UNIVERSITÉ
CÔTE D'AZUR

Paradigmes et Interprétation

Programmation à états : boîtes et enregistrements

Julien Provillard
julien.provillard@univ-cotedazur.fr

UNIVERSITÉ
CÔTE D'AZUR

PROGRAMMATION À ÉTATS

UNIVERSITÉ CÔTE D'AZUR

Etats

- Dans les langages que nous avons défini, nous avons des identificateurs.
- Ils sont introduit par un `let` ou lors de l'application d'une fonction.
- On peut substituer un identificateur : **sa valeur est fixe** !

```
{let {[x 1]}
  {let {[f {lambda {y} {+ x y}}]}
    ...
    {f 2}}}
```

```
{let {[f {lambda {y} {+ 1 y}}]}
  ...
  {f 2}}
```

UNIVERSITÉ CÔTE D'AZUR

Etats

- En `plait`, la valeur d'un identificateur peut changer.

```
{let {[x 1]}
  {let {[f {lambda {y} {+ x y}}]}
    {begin
      {set! x 10}
      {f 2}}}}
--> 12
```

- Les **variables** ont un **état**.
- En `plait`, il est fortement déconseillé d'utiliser la mutation de variables. Dans d'autres langages par contre …

UNIVERSITÉ CÔTE D'AZUR

Quand on peut se passer d'état : sommation

```
public static int sum(List<Integer> l) {
  int res = 0;
  for (Integer n : l) {
    res += n;
  }
  return res;
}
```

```
(define (sum [L : (Listof Number)]) : Number
  (if (empty? L)
      0
      (+ (first L) (sum (rest L)))))
```

UNIVERSITÉ CÔTE D'AZUR

Quand on peut se passer d'état : sommation

```
public static int sum(List<Integer> l) {
  int res = 0;
  for (Integer n : l) {
    res += n;
  }
  return res;
}
```

```
(define (sum [L : (Listof Number)] [res : Number]) : Number
  (local [(define (aux [L : (Listof Number)] [res : Number]) : Number
            (if (empty? L)
                res
                (aux (rest L) (+ (first L) res))))]
    (aux L 0)))
```

UNIVERSITÉ CÔTE D'AZUR

Quand on peut se passer d'état : sommation

```
public static int sum(List<Integer> l) {
  int res = 0;
  for (Integer n : l) {
    res += n;
  }
  return res;
}
```

```
(define (sum [L : (Listof Number)]) : Number
  (foldl + 0 L))))
```

UNIVERSITÉ CÔTE D'AZUR

Quand on peut se passer d'état : incrément

```
public static int incrAll(int[] t) {
  for (int i = 0; i < t.length; i++) {
    t[i]++;
  }
}
```

À chaque fois qu'on modifie une variable en Java, on définit une nouvelle valeur en `plait`.

```
(define (incrAll [L : (Listof Number)]) : (Listof Number)
  (map add1 L))
```

UNIVERSITÉ CÔTE D'AZUR

Pourquoi éviter les états?

- Tests faciles (seuls les valeurs importent)
  - `(test (incrAll '(1 2 3 4)) '(2 3 4 5))`
  - En Java `==` vs `equals`

- Pas d'effets de bord

```
(define now '(1 2 3 4))
(define later (incrAll now))
(equal? later now)
```

  - On sait que l'appel à `incrAll` n'a pas modifié `now`.

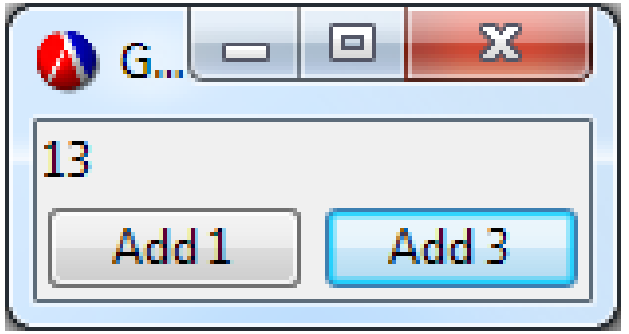
UNIVERSITÉ CÔTE D'AZUR

Les états peuvent être nécessaires

```
#lang racket
(require htdp/gui)

(define counter 0)
(define total-message (make-message (number->string counter)))
(define (make-incr-button label amount)
  (make-button label
               (lambda (evt)
                 (begin
                   (set! counter (+ counter amount))
                   (draw-message total-message (number->string counter))))))

(create-window (list (list total-message)
                     (list (make-incr-button "Add 1" 1)
                           (make-incr-button "Add 3" 3))))
```



Deux évènements possibles, une donnée commune.

UNIVERSITÉ CÔTE D'AZUR

Etats et effets de bord

- Les états sont un moyen pour différentes parties d'un programme de communiquer à l'aide d'**effets de bord** (modification d'une donnée visible par les deux blocs logiciels).

  \+ Il est possible de créer facilement des canaux de communication entre différents blocs logiciels.

  \- Les effets de bord peuvent être cachés et leur source difficile à identifier (debugging plus long).

UNIVERSITÉ
CÔTE D'AZUR

BOÎTES

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes

- Dans un langage impératif, on trouve la notion de variables.

```
(define counter 0)

(define (incr) : void
  (set! counter (add1 counter)))

(define (get-value) : Number
  counter)
```

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes

- Dans les langages fonctionnels qui se dotent d'une couche impérative minimale, on trouve plutôt la notion de boîte.

```
(define counter (box 0))

(define (incr) : void
  (set-box! counter (add1 (unbox counter))))

(define (get-value) : Number
  (unbox counter))
```

C'est la notion de référence en `Ocaml` par exemple.

UNIVERSITÉ CÔTE D'AZUR

Variables vs boîtes

- Dans les langages fonctionnels qui se dotent d'une couche impérative minimale, on trouve plutôt la notion de boîte.
- Les boîtes sont les seuls éléments du langage doté d'un état. On limite les effets de bord aux boîtes.

```
box : ('a -> (Boxof 'a)) ; création avec valeur initiale
unbox : ((Boxof 'a) -> 'a) ; accès au contenu
set-box! : ((Boxof 'a) 'a -> void) ; mutation du contenu
```

UNIVERSITÉ CÔTE D'AZUR

Les boîtes, des objets élémentaires

```
(let ([b (box 0)])
  (begin
    (set-box! b 1)
    (unbox b)))
```

```
class Box<T> {
  T val;
  Box(T val) {
    this.val = val;
  }
}

Box<Integer> b = new Box(0);

b.val = 1;
return b.val;
```

UNIVERSITÉ CÔTE D'AZUR

Grammaire

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

UNIVERSITÉ CÔTE D'AZUR

Représentation

Nouvelles expressions

```
(define-type Exp
  ...
  [boxC (val : Exp)]
  [unboxC (b : Exp)]
  [setboxC (b : Exp) (val : Exp)]
  [beginC (l : Exp) (r : Exp)])
```

Nouvelle valeur

```
(define-type Value
  [numV (n : Number)]
  [closV (par : Symbol) (body : Exp) (env : Env)]
  [boxV (b : (Boxof Value))])
```

UNIVERSITÉ CÔTE D'AZUR

Interprétation

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(boxE val) (boxV (box (interp val env)))]
    [(unboxE b)
     (type-case Value (interp b env)
       [(boxV bi) (unbox bi)]
       [else (error 'interp "not a box")])]
    [(setboxE b val)
     (type-case Value (interp b env)
       [(boxV bi) (let ([v (interp val env)])
                    (begin (set-box! bi v) v))]
       [else (error 'interp "not a box")])]
    [(beginE l r) (begin (interp l env) (interp r env))]))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation insatisfaisante

- Implémenter des boîtes à l'aide de boîte n'explique pas leur fonctionnement.

- On n'a pas besoin d'états pour interpréter les boîtes :
  - On prend la main sur tous les canaux de communication.
  - On passe explicitement les messages lors de l'interprétation.

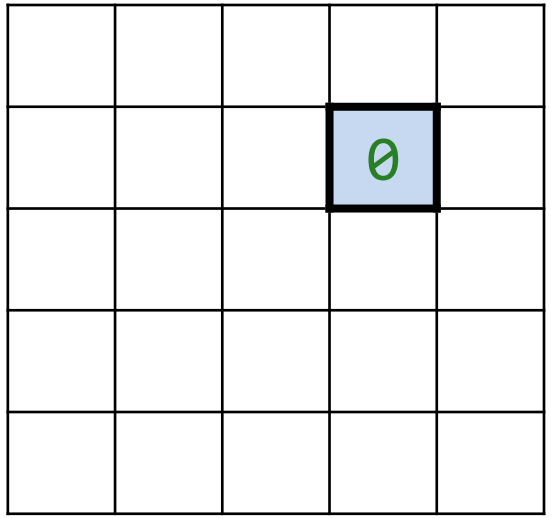
UNIVERSITÉ CÔTE D'AZUR

Boîtes et mémoire

- L'instruction box réserve un emplacement mémoire et initialise son contenu.

```
{let {[b {box 0}]}
  ...
```

Mémoire



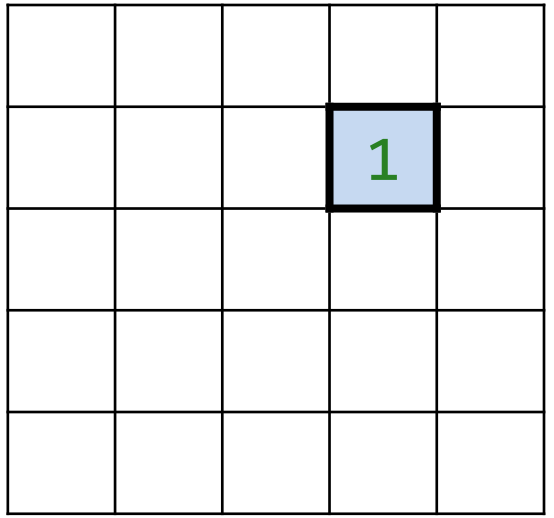
UNIVERSITÉ CÔTE D'AZUR

Boîtes et mémoire

- L'instruction `set-box!` modifie le contenu d'un emplacement mémoire qui a été réservé.

```
{let {[b {box 0}]}
  {begin
    {set-box! b 1}
    ...
```

Mémoire



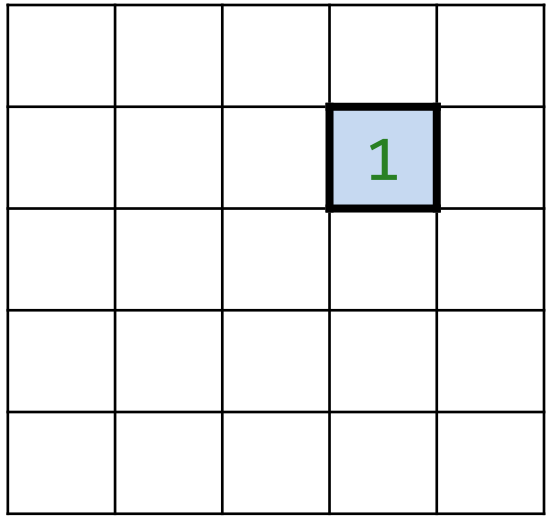
UNIVERSITÉ CÔTE D'AZUR

Boîtes et mémoire

- L'instruction unbox permet d'accéder au contenu d'un emplacement mémoire qui a été réservé.

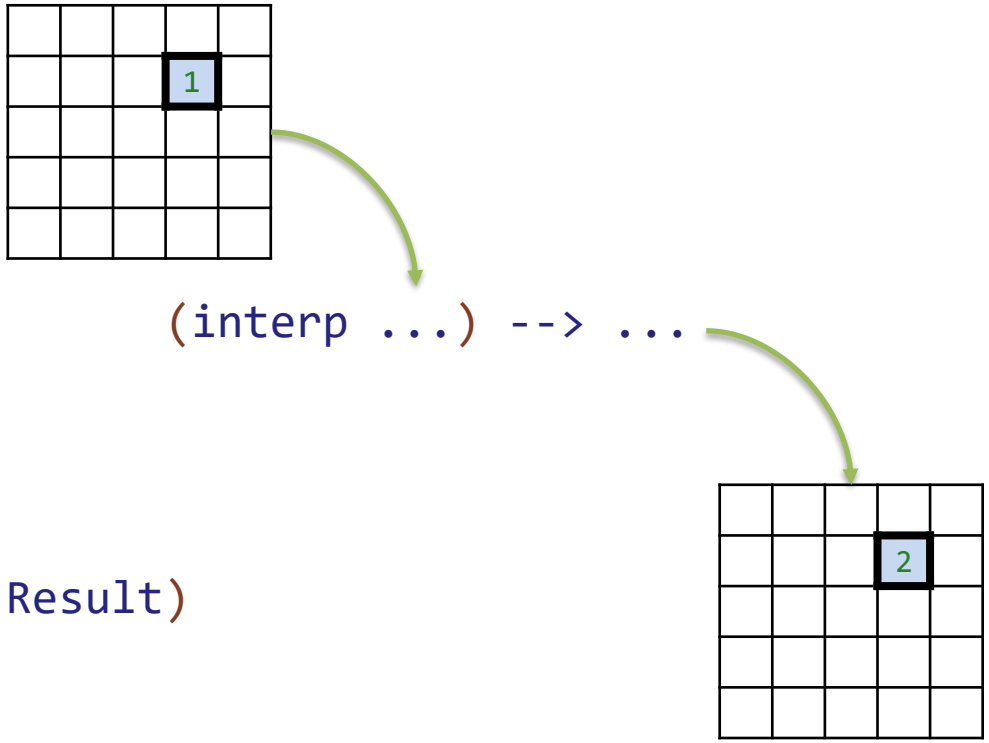
```
{let {[b {box 0}]}
  {begin
    {set-box! b 1}
    {unbox b}}}
--> 1
```

Mémoire



UNIVERSITÉ CÔTE D'AZUR

Mémoire et interprétation




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

(define-type Result
  [v*s (v : Value) (s : Store)])
```

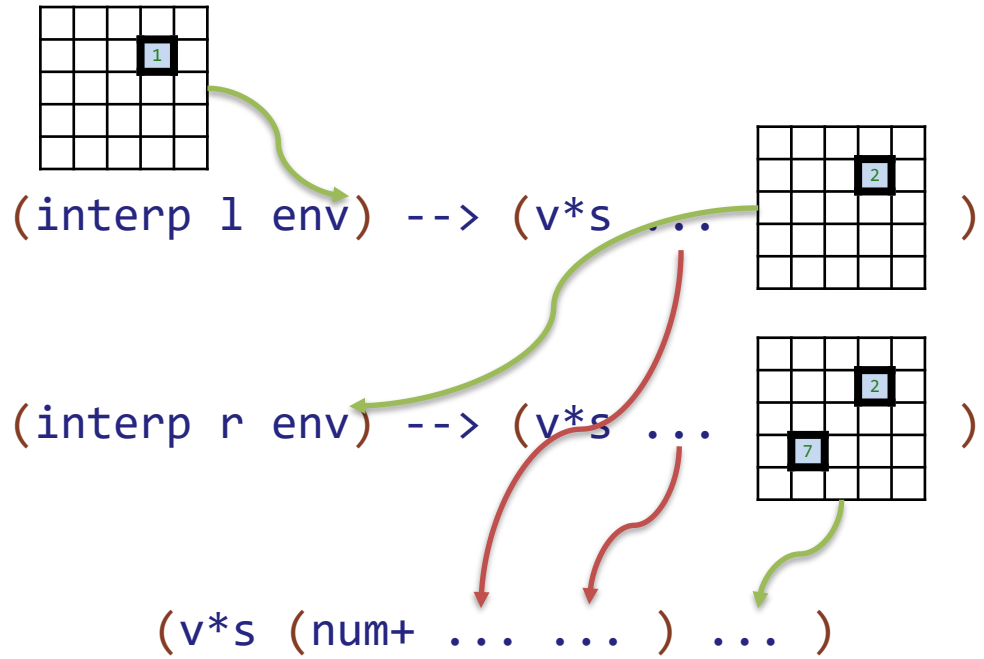
UNIVERSITÉ CÔTE D'AZUR

Mémoire et interprétation : addition

```
(num+ (interp l env) (interp r env))
```

UNIVERSITÉ CÔTE D'AZUR

Mémoire et interprétation : addition




```
(type-case Result (interp l env sto)
  [(v*s v-l sto-l)
       (type-case Result (interp r env sto-l)
          [(v*s v-r sto-r) (v*s (num+ v-l v-r) sto-r)])])
```

L'ordre d'évaluation des arguments devient important!

UNIVERSITÉ CÔTE D'AZUR

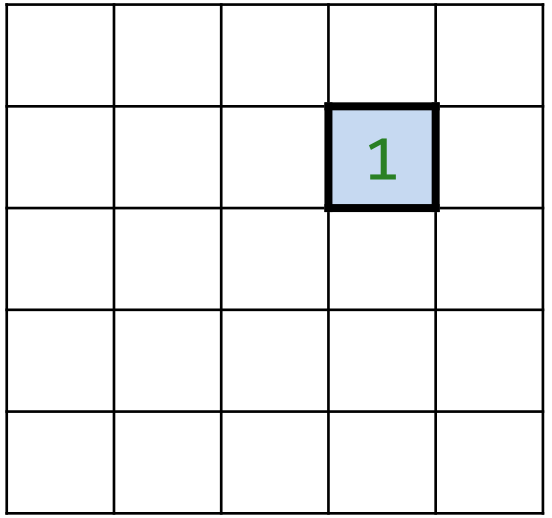
Représentation de la mémoire

```
(define-type-alias Location Number)

(define-type Storage
  [cell (location : Location) (val : Value)])

(define-type-alias Store (Listof Storage))
(define mt-store empty)
(define override-store cons)


(override-store (cell 9 (numV 1)) mt-store)
```



UNIVERSITÉ CÔTE D'AZUR

Utilisation de la mémoire : exemples

- Changements d'utilisation
  - Avant

```
interp : (Exp Env -> Value)

(test (interp (parse `2) mt-env) (numV 2))
```

  - Après

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

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

UNIVERSITÉ CÔTE D'AZUR

Utilisation de la mémoire : exemples

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

(test (interp (parse `{box 2}) mt-env mt-store)
      (v*s (boxV 1)
           (override-store (cell 1 (numV 2)) mt-store)))
```

UNIVERSITÉ CÔTE D'AZUR

Utilisation de la mémoire : exemples

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

(test (interp (parse `{set-box! {box 2} 3}) mt-env mt-store)
      (v*s (numV 3)
           (override-store (cell 1 (numV 3))
                           (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
    ...
    [(numE n) (v*s (numV n) sto)]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

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

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(plusE l r)
     (type-case Result (interp l env sto)
       [(v*s v-l sto-l)
        (type-case Result (interp r env sto-l)
          [(v*s v-r sto-r) (v*s (num+ v-l v-r) sto-r)])])]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(boxE val)
     (type-case Result (interp val env sto)
       [(v*s v-val sto-val)
        (let ([l (new-loc sto-val)])
          (v*s (boxV l) (override-store (cell l v-val) sto-val)))])]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (new-loc [sto : Store]) : Location
  (+ (max-address sto) 1))

(define (max-address [sto : Store]) : Location
  (if (empty? sto)
      0
      (max (cell-location (first sto)) (max-address (rest sto)))))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(unboxE b)
     (type-case Result (interp b env sto)
       [(v*s v-b sto-b)
        (type-case Value v-b
          [(boxV l) (v*s (fetch l sto-b) sto-b)]
          [else (error 'interp "not a box")])])]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (fetch [l : Location] [sto : Store]) : Value
  (cond
    [(empty? sto) (error 'interp "segmentation fault")]
    [(equal? l (cell-location (first sto))) (cell-val (first sto))]
    [else (fetch l (rest sto))]))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(setboxE b val)
     (type-case Result (interp b env sto)
       [(v*s v-b sto-b)
        (type-case Value v-b
          [(boxV l)
           (type-case Result (interp val env sto-b)
             [(v*s v-val sto-val)
              (v*s v-val (override-store (cell l v-val) sto-val))])]
          [else (error 'interp "not a box")])])]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env] [sto : Store]) : Result
  (type-case Exp e
    ...
    [(beginE l r)
     (type-case Result (interp l env sto)
       [(v*s v-l sto-l) (interp r env sto-l)]
    ...
```

UNIVERSITÉ CÔTE D'AZUR

Simplifier la manipulation des résultats

- Exemple de l'addition.

```
(type-case Result (interp l env sto)
  [(v*s v-l sto-l)
   (type-case Result (interp r env sto-l)
     [(v*s v-r sto-r) (v*s (num+ v-l v-r) sto-r)])])
```

- Le motif apparaît à de très nombreuses reprises.

```
(type-case Result call
  [(v*s v-id sto-id) body])
```

- Peut-on simplifier son écriture ?

UNIVERSITÉ CÔTE D'AZUR

Simplifier la manipulation des résultats

- À la place de

```
(type-case Result call
  [(v*s v-id sto-id) body])
```

- On veut pouvoir écrire

```
(with [(v-id sto-id) call] body)
```

UNIVERSITÉ CÔTE D'AZUR

Simplifier la manipulation des résultats

- Utilisation de macro

```
(define-syntax-rule (with [(v-id sto-id) call] body)
  (type-case Result call
    [(v*s v-id sto-id) body]))
```

```
(type-case Result (interp r env sto-l)
  [(v*s v-r sto-r) (v*s (num+ v-l v-r) sto-r)])
```

```
(with [(v-r sto-r) (interp r env sto-l)]
      (v*s (num+ v-l v-r) sto-r))
```

UNIVERSITÉ CÔTE D'AZUR

Simplifier la manipulation des résultats

```
(type-case Result (interp l env sto)
  [(v*s v-l sto-l)
   (type-case Result (interp r env sto-l)
     [(v*s v-r sto-r) (v*s (num+ v-l v-r) sto-r)])])
```

```
(with [(v-l sto-l) (interp l env sto)]
      (with [(v-r sto-r) (interp r env sto-l)]
            (v*s (num+ v-l v-r) sto-r)))]
```

UNIVERSITÉ
CÔTE D'AZUR

ENREGISTREMENTS

UNIVERSITÉ CÔTE D'AZUR

Exemples en C

```
struct point { int x, y; }
struct point p;
p.x = 1;
p.y = 2;
printf("%d\n", p.x + p.y);
```

- Un enregistrement est une structure de données qui agglomère d'autres données.
- On peut accéder à ses **champs** et les modifier.

UNIVERSITÉ CÔTE D'AZUR

Grammaire

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

New !

New !

New !

UNIVERSITÉ CÔTE D'AZUR

Exemples

Définition et accès

```
{let {[r {record [x 1] [y 2]}]}
  {get r x}}
--> 1

{let {[r {record [x 1] [y 2]}]}
  {get r y}}
--> 2

{let {[r {record [x 1] [y {+ 2 3}]}]}
  {get r y}}
--> 5
```

UNIVERSITÉ CÔTE D'AZUR

Exemples

Création et manipulation

```
{let {[make {lambda {v}
              {record
                [x {+ v 1}]
                [y {+ v 2}]}}]}
  {get {make 3} x}}
---> 4

{record [x 1] [y 2]}
---> {x = 1 y = 2}

{set {record [x 1] [y 2]} x 3}
---> {x = 3 y = 2}
```

UNIVERSITÉ CÔTE D'AZUR

Représentation

Nouvelles expressions

```
(define-type Exp
  ...
  [recordE (fields : (Listof Symbol)) (args : (Listof Exp))]
  [getE (record : Exp) (field : Symbol)]
  [setE (record : Exp) (field : Symbol) (arg : Exp)])
```

Et nouvelle valeur

```
(define-type Value
  ...
  [recV (fields : (Listof Symbol)) (vals : (Listof Value))])
```

On utilise deux listes et non pas une liste de couple pour être cohérent avec l'implémentation que l'on fera des objets et des classes.

UNIVERSITÉ CÔTE D'AZUR

Analyse syntaxique

```
(define (parse [s : S-Exp]) : Exp
  (cond
    ...
    [(s-exp-match? `{record [SYMBOL ANY] ...} s)
     (let ([sl (s-exp->list s)])
       (recordE (map (lambda (l) (s-exp->symbol (first (s-exp->list l)))) (rest sl))
                (map (lambda (l) (parse (second (s-exp->list l)))) (rest sl))))]
    [(s-exp-match? `{get ANY SYMBOL} s)
     (let ([sl (s-exp->list s)])
       (getE (parse (second sl)) (s-exp->symbol (third sl))))]
    [(s-exp-match? `{set ANY SYMBOL ANY} s)
     (let ([sl (s-exp->list s)])
       (setE (parse (second sl)) (s-exp->symbol (third sl)) (parse (fourth sl))))]
    ... ))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(recordE fds args)
     (recV fds (map (lambda (expr) (interp expr env)) args))]
    [(getE rec fd)
     (type-case Value (interp rec env)
       [(recV fds vs) (find fd fds vs)]
       [else (error 'interp "not a record")])]
    [(setE rec fd arg)
     (type-case Value (interp rec env)
       [(recV fds vs) (recV fds (update fd (interp arg env) fds vs))]
       [else (error 'interp "not a record")])]))
```

UNIVERSITÉ CÔTE D'AZUR

Fonctions intermédiaires

Recherche

```
(define (find [fd : Symbol] [fds : (Listof Symbol)] [vs : (Listof Value)]) : Value
  (cond
    [(empty? fds) (error 'interp "no such field")]
    [(equal? fd (first fds)) (first vs)]
    [else (find fd (rest fds) (rest vs))]))
```

Mise à jour

```
(define (update [fd : Symbol] [new-val : Value]
                [fds : (Listof Symbol)] [vs : (Listof Value)]) : (Listof Value)
  (cond
    [(empty? fds) (error 'interp "no such field")]
    [(equal? fd (first fds)) (cons new-val (rest vs))]
    [else (cons (first vs) (update fd new-val (rest fds) (rest vs)))]))
```

UNIVERSITÉ CÔTE D'AZUR

Mode de mise à jour

```
{let {[r1 {record
           [x {+ 1 2}]
           [y {* 3 4}]}]}
  {let {[r2 {set r1 x 5}]}
    {+ {get r1 x} {get r2 x}}}}
--> 8
```

- L'instruction `set` crée un nouvel enregistrement sans modifier l'ancien.

- C'est ce qu'on appelle la **mise à jour fonctionnelle des enregistrements**. On dit aussi que les données sont **persistantes**, elle garde la même valeur tout au long du programme.

UNIVERSITÉ CÔTE D'AZUR

Mode de mise à jour

```
{let {[r1 {record
           [x {+ 1 2}]
           [y {* 3 4}]}]}
  {let {[r2 {set r1 x 5}]}
    {+ {get r1 x} {get r2 x}}}}
--> 10
```

- A l'inverse, si `set` modifie un champ par mutation, l'ancien enregistrement est impacté.
- C'est ce qu'on appelle la **mise à jour impérative des enregistrements**.

UNIVERSITÉ CÔTE D'AZUR

Enregistrements mutables

- Valeur

```
(define-type Value
  ...
  [recV (fields : (Listof Symbol)) (vals : (Listof (Boxof Value)))])
```

UNIVERSITÉ CÔTE D'AZUR

Enregistrements mutables

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(recordE fds args)
     (recV fds (map (lambda (expr) (interp expr env)) args))]
    [(getE rec fd)
     (type-case Value (interp rec env)
       [(recV fds vs) (find fd fds vs)]
       [else (error 'interp "not a record")])]
    [(setE rec fd arg)
     (type-case Value (interp rec env)
       [(recV fds vs) (recV fds (update fd (interp arg env) fds vs))]
       [else (error 'interp "not a record")])]))
```

UNIVERSITÉ CÔTE D'AZUR

Enregistrements mutables

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(recordE fds args)
     (recV fds (map (lambda (expr) (box (interp expr env))) args))]
    [(getE rec fd)
     (type-case Value (interp rec env)
       [(recV fds vs) (find fd fds vs)]
       [else (error 'interp "not a record")])]
    [(setE rec fd arg)
     (type-case Value (interp rec env)
       [(recV fds vs) (recV fds (update fd (interp arg env) fds vs))]
       [else (error 'interp "not a record")])]))
```

UNIVERSITÉ CÔTE D'AZUR

Enregistrements mutables

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(recordE fds args)
     (recV fds (map (lambda (expr) (box (interp expr env))) args))]
    [(getE rec fd)
     (type-case Value (interp rec env)
       [(recV fds vs) (unbox (find fd fds vs))]
       [else (error 'interp "not a record")])]
    [(setE rec fd arg)
     (type-case Value (interp rec env)
       [(recV fds vs) (recV fds (update fd (interp arg env) fds vs))]
       [else (error 'interp "not a record")])]))
```

On modifie le champ impacté sur place. La valeur renvoyée sera la nouvelle valeur du champ (en `plait`, c'est `Void`).

UNIVERSITÉ CÔTE D'AZUR

Enregistrements mutables

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(recordE fds args)
     (recV fds (map (lambda (expr) (box (interp expr env))) args))]
    [(getE rec fd)
     (type-case Value (interp rec env)
       [(recV fds vs) (unbox (find fd fds vs))]
       [else (error 'interp "not a record")])]
    [(setE rec fd arg)
     (type-case Value (interp rec env)
       [(recV fds vs)
        (let ([val (interp arg env)])
          (begin (set-box! (find fd fds vs) val) val))]
       [else (error 'interp "not a record")])]))
```

UNIVERSITÉ CÔTE D'AZUR

Enregistrements mutables

- Mais interpréter les mutations des enregistrements avec les boîtes (ou les variables) de `plait`, c'est tricher !

- Que faut-il faire ?
- Réintroduire la mémoire explicite et allouer un emplacement mémoire pour chaque champ.