Haskell — Paresse, ordre strict et typage fort

IFT359 — cinquième thème

Benoît Fraikin

Département d'informatique
UNIVERSITÉ DE SHERBROOKE

28 juin 2012

Plan de la séance

Chronologie

de 1987 à 1998 (Paul Hudak et al., 2007)

- Hudak et Peyton Jones en 87
- inspiré par Miranda (propriétaire)
- Haskell v1.0 en 1990 (v1.1 en 91 et v1.2 en 92)
- Haskell v1.3 en 1996 (v1.4 en 97)
- Haskell 98 (99 et 02)
- Haskell HP 2011

Quelques acteurs principaux

- Paul Hudak (Yale U.)
- Simon Peyton Jones (Microsoft Research, Glasgow U.)
- Richard Bird (Oxford U.)
- Simon Marlow (Microsoft Research)
- Philip Wadler (Edinburgh U.)
- Graham Hutton (Notthingham U.)
- Eric Meijer (Microsoft Research)
- John Hugues (Chalmers U.)
- Ralph Lämmer (U. Koblenz-Landau, ex. Microsoft Research)
- Don Stewart (Galois Inc., Standard Chartered Bank)

Liste

Haskell

- est fortement typé
- est un langage fonctionnel pur
- a une évaluation en ordre strict (paresse)
- dispose du polymorphisme ad-hoc (surcharge)
- utilise des fonctions curryfiées
- utilise des opérateurs infixes par défaut

Les opérateurs infixes

Au choix du développeur

- notation classique
- fonction préfixée et sans parenthèse
- peut-être modifié par soucis de clarté

Exemples

```
ghci > 3 + 2
5
ghci > (+) 3 2
5
```

```
ghci > map succ [1,2,3,4]
[2,3,4,5]
ghci > succ `map` [1,2,3,4]
[2,3,4,5]
```

La curryfication

Choix par défaut

- Opérateurs et fonctions curryfiées
- Permet une grande souplesse d'utilisation
- Produit cartésien possible, mais peu recommandé

Exemples

```
ghci > :t f
f :: String → Integer → Char
ghci > :t g
g :: String × Integer → Char
```


Le polymorphisme ad-hoc

Surcharge des fonctions

- Concept classique
- Utile avec le typage fort pour éviter les noms différents
- Non disponible dans des langages comme OCaml
- Diffère du polymorphisme générique

Exemples

```
ghci > :t (+)
(+) :: Num a ⇒ a → a → a
```

```
ghci > 2.3 + 4
6.3
ghci > 3 + 4
7
```


La pureté

Pas de forme impérative

- pas d'affectation
- pas de boucle itérative (`for`, `while`)
- « forme » plus mathématique $\Longrightarrow$ récursivité
- nécessite un mécanisme pour les entrées/sorties
- apparition des **monades** comme solution

La paresse

La paresse est

- une implémentation de l'évaluation en ordre strict (normal)
- évite le problème de multiplication des calculs (*boxed value*)
- augmente la mémoire utilisée
- est un choix pour obtenir la pureté d'un point de vue pratique
- impose de repenser les itérateurs (**`foldl`** et **`foldr`**)
- simplifie les choix cependant

Le typage statique fort

Une notion fondamentale

- nécessaire pour comprendre comment développer
- fiabilité accrue
- efficacité en mémoire accrue à l'exécution
- impose de penser et comprendre les types avant tout
- pas d'indication explicite (la plupart du temps)
- gestion efficace par inférence de type

Développement

Outils

1. le compilateur : GHC
   - YHC
   - `http://codepad.org/`
2. l'interpréteur : GHCi
   - Hugs
   - `http://tryhaskell.org/`
3. la plateforme — piles inclus : Haskell Platform

Comprendre les types pour compiler

On retrouve

1. des types de bases
2. le type produit
3. le type liste
4. les types fonctions
5. les types génériques
6. les classes de types

Types et constructeurs de bases

Types internes

- **Bool**, **Char**, **Integer**
- **Int**, **Float**
- **String** (liste de **Char**)

Constructeurs primitifs

- les couples
- les listes

Exemple

```
indent = 4          :: Integer
space = 1.5         :: Double

b = ('a',1)         :: (Char,Integer)
key = fst b         :: Char

liste = [1,2,3,4] :: [Integer]
un = head liste   :: Integer

name = "Benoit"   :: String
```


Le type d'une fonction

Fonctions curryfiées

1. transformateur de types
2. exemple `a → b → c` (se lit `a → (b → c)`)
3. isomorphe à `(a,b) → c`

Exemple

```
add :: Integer → Integer → Integer
add x y = x + y

add' :: (Integer,Integer) → Integer
add' (x,y) = x + y
```


La généricité

Question

Quel est le type de la fonction **head** ?

des listes différentes

```
ghci > head [1,2,3]
1
ghci > head "abc"
'a'
```

Exemples

```
fst :: (a,b) → b
head :: [a] → a
take :: Int → [a] → [a]
zip :: [a] → [b] → [(a,b)]
id :: a → a
```


La généricité

Le polymorphisme paramétrique

- Le « vrai » polymorphisme
- La fonction doit pouvoir gérer tout type
- Utilisation dans le typage de *wild-cards* : `a`, `b`

Le terrible monomorphisme

- Limite certaines fonctions à un type fixe
- Pour des raisons d'efficacité
- `f = show` aura le type imposé par la 1er utilisation de `f`
- Préférence pour `f x = show x`

Écriture d'une fonction

Style

1. déclaratif
2. utilisation de *pattern matching*

Exemple

```
map :: (a → b) → [a] → [b]
map f []     = []
map f (x:xs) = f x : map f xs
```

Le *pattern matching*

Liaison avec *as*

```
f pivot []                    = [pivot]
f pivot l@(x:xs) | x == pivot = l
f pivot (x:xs)   | x > pivot  = f pivot xs
f pivot (x:xs)   | otherwise  = pivot : f x xs
```

wild-cards

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

Expression *case*

```
take n xs = case (n,xs) of
                (0,_)     → []
                (_,[])    → []
                (n,x:xs)  → x : take (n-1) xs
```


Limite de la généricité

Opérateurs arithmétiques

1. Doivent fonctionner pour les réels et les entiers
2. Deux types différents
3. Ne peux être généraliser à aucun type
4. Solution : polymorphisme ad-hoc ou surcharge

Typage de +

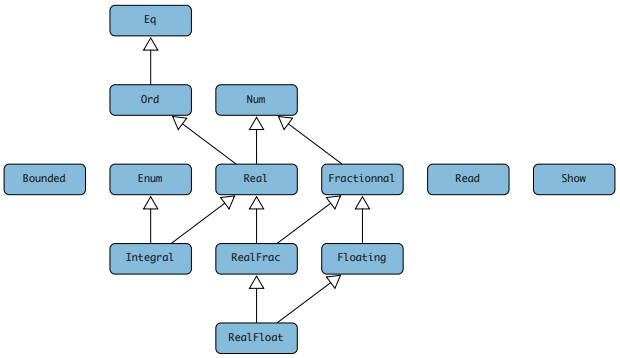
```
(+) :: (Num a) ⇒ a → a → a
```


Quelques classes de types

Les classiques

1. **Eq** définit les types comparables
2. **Ord** définit les types ordonnés
3. **Show** définit les types affichables
4. **Read** définit les types lisibles
5. **Num** définit les nombres
6. **Fractionnal** définit les nombres divisibles

Organigramme des types classes courants



Les opérateurs basiques

Les quatre classiques

1. **map** :: (a→b)→[a]→[b]
2. **filter** :: (a→bool)→[a]→[a]
3. **foldr** :: (a→b→b)→b→[a]→b
4. **foldl** :: (a→b→a)→a→[b]→a
5. **foldl'** :: (a→b→a)→a→[b]→a

Comparaison **foldr** vs. **foldl**

Développement de **foldl**

```
g xs x = odd x || xs
foldl g False [2,3,4]
foldl g False (2:[3,4])
foldl g (odd 2 || False) [3,4]
foldl g (odd 2 || False) (3:[4])
foldl g (odd 3 || (odd 2 || False)) [4]
foldl g (odd 3 || (odd 2 || False)) (4:[])
foldl g (odd 4 || (odd 3 || (odd 2 || False))) []
odd 4 || (odd 3 || (odd 2 || False))
True || (odd 3 || (odd 2 || False))
True
```

Comparaison `foldr` vs. `foldl`

Développement de `foldr`

```
g x xs = odd x || xs
foldr g False [2,3,4]
foldr g False (2:[3,4])
odd 2 || foldr g False [3,4]
True || foldr g False [3,4]
True
```


La version strict `foldl'`

Développement de `foldl'`

```
g xs x = odd x || xs
foldl' g False [2,3,4]
foldl' g False (2:[3,4])
foldl' g True [3,4]
foldl' g True (3:[4])
foldl' g False [4]
foldl' g False (4:[])
foldl' g True []
True
```


Structures existantes

Listes basiques

- **`Array`**
- `Bits`
- `ByteString`
- **`Complex`**
- `Graph`
- `Map`
- `Sequence`
- `Set`
- `Tree`

Nouveaux types

Mot-clé `type`

```
type Histogramme = [(Char,String)]
type Log a = (String,a)
```

Mot-clé `data`

```
data Shape = Circle | Rectangle | Triangle
data Nat = Zero | Succ Nat
data Liste a = Nil | Cons a (Liste a)
```


Comparaison

entre Racket et Haskell

Racket

- base : liste et paire
- typage faible
- simplifie la généricité

Haskell

- type algébrique
- typage fort
- augmente la cohérence

Les entrées/sorties ?

Les effets de bords

- Parfois désirables, parfois inévitables
- Notamment l'interaction avec l'extérieur

Solution

- Utilisation du type `IO a`
- Une monade
- une quoi ?
- « Une monade est juste un monoïde dans la catégorie des endofoncteurs. » Saunders Mac Lane

Monade `IO a`

Exemple

```
type Histogramme = [(Char,String)]

afficheH :: Histogramme → IO ()
afficheH h = mapM_ print_ligne h
    where print_ligne (l,f) = putStrLn $ l : ' ' : f

main = do texte ← getLine
          afficheH (calcul_histogramme texte)
          return -- n'est pas obligatoire
```


Conclusions

Haskell est un langage

- fonctionnel et efficace
- à la syntaxe concise (trop ?)
- encourageant un style déclaratif
- typée fort, pur et paresseux
- avec du polymorphisme paramétrique et ad-hoc
- disposant de monade !