

Paradigmes et Interprétation

Récurtivité et paresse

Julien Provillard

julien.provillard@univ-cotedazur.fr

RÉCURSIVITÉ

Définitions récursives

□ L'instruction `local` permet d'introduire des définitions récursives.

□ Définition classique

```
(local [(define x 1)]  
  (+ x 2))  
--> 3
```

Définitions récursives

□ L'instruction `local` permet d'introduire des définitions récursives.

□ Définition récursive

```
(local [(define (f x) (f x))]  
  (f 1))  
--> Boucle infinie
```

Définitions récursives

□ L'instruction `local` permet d'introduire des définitions récursives.

□ Définition récursive

```
(local [(define x x)]  
  x)
```

```
--> #<undefined>
```

□ On cherche à évaluer `x` pour définir `x`.

Depuis Racket 6.1, on ne peut plus reproduire ce comportement. À la place une erreur est renvoyée.

Définitions récursives

❑ L'instruction `local` permet d'introduire des définitions récursives.

❑ Définition récursive

```
(local [(define x (list x))]  
  x)  
--> (list #<undefined>)
```

❑ On cherche à évaluer `x` pour définir une liste contenant `x`.

❑ La valeur de `x` est réellement `#<undefined>`.

Définitions récursives

□ L'instruction `local` permet d'introduire des définitions récursives.

□ Définition récursive

```
(local [(define f (lambda (x) (f x)))]  
  (f 1))
```

--> Boucle infinie

□ La fonction `f` s'appelle elle-même, pourquoi n'y a-t-il pas de problème ici ?

□ Un lambda n'évalue pas son corps avant d'être appelé, `f` sera correctement défini à ce moment !

Définitions récursives

□ L'instruction `local` permet d'introduire des définitions récursives.

□ Définition récursive

```
(letrec ([f (lambda (x) (f x))])  
  (f 1))
```

--> Boucle infinie

□ On peut utiliser `letrec` à la place de `local`.

IMPLÉMENTATION MÉTA-CIRCULAIRE

Implémentation méta-circulaire

- ❑ On s'autorise le `letrec` de Racket pour implémenter `letrec`.
- ❑ L'idée principale est d'évaluer le membre droit du `letrec` dans un environnement qui contient déjà l'identifiant auquel il est lié.
- ❑ On doit donc utiliser une définition récursive.

Implémentation méta-circulaire

```
{letrec {[x 1]}  
  {+ x 2}}
```

```
(letrec ([val (interp (numE 1) (extend-env (bind 'x val) env))])  
  (interp (plusE (idE 'x) (numE 2))  
    (extend-env (bind 'x val) env)))  
--> (numV 3)
```

Implémentation méta-circulaire

```
{letrec {[f {lambda {x} {f x}}]}  
  {f 1}}
```

```
(letrec ([val (interp (lamE 'x (appE (idE 'f) (idE 'x)))  
                      (extend-env (bind 'f val) env))])  
  (interp (appE (idE 'f) (numE 1))  
    (extend-env (bind 'f val) env)))
```

expected a value from type Value, got: #<undefined>

❑ Pourquoi cette erreur ?

On évalue `val` ici alors qu'il n'est pas encore défini!

Implémentation méta-circulaire

```
{letrec {[f {lambda {x} {f x}}]}
  {f 1}}
```

❑ Quelle solution apporter ?

❑ Il faut différer l'évaluation dans l'environnement.

```
(letrec ([new-env (extend-env (bind 'f (lambda () val)) env)]
  [val (interp (lamE 'x (appE (idE 'f) (idE 'x)))
    new-env)])
  (interp (appE (idE 'f) (numE 1)) new-env))
--> Boucle infinie
```

Grammaire

```
<Exp> ::= <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {- <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {let {[<Symbol> <Exp>]} <Exp>}
        | {letrec {[<Symbol> <Exp>]} <Exp>}
        | {if <Exp> <Exp> <Exp>}
        | {lambda {<Symbol>} <Exp>}
        | {<Exp> <Exp>}
```

New !

Modification de l'environnement

- ❑ Les valeurs sont mises en attente dans le corps d'une fonction

```
(define-type Binding
  [bind (name : Symbol) (val : (-> Value))])
```

- ❑ Et forcées lors d'un appel à `lookup`

```
(define (lookup [n : Symbol] [env : Env]) : Value
  (cond
    [(empty? env) (error 'lookup "free identifier")]
    [(equal? n (bind-name (first env))) ((bind-val (first env)))] ; application
    [else (lookup n (rest env))]))
```

Implémentation

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(letrecE s rhs body)
     (letrec ([new-env (extend-env (bind s (lambda () val)) env)]
              [val (interp rhs new-env)])
       (interp body new-env))])
    ... ))
```


IMPLÉMENTATION PAR EXPANSION

Implémentation

□ De manière générale

```
{letrec {[name rhs]} body}
```

est équivalent à

```
{let {[name {mk-rec {lambda {name} rhs}}]} body}
```

et en réécrivant mk-rec

```
{let {[name {{lambda {body-proc}
  {let {[fX {lambda {fX}
    {let {[f {lambda {x} {{fX fX} x}}]}
    {body-proc f}}}}]}
  {fX fX}}]}
  {lambda {name} rhs}}]}
body}
```

IMPLÉMENTATION PAR MUTATION

Une approche plus classique

- ❑ Pour interpréter `fac`, il faut que l'identificateur `fac` soit présent dans l'environnement.

```
(letrec ([fac (lambda (n)
                (if (= n 0) 1 (* n (fac (- n 1)))))])
  (fac 6))
```

- ❑ Et bien, introduisons le!

```
(let ([fac 42])
  (begin
    (set! fac (lambda (n)
                (if (= n 0) 1 (* n (fac (- n 1)))))
    (fac 6)))
```

N'importe quelle valeur convient,
même `#<undefined>` !

Une approche plus classique

- ❑ Définir `letrec` par sucre syntaxique avec cette approche nécessite que notre langage contienne l'instruction `set!`.
- ❑ Mais pour implémenter cette approche dans un interpréteur, il suffit que le *langage d'implémentation* dispose de l'instruction `set!`.

Modification de l'environnement (encore)

- Les valeurs sont en attente de mutation dans les clôtures lexicales.

```
(define-type Binding
  [bind (name : Symbol) (val : (Boxof Value))])
```

- Après mutation, on les récupère dans `lookup`.

```
(define (lookup [n : Symbol] [env : Env]) : Value
  (cond
    [(empty? env) (error 'lookup "free identifier")]
    [(equal? n (bind-name (first env))) (unbox (bind-val (first env)))]
    [else (lookup n (rest env))]))
```

Implémentation

```
(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    ...
    [(letrecE s rhs body)
     (let ([b (box (undefV))])
       (let ([new-env (extend-env (bind s b) env)])
         (begin
           (set-box! b (interp rhs new-env))
           (interp body new-env))))])
    ... ))

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

PARESSE

Racket vs algèbre

□ En racket, on évalue les expressions de gauche à droite.

$$(+ (* 1 2) (- 4 3)) \longrightarrow (+ 2 (- 4 3)) \longrightarrow (+ 2 1)$$

□ En algèbre, on évalue les expressions dans l'ordre que l'on choisit.

$$(1 \times 2) + (4 - 3) \longrightarrow 2 + (4 - 3) \longrightarrow 2 + 1$$

ou

$$(1 \times 2) + (4 - 3) \longrightarrow (1 \times 2) + 1 \longrightarrow 2 + 1$$

Racket vs algèbre

□ En mathématiques, lorsqu'on voit

$$f(x, y) = x$$

$$g(x) = \dots$$

$$f(1, g(g(g(g(g(2))))))$$

□ On peut conclure directement que le résultat est 1. On ne se soucie pas de calculer les différents appels à g car la valeur n'est pas utilisée.

□ Pourquoi devrait-on le faire en programmation ?

Eviter les calculs inutiles

```
(define (f x y) x)
```

```
(define (ack m n)
  (cond
    [(zero? m) (+ n 1)]
    [(zero? n) (ack (- m 1) 1)]
    [else (ack (- m 1) (ack m (- n 1)))])))
```

```
(f 1 (ack 4 4))
```

❑ Temps d'attente **très long** pour arriver au résultat 1.

Eviter les calculs inutiles

```
(define (f x y) x)
```

```
(define (g x) (g x))
```

```
(f 1 (g 2))
```

❑ Ne donne jamais de résultat !

Eviter les calculs inutiles

```
(define (read-all-chars f)
  (let ([c (read-char f)])
    (if (equal? c eof)
        empty
        (cons c (read-all-chars f)))))
```

```
(define content (read-all-chars (open-input-file some-file)))
(if (equal? (first content) #\#)
    (do-something (rest content))
    (error 'parse "format exception"))
```

❑ Si on est dans le cas d'erreur, il n'est pas nécessaire d'évaluer entièrement `content`.

Listes 'infinies'

```
(define (numbers-from n)  
  (cons n (numbers-from (+ n 1))))
```

```
(define N (numbers-from 0))  
(list-ref N 1000)
```

- ☐ Le calcul de N ne termine pas !
- ☐ Pourtant, la seule chose qui nous intéresse, c'est son millième terme...
- ☐ Qui se calcule en temps fini !

Evaluation paresseuse

- ❑ Des langages comme C, Java ou Scheme sont dits **gloutons** (ou **stricts**) :
 - Une expression est évaluée dès qu'elle est rencontrée.
- ❑ Des langages comme Haskell sont dits **paresseux** :
 - Une expression n'est évaluée que si son résultat est nécessaire.
- ❑ Certains langages gloutons comme Caml ou Scheme ont des options pour autoriser l'évaluation paresseuse.

Evaluation paresseuse dans DrRacket

- ☐ Le package `plait` permet l'évaluation paresseuse des expressions via une option.
- ☐ Choisissez comme langage `#lang plait #:lazy`.
- ☐ Pour pouvoir déterminer si une expression a été évaluée, on utilise l'option de couverture syntaxique des tests.
- ☐ Dans la boîte de dialogue de **Sélectionner le langage...**
 - Dans la partie **Propriétés dynamiques** (en haut à droite), activez l'option **Couverture syntaxique de vos tests**.
- ☐ Après évaluation de la fenêtre de définition,
 - Si le code apparaît dans sa couleur normale, il a été évalué.
 - S'il apparaît en orange, il n'a pas été évalué.



example-lazy.rkt

Nos interpréteurs et la paresse

- ❑ Quels est l'impact de la paresse sur nos interpréteurs ?

- ❑ Exemple pour `ordresup.rkt`



`ordresup-lazy.rkt`

- ❑ On peut interpréter plus d'expressions, et de manière plus efficace !

Nos interpréteurs et la paresse

- ❑ Quels est l'impact de la paresse sur nos interpréteurs ?

- ❑ Exemple pour `letrec-mut.rkt`




`letrec-lazy.rkt`

- ❑ Comment résoudre le problème ?

- ❑ Il faut retirer les effets de bords et donc passer par le `letrec` de `plait`...
mais plus simplement qu'auparavant !

Modification de letrec-lazy.rkt

```
(define (interp e env)
  (type-case Exp e
    ...
    [(letrecE s rhs body)
     (letrec ([new-env (extend-env (bind s (interp rhs new-env)) env)])
       (interp body new-env))]
    ...
```



Plus besoin de différer explicitement.
L'évaluation paresseuse s'en charge.

Implémenter la paresse

□ On souhaite implémenter de manière paresseuse le langage de ordresup.rkt.

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

```
{{lambda {x} 1} {2 3}} → 1
{{lambda {x} x} {2 3}} → not a function
{let {[x {+ {lambda {y} 1} 2}]} 3} → 3
```

Implémenter la paresse

- ❑ Première possibilité : utiliser l'option `#:lazy`. C'est ce que l'on a fait pour `ordresup-lazy.rkt`.

```
(define (interp e env)
  (type-case Exp e
    ...
    [(appE f arg)
     (type-case Value (interp f env)
       [(closV par body c-env)
        (interp body (extend-env (bind par (interp arg env)) c-env))]
       [else (error 'interp "not a function")])]
    ... ))
```

- ❑ Mais cela ne nous apprend rien sur la paresse !

Implémenter la paresse

- ❑ Seconde possibilité : Implémenter l'interpréteur en `plait` et retarder explicitement les évaluations quand nécessaire.

```
(define (interp e env)
  (type-case Exp e
    ...
    [(appE f arg)
     (type-case Value (interp f env)
       [(closV par body c-env)
        (interp body (extend-env (bind par (delay arg env)) c-env))]
       [else (error 'interp "not a function")])]
    ... ))
```

Environnement : changements

- ☐ Seules deux expressions enrichissent l'environnement : la liaison local et l'application de fonction.
- ☐ Mais la liaison locale est désormais implémentée par sucre syntaxique à l'aide d'une application de fonction.
- ☐ L'environnement n'évolue que lors d'appels de fonction et ceux-ci retardent désormais l'évaluation de leur argument.
- ☐ L'environnement ne va donc plus contenir la valeur des arguments mais la promesse de calcul de ces valeurs.

Environnement : changements

□ Nouvelle représentation de l'environnement :

```
(define-type Binding  
  [bind (name : Symbol) (val : Thunk)])
```

```
(define-type Thunk  
  [delay (body : Exp) (env : Env)])
```


Implémenter la paresse

- ❑ Quand doit-on évaluer une promesse ?
- ❑ Quand on cherche sa valeur dans l'environnement. C'est-à-dire lorsqu'on interprète un identificateur.

```
(define (interp e env)
  (type-case Exp e
    ...
    [(idE s) (force (lookup s env))]
    ... ))
```

```
(define (force [t : Thunk]) : Value
  (type-case Thunk t
    [(delay e env) (interp e env))]))
```

Evaluation redondante

- ❑ Combien de fois évalue-t-on l'expression $\{+ \ 1 \ 2\}$ lorsqu'on évalue $\{\{\text{lambda } \{x\} \ + \ \{+ \ x \ x\} \ + \ x \ x\}\} \ + \ 1 \ 2\}$?
- ❑ En fait, à chaque fois que l'on cherche à évaluer l'identificateur x !
- ❑ On évalue plusieurs fois la même expression alors que le but de la paresse est de limiter le nombre d'évaluations !
- ❑ Il faut se souvenir du résultat d'une promesse si elle a déjà été forcée !

Mettre le résultat en cache

```
(define-type Thunk
  [delay (body : Exp) (env : Env) (mem : (Boxof (Optionof Value)))]])
```

```
(define (force [t : Thunk]) : Value
```

```
  (type-case Thunk t
```

```
    [(delay e env mem)
```

```
      (type-case (Optionof Value) (unbox mem)
```

```
        [(none) (let ([val (interp e env)])
```

```
          (begin
```

```
            (set-box! mem (some val))
```

```
            val))]
```

```
        [(some val) val]]))])
```

On se donne un emplacement mémoire pour stocker le résultat.

On n'évalue une promesse que si elle n'a pas déjà été forcée.


On mémorise alors le résultat.

Pour ne pas avoir à le réévaluer par la suite.

Mettre le résultat en cache

```

(define (interp [e : Exp] [env : Env]) : Value
  (type-case Exp e
    [(numE n) (numV n)]
    [(idE s) (force (lookup s env))]
    [(plusE l r) (num+ (interp l env) (interp r env))]
    [(multE l r) (num* (interp l env) (interp r env))]
    [(lamE par body) (closV par body env)]
    [(appE f arg)
     (type-case Value (interp f env)
       [(closV par body c-env)
        (interp body (extend-env (bind par (delay arg env (box (none)))) c-env))]
       [else (error 'interp "not a function")])]))
  
```



Terminologie

- ❑ **Appel par valeur** : correspond à l'évaluation gloutonne.
 - Racket, Java, C, Python
- ❑ **Appel par nom** : correspond à l'évaluation paresseuse sans mise en cache des résultats.
 - Inutilisé en pratique
- ❑ **Appel par nécessité** : correspond à l'évaluation paresseuse avec mise en cache des résultats.
 - Haskell, dialecte de Racket (`#lang plait` `#:lazy`)

Paresse et programmation à états

- ❑ La paresse et la programmation à états sont incompatibles.
- ❑ Dans la programmation à état, une **procédure** est une fonction qui ne renvoie pas de valeur. Elle est utilisée pour son effet de bord.
- ❑ L'évaluation paresseuse ne calcule que les valeurs utiles, en particulier les procédures ne sont pas évaluées.
- ❑ On peut dans tout de même avoir de la paresse ponctuellement, il faut alors la gérer explicitement. Par exemple, en Racket :

```
(lazy 1)           --> #<promise>  
(force (lazy 1))  --> 1
```