Paresse

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)
- 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

1,2,3,4,5,6,8,9,10,12,15,16,...

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))`
- *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

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?

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)