Slide 1

Programmation fonctionnelle

1 Introduction: Aperçu du paradigme fonctionnel

Programmation fonctionnelle = programmation *applicative* = programmation uniquement avec des fonctions et des valeurs

Paradigme *radicalement* différent du paradigme traditionnel (*impératif*):

- Tous les calculs sont faits uniquement en évaluant des *expressions* (au sens "pur", mathématique du terme, c'est-à-dire *sans effet de bord*)

$\Rightarrow$ Pas de notion de variable *modifiable*

$\Rightarrow$ Pas de *procédures*, pas d'objets

Slide 2

Programme impératif = programme caractérisé par un *état* implicite qui est modifié par des *instructions* avec des effets de bord:

```
fact = 1;
for[i = 1 to n]
  fact = fact * i;
```

Forme générale d'un programme fonctionnel = série de déclarations (constantes et fonctions) + une expression à évaluer

```
ident_1 = ...
ident_2 = ...
...
ident_k = ...
expression_a_evaluer
```

Note: une fonction, au sens mathématique du terme, n'est qu'une forme spéciale de constante!!

Slide 3

Impossible d'obtenir un langage qui soit plus expressif ou plus puissant en *éliminant* certaines constructions

Donc, les langages fonctionnels ont des caractéristiques additionnelles:

- Fonctions d'*ordre supérieur*: une fonction peut recevoir un argument ou retourner un argument qui est une fonction

  De façon plus générale: les fonctions sont des *citoyens de première classe* $\Rightarrow$ peuvent être manipulées de la même façon que les autres valeurs, transmises en argument, reçues en résultat, conservées dans des structures de données
- Utilisation du *pattern-matching*: la déclaration des différentes alternatives d'une fonction peut se faire en indiquant clairement les différents cas possibles, en fonction de la structure des arguments
- Utilisation de l'évaluation non-stricte et/ou paresseuse: les arguments effectifs lors d'un appel de fonction ne sont pas nécessairement évalués avant l'appel (*call-by-value*) comme dans les langages traditionnels mais soit en parallèle (évaluation indulgente), soit au besoin (évaluation paresseuse)

Slide 4

- Inférence de type: la plupart des langages *purement* fonctionnels (ce qui exclut LISP) sont des langages *fortement typés* et ce même s'il n'est pas nécessaire de spécifier les types $\Rightarrow$ le compilateur peut déduire (inférer) les types des expressions et vérifier que tout est valide.
- Polymorphisme: une fonction est dite *polymorphe* lorsqu'elle est associée à un type générique, qui décrit une *classe* de types possibles

  Exemple: la longueur d'une liste (ne dépend pas du type des éléments)

```
longueur :: [t] -> Int
longueur []       = 0
longueur (x : xs) = 1 + longueur xs
```

Slide 5

Pourquoi la programmation fonctionnelle est-elle intéressante?

- Style de programmation déclaratif, très près de ce qu'on écrirait en mathématiques
- Plus facile de raisonner à propos d'un programme, c'est-à-dire, de déterminer les propriétés satisfaites par le programme): on peut utilier le raisonnement par *substitution* (raisonnement *équationnel*): "*equals can be replaced by equals*", comme en mathématique

  Exemple: Soit `x = f a`. Alors, toutes les expressions suivantes, dans un langage *purement* fonctionnel, seront équivalentes:

```
x + x
2 * x
x + (f a)
(f a) + (f a)
```

  Contre-exemple dans un langage impératif:

```
int w = 0;
int f( int x ) { return( x  +  w++ ); }

f(0) != f(0)
```

Slide 6

- L'utilisation de fonctions d'ordre supérieur, du polymorphisme, du *pattern-matching*, de l'inférence de types et de l'évaluation non-stricte conduit à des solutions souvent simples, élégantes et *succinctes* $\Rightarrow$ excellent pour le *prototypage*

  Mais ...parfois difficile à débugger, parce que très difficile (impossible ;-(d'ajouter des traces: imprimer est un effet de bord et une expression *ne peut pas* avoir d'effets de bord

Quelques langages de programmation fonctionnels utilisés de nos jours:

- Lisp: langage *impur* (programmation impérative, OO, etc.)
- ML: langage strict quasi-pur (notion de `ref`érence)
- `Haskell`: langage purement fonctionnel non-strict paresseux
- `pH` (`parallel Haskell`): noyau purement fonctionel, non-strict mais non-paresseux, pour la programmation parallèle

Slide 7

2 Étude de cas: `Haskell`

`Haskell` ≈ standard *de facto* pour la programmation fonctionnelle pure non-stricte:

- Introduit à la fin des années 80 suite à la prolifération des langages fonctionnels purs et non-stricts (SASL, KRC, Miranda, LML, Hope, Alfl, Orwell)
- Défini par un groupe international de chercheurs qui ont voulu définir un langage moderne et complet ⇒ Langage complexe

Fonctions récursives simples et *pattern-matching*

Version classique de la fonction factorielle `fact`:

```
fact n = if n == 0
           then 1
           else n * fact (n-1)
```

Version avec *pattern-matching*:

Slide 8

```
fact 0 = 1
fact n = n * fact (n-1)
```

Version avec gardes:

```
fact n
| n == 0    = 1
| n > 0     = n * fact (n-1)
```

Slide 9

Fonctions avec arguments multiples: fonctions curryfiées

Diverses versions d'une fonction pour additionner deux nombres:

```
-- Version avec un argument (paire de nombres)
plus_paire :: (Int, Int) -> Int
plus_paire (x, y) = x + y

-- Version avec deux arguments
plus :: Int -> Int -> Int
plus x y = x + y

-- Diverses utilisations
x = plus_paire (2, 4)    -- (_, _) cree un tuple (paire)
x = plus 2 4             -- Prefixe, sans "(" ... ")"
x = 2 `plus` 4           -- Infixe
```

Slide 10

Exemple: les racines réelles d'un polynome du second degré

```
racines :: Float -> Float -> Float -> String
racines a b c
| discriminant > 0.0
    = "Deux racines: " ++ show r_plus ++ ", " ++ show r_moins
| discriminant == 0.0
    = "Une racine: " ++ show r0
| otherwise
    = "Aucune racine reelle"
    where
      discriminant = b^2 - 4.0*a*c
      r0 = -b / (2.0 * a)
      r_plus = (-b + sqrt(discriminant)) / (2.0*a)
      r_moins = (-b - sqrt(discriminant)) / (2.0*a)
```

Slide 11

Applications partielles d'une fonction avec plusieurs arguments:

```
plus1 :: Int -> Int
plus1 = plus 1

-- Utilisation de la forme prefixe pour un operateur infixe
plus2 = (+) 2
fois3 = (*) 3

plus1 2    -- plus1 2 == plus 1 2 == 1 + 2

plus2 3    -- plus2 3 == (+) 2 3 == 2 + 3 == 5
fois3 4    -- fois3 4 == (*) 3 4 == 3 * 4 == 12
```

Lambda-expressions

Une λ-expression sert à définir une fonction *anonyme*, donc à introduire une valeur qui est une fonction

- Une fonction qui reçoit un argument entier `x` et qui retourne la valeur entière suivante: `\x -> x+1`

Slide 12

- Une fonction qui détermine si un entier est nul: `\x -> x == 0`
- Une fonction qui reçoit deux entiers et qui retourne leur somme: `\x -> \y -> x + y`

Il est évidemment possible de définir des constantes qui seront associées à ces λ-expressions:

```
inc :: Int -> Int
inc = \x -> x + 1

estZero :: Int -> Bool
estZero = \x -> x == 0

plus :: Int -> Int -> Int
plus = \x -> \y -> x + y
```

Slide 13

Les structures de listes

Une liste est une séquence ordonnée d'éléments (≈ `sequence`s de `Spec`)

Si `T` est un type, alors l'expr. de type `[T]` dénote le type *liste de* `T`

```
-- Valeur :: Type
[1, 2, 3, 4, 1, 4] :: [Int]
[True]             :: [Bool]
['a', 'b', c']     :: [Char]
[]                 :: [t]    -- Pour un type t arbitraire
[2..5]             :: [Int]

-- Constructeur ":"
5 : [] == [5]
'a' : 'b' : 'a' : [] == ['a', 'b', 'a']
[1, 2] : [[3, 3], [5]] == [[1, 2], [3, 3], [5]]
```

Slide 14

Type de l'opérateur "`:`":

```
(:)  ::  t -> [t] -> [t]
```

L'ensemble des valeurs de type `[T]` est défini par induction comme suit:

- `[]` est un élément du type `[T]`
- Soient `v` une valeur de type `T`, `l` une valeur de type `[T]`. Alors on a que la valeur `v : l` est un élément du type `[T]`

En conséquence, les fonctions sur les listes sont souvent définies de façon inductive comme suit:

```
f :: [T] -> TypeResultat
f []       = ...            -- Cas de base.
f (x : xs) = ...            -- Cas inductif.
```

Slide 15

Quelques exemples de fonctions définies de façon inductive:

```
longueur :: [t] -> Int
longueur [] = 0
longueur (_ : xs) = 1 + longueur xs
                  -- "_" == "wildcard"

doubler :: [Int] -> [Int]
doubler [] = []
doubler (x : xs) = (2*x) : doubler xs
```

Autre opérateur important = concaténation de deux listes “++”:

```
[] ++ [10, 20] == [10, 20]
[10, 20] ++ [20, 30] == [10, 20, 20, 30]
```

Définition inductive de (++):

```
(++): [t] -> [t] -> [t]
(++)  []       l1 = l1
(++)  (x : xs) l1 = x : (xs ++ l1)
```

Slide 16

Autres exemples de fonctions sur les listes

Tri par insertion:

```
triInsertion :: [Int] -> [Int]
triInsertion []       = []
triInsertion (x : xs) = inserer x (triInsertion xs)

inserer :: Int -> [Int] -> [Int]
inserer v [] = [v]
inserer v (x : xs)
  | v <= x     = v : x : xs
  | otherwise  = x : (inserer v xs)
```

Slide 17

Compréhensions de listes

Approche différente, inspirée de la notation *ensembliste*, pour écrire des expressions avec des listes:

```
l1 = [ 5, 20, 7 ]

l2 = [ 2*x | x <- l1 ]      -- l2 == [10, 40, 14]

l3 = [ estPair x | x <- l1 ++ l2 ]
     where estPair x = x ‘mod‘ 2 == 0
     -- l3 == [False, True, False, True, True, True]

doubler :: [Int] -> [Int]
doubler  l = [ 2*x | x <- l ]
```

Slide 18

Fonctions d’ordre supérieur: `map`

Lorsqu’on examine le patron de récursion des fonctions suivantes, on s’aperçoit qu’il est semblable d’une fonction à une autre:

```
incrementer :: [Int] -> [Int]
incrementer [] = []
incrementer (x : xs) = (x+1) : incrementer xs

doubler :: [Int] -> [Int]
doubler [] = []
doubler (x : xs) = (2*x) : doubler xs

tripler :: [Int] -> [Int]
tripler [] = []
tripler (x : xs) = (3*x) : tripler xs
```

Slide 19

Il est possible de généraliser toutes ces formes en définissant une fonction plus générale qui reçoit en argument la fonction à appliquer sur chacun des éléments de la liste:

```
map :: (t -> t) -> [t] -> [t]
map f [] = []
map f (x : xs) = (f x) : map f xs

-- Ou encore:    map f l = [f x | l <- x]

incrementer x = x + 1
doubler     x = 2 * x

incrementer :: [Int] -> [Int]
incrementer l = map incrementer l

doubler :: [Int] -> [Int]
doubler l = map doubler l

tripler :: [Int] -> [Int]
tripler = map (\x -> 3*x)
```

Slide 20

Fonctions d'ordre supérieur: `filter`

Une liste peut être vue comme une forme *stream*. Une notion souvent utile sur un *stream* est celle d'un *filtre*, qui ne laisse passer que les éléments du *stream* qui satisfont une certaine propriété `p`.

```
filter :: (t -> Bool) -> [t] -> [t]
filter p [] = []
filter p (x : xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs

filter (\x -> x `mod` 2 == 0)  [10, 20, 3, 7, 0] == [10, 20, 0]
filter (\x -> x >= 10)         [10, 20, 3, 7, 0] == [10, 20]
```

Définition de `filter` avec une compréhension de listes:

```
filter p l = [ x | x <- l, p x ]
```

Note: Les expressions qui suivent "`,`" servent à sélectionner les items de `l` à être utilisés dans l'expression à gauche de "`|`".

Slide 21

Fonctions d'ordre supérieur: `fold`

Il est possible de trouver un patron commun à toutes ces fonctions:

```
longueur :: [t] -> Int
longueur []       = 0
longueur (x : xs) = 1 + longueur xs

somme :: [Int] -> Int
somme []       = 0
somme (x : xs) = x + somme xs

contientZero :: [Int] -> Bool
contientZero []       = False
contientZero (x : xs) = (x == 0) || contientZero xs

triInsertion :: [Int] -> [Int]
triInsertion []       = []
triInsertion (x : xs) = x `inserer` (triInsertion xs)
```

Slide 22

```
fold :: (t1 -> t2 -> t2) -> t2 -> [t1] -> t2
fold op valBase []       = valBase
fold op valBase (x : xs) = x `op` (fold op valBase xs)


longueur = fold f 0
             where f _ long = 1+long

somme = fold (+) 0

contientZero = fold (\x -> \y -> x == 0 || y) False

triInsertion = fold inserer []
```

De façon générale:

```
fold op b [x1, ..., xn]
== (op x1 (op x2 (op x3 ... (op xn b) ... )))
== x1 `op` (x2 `op` (x3 ...  `op` (xn `op` b)))
```

Slide 23

Autres exemples:

```
produit :: [Int] -> Int
produit = fold (*) 1

concatener :: [t] -> [t] -> [t]
concatener a b = fold (:) b a

doubler = fold (\x -> \xs -> (2*x) : xs) []
```

Si op est une opération associative:

```
fold op b [x1, ..., xn] == x1 ‘op‘ x2 ‘op‘ ... xn ‘op‘ b
```

Si op est une opération associative et commutative $\Rightarrow$ les opérations op peuvent être appliquées dans n’importe quel ordre, y compris en parallèle

Slide 24

Résolution de problème par une approche diviser-pour-régner

```
diviserPourRegner
     estSimple              -- (t_prob -> Bool)               ->
     resoudreProblemeSimple -- (t_prob -> t_soln)             ->
     decomposerProbleme     -- (t_prob -> [t_prob])           ->
     combinerSolutions      -- (t_prob -> [t_soln] -> t_soln) ->
     probleme               --  t_prob                        ->
                            --  t_soln
    = resoudreProbleme probleme
      where
      resoudreProbleme probleme
      | estSimple probleme = resoudreProblemeSimple probleme
      | otherwise          = combinerSolutions probleme sousSolutions
          where sousSolutions = map resoudreProbleme sousProblemes
                sousProblemes = decomposerProbleme probleme
```

Slide 25

Quelques exemples d'application:

```
quicksort :: [Int] -> [Int]
quicksort l = diviserPourRegner estVide vide partitionner combiner l
    where
     estVide l = l == []
     vide l = []
     partitionner (x : xs) = [ filter ((<=) x) xs,
                               [x],
                               filter ((>=) x) xs ]
     combiner probleme solutions = fold (++) [] solutions


fibo n = diviserPourRegner estCasBase un partitionner combiner n
    where
    estCasBase n = n <= 1
    un n = 1
    partitionner n = [n-1, n-2]
    combiner probleme solutions = fold (+) 0 solutions
```

Slide 26

L'opération `fold` avec une approche diviser-pour-régner:

```
fold op base l = diviserPourRegner estVide retournerBase split combiner l
    where
    estVide l = l == []
    retournerBase l = base
    split l = [take m l, drop m l]
              where m = (length l) `div` 2
    combiner probleme [x, y] = x `op` y

take :: Int -> [t] -> [t]
take n []       = []
take 0 l        = []
take n (x : xs) = x : (take (n-1) xs)

drop :: Int -> [t] -> [t]
drop n []       = []
drop 0 l        = l
drop n (x : xs) = drop (n-1) xs
```

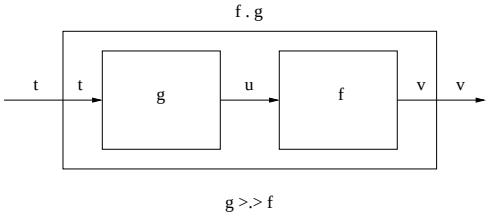
Slide 27

Composition de fonctions

Comme en mathématiques, il est possible de composer des fonctions:

```
(.) :: (u -> v) -> (t -> u) -> (t -> v)
(f . g) x = f (g x)

-- Composition dans l'autre sens, avec g avant f
infixl 9 >.>   -- "infixl" introduit un nouvel operateur infixe
(>.>) :: (t -> u) -> (u -> v) -> (t -> v)
(g >.> f) x = f (g x)
```



Donc:

```
g >.> f = f . g
```

Slide 28

Quelques exemples:

```
incrementer2 :: [Int] -> [Int]
incrementer . incrementer

quadrupler :: [Int] -> [Int]
quadrupler = doubler . doubler


deuxFois f = f . f

incrementer2' = deuxFois incrementer
quadrupler' = deuxFois doubler

nFois :: Int -> (t -> t) -> (t -> t)
nFois 0 f = id
nFois n f = f >.> nFois (n-1) f
```

Slide 29

Fonctions d'ordre supérieur: `pipeline`

On veut créer un pipeline à partir d'une liste de fonctions `f_1`, . . . , `f_n` avec les types suivants:

```
f_i: t -> t
```

On veut créer un pipeline de la forme suivante:

```
f_1 >.> f_2 >.> ... >.> f_n
```

Solution:

```
pipeline :: [t -> t] -> [t] -> [t]
pipeline [] l = l
pipeline (f : fs) l = pipeline fs (map f l)

pipeline [(+2), (*3), (-1)] [1..4]
== pipeline [(*3), (-1)] (map (+2) [1..4])
== pipeline [(*3), (-1)] [3, 4, 5, 6]
== pipeline [(-1)] [9, 12, 15, 18]
== pipeline [] [8, 11, 14, 17]
== [8, 11, 14, 17]
```

Slide 30

```
-- Autre solution equivalente ... mais plus esoterique
pipeline fs = fold (>.>) id (map map fs)

id :: t -> t
id x = x


pipeline [(+2), (*3), (-1)] [1..4]
== fold (>.>) id [map (+2), map (*3), map (-1)] [1..4]
== ((map (+2)) >.> (map (*3)) >.> (map (-1)) >.> id) [1..4]
== ((map (*3)) >.> (map (-1)) >.> id) [3, 4, 5, 6]
== ((map (-1)) >.> id) [9, 12, 15, 18]
== id [8, 11, 14, 17]
== [8, 11, 14, 17]
```

Slide 31

Évaluation paresseuse

Dans les langages impératifs traditionnels, lorsqu'on effectue un appel de fonction ou de procédure, soit l'argument est évalué *avant* l'appel (appel-par-valeur), soit l'adresse de l'entité à transmettre est déterminée avant l'appel (appel-par-référence)

Dans un langage fonctionnel *paresseux* (non-obstant les *optimisations* possibles) un argument n'est évalué que si il est *vraiment requis* = "appel par nécessité" (*call-by-need*)

En d'autres mots, une expression utilisée comme argument est évaluée 1 ou 0 fois, selon que l'argument formel correspondant est utilisé ou non

Un exemple simple: l'expression `f 0` n'est pas évaluée $\Rightarrow$ *pas* d'erreur de division par 0

```
f x  = 1 / x
g y  = 1

g (f 0) == 1
```

Slide 32

Autre exemple simple: le programme suivant *ne génère pas* de récursivité infinie, parce que l'appel à `f` à droite de "`:`" ne s'effectue que si on utilise la partie `tail` (le constructeur "`:`" est lui aussi paresseux):

```
head (x : _)  = x
tail (_ : xs) = xs

f x = x : f (x + 1)

head (tail (f 0)) == 1
```

L'évaluation paresseuse est utile:

- Pour la manipulation de structures de données (*potentiellement*) infinies
- Pour bien modulariser certains programmes en créant un couplage faible entre le producteur d'une structure de données et son consommateur

Slide 33

Exemple: Recherche de nombres premiers à l'aide du crible d'Eratosthène

```
premiers :: [Int]
premiers = cribleEratosthene [2..]
           where  cribleEratosthene (x : xs)
                      = x : (cribleEratosthene (filtrerMultiples x xs))

-- Supprime tous les multiples de p de la liste l.
filtrerMultiples :: Int -> [Int] -> [Int]
filtrerMultiples p l = [x | x <- l, not(x `mod` p == 0)]

-- Les 100 premiers nombres premiers
take 100 premiers

-- Tous les nombres premiers strictement inferieurs a 100.
takewhile (< 100) premiers

takewhile :: (t -> Bool) -> [t] -> [t]
takewhile p [] = []
takewhile p (x : xs)
  | p x       = x : takewhile p xs
  | otherwise = []

-- Tous les nombres premiers entre 1000 et 9999.
takewhile (<= 9999) (dropwhile (< 1000) premiers)
```

Slide 34

```
-- Idem, mais avec composition
((dropwhile (< 1000)) >.> (takewhile (<= 9999))) premiers

dropwhile :: (t -> Bool) -> [t] -> [t]
dropwhile p [] = []
dropwhile p (x : xs)
  | p x       = dropwhile p xs
  | otherwise = x : xs
```

Propriétés intéressantes:

- On peut travailler avec la liste (potentiellement) infinie de tous les nombres premiers
- Permet une modularisation élégante du problème: on peut *découpler* le processus de génération des nombres premiers du processus de sélection $\Rightarrow$ facile de changer le processus de sélection

Slide 35

Autre exemple: calcul de la racine carrée d'un nombre par la méthode de Newton-Raphson

La méthode de Newton-Raphson pour le calcul d'une racine carrée est une méthode *itérative* de calcul par *approximations successives*

Soit n un nombre pour lequel on veut calculer $\sqrt{n}$

Soit a_0, une approximation initiale (par ex., $(n+1)/2$).

Soit $a_{i+1} = (a_i + N/a_i)/2$.

Alors, la suite suivante *converge* vers $\sqrt{n}$:

$$a_0, a_1, a_2, \ldots, a_i, a_{i+1}, \ldots$$

Solution `Haskell`:

```
prochaineApprox n ai = (ai + (n / ai)) / 2

genererApproximations n = approximations
  where
    approximations = a0 : map (prochaineApprox n) approximations
    a0 = (n+1) / 2
```

Slide 36

```
-- Racine carree ou l'approximation est acceptee si l'erreur absolue
-- est plus petite ou egale a eps.
racineCarree n eps =
  erreurInf eps (genererApproximations n)

erreurInf eps (a0 : a1 : as)
| abs (a0 - a1) <= eps = a1
| otherwise            = erreurInf eps (a1 : as)
```

```
-- Racine carree ou l'approximation est acceptee si l'erreur
-- relative est plus petite ou egale a eps.
racineCarree n eps =
  erreurRelativeInf eps (genererApproximations n)

erreurRelativeInf eps (a0 : a1 : as)
| abs(a0 - a1) / (abs a1) <= eps = a1
| otherwise                      = erreurRelativeInf eps (a1 : as)
```

Slide 37

Les difficultés de l'inférence de type

Soit le programme suivant:

```
fold :: (t1 -> t2 -> t2) -> t2 -> [t] -> t2
fold op valBase []       = valBase
fold op valBase (x : xs) = x ‘op‘ (fold op valBase xs)
```

Le compilateur génère alors l'erreur indiquée plus bas. Quel est le problème?

```
Errors:
"folds.hs", line 2, [59] Bad restriction
    (a -> c -> c) -> c -> [d] -> c
of type
    (a -> b -> b) -> b -> [a] -> b
identified tyvars
 in  (\A1_fold -> \A2_fold -> \A3_fold ->
case A3_fold of {
  ([]) -> A2_fold
| (:) x xs -> A1_fold x (fold A1_fold A2_fold xs)
}
)::(b -> c -> c) -> c -> [d] -> c
 in fold
```

3 Étude de cas: pH

pH = *parallel Haskell*

= Haskell indulgent + I-structures + M-structures

⇒ Non-strict mais non-paresseux

Langage strict vs. non-strict

Slide 38

Langage strict = si un argument effectif n'est pas défini (par ex., erreur ou boucle), alors le résultat de la fonction n'est pas défini

Stratégie de mise en oeuvre = appel-par-valeur ⇒ on évalue les arguments *avant* l'appel, ensuite on appelle la fonction

```
f x = x + f (x+1)
f' x = 1 / 0
g x = 0

g (f 0)  -- ... => ne termine jamais!
g (f' 0) -- ... => erreur (division par 0)
```

Langage non-strict = il est possible pour une fonction de recevoir un argument qui n'est pas défini et de quand même produire un résultat

Slide 39

Deux grandes stratégies de mise en oeuvre:

1. Approche paresseuse: on appele immédiatement la fonction sans évaluer les arguments. On évalue un argument uniquement lorsqu'il devient requis dans la fonction (appel-par-nécessité)

   ⇒ Une expression utilisée comme argument doit être encapsulée dans une *suspension* (dans un *glaçon*). Lorsqu'un argument est utilisé, on évalue la suspension (on fait *fondre* le glaçon).

2. Approche indulgente: aucun ordre d'évaluation *a priori* n'est fixé, il suffit de respecter les dépendances de données (si une expression utilise X, on doit attendre que X doit disponible)

   ⇒ Les arguments et la fonction peuvent être évalués *en parallèle* et l'appelant et l'appelé se synchronisent (≈ producteur/consommateur)

Slide 40

Ce que permet l'évaluation non-stricte *sans* paresse

Définitions récursives de structures de données

```
>  two2 = (2, fst two2)
>
>  ones = 1 : ones
>
>  data Graph T = Node T [Graph T]
>
>  nodes = [
>    Node "S0" [nodes!!1, nodes!!2],
>    Node "S1" [nodes!!3],
>    Node "S2" [nodes!!0, nodes!!3],
>    Node "S3" []
>  ]
```

Slide 41

Structures de données pour programmation dynamique

Programmation dynamique ≈ on utilise une structure de données pour conserver des résultats intermédiaires de façon à éviter de recalculer plusieurs fois la solution à un même problème

```
> fib' n =
>   let
>     l = 0 : 1 : [l!!(i-1) + l!!(i-2) | i <- [2..n]]
>   in
>     l!!n
```

Traversée unique de structures de données

L'évaluation non-stricte permet, dans certains cas, d'effectuer *une seule* traversée d'une structure de données vs. deux pour une solution stricte

Exemple: Un programme pour trouver les éléments d'une liste d'entiers qui sont supérieurs à la moyenne: on utilise `theAvg`, qui est le résultat de l'appel de `findAboveAvg'`, comme argument dans l'appel de cette même fonction!

Slide 42

```
> findAboveAvg [] =
>  error "The list must not be empty."
> findAboveAvg xs =
>   let
>    (theAvg, aboveAvg)
>        = findAboveAvg' theAvg xs 0 0 []
>   in
>     aboveAvg
>
> findAboveAvg' theAvg [] total nb aboveAvg =
>   (total / nb, aboveAvg)
> findAboveAvg' theAvg (x : xs) total nb aboveAvg =
>  let
>    aboveAvg' = if x > theAvg
>                  then (x : aboveAvg)
>                  else aboveAvg
>  in
>   findAboveAvg'
>      theAvg xs (total + x) (nb + 1) aboveAvg'
```

Slide 43

Ce que permet l'évaluation paresseuse

- Structures de données potentiellement infinies
- Solutions modulaires permettant un couplage faible producteurs et consommateurs
- Structures de données à coûts amortis: le fait de retarder l'éva-luation de certaines expressions permet de borner le coût de certaines séquences d'opérations
- *Combinator parsing* = Solutions fonctionnelles élégantes pour l'écriture d'analyseurs lexicaux et syntaxiques

Slide 44

Progr. fonctionnelle parallèle implicite

Caractéristique intéressante d'un programme fonctionnel = implicitement parallèle en fonction des dépendances de donneées: comme il n'y a que des expressions (sans effets de bord), alors deux expressions qui n'ont aucune *dépendance de données* peuvent être évaluées en parallèle

Exemple: calcul des racines réelles d'un polynôme du second degré

```
racines a b c
| discriminant > 0.0
    = "Deux racines: " ++ show r_plus ++ ", " ++ show r_moins
| discriminant == 0.0
    = "Une racine: " ++ show r0
| otherwise
    = "Aucune racine reelle"
    where
      discriminant = b^2 - 4.0*a*c
      r0 = -b / (2.0 * a)
      r_plus  = (-b + sqrt(discriminant)) / (2.0*a)
      r_moins = (-b - sqrt(discriminant)) / (2.0*a)
```

Slide 45

Mais, tous les langages fonctionnels n'ont pas le même potentiel de parallélisme: la stratégie d'appel des fonctions a un impact majeur sur le parallélisme pouvant être obtenu:

- Évaluation stricte $\Rightarrow$ on doit attendre que les arguments soient prêts avant de commencer l'exécution du corps de la fonction
- Évaluation paresseuse $\Rightarrow$ on doit attendre jusqu'au dernier moment (lorsqu'on est certain qu'elle est requise) pour évaluer une expression
- Évaluation indulgente $\Rightarrow$ on peut évaluer les arguments et le corps de la fonction *en parallèle* en synchronisant l'appelant avec l'appelé uniquement aux points d'utilisation des arguments

Slide 46

Exemple: liste des valeurs conservées dans un arbre binaire

```
> data tree = Leaf Int | Node tree tree
>
> leaves a = leavesAccum a [];
>
> leavesAccum (Leaf i) leaves
>   = i : leaves
> leavesAccum (Node left right) leaves
>   = leavesAccum left (leavesAccum right leaves)
```

- Évaluation stricte: on doit attendre que le coté droit soit terminé pour commencer du coté gauche
- Évaluation paresseuse: rien en parallèle car on attend jusqu'au dernier moment
- Évaluation indulgente: les deux cotés de l'arbre peuvent être parcourus en parallèle

Slide 47

Sémantique formelle des modes d'évaluation

À première vue, il semble y avoir les regroupements suivants:

- Évaluation non-stricte: indulgent + paresseux
- Évaluation stricte: strict

Mais ... les divers modes d'évaluation fonctionnelle peuvent être formalisés, dans un même cadre *parallèle*, à l'aide du π-calcul $\Rightarrow$ les modèles sémantiques pour les trois modes d'évaluation sont très semblables et on a alors:

- Évaluation agressive des arguments: strict + indulgent
- Évaluation retardée des arguments: paresseux

Donc: indulgent $\approx$ strict + non-strict

4 Étude de cas: π-calcul

π-calcul = langage (très primitif) de description de processus concurrents et parallèles (introduit par Robin Milner)

Langage *primitif* (très peu de constructions) parce que l'objectif de Milner était d'en arriver à identifier l'*essence* de la concurrence $\Rightarrow$ Identifier le plus petit ensemble de concepts pouvant être utilisés pour modéliser n'importe quel système concurrent, y compris les systèmes dont la configuration évolue de façon dynamique (processus mobiles)

Slide 48

Caractéristiques du π-calcul:

- Les processus communiquent par l'intermédiaire de canaux de communication
- La seule chose qui peut être transmise sur un canal . . . *est un canal*
- Pas de récursion: processus itératif

Description d'une variante du π-calcul tirée de "*Non-strictness is not laziness*" (ceci *n'est pas* la version la plus compacte du π-calcul):

- Syntaxe = Figure 6 (p. 22) de
- Règles de congruence structurelle (équivalences syntaxiques) = Figure 7 (p. 22)
- Règles sémantique = Figure 8 (p. 23)

Slide 49

Jusqu'à présent, Milner semble avoir vu juste: un grand nombre de systèmes concurrents (simples et complexes) ont pu être décrits à l'aide du π-calcul:

- Langages fonctionnels parallèles
- Langages OO concurrents
- Divers langages parallèles
- Processus mobiles
- . . .