

Programmation Fonctionnelle

Programmes interactifs

Luigi Santocanale
LIF, Aix-Marseille Université
Marseille, FRANCE

24 septembre 2013

Plan

Le type des actions

Actions élémentaires, constructions d'actions

Le jeu du pendu

Plan

Le type des actions

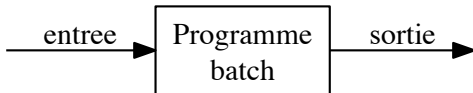
Actions élémentaires, constructions d'actions

Le jeu du pendu

Introduction

Nous avons vu – jusqu'à maintenant – comment utiliser Haskell pour écrire des programmes « *batch* » (français : par lots).

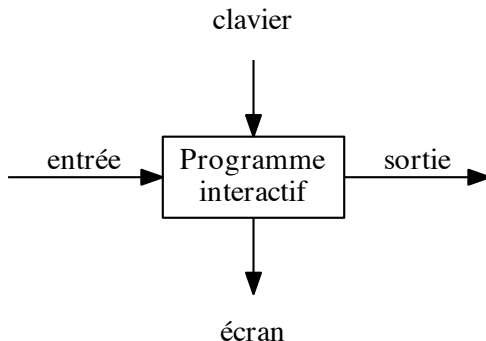
Un programme « *batch* » prend son entrée au début et retourne sa sortie à la terminaison :



Les programmes interactifs

Nous souhaitons utiliser Haskell aussi pour écrire des programmes interactifs :

- qui lisent du clavier,
- qui affichent des résultats à l'écran,
- au cours de l'exécution.



Le problème

Les programmes Haskell codent des fonctions mathématiques pures (pas de liens avec le “real world”).

Par ailleurs, lire du clavier et écrire à l'écran sont des

effets de bord.

- Le programmes Haskell n'ont pas des effets de bord.
- Les programmes interactifs ont des effets de bord.

La solution

On écrit un programme interactif en utilisant des types qui distinguent les **expressions pures** des **actions impures** (c'est-à-dire, qui peuvent entraîner des effets de bord).

I0 a

*Le type des actions (impures)
qui retournent un valeur de type a .*

Par exemple :

- IO Char : le type des actions qui retournent un caractère,
- IO () : le type des actions qui produisent des effets de bord seulement
(et ne retournent pas de valeurs).

Remarque :

- () est le type des tuplets de longueur 0, c'est-à-dire sans composantes.

La structure du type des actions

World

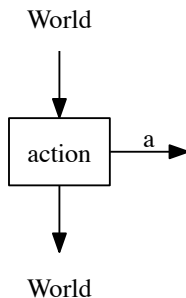
Le type des interactions avec le monde réel.

Une action

- interagit avec le monde,
- en I,
- en O,
- retourne un valeur de type a.

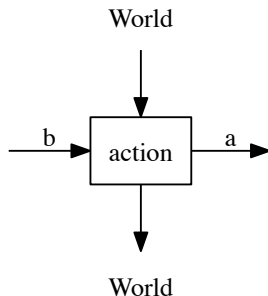
Ainsi nous avons

```
type IO a = World -> (a,World)
```



Les actions avec paramètres

Pour obtenir le schéma des actions qui prennent un argument de type `b` :



il suffit de considérer le type

$$b \rightarrow IO\ a$$

c'est-à-dire

$$b \rightarrow World \rightarrow (a, World)$$

Plan

Le type des actions

Actions élémentaires, constructions d'actions

Le jeu du pendu

Actions élémentaires

Des nombreuses actions sont définies dans `Prelude`, parmi lesquelles les trois primitives suivantes :

- `getChar :: IO Char`

lit un caractère au clavier, affiche ce caractère à l'écran, et retourne le caractère comme valeur.

- `putChar :: Char -> IO ()`

*écrit le caractère passé en paramètre à l'écran,
et ne retourne aucun résultat.*

- `return :: a -> IO a`

*l'action `return v` retourne la valeur `v`,
sans faire aucune interaction.*

Sequencage, enchaînement

On peut enchaîner une séquence d'actions, en utilisation le mot clés `do`.

On obtient ainsi une nouvelle action.

Par exemple :

```
getTwoChars :: IO (Char,Char)
getTwoChars = do
    x <- getChar
    y <- getChar
    return (x,y)
```

La structure de `do`

En fait, `do` est une construction dérivée (anglais : « syntactic sugar ») à partir de l'opérateur binaire `>>=` .

On peut écrire le programme précédent—de façon équivalente—par :

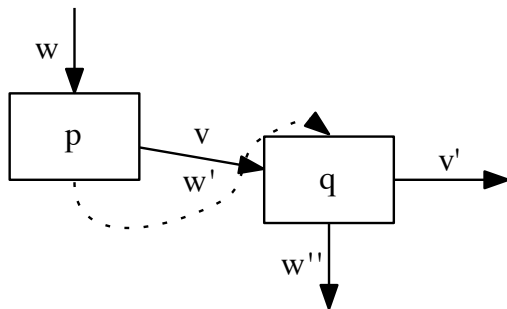
```
getTwoChars2 :: IO (Char, Char)
getTwoChars2 =
    getChar >>= \x ->
    getChar >>= \y ->
    return (x,y)
```

L'opérateur $\gg=$

Cet opérateur – appelé *et après* ou « *then* » en anglais – sert pour enchaîner une action avec une action avec paramètres :

$$(\gg=) :: IO\ a \rightarrow (a \rightarrow IO\ b) \rightarrow IO\ b$$
$$p \gg= q = \backslash w \rightarrow \text{let } (v, w') = p\ w \text{ in } q\ v\ w'$$

En diagrammes :



Des actions dérivées

- Lire une chaîne de caractères du clavier :

```
getLine :: IO String
getLine = do
    x <- getChar
    if x == '\n' then
        return []
    else
        do xs <- getLine
           return (x:xs)
```

Des actions dérivées (encore)

- Écrire une chaîne à l'écran :

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do
    putChar x
    putStr xs
```

- Écrire une chaîne à l'écran et aller à la nouvelle ligne :

```
putStrLn :: String -> IO ()
putStrLn xs = do
    putStr xs
    putChar '\n'
```

Un exemple

Nous pouvons maintenant définir une action qui demande une chaîne et affiche ensuite sa longueur :

```
strlen :: IO ()
strlen = do
    putStrLn "Rentrez une chaine : "
    xs <- getLine
    putStrLn "La chaine a "
    putStrLn (show (length xs))
    putStrLnLn " caracteres."
```

Par exemple :

```
> strlen  
Enter a string: abcde  
The string has 5 characters
```

Remarque :

- L'évaluation de l'action `strlen` exécute ses effets de bord, le résultat final étant ignoré.

Plan

Le type des actions

Actions élémentaires, constructions d'actions

Le jeu du pendu

Le jeu du pendu

Considérez la version suivante du jeu du pendu :

- Le premier joueur pense (et rentre dans l'ordi) un mot secret.
- L'autre joueur essaye de deviner ce mot, par des hypothèses successives.
- A chaque hypothèse, l'ordinateur montre quelle lettre de mot secret a une occurrence dans l'hypothèse.
- Le jeu se termine quand le mot secret est deviné.

Nous implementons ce jeu en Haskell, avec une approche « *top down* » .

Le début est comme suit :

```
pendu :: IO ()
pendu = do
  putStrLn "Rentrez le mot secret : "
  mot <- sgetLine
  putStrLn "Essayez de le deviner :"
  deviner mot
```

L'action `sgetLine` lit une ligne de texte du clavier, affichant à l'écran chaque caractère comme un tiré :

```
sgetLine :: IO String
sgetLine = do
    x <- getCh
    if x == '\n' then
        do
            putChar x
            return []
    else
        do
            putChar '-'
            xs <- sgetLine
            return (x:xs)
```


Remarque :

- L'action `getCh` lit un caractère du clavier, sans l'afficher à l'écran.

Elle peut se définir comme suit :

```
getCh :: IO Char
getCh  = do
    hSetEcho stdin False
    c <- getChar
    hSetEcho stdin True
    return c
```

Pour cela, il faut préalablement importer le module `System.IO` :

```
import System.IO
```

La fonction `deviner` est la boucle principale. Elle demande et analyse un mot, jusqu'à ce que le secret est deviné et le jeu se termine ainsi.

```
deviner :: String -> IO ()
deviner mot = do
  putStr "> "
  xs <- getLine
  if xs == mot then
    putStrLn "Bien devine'!"
  else
    do
      putStrLn (diff mot xs)
      deviner mot
```

La fonction `diff` montre les caractères d'une chaîne qui appartiennent à l'autre chaîne :

```
> diff "haskell" "pascal"  
"-as--ll"
```

```
diff :: String -> String -> String  
diff xs ys =  
    [if elem x ys then x else '-' | x <- xs]
```

Exercice

Implémentez, en Haskell, le jeu du *nim*. Les règles de ce jeu sont comme suit :

- L'échiquier (la table) contient 5 lignes d'étoiles :

```
1: * * * * *
2: * * * *
3: * * *
4: * *
5: *
```

- Deux joueurs enlèvent – alternativement – une ou plusieurs étoiles d'une seule ligne.
- Le gagnant est le joueur qui enlève la dernière(s) étoile(s) de l'échiquier.

Conseil : Représentez l'échiquier comme une liste de 5 entier qui représente les étoiles encore à enlever. Par exemple, la position initiale est $[5, 4, 3, 2, 1]$.