get-frame up)
                    ,over)))
```

Conversion de fermeture (31)


```
((set! ,var ,E1)
 (let* ((p (cte-lookup cte var))
        (up (car p))
        (over (cdr p)))
   `(vector-set! ,(get-frame up)
                 ,over
                 ,(cc E1))))

((lambda ,params ,E)
 (let ((new-cte
        (cons (cons #f params)
              cte)))
   `(make-closure
     (lambda (env)
       ,(closurec E new-cte))
     env)))
```

Conversion de fermeture (32)


```
((if ,E1 ,E2)
 `(if ,(cc E1) ,(cc E2)))
((if ,E1 ,E2 ,E3)
 `(if ,(cc E1) ,(cc E2) ,(cc E3)))

((,E0 . ,Es)
 (if (primitive? E0)
     `(,E0
       ,@(map cc Es))
     `(let ((clo ,(cc E0)))
        ((closure-code clo)
         (vector (closure-env clo)
                 ,@(map cc Es))))))

(,_
 (error "unknown expression" expr))))
```

Conversion de fermeture (33)

- Exemple de traduction :

```
> (closure-conv
    '(lambda (x) (lambda (y) (list x y))))

(make-closure
 (lambda (env)
   (make-closure
    (lambda (env)
      (let ((clo
             (vector-ref
               (vector-ref (vector-ref env 0)
                           0)
               1)))
        ((closure-code clo)
         (vector (closure-env clo)
                 (vector-ref (vector-ref env 0)
                             1)
                 (vector-ref env 1)))))
    env))
 env)
```

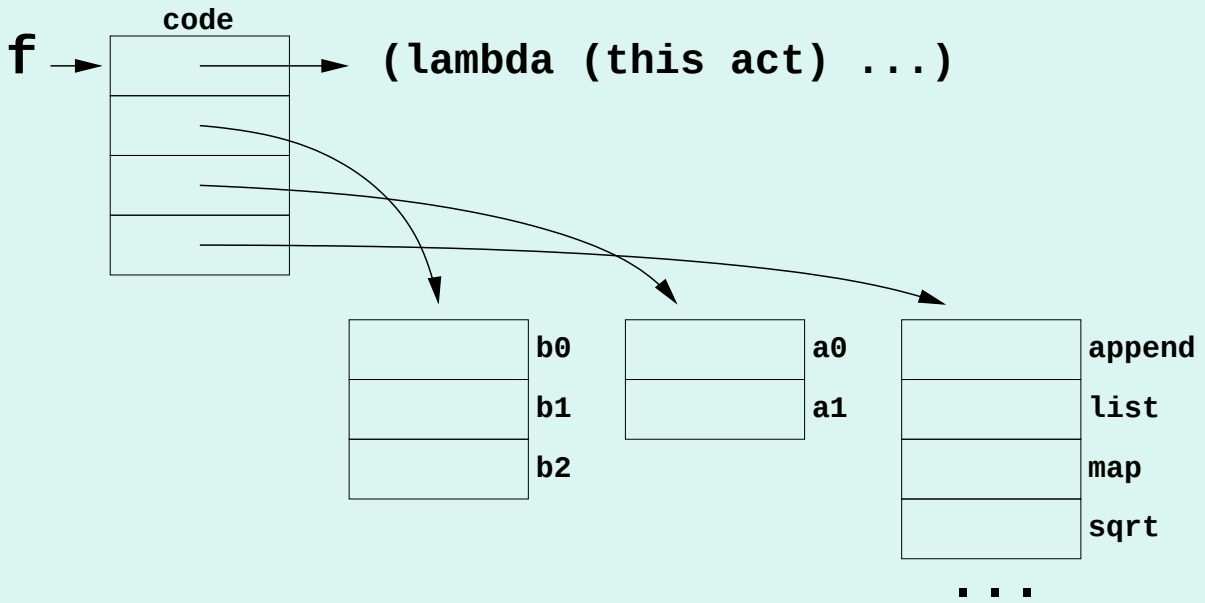
Conversion de fermeture (34)

- Avec variables renommées pour rendre plus lisible :

```
> (closure-conv
    '(lambda (x) (lambda (y) (list x y))))

(make-closure
 (lambda (env1)
   (make-closure
    (lambda (env2)
      (let ((clo
             (vector-ref
               (vector-ref (vector-ref env2 0)
                           0)
               1)))
        ((closure-code clo)
         (vector (closure-env clo)
                 (vector-ref (vector-ref env2 0)
                             1)
                 (vector-ref env2 1))))))
    env1))
 global-env)
```

Conversion de fermeture (35)

- Peut-on améliorer la représentation avec “display”?
- Oui : **intégrer le display à la fermeture**




```
(define make-closure vector)

(define (closure-code clo) (vector-ref clo 0))

(define (closure-ref clo i) (vector-ref clo (+ i 1)))
```

Conversion de fermeture (36)

- Traduction des lambda-expressions et appel de fermeture :

```
(lambda ...) -> (make-closure
                  (lambda (this act) ...)
                  act
                  (closure-ref this 0)
                  (closure-ref this 1)
                  ...)

(F X Y Z)    -> (let ((clo F))
                  ((closure-code clo)
                   clo
                   (vector X Y Z)))
```

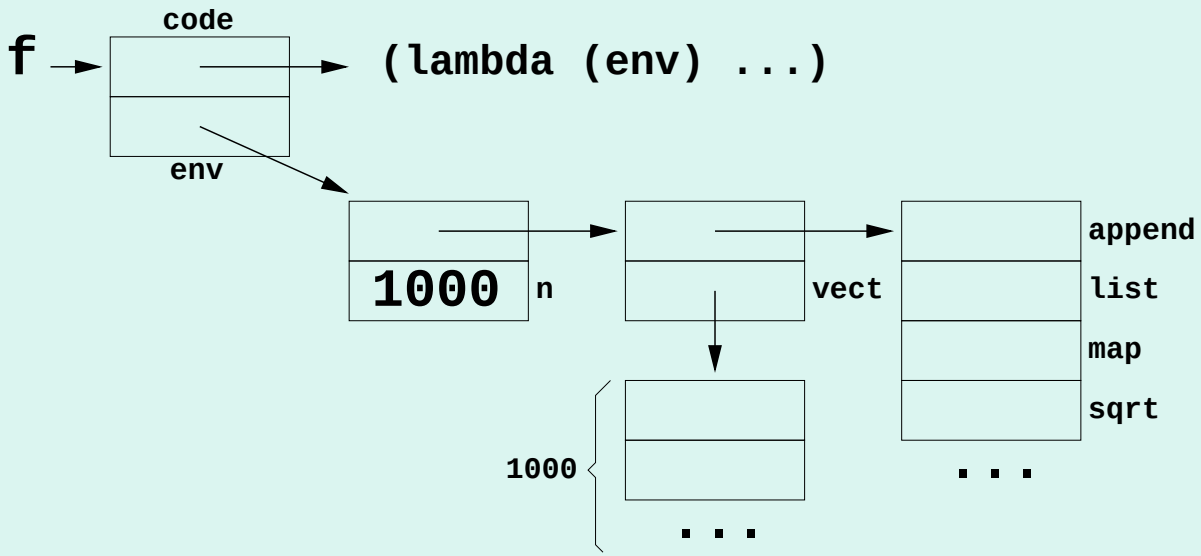
Conversion de fermeture (37)

- Toutes ces représentations d'environnement ont un **problème de rétention mémoire**
- Il se peut qu'une fermeture retienne des variables dans son environnement qui sont **mortes** (i.e. qui ne sont plus utiles pour le reste du calcul)
- Exemple :

```
(define (test vect)
  (let ((n (vector-length vect)))
    (lambda (x)
      (list x n))))

(define f (test (make-vector 1000)))

(f 17) => (17 1000)
(f 50) => (50 1000)
```

Conversion de fermeture (38)

- La fermeture créée sera comme suit

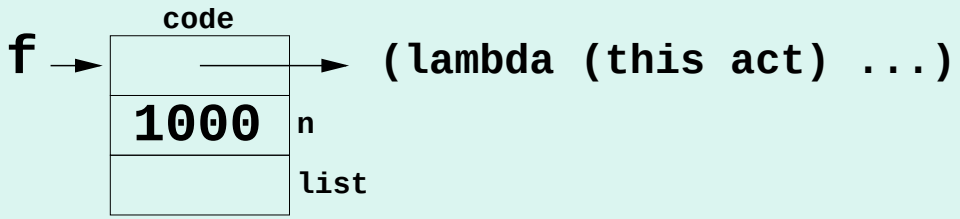


- Mais la variable `vect` est inutile pour l'exécution du corps de la fonction `f`
  - Le GC ne peut pas se débarasser du vecteur tant que la fermeture est accessible
  - L'environnement est pollué par des variables inutiles (perte d'espace, accès lent)

Conversion de fermeture (39)

- La **représentation plate** des fermetures offre une solution à ce problème

- Idée : stocker directement dans la fermeture seulement les variables que la fonction utilise, i.e. les **variables libres** de la fonction



- `this` permet d'accéder aux variables libres de la fonction et `act` permet d'accéder aux paramètres de la fonction

Conversion de fermeture (40)

- Cette représentation peut prendre moins ou plus d'espace que les autres représentations (ça dépend du contexte)
- De plus, telle quelle, cette représentation ne permet pas l'affectation aux variables libres
- La cause c'est qu'une variable peut être dupliquée dans plus d'une fermeture
- La représentation plate prend une **copie de chaque variable libre**

Conversion de fermeture (41)

- Exemple :

```
(define test
  (lambda (n)
    (cons (lambda (x)
            (begin
              (set! n (+ n x))
              n))
          (lambda (x)
            (begin
              (set! n (- n x))
              n)))))

(define p (test 0))
(define inc (car p))
(define dec (cdr p))

(inc 3) => 3
(inc 3) => 6
(dec 1) => 5
```

Conversion de fermeture (42)

- Conversion de fermeture avec environnements chaînés :

```
(define test
  (make-closure
   (lambda (env1)
     (cons (make-closure
            (lambda (env2)
              (begin
                (vector-set!
                 (vector-ref env2 0)
                 1
                 (+ (vector-ref (vector-ref env2 0) 1)
                    (vector-ref env2 1)))
                (vector-ref (vector-ref env2 0) 1)))
            env1)
           (make-closure
            (lambda (env3)
              (begin
                (vector-set!
                 (vector-ref env3 0)
                 1
                 (- (vector-ref (vector-ref env3 0) 1)
                    (vector-ref env3 1)))
                (vector-ref (vector-ref env3 0) 1)))
            env1)))
   global-env))
```

Conversion de fermeture (43)

- Pour traiter les affectations correctement il faut
  - Identifier les variables qui sont **mutables** (qui sont la cible d'un `set!`)
  - Créer des cellules pour contenir leur valeur
  - Les fermetures partageront ces cellules

- Note : heureusement, en Scheme l'affectation est peu souvent employée

Conversion de fermeture (44)

- Étapes de traduction :
  - **Alpha-conversion** qui donne un nom unique à toutes les variables (pour simplifier le reste de la traduction)
  - **Analyse de mutation** (“mutation analysis”)
  - **Conversion des affectations** (“assignment conversion”)
  - **Conversion des fermetures** (“closure conversion”)

Conversion de fermeture (45)


```
(define (alpha-conv expr)
  (alphac expr '()))

(define (alphac expr env)

  (define (rename v)
    (cond ((assq v env) => cdr)
          (else v)))

  (match expr

    (,const when (constant? const)
     expr)

    (,var when (variable? var)
     (rename var))

    ((set! ,var ,E1)
     `(set! ,(rename var)
            ,(alphac E1 env)))
```

Conversion de fermeture (46)


```
((lambda ,params ,E)
 (let* ((fresh-params
         (map (lambda (p) (cons p (gensym)))
              params))
        (new-env
         (append fresh-params env)))
   `(lambda ,(map cdr fresh-params)
      ,(alphac E new-env))))

((if ,E1 ,E2)
 `(if ,(alphac E1 env)
      ,(alphac E2 env)))
((if ,E1 ,E2 ,E3)
 `(if ,(alphac E1 env)
      ,(alphac E2 env)
      ,(alphac E3 env)))

((,E0 . ,Es)
 `(,(if (primitive? E0) E0 (alphac E0 env))
   ,@(map (lambda (e) (alphac e env))
          Es)))

(,_
 (error "unknown expression" expr))))
```

Conversion de fermeture (47)


```
(define (mv expr)
  (match expr
    (,const when (constant? const)
     '())

    (,var when (variable? var)
     '())

    ((set! ,var ,E1)
     (union (list var) (mv E1)))

    ((lambda ,params ,E)
     (mv E))

    ((if ,E1 ,E2)
     (union (mv E1) (mv E2)))
    ((if ,E1 ,E2 ,E3)
     (union (mv E1) (union (mv E2) (mv E3))))

    ((,E0 . ,Es)
     (union (if (primitive? E0) '() (mv E0))
            (apply union (map mv Es))))

    (,_
     (error "unknown expression" expr))))
```

Conversion de fermeture (48)


```
(define (assign-conv expr)
  (assignc expr
           (union (mv expr) (fv expr))))

(define (assignc expr mut-vars)

  (define (mutable? v) (memq v mut-vars))

  (match expr

    (,const when (constant? const)
     expr)

    (,var when (variable? var)
     (if (mutable? var)
         `(car ,var)
         var))

    ((set! ,var ,E1)
     `(set-car! ,var
                ,(assignc E1 mut-vars)))
```

Conversion de fermeture (49)


```
((lambda ,params ,E)
 (let* ((mut-params
         (map (lambda (p) (cons p (gensym)))
              (keep mutable? params)))
        (params2
         (map (lambda (p)
                (if (mutable? p)
                    (cdr (assq p mut-params))
                    p))
              params)))
   `(lambda ,params2
      ,(if (null? mut-params)
           (assignc E mut-vars)
           `((lambda ,(map car mut-params)
               ,(assignc E mut-vars))
             ,@(map (lambda (x) `(list ,(cdr x)))
                    mut-params))))))
```

Conversion de fermeture (50)


```
((if ,E1 ,E2)
 `(if ,(assignc E1 mut-vars)
      ,(assignc E2 mut-vars)))
((if ,E1 ,E2 ,E3)
 `(if ,(assignc E1 mut-vars)
      ,(assignc E2 mut-vars)
      ,(assignc E3 mut-vars)))

((,E0 . ,Es)
 `(,(if (primitive? E0) E0 (assignc E0 mut-vars))
   ,@(map (lambda (e) (assignc e mut-vars))
          Es)))

(,_
 (error "unknown expression" expr))))
```

Conversion de fermeture (51)


```
(define (closure-conv expr)
  (closurec expr '()))

(define (closurec expr cte)

  (define (pos id)
    (let ((x (memq id cte)))
      (and x
           (- (length cte)
              (length x)))))

  (match expr

    (,const when (constant? const)
     expr)

    (,var when (variable? var)
     (let ((p (pos var)))
       (if p
           `(closure-ref $this ,p)
           var)))
```

Conversion de fermeture (51)


```
((lambda ,params ,E)
 (let ((new-cte (fv expr)))
   `(make-closure
     (lambda ($this ,@params)
       ,(closurec E new-cte))
     ,@(map (lambda (v) (closurec v cte))
            new-cte))))

((if ,E1 ,E2)
 `(if ,(closurec E1 cte) ,(closurec E2 cte)))
((if ,E1 ,E2 ,E3)
 `(if ,(closurec E1 cte) ,(closurec E2 cte) ,(closurec

((,E0 . ,Es)
 (if (primitive? E0)
     `(,E0
       ,@(map (lambda (e) (closurec e cte))
              Es))
     `(let (($clo ,(closurec E0 cte)))
        ((closure-code $clo)
         $clo
         ,@(map (lambda (e) (closurec e cte))
                Es)))))

(,_
```

Conversion de fermeture (52)

- Traduction de l'exemple `inc/dec` :

```
(define test
  (make-closure
   (lambda ($this g99)
     (let (($clo (make-closure
                  (lambda ($this n)
                    (cons (make-closure
                           (lambda ($this x)
                             (begin
                               (set-car!
                                (closure-ref $this 0)
                                (+ (car (closure-ref $this 0)) x))
                               (car (closure-ref $this 0))))
                           n)
                          (make-closure
                           (lambda ($this x)
                             (begin
                               (set-car!
                                (closure-ref $this 0)
                                (- (car (closure-ref $this 0)) x))
                               (car (closure-ref $this 0))))
                           n))))))
       ((closure-code $clo) $clo (list g99)))))

(define p (let (($clo test)) ((closure-code $clo) $clo 0)))
(define inc (car p))
(define dec (cdr p))

(let (($clo inc)) ((closure-code $clo) $clo 3)) => 3
(let (($clo inc)) ((closure-code $clo) $clo 3)) => 6
(let (($clo dec)) ((closure-code $clo) $clo 1)) => 5
```

Conversion de fermeture (53)

- Application à l'interprète rapide (dans le but de sérialiser les fermetures)

```
(define (ev expr cte)

  (cond ((constant? expr)
         (let ((val (constant-val expr)))
           (lambda (rte) val)))

        ((variable? expr)
         (let ((pos (cte-lookup cte expr)))
           (lambda (rte) (rte-lookup rte pos))))

        ((lambda? expr)
         (let ((b (ev (lambda-body expr)
                      (cte-extend cte
                                  (lambda-params expr)))))
           (lambda (rte)
             (lambda args
               (b (rte-extend rte args))))))

        ...))
```

Conversion de fermeture (54)


```
(define (ev expr cte)

  (cond ((constant? expr)
         (let ((val (constant-val expr)))
           (make-closure
            (lambda ($this rte)
              (closure-ref $this 0)) ;; val
            val)))

        ((variable? expr)
         (let ((pos (cte-lookup cte expr)))
           (make-closure
            (lambda ($this rte)
              (rte-lookup rte
                          (closure-ref $this 0))) ;; pos
            pos)))
```

Conversion de fermeture (55)


```
((lambda? expr)
 (let ((b (ev (lambda-body expr)
              (cte-extend cte
                          (lambda-params expr)))))
   (make-closure
    (lambda ($this rte)
      (make-closure
       (lambda ($this . args)
         (let (($clo (closure-ref $this 0))) ;; b
           ((closure-code $clo)
            $clo
            (rte-extend
             (closure-ref $this 1) ;; rte
             args))))
       (closure-ref $this 0)
       rte))
    b)))

...))
```

Lambda-lifting (1)

- Le **lambda-lifting** est une transformation qui permet d'**éviter de créer des fermetures** dans certains cas
- C'est utile pour réduire la quantité de mémoire allouée par un programme, et donc de réduire l'utilisation du garbage-collector
- S'applique aux lambda-expressions qui sont liées à des variables par des `let`, `let*`, `letrec`, ou définitions internes

Lambda-lifting (2)

- Exemple simple de programme transformable par lambda-lifting :

```
(define f
  (lambda (a b)
    (let ((g (lambda (x) (+ x a))))
      (g (g b)))))

(f 1 100)
(f 2 500)
```

- Normalement il faudrait créer une fermeture pour la lambda-expression `(lambda (x) (+ x a))` pour que la fonction se souvienne de la valeur de `a` dans chaque appel à `f`

Lambda-lifting (3)

- Mais si on passe explicitement la valeur de `a` en paramètre, on élimine le besoin de créer une fermeture :

```
(define f
  (lambda (a b)
    (let ((g (lambda (a x) (+ x a))))
      (g a (g a b)))))

(f 1 100)
(f 2 500)
```

Lambda-lifting (4)

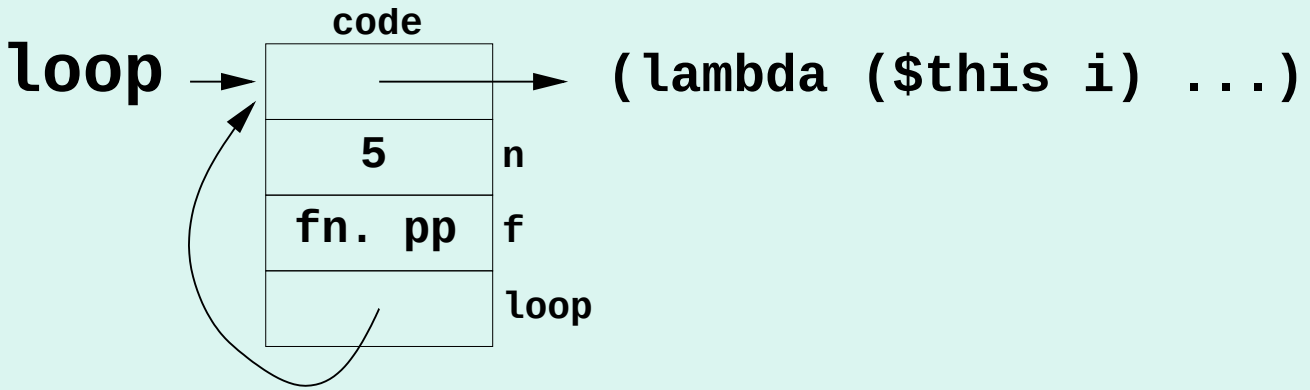
- Exemple de fonction récursive transformable par lambda-lifting :

```
(define repeter
  (lambda (n f)
    (letrec ((loop
              (lambda (i)
                (if (<= i n)
                    (begin
                      (f i)
                      (loop (+ i 1)))))))
      (loop 1))))

(repeter 5 pp)
```

- On aurait pu utiliser un `let`-nommé ou une définition interne (ce serait tout à fait équivalent)

Lambda-lifting (5)

- La fermeture créée pour `loop` permet d'accéder aux variables `n`, `f` et `loop` (à noter que dans ce cas particulier `loop` = `$this`) :




```
(define repeter
  (lambda (n f)
    (letrec ((loop
              (lambda (i)
                (if (<= i n)
                    (begin
                      (f i)
                      (loop (+ i 1)))))))
      (loop 1))))

(repeter 5 pp)
```

Lambda-lifting (6)

- Traduction par conversion de fermeture :

```
define repeter
(make-closure
 (lambda ($this n f)
  (let ((loop
         (make-closure
          (lambda ($this i)
           (if (<= i (closure-ref $this 0))
               (begin
                (let (($clo (closure-ref $this 1)))
                  ((closure-code $clo) $clo i))
                (let (($clo (closure-ref $this 2)))
                  ((closure-code $clo) $clo (+ i 1))))))
          n
          f
          #f)))
   (closure-set! loop 2 loop)
   (let (($clo loop))
     ((closure-code $clo) $clo 1))))))
```

Lambda-lifting (7)

- Peut-on éviter de créer une fermeture pour `loop`?
- Il s'agit de trouver une autre façon pour que le corps de `loop` soit informé de la valeur des variables `n`, `f` et `loop`
- Solution : **passer ces valeurs en paramètre** à `loop`!
- Cela fonctionne même pour l'affectation si on a fait une conversion d'affectation au préalable

Lambda-lifting (8)

- Nouveau code sans fermeture :

```
(define repeter
  (lambda (n f)
    (let ((loop
           (lambda (n f loop i)
             (if (<= i n)
                 (begin
                   (f i)
                   (loop n f loop (+ i 1)))))))
      (loop n f loop 1))))
```

- Remarques :
  - le `letrec` a été remplacé par un `let`
  - la fonction `loop` n'a plus de variables libres

Lambda-lifting (9)

- Idée de base de la transformation :
  - Pour toute variable v qui est liée à une lambda-expression l et qui apparait seulement dans des appels de fonction en position fonctionnelle, c'est-à-dire $(v \ \ldots)$:
    - Ajouter les variables libres de l au début de la liste de paramètres formels de l
    - Ajouter les variables libres de l au début de la liste de paramètres actuels de chaque appel de fonction $(v \ \ldots)$

Lambda-lifting (10)

- Cela ne fonctionne pas tout à fait lorsqu'il y a plus d'une fonction à transformer :

```
(define repeter
  (lambda (n z f)
    (letrec ((g
              (lambda (x) (* x z)))
             (loop
              (lambda (i)
                (if (<= i n)
                    (begin
                      (f (g i))
                      (loop (+ i 1)))))))
      (loop 1))))

(repeter 5 10 pp) => 10 20 30 40 50
```

Lambda-lifting (11)

- La transformation donne :

```
(define repeter
  (lambda (n z f)
    (let ((g
           (lambda (z x) (* x z)))
          (loop
           (lambda (n f loop g i)
             (if (<= i n)
                 (begin
                   (f (g z i))
                   (loop n f loop g (+ i 1)))))))
      (loop n f loop g 1))))
```

- `z` est maintenant une variable libre de `loop`

Lambda-lifting (12)

- Il faut répéter la transformation jusqu'à ce qu'il n'y ait plus de changement :

```
(define repeter
  (lambda (n z f)
    (let ((g
           (lambda (z x) (* x z)))
          (loop
           (lambda (z n f loop g i)
             (if (<= i n)
                 (begin
                   (f (g z i))
                   (loop z n f loop g (+ i 1)))))))
      (loop z n f loop g 1))))
```

Lambda-lifting (13)

- Le lambda-lifting ne s'applique pas lorsque les lambda-expressions ne sont pas directement liées à des variables, ou bien lorsque ces variables ne sont pas uniquement utilisées en position fonctionnelle

- Dans ces cas, il se peut que :
  - les sites d'appel ne soient pas dans la portée (lexicale) des variables libres
  - les sites d'appel de la fermeture soient aussi des sites d'appel de d'autres fermetures (donc il y a une possibilité de conflit d'ajout de paramètres actuels)

Lambda-lifting (14)

- Exemple de programme qui n'est pas transformable par lambda-lifting :

```
(define f
  (lambda (a b)
    (let ((g (lambda (x) (+ x a))))
      (list g
            (g (g b))))))

(define x (f 1 100))

(define h (car x))

(h 2) ;; comment transformer cet appel?
```

Lambda-lifting (15)

- Autre programme qui n'est pas transformable par lambda-lifting :

```
(define map
  (lambda (f lst)
    (if (null? lst)
        '()
        (cons (f (car lst))
              (map f (cdr lst))))))

(define g
  (lambda (a b c lst)
    (list (map (lambda (x) (* x (+ a b))) lst)
          (map (lambda (y) (+ y c)) lst))))
```

- Puisque 2 fermetures peuvent se faire appeler par l'appel `(f (car lst))` et que ces fermetures n'ont pas le même nombre de variables libres, comment transformer cet appel?