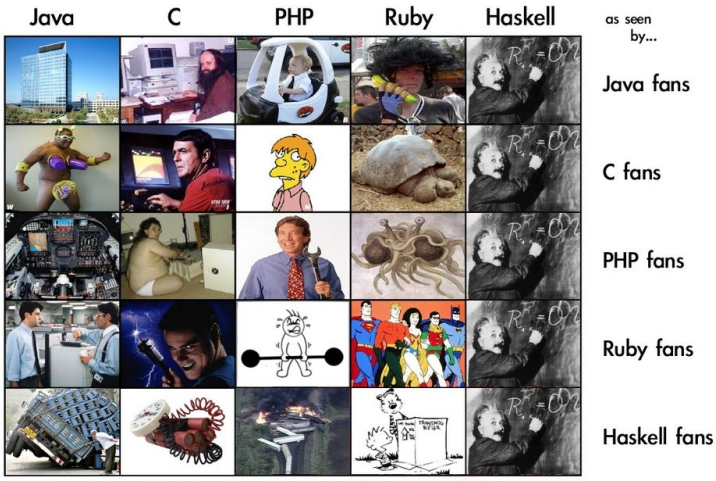
PF CM - Programmation fonctionnelle

2023-08-21

Contents

Généralités sur la PF



Intuition

- programmer à base de fonctions (pures)
- une fonction prend des paramètres et retourne un résultat
- une fonction peut prendre en paramètre des fonctions
- une fonction peut retourner une fonction

John Hughes, Why Functional Programming Matters

« Functional programming is so called because a program consists entirely of functions. The main program itself is written as a function which receives the program's input as its argument and delivers the program's output as its result. Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions until at the bottom level the functions are language primitives. »

Quelques paradigmes de programmation

paradigme	langage	principe
impératif	Fortran (1954)	exécuter des instructions successivement pour modifier l'état courant
fonctionnel	Lisp (1958)	appliquer des fonctions imbriquées sans effet de bord
objet	Smalltalk (1972)	faire interagir des briques logicielles représentant des concepts
. . .		

→ certains langages supportent plusieurs paradigmes

Langages fonctionnels

- intérêts : expressivité, découplage du code, réduction des sources d'erreurs, asynchronisme...
- applications : compilateurs, web back-end, scripts...
- caractéristiques : fonctions d'ordre supérieur, récursivité, listes...
- influence dans les langages "classiques" : Rust, Scala, JavaScript, C++, Java...
- langages fonctionnels : type Lisp, type ML

Langages fonctionnels de type Lisp

- exemples de langages : Common Lisp, Scheme, Emacs Lisp, Clojure...
- caractéristiques classiques :
  - expressions symboliques
  - syntaxe simple et élégante (si on aime les parenthèses)
  - homoïconicité
  - typage dynamique
- exemple de code en Common Lisp :

```
(defun factorielle (n)
    (if (= n 0)
        1
        (* n (factorielle(- n 1)))))
```

Langages fonctionnels de type ML

- exemples de langages : Haskell, OCaml...
- caractéristiques classiques :
  - types algébriques
  - inférence de types
  - filtrage par motifs
  - typage statique
- exemple de code en OCaml :

```
let rec factorielle = function
    | 0 -> 1
    | n -> n * factorielle (n - 1) ;;
```

Découverte d'Haskell

Présentation du langage

- langage de type ML créé en 1990 par un comité dédié
- normes Haskell 98, Haskell 2010
- caractéristiques remarquables :
  - purement fonctionnel
  - évaluation paresseuse
  - monades
- compilateur : GHC
- outils de gestion de projet : stack, cabal...
- utilisations : recherche académique, industrie, enseignement

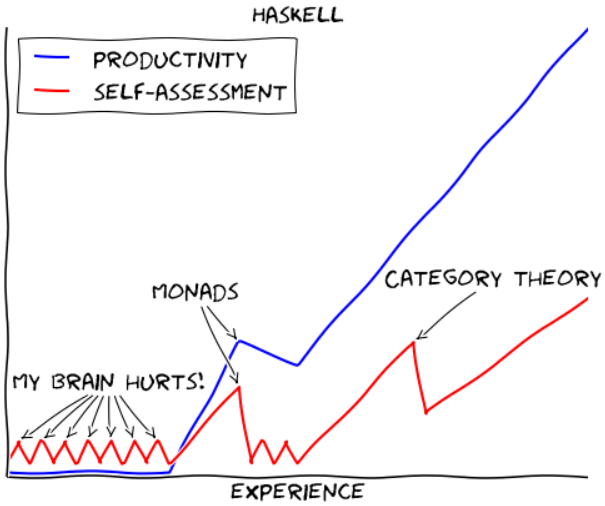


Figure 1: Learning Curves (for different programming languages)

Quelques liens

- site web officiel
- wikibook Haskell
- What I Wish I Knew When Learning Haskell
- State of the Haskell ecosystem
- Apprendre Haskell vous fera le plus grand bien !
- Real World Haskell
- Haskell for Imperative Programmers [video]

Exemples de code

- fonction de tri rapide :

```
quicksort [] = []
quicksort (p:xs) = quicksort lesser ++ [p] ++ quicksort greater
    where lesser = filter (< p) xs
          greater = filter (>= p) xs
```

- liste des nombres premiers :

```
primes = sieve [2..]
    where sieve (p:xs) = p : sieve [x | x <- xs, x `mod` p /= 0]
```

Exécuter du code, avec GHC

- trois possibilités :
  - compilation dans un fichier binaire (ghc)
  - compilation à la volée (runghc)
  - interpréteur interactif (ghci)

- example de fichier source `hello.hs` :

```
main = putStrLn "Hello"
```

- compilation dans un fichier binaire :

```
$ ghc -Wall hello.hs
[1 of 1] Compiling Main             ( hello.hs, hello.o )
Linking hello ...

$ ./hello
Hello
```

- compilation à la volée :

```
$ runghc -Wall hello.hs
Hello
```

- interpréteur interactif :

```
$ ghci
GHCi, version 8.6.5: http://www.haskell.org/ghc/  :? for help

Prelude> :load hello.hs
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, one module loaded.

*Main> main
Hello
```

Dans un projet Cabal + Nix

```
[monprojet]$ nix-shell

[nix-shell:monprojet]$ cabal build
...
Building executable 'monprojet' for monprojet-1.0..

[nix-shell:monprojet]$ cabal run monprojet
42

[nix-shell:monprojet]$ cabal test
...
1 of 1 test suites (1 of 1 test cases) passed.

[nix-shell:monprojet]$ exit
```

Figure 2: xkcd

Syntaxe de Base

Valeur, type, expression, fonction

- valeur : donnée concrête → `42`, `"toto"`...
- type : ensemble de valeurs, avec des propriétés/fonctionnalités communes → `Int`, `String`...
- expression : "morceau de code" pouvant être calculé (donc avec une valeur et un type) → `21*2`
- fonction : expression avec des paramètres (et éventuellement un nom) → `f x = x * 2`

Définir des "variables"

- associer un nom à une expression
- une variable ne peut pas être redéfinie
- le nom doit commencer par une minuscule

```
n = 42              -- definit une constante n
f x = 2 * x - 3     -- definit une fonction f
main = do
    print n         -- affiche la valeur de n
    print (f 4)     -- applique f sur 4 et affiche
```

Définir des variables locales

- avec le mot-clé `where` :

```
tropMaigre taille poids = imc < 18.5
    where imc = poids / t2
          t2 = taille * taille
```

- avec le mot-clé `let in` :

```
tropMaigre taille poids =
    let imc = poids / t2
        t2 = taille * taille
```

```
    in imc < 18.5
```

Indentation

- l'indentation compte en Haskell

```
...
    where t2 = taille * taille
          imc = poids / t2      -- ok
...
    where t2 = taille * taille
    imc = poids / t2            -- erreur d'indentation
```

- utiliser des espaces et non des tabs

Parenthèses et priorités

- les parenthèses servent uniquement à indiquer des priorités

```
Prelude> f x = 2 * (x - 3)
```

- contrairement à beaucoup de langages, les parenthèses ne servent pas à indiquer l'évaluation de fonction

```
Prelude> f 4
2
```

Commentaires

- deux types de commentaires :

```
-- commentaire de fin de ligne

{- commentaire
   multi-ligne {- imbricable -} -}
```

Mot-clés réservés

```
case         if          let
class        import      module
data         in          newtype
default      infix       of
deriving     infixl      then
do           infixr      type
else         instance    where
```

Programme principal

- `exemple1.hs` :

```haskell
main = do
    putStrLn "Entrez votre nom : "
    nom <- getLine
    let prefix = "Bonjour "
    putStrLn (prefix ++ nom ++ " !")
```

```
$ runghc exmple1.hs
Entrez votre nom : Roger
Bonjour Roger !
```

- exemple2.hs :

```haskell
import System.Environment
main = do
    args <- getArgs
    print args
```

```
$ runghc exemple2.hs blabla 42
["blabla","42"]
```

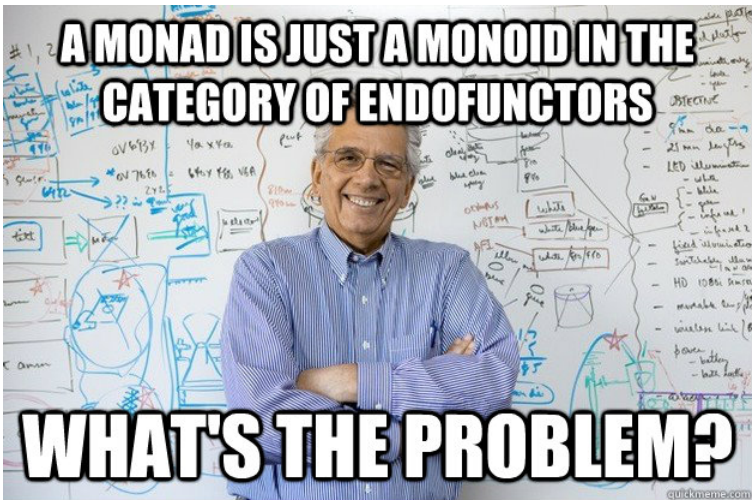
Transparence référentielle

- Propriété de transparence référentielle :

  on obtient un programme équivalent si on remplace une expression par une expression de même valeur

- un élément est “pur” s'il respecte la propriété de transparence référentielle
- donc un langage fonctionnel pur :
  - n'autorise pas d'effets de bord (variables mutables, boucles, entrées/sorties...)
  - n'a pas de “structure de contrôle” (`if-then`...)
  - ne permet pas de “debugger à coup de printf”

Système de types



Type de données

- ensemble des valeurs possibles et opérations autorisées
- toute donnée ou expression possède un type

Le typage en Haskell

- typage statique fort
- types polymorphes
- classes de types
- inférence de types
- types élémentaires (`Bool`, `Int`, `Double`…)
- types composés (tuples, listes…)

Typage faible/fort, statique/dynamique

- typage statique/dynamique :
  - statique : vérification de type à la compilation
  - dynamique : vérification de type à l'exécution
- typage faible/fort :
  - traduit l'exigence du compilateur lors de la vérification de type
- principales écoles :
  - typage statique fort (Haskell, Ada…) : fiabilité, performance
  - typage dynamique (Lisp, Python…) : flexibilité

Typage par inférence

- le compilateur détermine les types automatiquement (ou presque)
- mais on peut les écrire explicitement, et c'est la convention en Haskell :

```haskell
doubler :: Int -> Int
doubler x = 2 * x
```

- l'inférence est un calcul du type le plus général possible :

```
doubler :: Num a => a -> a
doubler x = 2 * x
```

(ne pas confondre avec la déduction de type, par exemple `auto` en C++)

- avec ghci, on peut demander le type d'une expression :

```
Prelude> :type abs
abs :: (Ord a, Num a) => a -> a

Prelude> :type 12.0
12.0 :: Fractional a => a
```

Types polymorphes

- une fonction peut être valide quelque soit le type de son paramètre
- Haskell permet de définir une fonction avec des types «polymorphes»
- et d'appliquer cette fonction pour différents types «concrets»
- notion différente du polymorphisme de la POO

```
Prelude> identite x = x

Prelude> :type identite
identite :: t -> t
-- fonction d'un type quelconque t vers t

Prelude> identite 4
4

Prelude> identite "blabla"
"blabla"
```

Classes de types

- une fonction peut être valide quelque soit le type de son paramètre, sous réserve qu'il supporte certaines opérations
- Haskell permet de définir des classes de types, c'est-à-dire des ensembles d'opérations que doivent supporter les types de la classe
- quelques classes prédéfinies : `Eq`, `Ord`, `Show`, `Read`, `Num`...
- un type peut appartenir à plusieurs classes
- notion différente des classes de la POO

```
Prelude> carre x = x * x

Prelude> :type carre
carre :: Num a => a -> a
-- fonction de a vers a où a est un type de classe Num

Prelude> carre 2
```

```
4

Prelude> carre 2.1
4.41
```

Quelques types élémentaires

- `Bool`
  - booléens : `True`, `False`
  - opérateurs booléens : `&&`, `||`, `not`

```
Prelude> True || False
True

Prelude> not False
True

Prelude> 3 > 7
False
```

- `Int, Integer`
  - nombres entiers : `42`, `-12`...
  - précision fixe, précision arbitraire
  - petit conseil : toujours mettre les nombres négatifs entre parenthèses
- `Float, Double`
  - nombres réels : `42.0`, `-3.2`...
  - simple précision, double précision

```
Prelude> 14 + 4 * (-7)
-14

Prelude> 2.0 + 3
5.0

Prelude> 7 / 3
2.3333333333333335

Prelude> 3::Float
3.0
```

- `Char`
  - caractères : `'a'`, `'\n'`...
- `String`
  - chaines de caractères : `"toto"`, `"a"`...
  - listes de `Char`

```
Prelude> "foo" ++ "bar"
"foobar"

Prelude> reverse "foobar"
```

```
"raboof"
```

Quelques classes de types

classe	fonctions de la classe
Eq	== /=
Ord	< <= > >= min max
Show	show
Read	read
Enum	succ pred
Bounded	minBound maxBound
Num	+ - * negate abs signum
Integral	div mod
Fractional	/ recip

Instances prédéfinies types-classes

	Eq	Ord	Num	Show	Read	Enum	autres
Bool	X	X		X	X	X	
Char	X	X		X	X	X	
String	X	X		X	X		
Int	X	X	X	X	X	X	Integral
Integer	X	X	X	X	X	X	Integral
Float	X	X	X	X	X	X	Fractional
Double	X	X	X	X	X	X	Fractional

```
Prelude> "toto" == "tata"
False

Prelude> show 12
"12"

Prelude> succ 'a'
'b'

Prelude> abs (-12.3)
12.3

Prelude> recip 2
0.5

Prelude> mod 1337 42
35
```

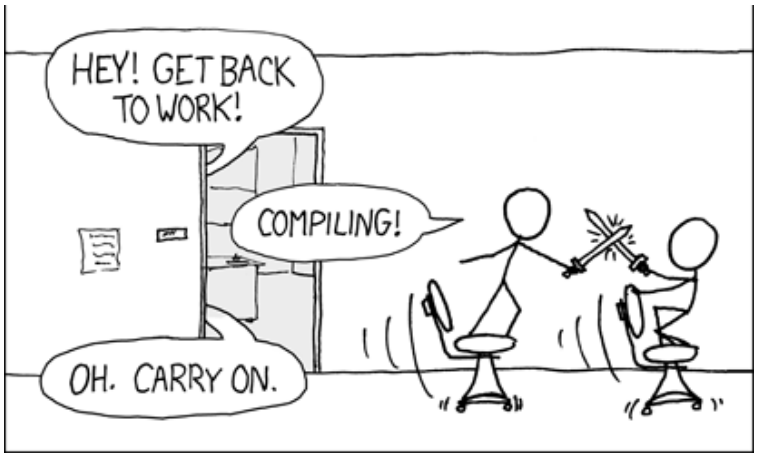
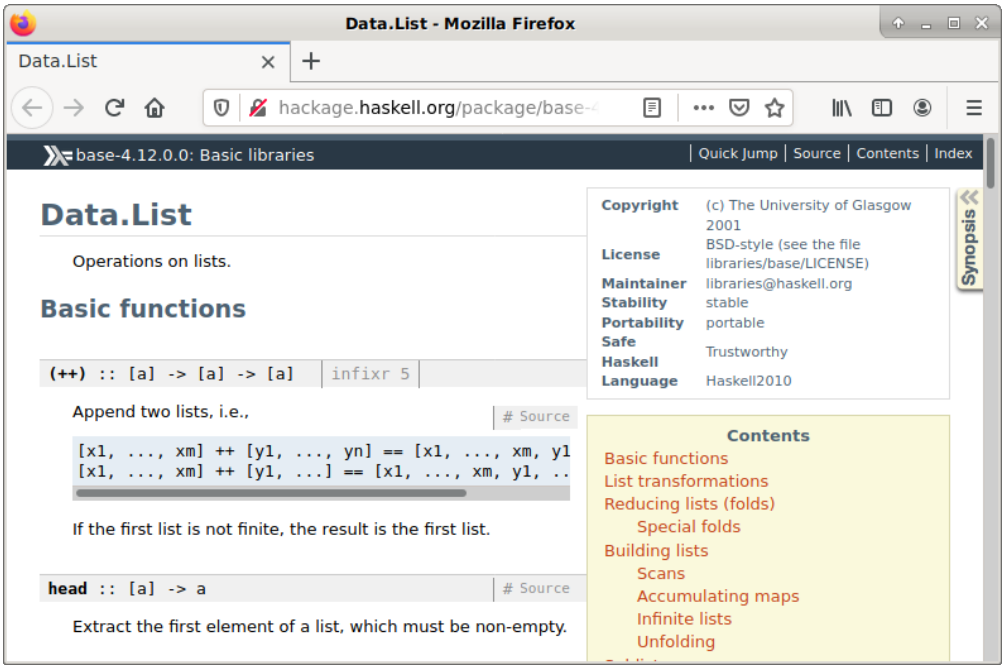


Figure 3: xkcd

Modules

Principe des modules

- module = fichier de code réutilisable
- bibliothèque = ensemble de modules
- bibliothèques communautaires: hackage
- exemple: module `Data.List` de la bibliothèque `base`
- permet la compilation séparée/incrémentale



Importer un module

- importer directement tout le module :

```
Prelude> import Data.List

Prelude Data.List> head "foobar"
'f'
```

- importer une partie du module :

```
Prelude> import Data.List (head)

Prelude Data.List> head "foobar"
'f'
```

- importer le module via un alias :

```
Prelude> import qualified Data.List as DL

Prelude DL> DL.head "foobar"
'f'
```

Définir un module

- déclarer le module avec le mot-clé `module`
- dans un fichier de même nom
- le nom d'un module doit commencer par une majuscule
- fichier `MyMath.hs` :

```
module MyMath where

mul2 :: Int -> Int
mul2 n = n * 2
```

- fichier `main.hs` :

```
import MyMath

main = print (mul2 21)
```

- exécution :

```
$ runghc main.hs
42
```

Entrées/sorties (IO)

Problématique des entrées/sorties

- exemple d'entrées/sorties : affichage écran, saisie clavier...

Figure 4: FunctionalGi

- problème : effet de bord, ne respecte pas la transparence référentielle
- par exemple, exécuter deux fois une fonction qui fait une saisie clavier peut donner deux résultats différents
- solution en Haskell : expliciter les fonctions qui font des entrées/sorties avec un type particulier (IO)

Le type “IO”

- exemple de “fonction pure” :

```
Prelude> :t negate
negate :: Num a => a -> a
```

- exemples de “fonction IO” :

```
Prelude> :t getLine
getLine :: IO String

Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

- type de la fonction `main` :

```
main :: IO ()
main = putStrLn "Hello World!"
```

Quelques fonctions de base

```
putStr :: String -> IO ()

putStrLn :: String -> IO ()
```

```haskell
print :: Show a => a -> IO ()

getLine :: IO String

getArgs :: IO [String]

getEnv :: String -> IO String

readFile :: FilePath -> IO String
```

La notation “do”

- permet de définir une séquence d’entrées/sorties
- mot-clé `do`, puis chaque ligne fait une entrée/sortie
- `<-` permet d’extraire la valeur d’une entrée/sortie
- `let` permet de définir une valeur pure
- (en vrai la notation `do` est plus générale que `IO`)
- `hello.hs` :

```haskell
main = do
    putStrLn "Qui est-ce ?"
    nom <- getLine
    let accueil = "Bonjour " ++ nom ++ " !"
    putStrLn accueil
```

- exécution :

```
$ runghc hello.hs

Qui est-ce ?
Julien
Bonjour Julien !
```

Mot-clé “return”

- permet d’encapsuler une valeur dans un `IO`
- différent du `return` de la plupart des langages : ne contrôle pas le flux d’instructions

```haskell
Prelude> :t return
return :: Monad m => a -> m a
```

- `mul2.hs` :

```haskell
mul2Log :: Int -> IO Int    -- car la fonction fait des IO
mul2Log n = do
```

```haskell
    let n2 = n * 2          -- n2 est un Int
    putStr "dans mul2Log: "
    print n2
    return n2               -- on retourne un IO Int

main = do
    x <- mul2Log 21
    putStr "dans main: "
    print x
```

- exécution :

```
$ runghc mul2.hs
dans mul2Log: 42
dans main: 42
```

Pattern matching

Expressions conditionnelles: if

- produit une valeur, selon 2 cas possibles
- dans les 2 cas, la valeur produite doit avoir le même type
- contrairement aux langages impératifs, le `else` est obligatoire

```haskell
main :: IO ()
main = do
    line <- getLine
    let x = read line :: Int
        absX = if x < 0 then -x else x
    print absX
```

Expressions conditionnelles : case

- produit une valeur selon un nombre quelconque de cas possibles

```haskell
main :: IO ()
main = do
    line <- getLine
    let str = case line of
                "0" -> "zero"
                "1" -> "un"
                _   -> "plusieurs"
    putStrLn str
```

Définition de fonction par des gardes

- même principe que le `case` mais pour définir une fonction
- le cas retenu est celui qui correspond à la première condition vérifiée

```haskell
import System.Environment

formatMessage :: [String] -> String
formatMessage args
    | l == 0 = "Erreur : indiquez votre nom"
    | l == 1 = "Salut " ++ head args ++ " !"
    | otherwise = "Erreur : trop de parametres"
    | l > 3 = "ce cas ne sera jamais atteint"
    where l = length args

main :: IO ()
main = do
    args <- getArgs
    putStrLn (formatMessage args)
```

Définition de fonction par pattern matching

- idem que les gardes (définir une fonction selon les cas possibles)
- mais en testant des motifs de paramètres (par déconstruction)

```haskell
import System.Environment

formatMessage :: [String] -> String
formatMessage [] = "Erreur : indiquez votre nom"
formatMessage ["julien"] = "Oh non, pas lui !"
formatMessage [x] = "Salut " ++ x ++" !"
formatMessage _ = "Erreur : trop de parametres"
formatMessage (x:xs) = "ce cas ne sera jamais atteint"

main :: IO ()
main = do
    args <- getArgs
    putStrLn (formatMessage args)
```

Récapitulatif

	définit quoi	quel test	combien de cas
`if`	expression	booléen	2
`case`	expression	valeur	n
gardes	fonction	booléen	n
pattern matching	fonction	valeur	n

Listes, tuples, Maybe

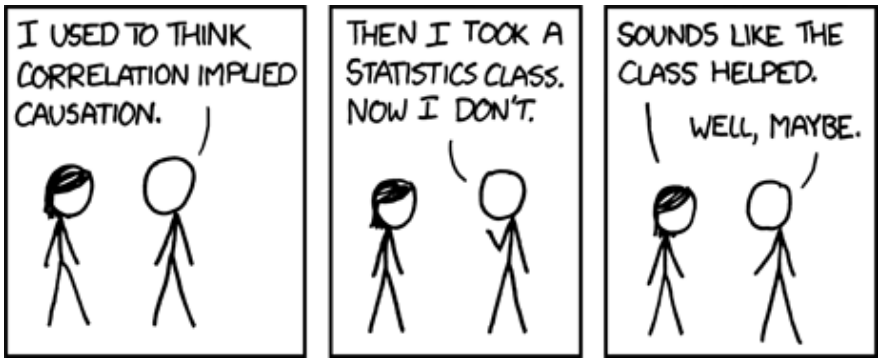


Figure 5: xkcd

Listes

- suite d'éléments de même type et de taille quelconque (éventuellement infinie)
- liste vide : `[]`
- opérateur de construction : `13:37:[]`
- syntaxe simplifiée : `[13,37]`
- fonctions prédéfinies : `head length null elem take drop`...
- opérateurs : `++ !!`
- exemples de listes :

```
Prelude> 1:2:3:[]           -- construction de liste
[1,2,3]

Prelude> [1,2,3]            -- construction simplifiée
[1,2,3]

Prelude> [1,2,3] ++ [4,5]   -- concaténation de deux listes
[1,2,3,4,5]

Prelude> :type [2.5, 12]    -- type d'une liste
[2.5, 12] :: Fractional t => [t]

Prelude> ['t','o','t','o']  -- liste/chaine de caractères
"toto"
```

```
Prelude> :type "toto"        -- type d'une chaîne de caractères
"toto" :: [Char]
```

```
Prelude> [1..3]              -- liste de nombres
[1,2,3]

Prelude> [0, -3 .. -10]      -- avec un pas donné
[0,-3,-6,-9]

Prelude> take 5 [0,-3..]     -- utilisation d'une liste infinie
[0,-3,-6,-9,-12]

Prelude> head [1..]          -- tête de liste
1

Prelude> tail [1..4]         -- queue de liste
[2,3,4]
```

- the list monster :

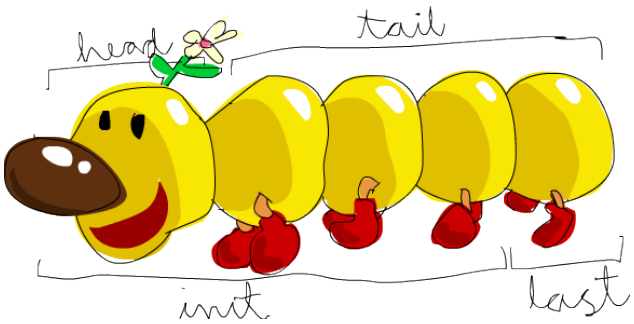


Figure 6: Miran Lipovača

- pattern matching de listes (avec une fonction) :

```
f :: [a] -> ...
f l = ...
f (x:xs) = ...
f (_:xs) = ...
f (x:_) = ...
f l@(x:xs) = ...
f (x1:x2:xs) = ...
```

- pattern matching de listes (avec des variables) :

```
l1 = [1..3]

[x1, x2, x3] = l1    -- définit x1, x2 et x3 à partir de l1

(x:xs) = l1          -- définit x et xs
```

Tuples (n-uplets)

- suite d'éléments de types éventuellement différents mais prédéfinis
- syntaxe : `(1, "toto", 4.2)`
- fonctions prédéfinies : `fst snd zip zip3`...
- exemples de tuples :

```
Prelude> ("toto", 42)
("toto",42)

Prelude> fst ("toto", 42)
"toto"

Prelude> :type ("toto", 42, True)
("toto", 42, True) :: Num t => ([Char], t, Bool)

Prelude> zip ["toto", "tata", "titi"] [42, 13, 37]
[("toto",42),("tata",13),("titi",37)]
```

- pattern matching de tuples (fonction) :

```
f :: (a,b) -> ...
f p = ...
f (x,y) = ...
f (_,y) = ...
f (x,_) = ...
f p@(x,y) = ...
```

- pattern matching de tuples (variables) :

```
t1 = (1, "toto", 4.2)

(e1, e2, e3) = t1

(x1, _, _) = t1
```

Le type Maybe

- permet de représenter une valeur optionnelle
- type (polymorphe) : `Maybe a`
- valeurs : `Nothing` ou `Just a`
- bibliothèque `Data.Maybe` : `maybe fromMaybe isJust`...
- exemples de Maybe :

```
Prelude> Just "foobar"
Just "foobar"

Prelude> :type Just "foobar"
Just "foobar" :: Maybe [Char]
```

```
Prelude> Just 42
Just 42

Prelude> :type Just 42
Just 42 :: Num a => Maybe a

Prelude> :type Nothing
Nothing :: Maybe a
```

```
import System.Environment
import Data.Maybe

-- fonction qui retourne l'éventuelle tête de liste
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:xs) = Just x

main = do
    args <- getArgs
    print (safeHead args)
```

- pattern matching de Maybe (fonction) :

```
f :: Maybe Int -> String
f Nothing  = "rien"
f (Just 0) = "zero"
f (Just 1) = "un"
f _        = "autre"
```

- pattern matching de Maybe (expression) :

```
x :: Maybe Int
x = Just 42

y :: Int
y = case x of
        Just n -> n
        Nothing -> 0
```

Fonctions récursives

Rappel sur les fonctions

- type et définition :

```haskell
mulHead :: Int -> [Int] -> Maybe Int
mulHead _ [] = Nothing
mulHead k (x:_) = Just (k*x)
```

- évaluation :

```
*Main> mulHead 2 []
Nothing

*Main> mulHead 2 [21, 3]
Just 42
```

Principe de la récursivité

- une fonction récursive est une fonction dont la définition utilise la fonction elle-même
- correspond à une définition par récurrence, par exemple :

$$n! = \begin{cases} 1 \text{ si } n = 1 \\ n \times (n-1)! \text{ sinon} \end{cases}$$

- permet d'implémenter des répétitions sans utiliser de boucle (et sans effet de bord)
- également utile pour faire des preuves d'algorithmes

Fonctions récursives en Haskell

- expressions conditionnelles :

```haskell
factorielle :: Int -> Int
factorielle n =
    if n == 1
        then 1
        else n * factorielle (n-1)
```

```haskell
factorielle :: Int -> Int
factorielle n =
    case n of
        1 -> 1
        _ -> n * factorielle (n-1)
```

- gardes :

```haskell
factorielle :: Int -> Int
factorielle n
    | n == 1 = 1
    | otherwise = n * factorielle (n-1)
```

- filtrage par motif :

```haskell
factorielle :: Int -> Int
factorielle 1 = 1
factorielle n = n * factorielle (n-1)
```

Récursivité sur des listes

- déconstruction de la liste via un filtrage par motif (ou autres)
- exemple (calcul de la taille d'une liste d'entiers) :

```haskell
taille :: [Int] -> Int
taille [] = 0
taille (_:xs) = 1 + taille xs

main = print (taille [13,37])
```

Récursivité terminale

- l'appel récursif fournit directement la valeur de retour
- intérêt : coût mémoire constant (pas d'empilement des appels récursifs)
- exemple de récursivité non-terminale (exemple précédent) :

```haskell
taille :: [Int] -> Int
taille [] = 0
taille (_:xs) = 1 + taille xs

main = print (taille [13,37])
```

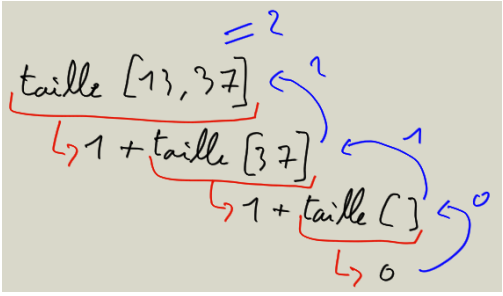


Figure 7: trace d'une fonction récursive non-terminale

- exemple équivalent en récursivité terminale :

```haskell
taille :: [Int] -> Int -> Int
taille [] n = n
taille (_:xs) n = taille xs (n+1)

main = print (taille [13,37] 0)
```

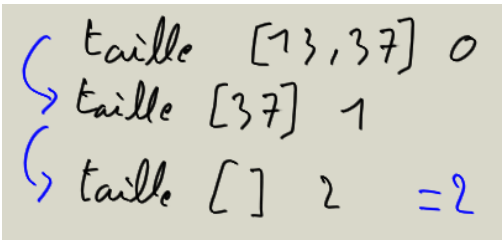


Figure 8: trace d'une fonction récursive terminale

Fonction auxiliaire

- problème : réécrire une fonction en récursivité terminale peut nécessiter un paramètre supplémentaire (et dont l'initialisation dépend de l'implémentation)
- solution : implémenter la fonction récursive terminale comme une fonction auxiliaire d'une fonction principale :

```haskell
taille :: [Int] -> Int
taille liste = tailleAux liste 0
    where tailleAux [] n = n
          tailleAux (_:xs) n = tailleAux xs (n+1)

main = print (taille [13,37])
```

Écrire une fonction récursive (ou pas)

1. écrire le type de la fonction :

```haskell
taille :: [Int] -> Int
```

2. énumérer les cas :

```haskell
taille [] =
taille (x:xs) =
```

3. écrire les cas triviaux :

```haskell
taille [] = 0
taille (x:xs) =
```

4. écrire les cas restants :

```haskell
taille [] = 0
taille (x:xs) = 1 + (taille xs)
```

5. simplifier, optimiser, généraliser :

```haskell
taille :: Num b => [a] -> b
taille = foldr (\_ n -> n+1) 0
```

Traitements de listes

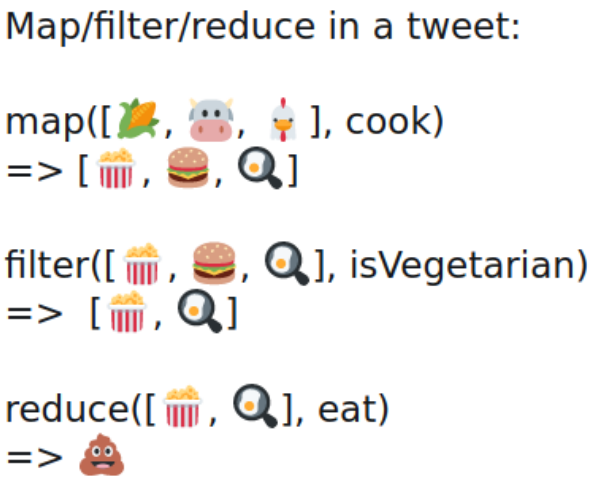


Figure 9: @steveluscher

Rappel sur les listes

- structure de données récursive (définie avec `[]` et `:`)
- traitement avec des fonctions récursives
- traitements types : mapping, filtrage, réduction

- déjà implémentés dans la lib de base de Haskell
- en programmation impérative, on utiliserait des boucles

Mapping de liste

- appliquer une fonction sur chaque élément d'une liste
- exemple dans ghci :

```
Prelude> map (*2) [1..5]
[2,4,6,8,10]

Prelude> map (\ x -> "toto " ++ show x) [13, 37, 42]
["toto 13","toto 37","toto 42"]
```

- exemple dans du code source :

```
fois2 :: Num a => [a] -> [a]
fois2 = map (*2)

main = print (fois2 [1..5])
```

- exemple avec une fonction récursive :

```
fois2 :: Num a => [a] -> [a]
fois2 [] = []
fois2 (x:xs) = (x*2):(fois2 xs)

main = print (fois2 [1..5])
```

Filtrage de liste

- sélectionner les éléments d'une liste qui vérifient un prédicat
- exemple avec ghci :

```
Prelude> filter (\ x -> x `mod` 2 == 0) [13, 37, 42]
[42]

Prelude> filter even [13, 37, 42]
[42]

Prelude> filter odd [13, 37, 42]
[13,37]
```

- exemple dans du code source :

```
pairs :: Integral a => [a] -> [a]
pairs = filter even
```

```haskell
main = print (pairs [1..5])
```

- exemple avec une fonction récursive :

```haskell
pairs :: Integral a => [a] -> [a]
pairs [] = []
pairs (x:xs) = if even x then x:(pairs xs) else pairs xs

main = print (pairs [1..5])
```

Réduction de liste

- faire un calcul avec les éléments d'une liste
- exemple avec ghci :

```haskell
Prelude> foldr (+) 0 [1..4]
10

Prelude> foldr (\ x acc -> show x ++ " " ++ acc) "" [1..4]
"1 2 3 4 "
```

- exemple dans du code source :

```haskell
somme :: Num a => [a] -> a
somme = foldr (+) 0

main = print (somme [1..5])
```

- exemple avec une fonction récursive :

```haskell
somme :: Num a => [a] -> a
somme [] = 0
somme (x:xs) = x + (somme xs)

main = print (somme [1..5])
```

- réductions prédéfinies : `sum` `product` `concat` `length` `minimum`...

Sens d'une réduction

- réduction depuis la droite :

```haskell
Prelude> foldr (:) [] [1..5]
[1,2,3,4,5]
```

- réduction depuis la gauche :

```
Prelude> foldl (flip (:)) [] [1..5]
[5,4,3,2,1]

-- flip échange les paramètres de (:)
```

Coût d'une réduction

- si l'opérateur est commutatif, on peut utiliser `foldr` ou `foldl` indifféremment mais...
- `foldr` est récursif non-terminal
- `foldl` est récursif terminal mais évalué paresseusement
- `foldl'` est récursif terminal à évaluation stricte

en général, utiliser `foldl'` ou `foldr`

Listes en compréhension (principe)

- syntaxe concise pour construire des listes :

```
[x/2 | x<-[1..4], even x]
```

- inspirée des math : $\{x/2 \mid x \in [\![1,4]\!], x \text{ pair}\}$
- n'ajoute pas vraiment de fonctionnalité mais très pratique
- trois composants : génération, mapping, filtrage

Listes en compréhension (génération)

- elles sont construites à partir de listes de base :

```
Prelude> [x | x<-[0,2..12]]
[0,2,4,6,8,10,12]
```

- produit cartésien :

```
Prelude> [(x,y) | x<-[1..2], y<-[1..2]]
[(1,1),(1,2),(2,1),(2,2)]
```

- référence vers des variables locales précédentes :

```
Prelude> [(x,y) | x<-[1..4], y<-[1..x]]
[(1,1),(2,1),(2,2),(3,1),(3,2),(3,3),(4,1),(4,2),(4,3),(4,4)]

Prelude>[x | xs<-["toto","tata"], x<-xs]
"tototata"
```

Listes en compréhension (mapping)

- une fonction est appliquée sur chaque élément généré :

```
Prelude> [2*x | x<-[1..6]]
[2,4,6,8,10,12]

Prelude> [(x,y,x+y) | x<-[1..2], y<-[1..2]]
[(1,1,2),(1,2,3),(2,1,3),(2,2,4)]

Prelude> [x+y | x<-[1..2], y<-[1..2]]
[2,3,3,4]

Prelude> [Data.Char.toUpper x | x<-"toto"]
"TOTO"

Prelude> [(Data.Char.toUpper x):xs | (x:xs)<-["toto","tata"]]
["Toto","Tata"]
```

Listes en compréhension (filtrage)

- un prédicat sélectionne les éléments générés :

```
Prelude> [x | x<-[1..6], even x]
[2,4,6]

Prelude> [x | x<-"tototata", elem x "aeiouy"]
"ooaa"

Prelude> [x | x<-[1..42], 42 `mod` x == 0]
[1,2,3,6,7,14,21,42]

Prelude> [(x,y,z) | x<-[1..7], y<-[x..7], z<-[y..7], x+y+z==7]
[(1,1,5),(1,2,4),(1,3,3),(2,2,3)]
```

Réimplémenter map

- avec une fonction récursive :

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = (f x) : map f xs
```

- avec une liste en compréhension :

```
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x<-l]
```

Réimplémenter filter

- avec une fonction récursive :

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter f (x:xs) = if f x then x:(filter f xs)
                         else filter f xs
```

- avec une liste en compréhension :

```haskell
filter :: (a -> Bool) -> [a] -> [a]
filter p l = [x | x<-l, p x]
```

Réimplémenter fold

- `foldr`, avec une fonction récursive :

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)
```

- `foldl`, avec une fonction récursive :

```haskell
foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

- avec une liste en compréhension : pas d'implémentation

Mapping et entrées/sorties

- `map` applique une fonction pure
- `mapM_` applique une action IO (dans un contexte IO) :
- exemple dans ghci :

```haskell
Prelude> mapM_ print [13,37]
13
37

Prelude> :t mapM_
mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
```

- exemple dans du code source :

```haskell
main :: IO ()
main = do
    mapM_ print [21, 7]
    print [21, 7]
    print (map (*2) [21, 7])
```

```
21
7
[21,7]
[42,14]
```

Fonctions d'ordres supérieurs

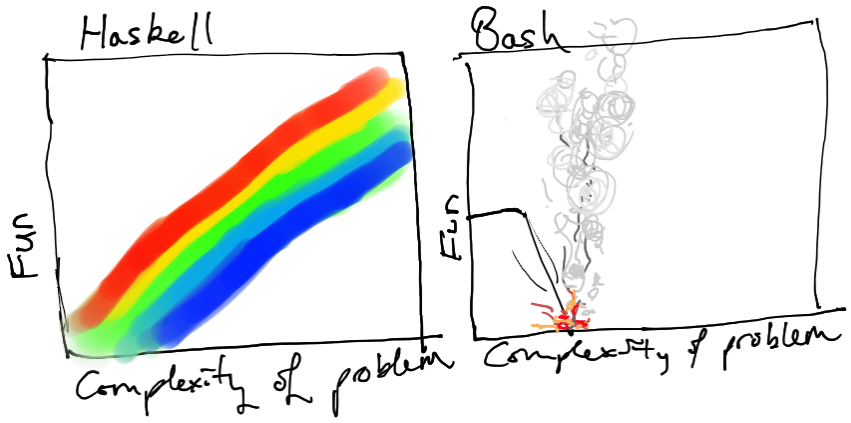


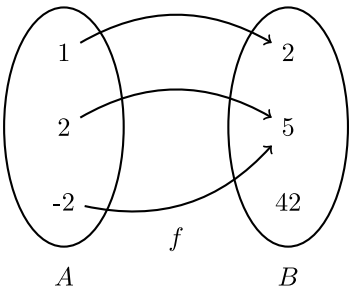
Figure 10: Functional programming in sh

Notion de fonction

- une fonction f d'un ensemble A vers un ensemble B, fait correspondre, à chaque élément $x \in A$, un élément $f(x) \in B$:
  - A est appelé le domaine de f
  - B est appelé le codomaine de f
  - $f(x)$ est appelé l'image de x par f
- notation :

$$\begin{aligned} f : \quad & A \to B \\ & x \mapsto f(x) \end{aligned}$$

- exemple :



Définir une fonction explicitement

- en math :

$$\begin{array}{rll} f : & \mathbb{Z} & \to \mathbb{N}^* \\ & 1 & \mapsto 2 \\ & 2 & \mapsto 5 \\ & -2 & \mapsto 5 \end{array}$$

- en Haskell :

```haskell
f :: Int -> Int
f 1 = 2
f 2 = 5
f (-2) = 5
```

Définir une fonction avec une expression

- en math :

$$\begin{array}{rl} f : & \mathbb{Z} \to \mathbb{N}^* \\ & x \mapsto x^2 + 1 \end{array}$$

- en Haskell :

```haskell
f :: Int -> Int
f x = x^2 + 1
```

Composition de fonctions

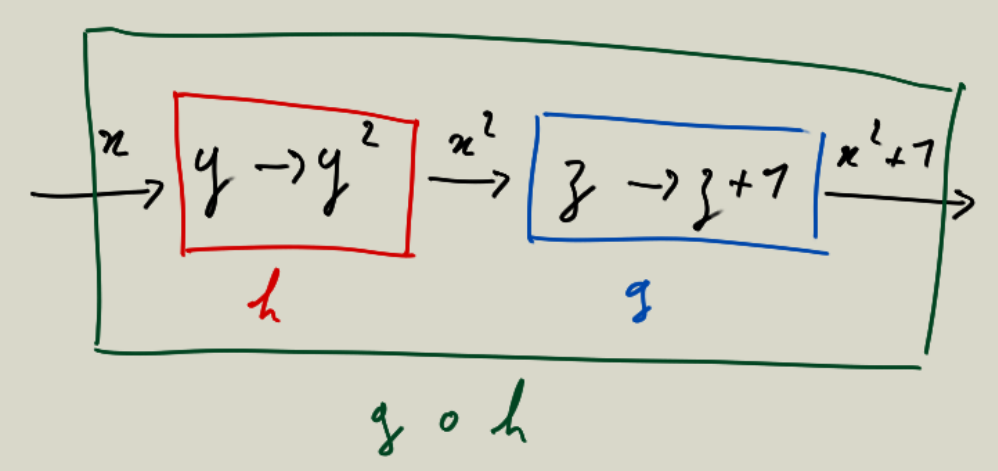
- en math :

$$\begin{array}{rl} h : & \mathbb{Z} \to \mathbb{N} \\ & y \mapsto y^2 \\ g : & \mathbb{N} \to \mathbb{N}^* \\ & z \mapsto z + 1 \\ f = & g \circ h \end{array}$$

- en Haskell :

```haskell
h :: Int -> Int
h y = y^2
g :: Int -> Int
g z = z+1
f = g . h
```

- en “boi-boites” :



Fonction anonyme

- on peut créer une fonction sans lui donner de nom
- simplifie parfois la syntaxe
- également appelé «lambda», en référence au lambda-calcul
- exemple :

```haskell
divisible2et3 :: [Int] -> [Int]
divisible2et3 xs = filter (\x -> even x && x `mod` 3 == 0) xs
```

Appliquer/évaluer une fonction

- en math : $y = f(2)$
- en Haskell : `y = f 2`
- transparence référentielle : appliquer une fonction sur un argument donné produit toujours le même résultat (pas d'effet de bord)

Type d'une fonction

- `fonction :: domaine -> codomaine`
- types concrets, types polymorphes, classes de types

```haskell
Prelude> :type not
not :: Bool -> Bool

Prelude> :type (\x -> x^2 + 1)
(\x -> x^2 + 1) :: Num a => a -> a

Prelude> :type 2
2 :: Num a => a

Prelude> :type (+)
(+) :: Num a => a -> a -> a
```

Fonction à plusieurs variables

- en Haskell, une fonction prend 1 argument et produit 1 résultat
- forme non curryfiée : on utilise un tuple

```haskell
add :: (Int,Int) -> Int
add (x,y) = x+y
```

- forme curryfiée : on retourne une fonction

```haskell
add :: Int -> (Int -> Int)
add x = (\y -> x + y)
```

- implémentations (curryfiées) équivalentes :

```haskell
add :: Int -> Int -> Int

add x = (\y -> x + y)

add = (\x -> (\y -> x + y))

add = (\x y -> x + y)

add x y = x + y

add = (+)
```

- curryfier ou décurryfier une fonction : `curry` `uncurry`
- en pratique : privilégier la forme curryfiée

Ordre d'une fonction

- ordre 0 : valeur simple

```haskell
f :: Int  -- "fonction" d'ordre 0 (i.e. une variable)
...
```

- ordre 1 : fonction qui retourne une valeur simple

```haskell
f :: Int -> Int  -- fonction d'ordre 1
...
```

- ordre n : fonction qui retourne une fonction d'ordre $n-1$

```haskell
f :: Int -> Int -> Int  -- fonction d'ordre 2
...
```

Application partielle

- appliquer une fonction «sur une partie des paramètres»
- retourne donc une fonction

```haskell
-- fonction "à 2 paramètres"
add :: Int -> Int -> Int
add x y = x + y

-- fonction "à 1 paramètre"
-- obtenue par application partielle de add
increment :: Int -> Int
increment y = add 1 y

-- autre implémetation possible :
increment' :: Int -> Int
increment' = add 1
```

Passer une fonction en paramètre

- opération très classique en programmation fonctionnelle :

```haskell
applyTwice :: (Int -> Int) -> Int -> Int
              -- ici, les parenthèses sont obligatoires
applyTwice f x = f (f x)
main = print (applyTwice (\x -> x*2) 3)
```

- autre implémentation :

```haskell
applyTwice :: (Int -> Int) -> Int -> Int
applyTwice f = f . f
main = print (applyTwice (*2) 3)
```

- équivalent en C++ :

```cpp
#include <functional>
#include <iostream>

int applyTwice(std::function<int(int)> f, int x) {
    return f(f(x));
}

int main() {
    std::cout << applyTwice([](int x){return x*2;}, 3)
              << std::endl;
    return 0;
}
```

Notion de fermeture de fonction

- éléments définissant la fonction mais non passés en paramètre

- exemple en Haskell :

```haskell
add :: Int -> Int -> Int
add x y = f y
    where f y' = x + y'   -- définition d'une fonction f
                          -- x est capturé dans la fermeture de f
```

- équivalent en C++ :

```cpp
int add(int x, int y) {
    // en C++, on indique la fermeture des lambda
    // ici, la lambda f capture la variable x par valeur
    std::function<int(int)> f = [x](int yp){return x + yp;};
    return f(y);
}
```

- dans un langage à effets de bord, une fermeture peut devenir invalide

```cpp
std::function<int(int)> makeDoubleur() {
    int x = 2;
    // ici on capture une variable locale par référence
    // la lambda utilisera une variable qui ne sera plus valide
    return [&x](int yp) { return x*yp; };
}
int main() {
    auto f1 = makeDoubleur();
    auto f2 = makeDoubleur();
    std::cout << f1(1) << " " << f2(1) << std::endl;
    return 0;
}
```

```
$ ./a.out
2 32639
```

- cette erreur est impossible dans un langage fonctionnel pur

Fonction totale, fonction partielle

- fonction totale : définie pour toute valeur de ses paramètres
- fonction partielle : non définie pour certaines valeurs
- par exemple, `head` est partielle :

```
Prelude> head [1..]
1

Prelude> head []
*** Exception: Prelude.head: empty list
```

- exemple de fonction partielle :

```haskell
head1 :: [a] -> a
head1 (x:_) = x
```

- exemple de fonction partielle, avec message d'erreur :

```haskell
head2 :: [a] -> a
head2 [] = error "empty list"
head2 (x:_) = x
```

- exemple de fonction totale :

```haskell
head3 :: [a] -> Maybe a
head3 [] = Nothing
head3 (x:_) = Just x
```

- Haskell ne vérifie pas la totalité
- mais il existe des bibliothèques explicitant ou évitant les fonctions partielles : RIO, Relude
- ainsi que des langages totaux inspirés de Haskell : Idris, Agda

Notation « point-free »

- sans expliciter les paramètres (points du domaine)
- exemple en notation classique :

```haskell
f :: Int -> Bool
f x =  2 * x > 42
```

- exemple en notation point-free :

```haskell
f :: Int -> Bool
f = (>42) . (2*)
```

Opérateurs d'évaluation

- `$` évalue (paresseusement) l'expression qui suit, par exemple :

```haskell
print (abs (negate 2))
print $ abs (negate 2)
print $ abs $ negate 2
print (abs $ negate 2)
```

- évaluation stricte avec `$!` (parfois plus performant)

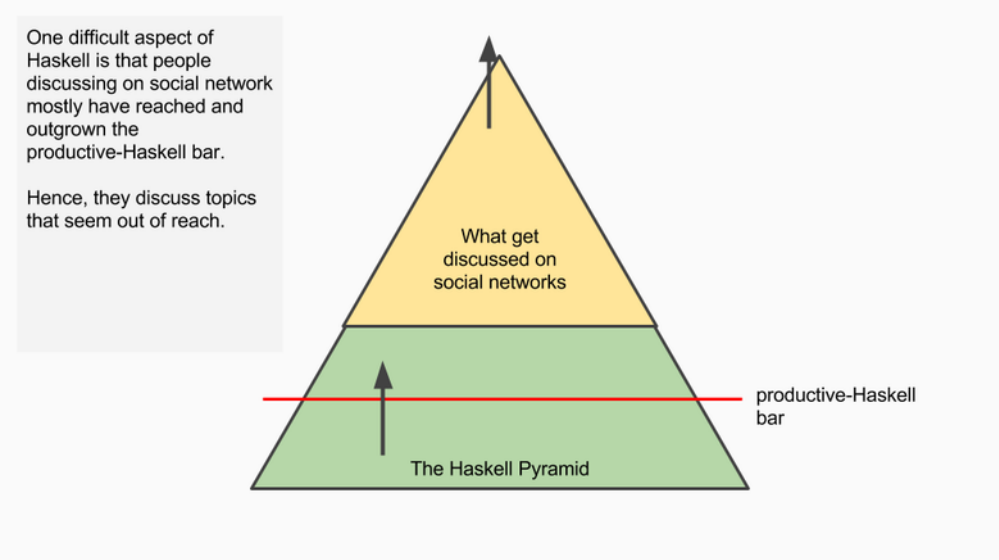


Figure 11: The Haskell Pyramid

Conclusion

Ce qu'on a abordé dans ce module

- notions de programmation fonctionnelle :
  - expressions, fonctions
  - fonctions d'ordres supérieurs
  - structures de données
  - ...
- découverte du système de types
- application en Haskell

Ce qu'on abordera dans le module PFA (peut-être)

- système de types (types algébriques, classes de types)
- applications (développement web, compilation...)

Une dernière remarque

> Haskell est utilisé dans l'enseignement et dans l'industrie mais aussi dans la recherche. Si dans la doc, vous lisez "intuitively a profunctor is a bifunctor where the first argument is contravariant and the second argument is covariant" mais que ça ne vous parait pas vraiment intuitif, **ce n'est pas grave**. Vous pouvez quand même utiliser et apprécier le langage.