UNIVERSITÉ
CÔTE D'AZUR

Paradigmes et Interprétation

Programmation à états : variables

Julien Provillard
julien.provillard@univ-cotedazur.fr

UNIVERSITÉ
CÔTE D'AZUR

VARIABLES

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes

- Comment traduire ?

```
{let {[x 1]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 10}
      {f 2}}}}
```

```
{let {[x {box 1}]}
  {let {[f {box {lambda {y}
                  {+ {unbox x} y}}}]}
    {begin
      {set-box! x 10}
      {{unbox f} 2}}}}
```

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes

- Comment traduire ?

```
{let {[x 1]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 10}
      {f 2}}}}
```

```
{let {[x {box 1}]}
  {let {[f {box {lambda {y}
                  {+ {unbox x} {unbox y}}}}]}
    {begin
      {set-box! x 10}
      {{unbox f} {box 2}}}}}
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation des variables

- Les boîtes permettent de mimer le comportement des variables.
- Mais elles sont beaucoup plus lourde à l'usage.

- Il suffirait que les variables prennent la place des boîtes :

```
(define-type Binding
  [bind (name : Symbol) (location : Location)])
```

- Tous les identificateurs passent par une indirection dans la mémoire !

UNIVERSITÉ CÔTE D'AZUR

Grammaire

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
        | {lambda {<Symbol>} <Exp>}
        | {<Exp> <Exp>}
        | {set! <Symbol> <Exp>}
        | {begin <Exp> <Exp>}
```

New !

UNIVERSITÉ CÔTE D'AZUR

Utilisation des variables : exemples

```
interp : (Exp Env Store -> Result)

(test (interp (parse `2) mt-env mt-store)
      (v*s (numV 2) mt-store))
```

UNIVERSITÉ CÔTE D'AZUR

Utilisation des variables : exemples

```
interp : (Exp Env Store -> Result)

(test (interp (parse `{let {[x 2]} x}) mt-env mt-store)
      (v*s (numV 2)
           (override-store (cell 1 (numV 2)) mt-store)))

(test (interp (parse `x)
              (extend-env (bind 'x 1) mt-env)
              (override-store (cell 1 (numV 2)) mt-store))
      (v*s (numV 2)
           (override-store (cell 1 (numV 2)) mt-store)))
```

UNIVERSITÉ CÔTE D'AZUR

Utilisation des variables : exemples

```
interp : (Exp Env Store -> Result)

(test (interp (parse `{{lambda {x} {+ x x}} 2}) mt-env mt-store)
      (v*s (numV 4)
           (override-store (cell 1 (numV 2)) mt-store)))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(idE s) (v*s (fetch (lookup s env) sto) sto)]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(letE s rhs body)
     (with [(v-rhs sto-rhs) (interp rhs env sto)]
           (let ([l (new-loc sto-rhs)])
             (interp body
                     (extend-env (bind s l) env)
                     (override-store (cell l v-rhs) sto-rhs))))]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(appE f arg)
     (with [(v-f sto-f) (interp f env sto)]
           (type-case Value v-f
             [(closV par body c-env)
              (with [(v-arg sto-arg) (interp arg env sto-f)]
                    (let ([l (new-loc sto-arg)])
                      (interp body
                              (extend-env (bind par l) c-env)
                              (override-store (cell l v-arg) sto-arg))))]
             [else (error 'interp "not a function")]))]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(setE var val)
     (let ([l (lookup var env)])
       (with [(v-val sto-val) (interp val env sto)]
             (v*s v-val (override-store (cell l v-val) sto-val))))]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes (bis)

- Les boîtes peuvent simuler les variables.

```
{let {[x 1]}
  {let {[f {lambda {y}
             {+ x y}}]}
    {begin
      {set! x 10}
      {f 2}}}}
```

```
{let {[x {box 1}]}
  {let {[f {box {lambda {y}
                  {+ {unbox x} {unbox y}}}}]}
    {begin
      {set-box! x 10}
      {{unbox f} {box 2}}}}}
```

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes (bis)

- L'inverse est-il vrai ?

```
{let {[fill {lambda {b}
              {set-box! b 1}}]}
  {let {[b {box 0}]}
    {begin
      {fill b}
      {unbox b}}}}
```

s'évalue en `(numV 1)`.

Les boîtes sont des valeurs.

```
{let {[fill {lambda {v}
              {set! v 1}}]}
  {let {[v 0]}
    {begin
      {fill v}
      v}}}
```

s'évalue en `(numV 0)`.

Pas les variables.

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes (bis)

- Pour pouvoir simuler les boîtes avec les variables, il faut les encapsuler dans une valeur : une clôture lexicale.
- En fait, une clôture va gérer l'accès et une deuxième la modification.

```
(define (crate val)
  (pair (lambda () val) ; équivalent de unbox
        (lambda (new-val) (set! val new-val)))) ; équivalent de set-box!

(define (uncrate c)
  ((fst c))) ; extraction et appel de l'équivalent de unbox

(define (set-crate! c val)
  ((snd c) val)) ; extraction et appel de l'équivalent de set-box!
```

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes (bis)

```
{let {[fill {lambda {b}
              {set-box! b 1}}]}
  {let {[b {box 0}]}
    {begin
      {fill b}
      {unbox b}}}}
```

```
{let {[fill {lambda {c}
              {set-crate! c 1}}]}
  {let {[c {crate 0}]}
    {begin
      {fill c}
      {uncrate c}}}}
```

Mais pour que cela fonctionne dans le langage interne, il faudrait implémenter les paires…

On peut néanmoins conclure que les boîtes et les variables ont la même expressivité.

UNIVERSITÉ CÔTE D'AZUR

Appel par valeur

- Lorsqu'on passe une variable comme argument d'une fonction, cela revient à passer sa valeur. C'est ce qu'on nomme l'**appel par valeur** des fonctions.

```
(define (swap x y)
  (let ([tmp x])
    (begin (set! x y) (set! y tmp))))

(let ([a 1] [b 2])
  (begin (swap a b) a))
--> 1
```

x et y sont des variables qui contiennent les mêmes valeurs que a et b.
Ce sont les valeurs de x et y qui sont échangées, pas celles de a et b !

UNIVERSITÉ CÔTE D'AZUR

Appel par valeur

- Lorsqu'on passe une variable comme argument d'une fonction, cela revient à passer sa valeur. C'est ce qu'on nomme l'**appel par valeur** des fonctions.

```
{let {[incr {lambda {x} {set! x {+ x 1}}}]}
  {let {[a 0]}
    {begin
      {incr a}
      a}}}
--> (numV 0)
```

On a bien implémenté l'appel par valeur dans l'interpréteur.
Mais qu'aurait-on dû faire si on avait voulu obtenir `(numV 1)` en résultat ?

UNIVERSITÉ CÔTE D'AZUR

Parallèle avec les boîtes

```
{let {[incr {lambda {x} {set! x {+ x 1}}}]}
  {let {[a 0]}
    {begin
      {incr a}
      a}}}
```

```
{let {[incr {box {lambda {x} {set-box! x {+ {unbox x} 1}}}}]}
  {let {[a {box 0}]}
    {begin
      {{unbox incr} {box {unbox a}}}
      {unbox a}}}}
```

UNIVERSITÉ CÔTE D'AZUR

Parallèle avec les boîtes

```
{let {[incr {lambda {x} {set! x {+ x 1}}}]}
  {let {[a 0]}
    {begin
      {incr a}
      a}}}
```

```
{let {[incr {box {lambda {x} {set-box! x {+ {unbox x} 1}}}}]}
  {let {[a {box 0}]}
    {begin
      {{unbox incr} a} ; {{unbox incr} {box {unbox a}}}
      {unbox a}}}}
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(appC f arg)
     (with [(v-f sto-f) (interp f env sto)]
           (type-case Value v-f
             [(closV par body c-env)
              (type-case Exp arg
                [(idE s) (interp body
                                 (extend-env (bind par (lookup s env)) c-env)
                                 sto-f)]
                [else ... ])]
             [else (error 'interp "not a function")]))]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Appel par référence

- Lors d'un **appel par référence**, si une variable est passée en argument, aucune variable fraîche n'est créée pour contenir sa valeur. On utilise directement la variable passée en argument.

- Le code C suivant est-il un appel par référence ?

```
void swap(int* i, int* j) {
  int tmp = *i;
  *i = *j;
  *j = tmp;
}

int x = 0, y = 1;
swap(&x, &y);
```

Non ! C'est un appel par valeur !
Mais la valeur qui est copiée est une adresse mémoire.
On ne peut pas les manipuler dans notre langage.