Lazi : introduction pratique au langage informatique

Emmanuel Chantréau

18 décembre 2016

Table des matières

1 Introduction

Le but de ce texte est de présenter le langage informatique Lazi de la manière la plus pratique possible, de sorte que le lecteur puisse avoir une idée du « look and feel » de la programmation en Lazi. La contrepartie de ce type de présentation est le manque d'éclairage sur les principes de base, qui ont en Lazi un rôle primordial. C'est pourquoi nous ne présentons ici que quelques éléments du langage, par exemple nous omettons les types.

2 Ressemblance avec Haskell

Lazi et Haskell on des points communs importants, et un « look and feel » de base proches. La syntaxe Lazi reprend le côté léger de celle d'Haskell mais d'une autre manière, en particulier le positionnement (nombre d'espaces au début des lignes) n'a pas de conséquence sur l'analyse syntaxique. Les personnes venant de langages impératifs pourront utiliser les ressources d'Haskell pour se familiariser avec la programmation fonctionnelle pure.

3 Exemple de la définition d'une fonction

Pour les premiers éléments de langage nous allons utiliser cet exemple de définition d'une fonction :

```
///                                  Appartenance à une liste

/*
Pour une comparaison `eq', la liste `l' contient-elle un élément comparable à x.
*/
$Def inList = $F eq,l,x -> existsOnList l ( eq x )
```

4 Les commentaires

C'est la même syntaxe qu'en C++ : « `//` » pour un commentaire d'une ligne et « `/* ... */` » pour les commentaires multi-lignes.

Comme le montre l'exemple, nous utiliser un / de plus, pour déclencher une coloration syntaxique spéciale (configuration pour kate).

5 $Def

`$Def` permet de définir un nom (ici `inList`). En C++ on distingue la définition des noms suivant qu'il s'agit d'une fonction, d'une donnée ou d'un type. En haskell, les données et les fonctions sont unifiées et on ne distingue que la définition des types et des données/fonctions. En Lazi tout est unifié et il n'y a qu'une sorte de définition.

On peut remarquer que la définition de inList n'est pas explicitement délimitée (par exemple par des accolades comme en C++). En Lazi, la définition continue jusqu'à la prochaine notation globale (comme « `$Def` » ou « `$Nota` »). Ce choix a pour but d'alléger le code source.

6 $F

« `$F` » permet d'exprimer une fonction, suit la liste des variables, « `->` » puis le corps de la fonction.

7 Fonctions et arguments

L'application des arguments en Lazi fonctionne comme en Haskell (du point de vue théorique, c'est un langage fonctionnel pure basé sur le lambda calcul réduit aux applications).

En Lazi les données et les fonctions sont unifiées. Si vous avez deux choses x et y, vous pouvez appliquer y à x pour former la chose « x y » (x a le rôle de la fonction et y de l'argument).

Par exemple si vous avez une fonction « sum » et deux entiers x et y que vous voulez additionner grâce à sum :
Vous appliquez x à sum, cela constitue la chose « sum x ». On peut voir « sum x » comme une fonction qui prend un argument n et retourne x+n. Vous pouvez maintenant appliquer y à « sum x », le résultat sera « (sum x) y » que l'on note aussi « sum x y ».

Cette forme d'application des arguments aux fonctions devient vite naturelle et pratique. Par exemple si vous avez une fonction d'égalité « eq », et une fonction prenant en argument une propriété, si vous voulez passer en argument la propriété « être égal à x », il vous suffit de passer « eq x ».

8 Récursivité

Pour utiliser la récursivité il vous suffit d'utiliser le nom défini dans sa définition. Par exemple pour définir une fonction retournant la liste infinie remplie par x :

```
$Def infiniteList = $F x -> infiniteList x :l x
```

`:l` ajoute l'élément x à la fin de la liste « `infiniteList x` ».

Remarquons que ce code ne fonctionnerait pas en langage impératif car la fonction bouclerait. Mais Lazi est un langage à évaluation paresseuse (comme Haskell), c'est à dire qu'il ne calcul que ce qu'il a besoin.

Par exemple pour prendre le dernier élément de la liste `infiniteList a`, il calcul jusqu'à obtenir `infiniteList a :l a`, hors prendre le dernier élément de `infiniteList a :l a` ne nécéssite pas de calculer toute la liste (de même que si je vous demande « Quel est le dernier élément d'une liste se terminant par 0 ? » vous n'avez pas besoin de me demander de quoi est constituée toute la liste pour répondre).

N'oublions pas que les données et les fonctions sont unifiées, par exemple pour définir la liste infinie remplie de zéro : `$Def infiniteListZero = infiniteListZero :l 0`

Et nous pouvons tout aussi bien définir des types récursifs ou encore paramétrés, mais nous n'irons pas jusque là dans cette documentation.

9 Simplifier le parenthèsage

Les langages fonctionnels sont réputés nécessiter trop de parenthèses, par exemple pour appliquer x y puis z t à f il faut écrire `f (x y) (z t)` , rappelons que c'est la même chose que `(f (x y)) (z t)` . En C++ on écrirait `f( x(y) , z(t) )`.

Nous avons deux syntaxes en Lazi aidant à l'utilisation minimale des parenthèses. Je vous conseille de prendre l'habitude d'utiliser ces deux aides syntaxiques car elles permettent un code plus agréable à lire.

9.1 Les parenthèses à virgule

En Lazi, une notation transforme une expression
`( x1 , x2 , ... , xn )` en `( x1 ) ( x2 ) ... ( xn )`
où x1 ... xn sont des expressions quelconques.
Donc nous pouvons écrire notre expression précédente « `f(x y, z t)` » mais aussi « `(f ,x y, z t)` » voir même « `(f, x y) (z t)` »

9.2 L'application à priorité minimale

La construction « f x » est l'application de x sur f. La priorité de l'application dépasse tout opérateur, par exemple `infiniteList x :l x` se lit `(infiniteList x) :l x` car l'application entre infiniteList et x a une priorité (ou précédence) plus grande que l'opérateur `:l`.

Il existe en Lazi un opérateur noté « . » qui représente l'application (donc `(x y) = (x . y)`) , qui est associatif à droite et avec une priorité strictement minimale (rien n'a une priorité égale ou inférieure).

Donc, par exemple, on peut remplacer `f ( a b ( c d ( e f ) ) )` par `f . a b . c d . e f`

Vous pouvez utiliser la règle suivante pour utiliser cet opérateur : tout ce qui est à droite d'un point est une expression (même si elle utilise aussi cet opérateur). Donc par exemple `f . a b . c d . e f` se traduit en `f ( a b . c d . e f)` . Cette notation est donc très pratique quand on applique une fonction à une grosse expression, car allors on peut remplacer de grande parenthèses par un point qui signifie « application avec tout ce qui est à droite ».

10 Correspondance par motif

Imaginons que nous voulions définir une fonction qui inverse les deux éléments d'une paire. Sachant que la notation `$P[x,y]` permet de construire une paire nous pourrions écrire :

```
$Def reversePair = $F p -> $P[ pairSecond p , pairFirst p ]
```

Mais nous pouvons simplifier cette définition par :

```
$Def reversePair = $F $P[x,y] -> $P[y,x]
```

La correspondance par motif n'a pratiquement pas de limite, on peut l'imbriquer ou encore nommer (en utilisant `@`)) l'argument entier. Par exemple :

```
$Def flat = $F $P[x,p@$P[y,z]] -> $L[x,p,y,z]
```

11 Définition d'un opérateur

Dans l'exemple ci-dessous nous définissons une fonction de composition de deux fonctions à un argument (la fonction mathématique « ∘ ») et un opérateur associé à cette fonction.

```
/*
Composition de deux fonctions f et g.
*/
$Def functionCompose = $F f,g,x -> f . g x

$Nota .o infix left 10 functionCompose
```

Remarque : peut être que la définition strictement équivalente suivante vous est plus parlante :
`$Def functionCompose = $F f,g -> $F x -> f . g x`

.o devient donc un opérateur infix, associatif à gauche de priorité 10 et tel que `x .o y = functionCompose x y`

12 Le style de syntaxe Lazi

12.1 Place à la nouveauté

Lazi a un style de syntaxe qui découle de sa nature spéciale. L'idée est de laisser toujours de l'espace pour de nouvelles notations. Un minimum de symboles sont réservés, par exemple Haskell définit une liste par `[ x , y ... ]` alors qu'en Lazi c'est `$L[ x, y ... ]`. Quand Lazi sera mature, même des notations complexes comme « `$F ->` » seront définissables en Lazi, il est donc important de ne pas s'accaparer les symboles pour laisser de la place à la nouveauté. C'est pourquoi en Lazi « x.y » est syntaxiquement invalide : il faut laisser un espace entre les opérateurs et les noms, de la sorte on peut définir l'opérateur « .o » sans risquer de confusion avec l'opérateur « . ».

Les symboles qui peuvent être collés aux expressions sont donc rares : la virgule et les symboles de délimitations pour la partie intérieure (parnthèses, crochets etc).

Certains symboles ont un rôle fixés :
— les parenthèses : servent toujours à délimiter une expression
— la virgule : sert de séparateur
— $Texte[...] : Délimite une syntaxe spéciale (par exemple `$L[ x,y,z]` est la notation pour la liste d'éléments x, y et z).

12.2 Place à la clarté

Le programmeur n'a pas besoin de se soucier des optimisations basiques, car elles sont faites par les programmes de traduction et d'interprétation. Il peut donc se concentrer sur la clarté de son code.

Exemples de formules ayant un temps de calcul strictement identiques :

`($F x,y -> f a b x y)`	`f a b`	Car `$F x -> g x` est égal à `g`
`$Let x = y z, f x`	f . y z	Car x n'apparaît qu'une fois
`f (g . x y, h . x y)`	`$Let a = x y, f (g a, h a)`	« x y » est factorisé

Remarque : la factorisation d'expressions identiques se fait complètement intra-formule (mêmes avec les variables), mais pas entre formules différentes (donc pas entre les valeurs des noms définis).

13 Un exemple plus gros

```
$Def listFoldByIter = $F it ->
  recurse $F lf,f,r,l ->
    if
    (
      isEmptyList l
    ,
      r
    ,
      it ( lf f , f . listLast l , r , listInit l )
    )
```

Sans chercher à comprendre le but de cette fonction, on peut noter :

— L'utilisation d'une lambda fonction récursive (`recurse $F lf, ...`), `lf` est la variable représentant la lambda fonction récursive.

— L'utilisation du classique « if then else ». Lazi ne fournit pas de notation spéciale car je trouve la syntaxe générale suffisante pour rendre le code source clair.