Paresse

Stratégies d'évaluation

```
(fun x -> x*x) (3+4)
```

- stratégie *innermost* réduire les *radicaux* les plus internes
- stratégie *outermost* réduire les radicaux les plus externes

Stratégies d'évaluation

```
(fun x -> x*x) (3+4)
```

- stratégie *innermost* réduire les *radicaux* les plus internes
- stratégie *outermost* réduire les radicaux les plus externes
- OCaml, SML: innermost
- Haskell privilégie le jeu au large: outermost
  - on n'exécute un calcul que si on en a besoin
  - on peut ainsi éviter des calculs inutiles `fst (e1, e2)`
- innermost et outermost donnent le même résultat

Stratégies d'évaluation

```
(fun x -> x*x) (3+4)
```

- stratégie *innermost* réduire les *radicaux* les plus internes
- stratégie *outermost* réduire les radicaux les plus externes
- OCaml, SML: innermost
- Haskell privilégie le jeu au large: outermost
  - on n'exécute un calcul que si on en a besoin
  - on peut ainsi éviter des calculs inutiles `fst (e1, e2)`
- innermost et outermost donnent le même résultat
  - en l'absence d'effets de bord
  - sur un calcul qui termine `(fun x -> 3)` $\mathtt{e}_{\mathrm{loop}}$
  - 'davantage' de calculs terminent en outermost

(Thm: si une exécution termine, outermost la trouve)

Innermost vs outermost

- outermost
  - moins de divergences
  - le calcul a lieu ‘quand il le faut’ (difficile à prévoir)
  - programmation naturelle avec des données infinies
- la stratégie outermost est adaptée dans le cadre de la programmation fonctionnelle *pure* (Haskell)
  - en Caml, les effets de bord sont déclenchés *à un moment donné*: cela s’accomode mal avec la paresse
  - la plupart des langages sont innermost
  - Algol: combinaison de outermost et références

(mais stratégie d’exécution bien spécifique)

Innermost vs outermost

- outermost
  - moins de divergences
  - le calcul a lieu ‘quand il le faut’ (difficile à prévoir)
  - programmation naturelle avec des données infinies
- la stratégie outermost est adaptée dans le cadre de la programmation fonctionnelle *pure* (Haskell)
  - en Caml, les effets de bord sont déclenchés *à un moment donné*: cela s’accomode mal avec la paresse
  - la plupart des langages sont innermost
  - Algol: combinaison de outermost et références

    (mais stratégie d’exécution bien spécifique)
- outermost implémentée naïvement duplique les calculs

```
(fun x -> x*x) (3+4)
```

Innermost vs outermost

- outermost
  - moins de divergences
  - le calcul a lieu ‘quand il le faut’ (difficile à prévoir)
  - programmation naturelle avec des données infinies
- la stratégie outermost est adaptée dans le cadre de la programmation fonctionnelle *pure* (Haskell)
  - en Caml, les effets de bord sont déclenchés *à un moment donné*: cela s’accomode mal avec la paresse
  - la plupart des langages sont innermost
  - Algol: combinaison de outermost et références

    (mais stratégie d’exécution bien spécifique)
- outermost implémentée naïvement duplique les calculs

```
(fun x -> x*x) (3+4)
```

paresse = outermost + partage

Le partage

- les expressions sont représentées par des *graphes* (et non des arbres)

$$\texttt{(fun x -> x*x) (3+4)} \rightsquigarrow (\bullet \times \bullet)\ \texttt{(3+4)}$$

graphe $\leftrightarrow$ partage $\leftrightarrow$ toute expression est évaluée au plus 1 fois

Le partage

- les expressions sont représentées par des *graphes* (et non des arbres)

  $$\texttt{(fun x -> x*x) (3+4)} \rightsquigarrow (\bullet \times \bullet)\ \texttt{(3+4)}$$

  graphe $\leftrightarrow$ partage $\leftrightarrow$ toute expression est évaluée au plus 1 fois

- structures cycliques et partage
  - soit la “définition Haskell”

    ```
    let rec iterate f x = x::(map f (iterate f x))
    ```

Le partage

- les expressions sont représentées par des *graphes* (et non des arbres)

  `(fun x -> x*x) (3+4)` $\leadsto$ $(\bullet \times \bullet)$ `(3+4)`

  graphe $\leftrightarrow$ partage $\leftrightarrow$ toute expression est évaluée au plus 1 fois

- structures cycliques et partage
  - soit la "définition Haskell"

    ```
    let rec iterate f x = x::(map f (iterate f x))
    ```
  - si on fait une définition locale...

    ```
    let iterate f x =
     let rec xs = x::(map f xs) in
    xs
    ```

    ou encore

    ```
    let iterate f x = xs
     where rec xs = x::(map f xs)
    ```

$\longrightarrow$ nécessité d'indiquer le partage

Réduction de graphes

- réduction de graphes: stratégie outermost avec partage
- les langages de programmation fonctionnelle *pure* exploitent souvent la *réduction de graphes* . . . c'est le cas de Haskell

Réduction de graphes

- réduction de graphes: stratégie outermost avec partage
- les langages de programmation fonctionnelle *pure* exploitent souvent la *réduction de graphes* ... c'est le cas de Haskell
- la *G machine*: machine abstraite développée pour la compilation de langages purement fonctionnels

  S. Peyton-Jones – *"The implementation of functional programming languages"*

  on utilise *un graphe* et *une pile*

Réduction de graphes

- réduction de graphes: stratégie outermost avec partage
- les langages de programmation fonctionnelle *pure* exploitent souvent la *réduction de graphes* . . . c'est le cas de Haskell
- la *G machine*: machine abstraite développée pour la compilation de langages purement fonctionnels

  S. Peyton-Jones – *"The implementation of functional programming languages"*

  on utilise *un graphe* et *une pile*
- Caml: machine abstraite plus 'traditionnelle', exploitant essentiellement une pile pour les appels de fonctions

Memoization

- digression: l'idée d'éviter de répéter un même calcul rappelle la technique de la *memoization* (tabulation)
  - stocker les valeurs déjà calculées pour ne pas les recalculer
  - utilisation typique: approches de type *diviser pour règner*

Memoization

- digression: l'idée d'éviter de répéter un même calcul rappelle la technique de la *memoization* (tabulation)
  - stocker les valeurs déjà calculées pour ne pas les recalculer
  - utilisation typique: approches de type *diviser pour règner*
- parlons d'abord de *récursion ouverte*
  - on commence par écrire la factorielle 'sans `rec`'

```
# let fact f n = if n<=1 then 1 else n*(f (n-1));;
val fact :  (int -> int) -> int -> int = <fun>
```

Memoization

- digression: l'idée d'éviter de répéter un même calcul rappelle la technique de la *memoization* (tabulation)
  - stocker les valeurs déjà calculées pour ne pas les recalculer
  - utilisation typique: approches de type *diviser pour règner*
- parlons d'abord de *récursion ouverte*
  - on commence par écrire la factorielle 'sans `rec`'

    ```
    # let fact f n = if n<=1 then 1 else n*(f (n-1));;
    val fact : (int -> int) -> int -> int = <fun>
    ```
  - puis on fait un nœud

    ```
    # let rec factorielle k = fact factorielle k;;
    val factorielle : int -> int = <fun>
    # factorielle 4;;
    - : int = 24
    ```

Memoization

- digression: l'idée d'éviter de répéter un même calcul rappelle la technique de la *memoization* (tabulation)
  - stocker les valeurs déjà calculées pour ne pas les recalculer
  - utilisation typique: approches de type *diviser pour règner*
- parlons d'abord de *récursion ouverte*
  - on commence par écrire la factorielle 'sans `rec`'

    ```
    # let fact f n = if n<=1 then 1 else n*(f (n-1));;
    val fact : (int -> int) -> int -> int = <fun>
    ```
  - puis on fait un nœud

    ```
    # let rec factorielle k = fact factorielle k;;
    val factorielle : int -> int = <fun>
    # factorielle 4;;
    - : int = 24
    ```
- on utilise ensuite cette idée pour mémoriser les calculs

  Démo `demo_memoization.ml`

La même chose, en version "directe"

```
let h = Hashtbl.create 23;;
let rec fib = function
  | 0 | 1 -> 1
  | n ->
      let x = try Hashtbl.find h (n-1)
      with Not_found ->
        let x' = fib (n-1) in Hashtbl.add h (n-1) x';x'
      and y = try Hashtbl.find h (n-2)
      with Not_found ->
        let y' = fib (n-2) in Hashtbl.add h (n-2) y';y'
      in x+y
```

NB: avoir la table `h` globale est mieux que d'en faire une à chaque fois

Retour à la paresse, Haskell

- comparons `fold_left` et `fold_right`

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> fold_left f (f accu a) l
```

```
let rec fold_right f l accu =
 match l with
 | [] -> accu
 | a::l -> f a (fold_right f l accu)
```

Retour à la paresse, Haskell

- comparons `fold_left` et `fold_right`

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> fold_left f (f accu a) l
```

```
let rec fold_right f l accu =
 match l with
 | [] -> accu
 | a::l -> f a (fold_right f l accu)
```

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> strict (fold_left f) (f accu a) l
```

en Haskell

Retour à la paresse, Haskell

- comparons `fold_left` et `fold_right`

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> fold_left f (f accu a) l
```

```
let rec fold_right f l accu =
 match l with
 | [] -> accu
 | a::l -> f a (fold_right f l accu)
```

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> strict (fold_left f) (f accu a) l
```

en Haskell

- `fold_left` est récursive terminale:

  si on est outermost, espace en $\mathcal{O}(n)$

Retour à la paresse, Haskell

- comparons `fold_left` et `fold_right`

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> fold_left f (f accu a) l
```

```
let rec fold_right f l accu =
 match l with
 | [] -> accu
 | a::l -> f a (fold_right f l accu)
```

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> strict (fold_left f) (f accu a) l
```

en Haskell

- `fold_left` est récursive terminale:

  si on est outermost, espace en $\mathcal{O}(n)$

- en Haskell, mot-clef `strict` pour "forcer" l'évaluation

  `strict f x`

  - on évalue l'argument avant de le passer à la fonction: innermost

Retour à la paresse, Haskell

- comparons `fold_left` et `fold_right`

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> fold_left f (f accu a) l
```

```
let rec fold_right f l accu =
 match l with
 | [] -> accu
 | a::l -> f a (fold_right f l accu)
```

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> strict (fold_left f) (f accu a) l
```

en Haskell

- `fold_left` est récursive terminale:

  si on est outermost, espace en $\mathcal{O}(n)$
- en Haskell, mot-clef `strict` pour “forcer” l’évaluation

  `strict f x`
  - on évalue l’argument avant de le passer à la fonction: innermost
  - calcul en place pour `fold_left` $\mathcal{O}(1)$
  - c’est “moral”: tirer parti de la récursivité terminale présuppose de maîtriser le flot du calcul, i.e., s’appuyer sur un ordre d’évaluation

Retour à la paresse, Haskell

- comparons `fold_left` et `fold_right`

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> fold_left f (f accu a) l
```

```
let rec fold_right f l accu =
 match l with
 | [] -> accu
 | a::l -> f a (fold_right f l accu)
```

```
let rec fold_left f accu l =
 match l with
 | [] -> accu
 | a::l -> strict (fold_left f) (f accu a) l
```

en Haskell

- `fold_left` est récursive terminale:

  si on est outermost, espace en $\mathcal{O}(n)$

- en Haskell, mot-clef `strict` pour "forcer" l'évaluation

  `strict f x`

  - on évalue l'argument avant de le passer à la fonction: innermost
  - calcul en place pour `fold_left` $\mathcal{O}(1)$
  - c'est "moral": tirer parti de la récursivité terminale présuppose de maîtriser le flot du calcul, i.e., s'appuyer sur un ordre d'évaluation

- idem `if.. then.. else`: flot du calcul *imposé*

Paresse en OCaml

- comment *geler* une expression? construction `lazy`
- fichier `lazy.ml`:

  (WARNING: some purple magic is going on here. Do not take this file as an example of how to program in Objective Caml. *)*

pourquoi?

Paresse en OCaml

- comment *geler* une expression? construction `lazy`
- fichier `lazy.ml`:

```
(* WARNING: some purple magic is going
on here.  Do not take this file as an
example of how to program in Objective
Caml. *)
```

pourquoi?

DÉMO `demo_lazy.ml`

Paresse en OCaml

- comment *geler* une expression? construction `lazy`
- fichier `lazy.ml`:

  (WARNING: some purple magic is going on here. Do not take this file as an example of how to program in Objective Caml. *)*

  pourquoi?

  Démo `demo_lazy.ml`
- `lazy` (1) type, (2) court-circuite l'évaluation
- un type de données 'infini': les flots
  Démo `demo_streams.ml`

Paresse, suite

- stratégies `(fun x -> (x*x)) (32*52)`
  - Caml est *innermost*: on évalue un argument avant de le passer à une fonction
  - paresse: *outermost* + partage (pour ne pas répéter les calculs)

L'infini paresseux

- en Haskell, pas de miracle, `loop 3` diverge
- en revanche, `(fun x -> 17) (loop 3)` $\rightsquigarrow$ `17`
  (on peut écrire `(loop 3)`, du moment qu'on n'y va pas)

L'infini paresseux

- en Haskell, pas de miracle, `loop 3` diverge
- en revanche, `(fun x -> 17) (loop 3)` $\rightsquigarrow$ `17`
  (on peut écrire `(loop 3)`, du moment qu'on n'y va pas)
- la paresse permet de manipuler naturellement des choses infinies
  - `let rec nats = 1::(List.map (fun x -> x+1) nats)`
    - infini $\leftrightarrow$ cyclicité
    - un calcul qui engendre une donnée infinie
    - ici le "`::`" protège des données qui sont encore des calculs

L'infini paresseux

- en Haskell, pas de miracle, `loop 3` diverge
- en revanche, `(fun x -> 17) (loop 3)` $\rightsquigarrow$ `17`
  (on peut écrire `(loop 3)`, du moment qu'on n'y va pas)
- la paresse permet de manipuler naturellement des choses infinies
  - `let rec nats = 1::(List.map (fun x -> x+1) nats)`
    - infini $\leftrightarrow$ cyclicité
    - un calcul qui engendre une donnée infinie
    - ici le "`::`" protège des données qui sont encore des calculs
  - le problème de Hamming: construire la liste d'entiers t.q.
    - la liste commence par 1
    - si x est dans la liste, alors aussi $2x$, $3x$ et $5x$
    - pas d'autre nombre

$\boxed{1,2,3,4,5,6,8,9,10,12,15,16,\ldots}$

Hamming – solution

```
let rec merge l1 l2 = match l1,l2 with
| (x::xs, y::ys) ->
    if x<y then x::(merge xs (y::ys))
    else if x=y then x::(merge xs ys)
    else y::(merge (x::xs) ys)
| [],_ -> l2
| _,[] -> l1
```

Hamming – solution

```
let rec merge l1 l2 = match l1,l2 with
| (x::xs, y::ys) ->
    if x<y then x::(merge xs (y::ys))
    else if x=y then x::(merge xs ys)
    else y::(merge (x::xs) ys)
| [],_ -> l2
| _,[] -> l1
```

```
let rec merge l1 l2 = match l1,l2 with
| (x::xs, y::ys) ->
    if x<y then x::(merge xs l2)
    else if x=y then x::(merge xs ys)        ← mieux
    else y::(merge l1 ys)
| [],_ -> l2
| _,[] -> l1
```

Infini en OCaml: calcul, données

- *calculs* infinis `let rec f x = x+(f (x+1))`
  `let rec f x = x::(f (x+1))`

Infini en OCaml: calcul, données

- *calculs* infinis `let rec f x = x+(f (x+1))`
  `let rec f x = x::(f (x+1))`
- *structures de données* infinies `let rec ones = 1::ones`
  - `List.hd ones`, `List.tl ones` $\rightsquigarrow$ ok
  - `List.length ones` $\rightsquigarrow$ divergence
  - `List.map (fun x->2*x) ones` $\rightsquigarrow$ la pile explose

Infini en OCaml: calcul, données

- *calculs* infinis `let rec f x = x+(f (x+1))`
  `let rec f x = x::(f (x+1))`
- *structures de données* infinies `let rec ones = 1::ones`
  - `List.hd ones`, `List.tl ones` $\rightsquigarrow$ ok
  - `List.length ones` $\rightsquigarrow$ divergence
  - `List.map (fun x->2*x) ones` $\rightsquigarrow$ la pile explose
- dans les deux cas on a une forme *gardée* de récursivité
  - garde = dans un cas `fun`, dans l'autre le constructeur
  - idée $\simeq$ similaire dans `fun () -> incr c`
- deux formes de calcul en Caml *(ôter la garde)*
  - application *(une fonction, son argument)*
    un calcul qui diverge est un calcul qui diverge
  - filtrage *(un motif, une valeur filtrée)*
    - on peut manipuler 'finiment' une structure 'infinie'
    - application d'une fonction $\neq$ application d'un constructeur
      à chaque pas de filtrage, on "déboîte" (au moins) un constructeur

Construire sans fin

- `let rec nats = 1::(List.map (fun x -> x+1) nats)`
  on peut appeler des fonctions au sein d'une définition de valeur récursive ($\neq$ Caml)

  (plus précisément: *fonction* calculant avec la valeur en train d'être définie, et *valeur* non fonctionnelle)
- l'application est liquide en Caml, elle est plus visqueuse en Haskell
- exemples de manipulation de structures infinies:
  - séries mathématiques (p.ex. développements limités)
  - processus interagissants
  - systèmes réactifs (cf. Lucid Synchrone)

Opter pour la paresse, ou pas

- avantages:
  - pas plus de pas de réduction qu'en innermost
  - éviter calculs inutiles / davantage de programmes terminent
  - la paresse est la bonne stratégie face à l'infini
    on décrit une structure de données infinie par l'intermédiaire d'un *processus* construisant progressivement la structure

Opter pour la paresse, ou pas

- avantages:
  - pas plus de pas de réduction qu'en innermost
  - éviter calculs inutiles / davantage de programmes terminent
  - la paresse est la bonne stratégie face à l'infini
    on décrit une structure de données infinie par l'intermédiaire d'un *processus* construisant progressivement la structure
- inconvénients:
  - questions d'espace moins bien maîtrisées
  - implémentation efficace difficile (filtrage, en particulier)
  - debugger?