la semaine prochaine, permutation entre cours d'algo et de prog

cours de prog: vendredi de 10h15 à 12h15 **EN AMPHI J**

cours de projet demain: salle E001

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
- 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)` e_{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)

- 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)

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

```
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?

DÉMO `demo_lazy.ml`

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

DÉMO `demo_streams.ml`