eBook Gratuit

APPRENEZ Haskell Language

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#haskell

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: haskell-language

It is an unofficial and free Haskell Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Haskell Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec le langage Haskell

Remarques

Haskell est un langage de programmation avancé, purement fonctionnel.

Caractéristiques:

- **Typiquement statiquement:** chaque expression dans Haskell a un type qui est déterminé au moment de la compilation. La vérification de type statique est le processus de vérification de la sécurité de type d'un programme en fonction de l'analyse du texte d'un programme (code source). Si un programme passe un vérificateur de type statique, le programme est assuré de satisfaire un ensemble de propriétés de sécurité de type pour toutes les entrées possibles.
- **Purement fonctionnel** : toute fonction dans Haskell est une fonction au sens mathématique. Il n'y a pas d'instructions ou d'instructions, seulement des expressions qui ne peuvent muter des variables (locales ou globales) ni des états d'accès tels que des nombres temporels ou aléatoires.
- **Concurrent:** son compilateur phare, GHC, est livré avec une bibliothèque parallèle très performante de ramasse-miettes et de concurrences légères contenant un certain nombre de primitives et d'abstractions utiles.
- **Evaluation paresseuse: les** fonctions n'évaluent pas leurs arguments. Retarde l'évaluation d'une expression jusqu'à ce que sa valeur soit nécessaire.
- **Usage général:** Haskell est conçu pour être utilisé dans tous les contextes et environnements.
- **Packages:** La contribution open source à Haskell est très active avec un large éventail de packages disponibles sur les serveurs de packages publics.

Le dernier standard de Haskell est Haskell 2010. En mai 2016, un groupe travaille sur la prochaine version, Haskell 2020.

La documentation officielle de Haskell est également une ressource complète et utile. Super endroit pour trouver des livres, des cours, des tutoriels, des manuels, des guides, etc.

Versions

Version	Date de sortie
Haskell 2010	2012-07-10
Haskell 98	2002-12-01

Examples

Bonjour le monde!

Un basique "Bonjour, Monde!" programme en Haskell peut être exprimé de manière concise en une ou deux lignes:

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

La première ligne est une annotation de type facultative, indiquant que `main` est une valeur de type `IO ()` , représentant une action E / S qui "calcule" une valeur de type `()` (lire "unit"; le tuple vide ne transmettant aucune information) en plus d'effectuer des effets secondaires sur le monde extérieur (ici, imprimer une chaîne au terminal). Cette annotation de type est généralement omise pour `main` car c'est son *seul* type possible.

Mettez ceci dans un fichier `helloworld.hs` et compilez-le en utilisant un compilateur Haskell, tel que GHC:

```
ghc helloworld.hs
```

L'exécution du fichier compilé entraînera la sortie `"Hello, World!"` en cours d'impression sur l'écran:

```
./helloworld
Hello, World!
```

Sinon, `runhaskell` ou `runghc` permettent d'exécuter le programme en mode interprété sans avoir à le compiler:

```
runhaskell helloworld.hs
```

La REPL interactive peut également être utilisée au lieu de la compilation. Il est livré avec la plupart des environnements Haskell, tels que `ghci` fourni avec le compilateur GHC:

```
ghci> putStrLn "Hello World!"
Hello, World!
ghci>
```

Vous pouvez également charger des scripts dans ghci à partir d'un fichier en utilisant `load` (ou `:l`):

```
ghci> :load helloworld
```

`:reload` (ou `:r`) recharge tout en ghci:

```
Prelude> :l helloworld.hs
[1 of 1] Compiling Main             ( helloworld.hs, interpreted )
```

```
<some time later after some edits>

*Main> :r
Ok, modules loaded: Main.
```

Explication:

Cette première ligne est une signature de type, déclarant le type de `main` :

```
main :: IO ()
```

Les valeurs de type `IO ()` décrivent des actions pouvant interagir avec le monde extérieur.

Étant donné que Haskell a un système de type Hindley-Milner à part entière qui permet l'inférence automatique de type, les signatures de type sont techniquement facultatives: si vous omettez simplement `main :: IO ()` , le compilateur pourra déduire le type de lui-même en analyser la *définition* du `main` . Cependant, il est considéré comme un mauvais style de ne pas écrire les signatures de type pour les définitions de niveau supérieur. Les raisons incluent:

- Les signatures de type dans Haskell sont une documentation très utile car le système de types est si expressif que vous pouvez souvent voir à quoi sert une fonction simplement en regardant son type. Cette «documentation» est facilement accessible avec des outils tels que GHCi. Et contrairement à la documentation normale, le vérificateur de type du compilateur s'assurera qu'il correspond réellement à la définition de la fonction!
- Les signatures de type *gardent les bogues locaux* . Si vous faites une erreur dans une définition sans fournir sa signature de type, le compilateur peut ne pas signaler immédiatement une erreur, mais simplement en déduire un type absurde, avec lequel il vérifie réellement. Vous pouvez alors recevoir un message d'erreur crypté lorsque vous *utilisez* cette valeur. Avec une signature, le compilateur est très efficace pour détecter les bogues là où ils se produisent.

Cette deuxième ligne fait le travail réel:

```
main = putStrLn "Hello, World!"
```

Si vous venez d'un langage impératif, il peut être utile de noter que cette définition peut également être écrite comme suit:

```
main = do {
   putStrLn "Hello, World!" ;
   return ()
   }
```

Ou de manière équivalente (Haskell a une analyse basée sur la mise en page, mais *méfiez-vous du mélange incohérent des onglets et des espaces,* ce qui compliquerait ce mécanisme):

```
main = do
    putStrLn "Hello, World!"
    return ()
```

Chaque ligne d'un bloc `do` représente un *calcul* monadique (ici, E / S), de sorte que tout `do` bloc `do` représente l'action globale composée de ces sous-étapes en les combinant de manière spécifique à la monade donnée (pour les E / S). cela signifie simplement les exécuter l'un après l'autre).

La syntaxe `do` est elle-même un sucre syntaxique pour les monades, comme `IO` ici, et `return` est une action sans opération produisant son argument sans effectuer aucun effet secondaire ou calcul supplémentaire pouvant faire partie d'une définition de monade particulière.

Ce qui précède est identique à la définition de `main = putStrLn "Hello, World!"` , parce que la valeur `putStrLn "Hello, World!"` a déjà le type `IO ()` . Vu comme une "déclaration", `putStrLn "Hello, World!"` peut être vu comme un programme complet, et vous définissez simplement `main` pour faire référence à ce programme.

Vous pouvez rechercher la signature de `putStrLn` ligne :

```
putStrLn :: String -> IO ()
-- thus,
putStrLn (v :: String) :: IO ()
```

`putStrLn` est une fonction qui prend une chaîne comme argument et `putStrLn` une action E / S (c'est-à-dire une valeur représentant un programme que le moteur d'exécution peut exécuter). Le moteur d'exécution exécute toujours l'action nommée `main` , il suffit donc de la définir comme étant égale à `putStrLn "Hello, World!"` .

Factorial

La fonction factorielle est un Haskell "Hello World!" (et pour la programmation fonctionnelle en général) dans le sens où elle démontre succinctement les principes de base de la langue.

Variation 1

```
fac :: (Integral a) => a -> a
fac n = product [1..n]
```

Démo en direct

- `Integral` est la classe des types de nombres entiers. Les exemples incluent `Int` et `Integer` .
- `(Integral a) =>` place une contrainte sur le type `a` pour être dans ladite classe
- `fac :: a -> a` dit que `fac` est une fonction qui prend un `a` et retourne `a`
- `product` est une fonction qui accumule tous les nombres d'une liste en les multipliant.
- `[1..n]` est une notation spéciale qui désuère `enumFromTo 1 n` et représente la plage de nombres `1 ≤ x ≤ n` .

Variation 2

```
fac :: (Integral a) => a -> a
fac 0 = 1
fac n = n * fac (n - 1)
```

Démo en direct

Cette variation utilise la correspondance de motif pour diviser la définition de fonction en cas séparés. La première définition est appelée si l'argument est `0` (parfois appelé la condition d'arrêt) et la deuxième définition autrement (l'ordre des définitions est significatif). Elle illustre également la récursivité en tant que `fac` .

Il est à noter que, en raison des règles de réécriture, les deux versions de `fac` seront compilées en un code machine identique lors de l'utilisation de GHC avec des optimisations activées. Donc, en termes d'efficacité, les deux seraient équivalents.

Fibonacci, en utilisant l'évaluation paresseuse

Une évaluation paresseuse signifie que Haskell évaluera uniquement les éléments de liste dont les valeurs sont nécessaires.

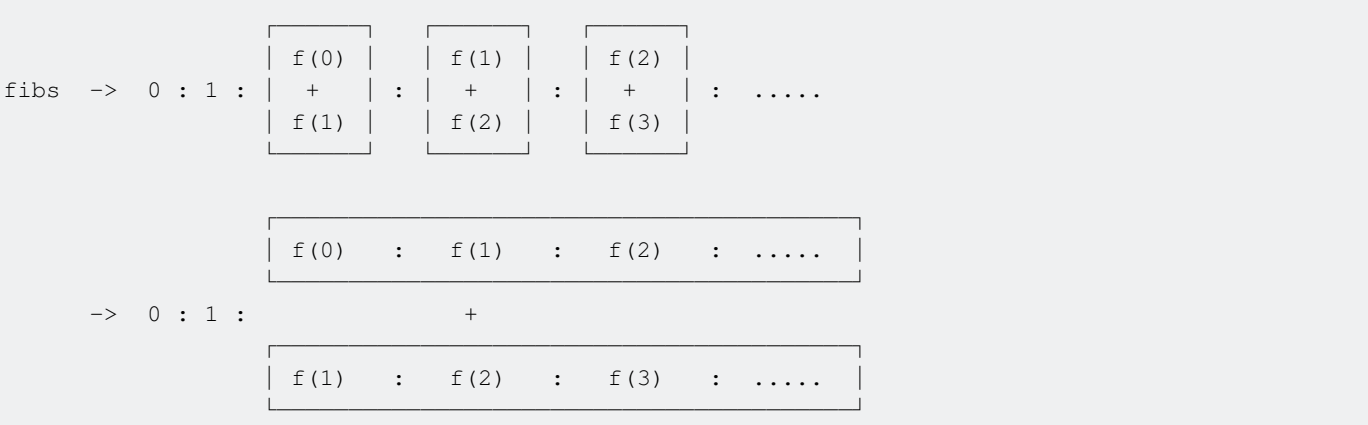
La définition récursive de base est la suivante:

```
f (0)  <-  0
f (1)  <-  1
f (n)  <-  f (n-1) + f (n-2)
```

S'il est évalué directement, ce sera *très* lent. Mais, imaginez que nous ayons une liste qui enregistre tous les résultats,

```
fibs !! n  <-  f (n)
```

alors



Ceci est codé comme:

```
fibn n = fibs !! n
    where
    fibs = 0 : 1 : map f [2..]
    f n = fibs !! (n-1) + fibs !! (n-2)
```

Ou même comme

```
GHCi> let fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`zipWith` crée une liste en appliquant une fonction binaire donnée aux éléments correspondants des deux listes qui lui sont données, donc `zipWith (+) [x1, x2, ...] [y1, y2, ...]` est égal à `[x1 + y1, x2 + y2, ...]` .

Une autre façon d'écrire des `fibs` est la fonction `scanl` :

```
GHCi> let fibs = 0 : scanl (+) 1 fibs
GHCi> take 10 fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

`scanl` construit la liste des résultats partiels que `foldl` produirait, travaillant de gauche à droite le long de la liste d'entrée. C'est-à-dire que `scanl f z0 [x1, x2, ...]` est égal à `[z0, z1, z2, ...] where z1 = f z0 x1; z2 = f z1 x2; ...`

Grâce à l'évaluation différée, les deux fonctions définissent des listes infinies sans les calculer entièrement. C'est-à-dire que nous pouvons écrire une fonction `fib` , en récupérant le nième élément de la séquence Fibonacci sans limite:

```
GHCi> let fib n = fibs !! n  -- (!!) being the list subscript operator
-- or in point-free style:
GHCi> let fib = (fibs !!)
GHCi> fib 9
34
```

Commencer

REPL en ligne

La manière la plus simple de commencer à écrire Haskell est probablement d'aller sur le site Web Haskell ou Try Haskell et d'utiliser la ligne REPL (read-eval-print-loop) en ligne sur la page d'accueil. Le REPL en ligne prend en charge la plupart des fonctionnalités de base et même certaines entrées-sorties. Il existe également un didacticiel de base qui peut être démarré en tapant l' `help` la commande. Un outil idéal pour commencer à apprendre les bases de Haskell et essayer certaines choses.

GHC (i)

GHCi est un environnement interactif fourni avec le *compilateur Glorious / Glasgow Haskell* . Le *GHC* peut être installé séparément, mais ce n'est qu'un compilateur. Pour pouvoir installer de nouvelles bibliothèques, des outils tels que *Cabal* et *Stack* doivent également être installés. Si vous utilisez un système d'exploitation de type Unix, l'installation la plus simple consiste à installer *Stack en* utilisant:

```
curl -sSL https://get.haskellstack.org/ | sh
```

Cela installe GHC isolé du reste de votre système, il est donc facile à supprimer. Toutes les commandes doivent cependant être précédées de `stack` . Une autre approche simple consiste à installer une plate-forme Haskell . La plateforme existe en deux versions:

1. La distribution **minimale** ne contient que *GHC* (compiler) et *Cabal* / *Stack* (pour installer et compiler les paquets)
2. La distribution **complète** contient en outre des outils pour le développement de projets, le profilage et l'analyse de la couverture. Un ensemble supplémentaire de packages largement utilisés est également inclus.

Ces plates-formes peuvent être installées en téléchargeant un programme d'installation et en suivant les instructions ou en utilisant le gestionnaire de paquets de votre distribution (notez que cette version n'est pas garantie pour être à jour):

- Ubuntu, Debian, Mint:

  ```
  sudo apt-get install haskell-platform
  ```

- Feutre:

  ```
  sudo dnf install haskell-platform
  ```

- Chapeau rouge:

  ```
  sudo yum install haskell-platform
  ```

- Arch Linux:

  ```
  sudo pacman -S ghc cabal-install haskell-haddock-api \
                 haskell-haddock-library happy alex
  ```

- Gentoo:

  ```
  sudo layman -a haskell
  sudo emerge haskell-platform
  ```

- OSX avec Homebrew:

```
brew cask install haskell-platform
```

- OSX avec MacPorts:

```
sudo port install haskell-platform
```

Une fois installé, il devrait être possible de démarrer *GHCi* en `ghci` commande `ghci` n'importe où dans le terminal. Si l'installation s'est bien passée, la console devrait ressembler à

```
me@notebook:~$ ghci
GHCi, version 6.12.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

éventuellement avec plus d'informations sur les bibliothèques chargées avant le `Prelude>` . Maintenant, la console est devenue une REPL Haskell et vous pouvez exécuter le code Haskell comme avec la REPL en ligne. Pour quitter cet environnement interactif, vous pouvez taper `:q` ou `:quit` . Pour plus d'informations sur les commandes disponibles dans *GHCi* , tapez `:?` comme indiqué dans l'écran de démarrage.

Parce qu'écrire les mêmes choses encore et encore sur une seule ligne n'est pas toujours pratique, il peut être judicieux d'écrire le code Haskell dans les fichiers. Ces fichiers ont normalement `.hs` pour une extension et peuvent être chargés dans la REPL en utilisant `:l` ou `:load` .

Comme mentionné précédemment, *GHCi* fait partie du *GHC* , qui est en fait un compilateur. Ce compilateur peut être utilisé pour transformer un fichier `.hs` avec du code Haskell en un programme en cours d'exécution. `.hs` fichier `.hs` pouvant contenir de nombreuses fonctions, une fonction `main` doit être définie dans le fichier. Ce sera le point de départ du programme. Le fichier `test.hs` peut être compilé avec la commande

```
ghc test.hs
```

Cela créera des fichiers d'objets et un exécutable s'il n'y avait pas d'erreurs et que la fonction `main` était définie correctement.

Des outils plus avancés

1. Il a déjà été mentionné précédemment en tant que gestionnaire de paquets, mais la *pile* peut être un outil très utile pour le développement de Haskell. Une fois installé, il est capable de

    - installer (plusieurs versions de) *GHC*
    - création de projet et échafaudage
    - gestion des dépendances
    - projets de construction et d'essais

- analyse comparative

2. IHaskell est un noyau de haskell pour IPython et permet de combiner du code (exécutable) avec le marquage et la notation mathématique.

Primes

Quelques variantes *les plus saillantes* :

Moins de 100

```
import Data.List ( (\\) )

ps100 = ((([2..100] \\ [4,6..100]) \\ [6,9..100]) \\ [10,15..100]) \\ [14,21..100]

   -- = (((2:[3,5..100]) \\ [9,15..100]) \\ [25,35..100]) \\ [49,63..100]

   -- = (2:[3,5..100]) \\ ([9,15..100] ++ [25,35..100] ++ [49,63..100])
```

Illimité

Tamis d'Eratosthenes, en utilisant le paquet data-ordlist :

```
import qualified Data.List.Ordered

ps   = 2 : _Y ((3:) . minus [5,7..] . unionAll . map (\p -> [p*p, p*p+2*p..]))

_Y g = g (_Y g)   -- = g (g (_Y g)) = g (g (g (g (...)))) = g . g . g . g . ...
```

Traditionnel

(un tamis de division d'essai sous-optimal)

```
ps = sieve [2..]
     where
     sieve (x:xs) = [x] ++ sieve [y | y <- xs, rem y x > 0]

-- = map head ( iterate (\(x:xs) -> filter ((> 0).(`rem` x)) xs) [2..] )
```

Division d'essai optimale

```
ps = 2 : [n | n <- [3..], all ((> 0).rem n) $ takeWhile ((<= n).(^2)) ps]

-- = 2 : [n | n <- [3..], foldr (\p r-> p*p > n || (rem n p > 0 && r)) True ps]
```

De transition

De la division d'essai au tamis d'Eratosthène:

```
[n | n <- [2..], []==[i | i <- [2..n-1], j <- [0,i..n], j==n]]
```

Le code le plus court

```
nubBy (((>1).).gcd) [2..]          -- i.e., nubBy (\a b -> gcd a b > 1) [2..]
```

`nubBy` provient également de `Data.List` , comme `(\\)` .

Déclaration des valeurs

Nous pouvons déclarer une série d'expressions dans le REPL comme ceci:

```
Prelude> let x = 5
Prelude> let y = 2 * 5 + x
Prelude> let result = y * 10
Prelude> x
5
Prelude> y
15
Prelude> result
150
```

Pour déclarer les mêmes valeurs dans un fichier, nous écrivons ce qui suit:

```
-- demo.hs

module Demo where
-- We declare the name of our module so
-- it can be imported by name in a project.

x = 5

y = 2 * 5 + x

result = y * 10
```

Les noms de module sont en majuscules, contrairement aux noms de variables.

Lire Démarrer avec le langage Haskell en ligne:
https://riptutorial.com/fr/haskell/topic/251/demarrer-avec-le-langage-haskell

Chapitre 2: Algèbre de type

Examples

Nombres naturels en algèbre de type

Nous pouvons établir un lien entre les types Haskell et les nombres naturels. Cette connexion peut être faite en attribuant à chaque type le nombre d'habitants qu'il possède.

Types d'union finis

Pour les types finis, il suffit de voir que l'on peut attribuer un type naturel à chaque nombre, en fonction du nombre de constructeurs. Par exemple:

```
type Color = Red | Yellow | Green
```

serait **3** . Et le type `Bool` serait **2** .

```
type Bool = True | False
```

Unicité jusqu'à l'isomorphisme

Nous avons vu que plusieurs types correspondraient à un seul numéro, mais dans ce cas, ils seraient isomorphes. C'est-à-dire qu'il y aurait une paire de morphismes `f` et `g` , dont la composition serait l'identité, reliant les deux types.

```
f :: a -> b
g :: b -> a

f . g == id == g . f
```

Dans ce cas, nous dirions que les types sont **isomorphes** . Nous considérerons deux types égaux dans notre algèbre tant qu'ils sont isomorphes.

Par exemple, deux représentations différentes du nombre deux sont trivialement isomorphes:

```
type Bit  = I    | O
type Bool = True | False

bitValue :: Bit -> Bool
bitValue I = True
bitValue O = False

booleanBit :: Bool -> Bit
booleanBit True  = I
booleanBit False = O
```

Parce que nous pouvons voir `bitValue . booleanBit == id == booleanBit . bitValue`

Un et zéro

La représentation du nombre **1** est évidemment un type avec un seul constructeur. Dans Haskell, ce type est canoniquement le type `()` , appelé Unit. Tout autre type avec un seul constructeur est isomorphe à `()` .

Et notre représentation de **0** sera un type sans constructeurs. C'est le type **Void** dans Haskell, tel que défini dans `Data.Void` . Cela équivaudrait à un type non habité, sans constructeurs de données:

```
data Void
```

Ajout et multiplication

L'addition et la multiplication ont des équivalents dans ce type algèbre. Ils correspondent aux **unions marquées** et aux **types de produits** .

```
data Sum a b = A a | B b
data Prod a b = Prod a b
```

Nous pouvons voir comment le nombre d'habitants de chaque type correspond aux opérations de l'algèbre.

De manière équivalente, nous pouvons utiliser `Either` et `(,)` comme constructeurs de type pour l'addition et la multiplication. Ils sont isomorphes à nos types précédemment définis:

```
type Sum' a b = Either a b
type Prod' a b = (a,b)
```

Les résultats attendus de l'addition et de la multiplication sont suivis par le type algèbre jusqu'à l'isomorphisme. Par exemple, on peut voir un isomorphisme entre 1 + 2, 2 + 1 et 3; comme 1 + 2 = 3 = 2 + 1.

```
data Color = Red | Green | Blue

f :: Sum () Bool -> Color
f (Left ())     = Red
f (Right True)  = Green
f (Right False) = Blue

g :: Color -> Sum () Bool
g Red   = Left ()
g Green = Right True
g Blue  = Right False

f' :: Sum Bool () -> Color
f' (Right ())   = Red
```

```
f' (Left True)  = Green
f' (Left False) = Blue

g' :: Color -> Sum Bool ()
g' Red   = Right ()
g' Green = Left True
g' Blue  = Left False
```

Règles d'addition et de multiplication

Les règles communes de commutativité, associativité et distributivité sont valables car il existe des isomorphismes triviaux entre les types suivants:

```
-- Commutativity
Sum a b           <=> Sum b a
Prod a b          <=> Prod b a
-- Associativity
Sum (Sum a b) c   <=> Sum a (Sum b c)
Prod (Prod a b) c <=> Prod a (Prod b c)
-- Distributivity
Prod a (Sum b c)  <=> Sum (Prod a b) (Prod a c)
```

Types récursifs

Des listes

Les listes peuvent être définies comme suit:

```
data List a = Nil | Cons a (List a)
```

Si nous traduisons cela dans notre algèbre de type, nous obtenons

> Liste (a) = 1 + a * Liste (a)

Mais nous pouvons maintenant substituer plusieurs fois *List (a)* dans cette expression pour obtenir:

> Liste (a) = 1 + a + a * a + a * a * a + a * a * a * a + ...

Cela a du sens si nous voyons une liste comme un type qui ne peut contenir qu'une valeur, comme dans `[]` ; ou toute valeur de type `a` , comme dans `[x]` ; ou deux valeurs de type `a` , comme dans `[x,y]` ; etc. La définition théorique de List que nous devrions en tirer serait:

```
-- Not working Haskell code!
data List a = Nil
            | One a
            | Two a a
            | Three a a a
            ...
```

Des arbres

Nous pouvons faire la même chose avec les arbres binaires, par exemple. Si nous les définissons comme:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Nous obtenons l'expression:

> Arbre (a) = 1 + a * Arbre (a) * Arbre (a)

Et si nous faisons les mêmes substitutions encore et encore, nous obtiendrons la séquence suivante:

> Arbre (a) = 1 + a + 2 (a * a) + 5 (a * a * a) + 14 (a * a * a * a) + ...

Les coefficients que nous obtenons ici correspondent à la séquence des nombres catalans, et le nième nombre catalan est précisément le nombre d'arbres binaires possibles avec n nœuds.

Dérivés

La dérivée d'un type est le type de son type de contexte à un trou. C'est le type que nous obtiendrions si nous faisons disparaître une variable de type à chaque point et additionnons les résultats.

Par exemple, nous pouvons prendre le triple type `(a,a,a)` et le dériver, en obtenant

```
data OneHoleContextsOfTriple = (a,a,()) | (a,(),a) | ((),a,a)
```

Ceci est cohérent avec notre définition habituelle de la dérivation, comme suit:

> d / da (a * a * a) = 3 * a * a

Plus d'informations sur ce sujet peuvent être lues sur cet article .

Les fonctions

Les fonctions peuvent être considérées comme exponentielles dans notre algèbre. Comme nous pouvons le voir, si nous prenons un type `a` avec n instances et un type `b` avec m instances, le type `a -> b` aura m `a -> b` la puissance de n instances.

Par exemple, `Bool -> Bool` est isomorphe à `(Bool,Bool)` , comme 2 * 2 = 2^2.

```
iso1 :: (Bool -> Bool) -> (Bool,Bool)
iso1 f = (f True,f False)

iso2 :: (Bool,Bool) -> (Bool -> Bool)
iso2 (x,y) = (\p -> if p then x else y)
```

Lire Algèbre de type en ligne: https://riptutorial.com/fr/haskell/topic/4905/algebre-de-type

Chapitre 3: Algorithmes de tri

Examples

Tri par insertion

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x < y     = x:y:ys
                | otherwise = y:(insert x ys)

isort :: Ord a => [a] -> [a]
isort [] = []
isort (x:xs) = insert x (isort xs)
```

Exemple d'utilisation:

```
> isort [5,4,3,2,1]
```

Résultat:

```
[1,2,3,4,5]
```

Tri par fusion

Ordonner la fusion de deux listes ordonnées

Préserver les doublons:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y    = x:merge xs (y:ys)
                    | otherwise = y:merge (x:xs) ys
```

Version descendante:

```
msort :: Ord a => [a] -> [a]
msort [] = []
msort [a] = [a]
msort xs = merge (msort (firstHalf xs)) (msort (secondHalf xs))

firstHalf  xs = let { n = length xs } in take (div n 2) xs
secondHalf xs = let { n = length xs } in drop (div n 2) xs
```

Il est défini de cette manière pour plus de clarté et non pour l'efficacité.

Exemple d'utilisation:

```
> msort [3,1,4,5,2]
```

Résultat:

```
[1,2,3,4,5]
```

Version bottom-up:

```
msort [] = []
msort xs = go [[x] | x <- xs]
    where
    go [a] = a
    go xs = go (pairs xs)
    pairs (a:b:t) = merge a b : pairs t
    pairs t = t
```

Tri rapide

```
qsort :: (Ord a) => [a] -> [a]
qsort [] = []
qsort (x:xs) = qsort [a | a <- xs, a < x]
                     ++ [x] ++
               qsort [b | b <- xs, b >= x]
```

Tri à bulles

```
bsort :: Ord a => [a] -> [a]
bsort s = case bsort' s of
               t | t == s    -> t
                 | otherwise -> bsort t
  where bsort' (x:x2:xs) | x > x2    = x2:(bsort' (x:xs))
                         | otherwise = x:(bsort' (x2:xs))
        bsort' s = s
```

Tri des permutations

Aussi connu sous le nom de bogosort .

```
import Data.List (permutations)

sorted :: Ord a => [a] -> Bool
sorted (x:y:xs) = x <= y && sorted (y:xs)
sorted _        = True

psort :: Ord a => [a] -> [a]
psort = head . filter sorted . permutations
```

Extrêmement inefficace (sur les ordinateurs actuels).

Tri de sélection

Le tri de sélection sélectionne l'élément minimum, de manière répétée, jusqu'à ce que la liste soit vide.

```
import Data.List (minimum, delete)

ssort :: Ord t => [t] -> [t]
ssort [] = []
ssort xs = let { x = minimum xs }
           in  x : ssort (delete x xs)
```

Lire Algorithmes de tri en ligne: https://riptutorial.com/fr/haskell/topic/2300/algorithmes-de-tri

Chapitre 4: Analyse HTML avec lentille taggy et lentille

Examples

Extraire le contenu du texte d'un div avec un identifiant particulier

Taggy-lens nous permet d'utiliser des objectifs pour analyser et inspecter des documents HTML.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.0 --install-ghc runghc --package text --package lens --package
taggy-lens

{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text.Lazy as TL
import qualified Data.Text.IO as T
import Text.Taggy.Lens
import Control.Lens

someHtml :: TL.Text
someHtml =
    "\
    \<!doctype html><html><body>\
    \<div>first div</div>\
    \<div id=\"thediv\">second div</div>\
    \<div id=\"not-thediv\">third div</div>"

main :: IO ()
main = do
    T.putStrLn
        (someHtml ^. html . allAttributed (ix "id" . only "thediv") . contents)
```

Filtrage des éléments de l'arbre

Trouvez div avec `id="article"` et supprimez toutes les balises script internes.

```
#!/usr/bin/env stack
-- stack --resolver lts-7.1 --install-ghc runghc --package text --package lens --package
taggy-lens --package string-class --package classy-prelude
{-# LANGUAGE NoImplicitPrelude #-}
{-# LANGUAGE OverloadedStrings #-}

import ClassyPrelude
import Control.Lens hiding (children, element)
import Data.String.Class (toText, fromText, toString)
import Data.Text (Text)
import Text.Taggy.Lens
import qualified Text.Taggy.Lens as Taggy
import qualified Text.Taggy.Renderer as Renderer

somehtmlSmall :: Text
```

```
somehtmlSmall =
    "<!doctype html><html><body>\
    \<div id=\"article\"><div>first</div><div>second</div><script>this should be
removed</script><div>third</div></div>\
    \</body></html>"

renderWithoutScriptTag :: Text
renderWithoutScriptTag =
    let mArticle :: Maybe Taggy.Element
        mArticle =
            (fromText somehtmlSmall) ^? html .
            allAttributed (ix "id" . only "article")
        mArticleFiltered =
            fmap
                (transform
                     (children %~
                      filter (\n -> n ^? element . name /= Just "script")))
                mArticle
    in maybe "" (toText . Renderer.render) mArticleFiltered

main :: IO ()
main = print renderWithoutScriptTag
-- outputs:
-- "<div id=\"article\"><div>first</div><div>second</div><div>third</div></div>"
```

Contribution basée sur la réponse SO de @ duplode

Lire Analyse HTML avec lentille taggy et lentille en ligne:
https://riptutorial.com/fr/haskell/topic/6962/analyse-html-avec-lentille-taggy-et-lentille

Chapitre 5: Application partielle

Remarques

Éclaircissons certaines idées fausses que les débutants pourraient faire.

Vous avez peut-être rencontré des fonctions telles que:

```
max :: (Ord a) => a -> a -> a
max m n
  | m >= n = m
  | otherwise = n
```

Les débutants afficheront généralement la fonction `max :: (Ord a) => a -> a -> a` qui prend deux arguments (valeurs) de type `a` et renvoie une valeur de type `a` . Cependant, ce qui se passe réellement, c'est que `max` **prend un argument** de type `a` et **renvoie une fonction** de type `a -> a` . Cette fonction prend alors un argument de type `a` et retourne une valeur finale de type `a` .

En effet, `max` peut être écrit comme `max :: (Ord a) => a -> (a -> a)`

Considérons le type signature de `max` :

```
Prelude> :t max
max :: Ord a => a -> a -> a

Prelude> :t (max 75)
(max 75) :: (Num a, Ord a) => a -> a

Prelude> :t (max "Fury Road")
(max "Fury Road") :: [Char] -> [Char]

Prelude> :t (max "Fury Road" "Furiosa")
(max "Fury Road" "Furiosa") :: [Char]
```

`max 75` et `max "Fury Road"` peuvent ne pas *ressembler* à des fonctions, mais en réalité, elles le sont.

La confusion provient du fait qu'en mathématiques et dans de nombreux autres langages de programmation courants, nous sommes autorisés à avoir des fonctions qui prennent plusieurs arguments. Cependant, dans Haskell, les **fonctions ne peuvent prendre qu'un seul argument** et peuvent renvoyer des valeurs telles que `a` ou des fonctions telles que `a -> a` .

Examples

Fonction d'ajout partiellement appliquée

Nous pouvons utiliser *une application partielle* pour "verrouiller" le premier argument. Après avoir appliqué un argument, on se retrouve avec une fonction qui attend un argument supplémentaire avant de renvoyer le résultat.

```
(+) :: Int -> Int -> Int

addOne :: Int -> Int
addOne = (+) 1
```

Nous pouvons alors utiliser `addOne` pour en ajouter un à un `Int` .

```
> addOne 5
6
> map addOne [1,2,3]
[2,3,4]
```

Renvoi d'une fonction partiellement appliquée

Renvoyer des fonctions partiellement appliquées est une technique pour écrire du code concis.

```
add :: Int -> Int -> Int
add x = (+x)

add 5 2
```

Dans cet exemple (+ x) est une fonction partiellement appliquée. Notez que le deuxième paramètre de la fonction add n'a pas besoin d'être spécifié dans la définition de fonction.

Le résultat de l'appel à l' `add 5 2` est sept.

Sections

La section est un moyen concis d'appliquer partiellement des arguments à des opérateurs infixés.

Par exemple, si nous voulons écrire une fonction qui ajoute "ing" à la fin d'un mot, nous pouvons utiliser une section pour définir succinctement une fonction.

```
> (++ "ing") "laugh"
"laughing"
```

Remarquez comment nous avons partiellement appliqué le deuxième argument. Normalement, nous ne pouvons appliquer que partiellement les arguments dans l'ordre spécifié.

Nous pouvons également utiliser la section gauche pour appliquer partiellement le premier argument.

```
> ("re" ++) "do"
"redo"
```

Nous pourrions écrire ceci de manière équivalente en utilisant une application partielle de préfixe normale:

```
> ((++) "re") "do"
"redo"
```

Une note sur la soustraction

Les débutants sectionnent souvent incorrectement la négation.

```
> map (-1) [1,2,3]
***error: Could not deduce...
```

Cela ne fonctionne pas car `-1` est analysé en tant que littéral `-1` plutôt que l'opérateur sectionné `-` appliqué à `1` . La fonction de `subtract` existe pour contourner ce problème.

```
> map (subtract 1) [1,2,3]
[0,1,2]
```

Lire Application partielle en ligne: https://riptutorial.com/fr/haskell/topic/1954/application-partielle

Chapitre 6: Arithmétique

Introduction

Dans Haskell, toutes les expressions (qui incluent les constantes numériques et les fonctions fonctionnant sur celles-ci) ont un type décidable. Au moment de la compilation, le vérificateur de type déduit le type d'une expression à partir des types des fonctions élémentaires qui le composent. Les données étant immuables par défaut, il n'y a pas d'opérations de "type casting", mais certaines fonctions copient des données et généralisent ou spécialisent les types dans des limites raisonnables.

Remarques

La hiérarchie des classes de caractères numériques

`Num` trouve à la racine de la hiérarchie numérique des classes de caractères. Ses opérations caractéristiques et certaines instances communes sont indiquées ci-dessous (celles chargées par défaut avec Prelude plus celles de `Data.Complex`):

```
λ> :i Num
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a
  {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
      -- Defined in ‘GHC.Num’
instance RealFloat a => Num (Complex a) -- Defined in ‘Data.Complex’
instance Num Word -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Double -- Defined in ‘GHC.Float’
```

Nous avons déjà vu la classe `Fractional` , qui nécessite `Num` et introduit les notions de "division" `(/)` et de réciproque d'un nombre:

```
λ> :i Fractional
class Num a => Fractional a where
  (/) :: a -> a -> a
  recip :: a -> a
  fromRational :: Rational -> a
  {-# MINIMAL fromRational, (recip | (/)) #-}
```

```
      -- Defined in ‘GHC.Real’
instance RealFloat a => Fractional (Complex a) -- Defined in ‘Data.Complex’
instance Fractional Float -- Defined in ‘GHC.Float’
instance Fractional Double -- Defined in ‘GHC.Float’
```

Les modèles de classe `Real` .. les vrais nombres. Il nécessite `Num` et `Ord` , il modélise donc un champ numérique ordonné. Comme contre-exemple, les nombres complexes ne sont *pas* un champ ordonné (c’est-à-dire qu’ils ne possèdent pas de relation d’ordre naturel):

```
λ> :i Real
class (Num a, Ord a) => Real a where
  toRational :: a -> Rational
  {-# MINIMAL toRational #-}
      -- Defined in ‘GHC.Real’
instance Real Word -- Defined in ‘GHC.Real’
instance Real Integer -- Defined in ‘GHC.Real’
instance Real Int -- Defined in ‘GHC.Real’
instance Real Float -- Defined in ‘GHC.Float’
instance Real Double -- Defined in ‘GHC.Float’
```

`RealFrac` représente des nombres qui peuvent être arrondis

```
λ> :i RealFrac
class (Real a, Fractional a) => RealFrac a where
  properFraction :: Integral b => a -> (b, a)
  truncate :: Integral b => a -> b
  round :: Integral b => a -> b
  ceiling :: Integral b => a -> b
  floor :: Integral b => a -> b
  {-# MINIMAL properFraction #-}
      -- Defined in ‘GHC.Real’
instance RealFrac Float -- Defined in ‘GHC.Float’
instance RealFrac Double -- Defined in ‘GHC.Float’
```

`Floating` (qui implique `Fractional`) représente les constantes et les opérations qui ne peuvent pas avoir une expansion décimale finie.

```
λ> :i Floating
class Fractional a => Floating a where
  pi :: a
  exp :: a -> a
  log :: a -> a
  sqrt :: a -> a
  (**) :: a -> a -> a
  logBase :: a -> a -> a
  sin :: a -> a
  cos :: a -> a
  tan :: a -> a
  asin :: a -> a
  acos :: a -> a
  atan :: a -> a
  sinh :: a -> a
  cosh :: a -> a
  tanh :: a -> a
  asinh :: a -> a
  acosh :: a -> a
```

```
  atanh :: a -> a
  GHC.Float.log1p :: a -> a
  GHC.Float.expm1 :: a -> a
  GHC.Float.log1pexp :: a -> a
  GHC.Float.log1mexp :: a -> a
  {-# MINIMAL pi, exp, log, sin, cos, asin, acos, atan, sinh, cosh,
              asinh, acosh, atanh #-}
      -- Defined in ‘GHC.Float’
instance RealFloat a => Floating (Complex a) -- Defined in ‘Data.Complex’
instance Floating Float -- Defined in ‘GHC.Float’
instance Floating Double -- Defined in ‘GHC.Float’
```

Attention: les expressions telles que `sqrt . negate :: Floating a => a -> a` sont parfaitement valides, ils peuvent renvoyer `NaN` ("not-a-number"), ce qui peut ne pas être un comportement voulu. Dans de tels cas, nous pourrions vouloir travailler sur le champ complexe (montré plus tard).

Examples

Exemples de base

```
λ> :t 1
1 :: Num t => t

λ> :t pi
pi :: Floating a => a
```

Dans les exemples ci-dessus, le vérificateur de type déduit une *classe de type* plutôt qu'un type concret pour les deux constantes. Dans Haskell, la classe `Num` est la classe numérique la plus générale (car elle englobe les entiers et les réels), mais `pi` doit appartenir à une classe plus spécialisée, car elle comporte une partie non nulle.

```
list0 :: [Integer]
list0 = [1, 2, 3]

list1 :: [Double]
list1 = [1, 2, pi]
```

Les types de béton ci-dessus ont été déduits par GHC. Des types plus généraux tels que `list0 :: Num a => [a]` auraient fonctionné, mais auraient également été plus difficiles à préserver (par exemple, si on avait consigné un `Double` sur une liste de `Num` s), en raison des réserves indiquées ci-dessus.

`Impossible de déduire (Fractional Int) ... '

Le message d'erreur dans le titre est une erreur courante des débutants. Voyons comment cela se passe et comment le réparer.

Supposons que nous devions calculer la valeur moyenne d'une liste de nombres; la déclaration suivante semblerait le faire, mais elle ne compilerait pas:

```
averageOfList ll = sum ll / length ll
```

Le problème est avec la fonction division `(/)` : sa signature est `(/) :: Fractional a => a -> a -> a` , mais dans le cas au-dessus du dénominateur (donné par `length :: Foldable t => ta -> Int`) est de type `Int` (et `Int` n'appartient pas à la classe `Fractional`) d'où le message d'erreur.

Nous pouvons corriger le message d'erreur avec `fromIntegral :: (Num b, Integral a) => a -> b` . On peut voir que cette fonction accepte les valeurs de tout type `Integral` et renvoie les valeurs correspondantes dans la classe `Num` :

```
averageOfList' :: (Foldable t, Fractional a) => t a -> a
averageOfList' ll = sum ll / fromIntegral (length ll)
```

Exemples de fonction

Quel est le type de `(+)` ?

```
λ> :t (+)
(+) :: Num a => a -> a -> a
```

Quel est le type de `sqrt` ?

```
λ> :t sqrt
sqrt :: Floating a => a -> a
```

Quel est le type de `sqrt . fromIntegral` ?

```
sqrt . fromIntegral :: (Integral a, Floating c) => a -> c
```

Lire Arithmétique en ligne: https://riptutorial.com/fr/haskell/topic/8616/arithmetique

Chapitre 7: Attoparsec

Introduction

Attoparsec est une bibliothèque d'analyse combinatoire qui "vise particulièrement à traiter efficacement les protocoles réseau et les formats de fichiers texte / binaires compliqués".

Attoparsec offre non seulement la rapidité et l'efficacité, mais aussi un retour en arrière et des entrées incrémentielles.

Son API reflète étroitement celle d'une autre bibliothèque de combineurs d'analyseurs, Parsec.

Il existe des sous-modules pour la compatibilité avec `ByteString` , `Text` et `Char8` . L'utilisation de l'extension de langue `OverloadedStrings` est recommandée.

Paramètres

Type	Détail
`Parser ia`	Le type de base pour représenter un analyseur. `i` est le type de chaîne, par exemple `ByteString` .
`IResult ir`	Le résultat d'une analyse, avec `Fail i [String] String` , `Partial (i -> IResult ir)` et `Done ir` tant que constructeurs.

Examples

Combinateurs

Le meilleur moyen d’analyser les entrées est d’utiliser des fonctions d’analyse plus larges, composées de fonctions plus petites et plus simples.

Disons que nous voulions analyser le texte suivant qui représente les heures de travail:

> Lundi: 0800 1600.

Nous pourrions les diviser en deux "jetons": le nom du jour - "lundi" - et une partie du temps "0800" à "1600".

Pour analyser un nom de jour, nous pourrions écrire ce qui suit:

```
data Day = Day String

day :: Parser Day
day = do
  name <- takeWhile1 (/= ':')
```

```
  skipMany1 (char ':')
  skipSpace
  return $ Day name
```

Pour analyser la partie du temps, nous pourrions écrire:

```
data TimePortion = TimePortion String String

time = do
    start <- takeWhile1 isDigit
    skipSpace
    end <- takeWhile1 isDigit
    return $ TimePortion start end
```

Maintenant, nous avons deux analyseurs pour nos parties individuelles du texte, nous pouvons les combiner dans un analyseur "plus grand" pour lire les heures de travail d'une journée entière:

```
data WorkPeriod = WorkPeriod Day TimePortion

work = do
    d <- day
    t <- time
    return $ WorkPeriod d t
```

puis exécutez l'analyseur:

```
parseOnly work "Monday: 0800 1600"
```

Bitmap - Analyse des données binaires

Attoparsec rend l'analyse des données binaires triviale. En supposant ces définitions:

```
import           Data.Attoparsec.ByteString (Parser, eitherResult, parse, take)
import           Data.Binary.Get            (getWord32le, runGet)
import           Data.ByteString            (ByteString, readFile)
import           Data.ByteString.Char8      (unpack)
import           Data.ByteString.Lazy       (fromStrict)
import           Prelude                    hiding (readFile, take)

-- The DIB section from a bitmap header
data DIB = BM | BA | CI | CP | IC | PT
           deriving (Show, Read)

type Reserved = ByteString

-- The entire bitmap header
data Header = Header DIB Int Reserved Reserved Int
              deriving (Show)
```

Nous pouvons analyser facilement l’en-tête d’un fichier bitmap. Ici, nous avons 4 fonctions d'analyseur qui représentent la section d'en-tête d'un fichier bitmap:

Tout d'abord, la section DIB peut être lue en prenant les 2 premiers octets

```
dibP :: Parser DIB
dibP = read . unpack <$> take 2
```

De même, la taille du bitmap, les sections réservées et le décalage des pixels peuvent être lus facilement:

```
sizeP :: Parser Int
sizeP = fromIntegral . runGet getWord32le . fromStrict <$> take 4

reservedP :: Parser Reserved
reservedP = take 2

addressP :: Parser Int
addressP = fromIntegral . runGet getWord32le . fromStrict <$> take 4
```

qui peuvent ensuite être combinés dans une fonction d'analyse plus grande pour l'en-tête entier:

```
bitmapHeader :: Parser Header
bitmapHeader = do
    dib <- dibP
    sz <- sizeP
    reservedP
    reservedP
    offset <- addressP
    return $ Header dib sz "" "" offset
```

Lire Attoparsec en ligne: https://riptutorial.com/fr/haskell/topic/9681/attoparsec

Chapitre 8: Banane réactive

Examples

Injection d'événements externes dans la bibliothèque

Cet exemple n'est lié à aucune boîte à outils graphique, comme le fait réactif-banana-wx, par exemple. Au lieu de cela, il montre comment injecter des actions `IO` arbitraires dans les machines FRP.

Le module `Control.Event.Handler` fournit une fonction `addHandler` qui crée une paire de valeurs `AddHandler a` et `a -> IO ()` . Le premier est utilisé par reactive-banana lui-même pour obtenir un `Event a` , alors que le second est une fonction simple utilisée pour déclencher l'événement correspondant.

```
import Data.Char (toUpper)

import Control.Event.Handler
import Reactive.Banana

main = do
    (inputHandler, inputFire) <- newAddHandler
```

Dans notre cas `a` paramètre du gestionnaire est de type `String` , mais le code qui permet de déduire du compilateur qui sera écrit plus tard.

Nous définissons maintenant le `EventNetwork` qui décrit notre système FRP. Ceci est fait en utilisant la fonction de `compile` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
```

La fonction `fromAddHandler` transforme `AddHandler a` valeur `Event a` , qui est traitée dans l'exemple suivant.

Enfin, nous lançons notre "boucle d'événements", qui déclencherait des événements sur la saisie de l'utilisateur:

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
    forever $ do
        input <- getLine
        inputFire input
```

Type d'événement

En mode réactif-banane, le type d' `Event` représente un flux de certains événements dans le temps. Un `Event` est similaire à un signal d'impulsion analogique dans le sens où il n'est pas continu dans le temps. Par conséquent, `Event` est une instance de la `Functor` types Functor uniquement. Vous ne pouvez pas combiner deux `Event` ensemble car ils peuvent tirer à des moments différents. Vous pouvez faire quelque chose avec la valeur [actuelle] d'un `Event` et y réagir avec une action `IO` .

Les transformations sur la valeur de l' `Event` sont effectuées en utilisant `fmap` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
        -- turn all characters in the signal to upper case
        let inputEvent' = fmap (map toUpper) inputEvent
```

Réagir à un `Event` se fait de la même manière. D'abord, vous le `fmap` avec une action de type `a -> IO ()` et vous la transmettez ensuite à la fonction `reactimate` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        inputEvent <- fromAddHandler inputHandler
        -- turn all characters in the signal to upper case
        let inputEvent' = fmap (map toUpper) inputEvent
        let inputEventReaction = fmap putStrLn inputEvent' -- this has type `Event (IO ())
        reactimate inputEventReaction
```

Maintenant, chaque fois que `inputFire "something"` est appelé `"SOMETHING"` , `"SOMETHING"` sera imprimé.

Type de comportement

Pour représenter des signaux continus, les caractéristiques réactives-bananes `Behavior a` type. Contrairement à `Event` , un `Behavior` est un `Applicative` , qui vous permet de combiner n `Behavior` aide d'une fonction pure n-aire (en utilisant `<$>` et `<*>`).

Pour obtenir un `Behavior a` partir de l' `Event a` il existe `accumE` fonction d' `accumE` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
```

`accumE` prend la valeur initiale de `Behavior` et un `Event` , contenant une fonction qui lui donnerait la nouvelle valeur.

Comme avec `Event` s, vous pouvez utiliser `fmap` pour travailler avec la valeur actuelle du `Behavior` , mais vous pouvez également les combiner avec `(<*>)` .

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
```

Pour réagir aux changements de `Behavior` , il y a une fonction de `changes` :

```
main = do
    (inputHandler, inputFire) <- newAddHandler
    compile $ do
        ...
        inputBehavior  <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        inputBehavior' <- accumE "" $ fmap (\oldValue newValue -> newValue) inputEvent
        let constantTrueBehavior = (==) <$> inputBehavior <*> inputBehavior'
        inputChanged <- changes inputBehavior
```

La seule chose à noter est que les `changes` renvoient l' `Event (Future a)` au lieu de l' `Event a` . À cause de cela, `reactimate'` doit être utilisé au lieu de `reactimate` . La justification derrière cela peut être obtenue de la documentation.

Activer les réseaux d'événements

`EventNetwork` retournés par `compile` doivent être activés avant que les événements réactivés aient un effet.

```
main = do
    (inputHandler, inputFire) <- newAddHandler

    eventNetwork <- compile $ do
        inputEvent <- fromAddHandler inputHandler
        let inputEventReaction = fmap putStrLn inputEvent
        reactimate inputEventReaction

    inputFire "This will NOT be printed to the console!"
    actuate eventNetwork
    inputFire "This WILL be printed to the console!"
```

Lire Banane réactive en ligne: https://riptutorial.com/fr/haskell/topic/4186/banane-reactive

Chapitre 9: Bases de données

Examples

Postgres

Postgresql-simple est une bibliothèque Haskell de niveau intermédiaire pour communiquer avec une base de données backend PostgreSQL. Il est très simple à utiliser et fournit une API sécurisée pour la lecture / écriture dans une base de données.

Exécuter une requête simple est aussi simple que:

```
{-# LANGUAGE OverloadedStrings #-}

import Database.PostgreSQL.Simple

main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query_ conn "select 2 + 2"  -- execute with no parameter substitution
  print i
```

Substitution de paramètre

PostreSQL-Simple prend en charge la substitution de paramètres pour les requêtes paramétrées sûres à l'aide de `query` :

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  [Only i] <- query conn "select ? + ?" [1, 1]
  print i
```

Exécution d'insertions ou de mises à jour

Vous pouvez exécuter des insertions / mises à jour de requêtes SQL en utilisant `execute` :

```
main :: IO ()
main = do
  -- Connect using libpq strings
  conn <- connectPostgreSQL "host='my.dbhost' port=5432 user=bob pass=bob"
  execute conn "insert into people (name, age) values (?, ?)" ["Alex", 31]
```

Lire Bases de données en ligne: https://riptutorial.com/fr/haskell/topic/4444/bases-de-donnees

Chapitre 10: Bifunctor

Syntaxe

- bimap :: (a -> b) -> (c -> d) -> pac -> pbd
- d'abord :: (a -> b) -> pac -> pbc
- second :: (b -> c) -> pab -> pac

Remarques

Un fonctionnement du broyeur `Functor` est covariant dans un paramètre de type *unique* . Par exemple, si `f` est un `Functor` , alors on attribue une `fa` et une fonction de la forme `a -> b` , on peut obtenir un `fb` (en utilisant `fmap`).

Un `Bifunctor` est covariant dans *deux* paramètres de type. Si `f` est un `Bifunctor` , on donne alors une `fab` , et deux fonctions, une de `a -> c` et une autre de `b -> d` , on peut alors obtenir un `fcd` (en utilisant `bimap`).

`fmap first` devrait penser à un `fmap` sur le premier paramètre de type, `second` à un `fmap` sur le second et `bimap` devrait être conçu comme mappant deux fonctions de façon covariante respectivement sur les paramètres de premier et de second type.

Examples

Instances courantes de Bifunctor

Tuples à deux éléments

`(,)` est un exemple de type qui a une instance `Bifunctor` .

```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

`bimap` prend une paire de fonctions et les applique aux composants respectifs du tuple.

```
bimap (+ 2) (++ "nie") (3, "john") --> (5,"johnnie")
bimap ceiling length (3.5 :: Double, "john" :: String) --> (4,4)
```

Either

`Either` 'une des instances de `Bifunctor` sélectionne une des deux fonctions à appliquer selon que la valeur est `Left` ou `Right` .

```
instance Bifunctor Either where
    bimap f g (Left x) = Left (f x)
    bimap f g (Right y) = Right (g y)
```

premier et deuxième

Si le mappage covariant sur le premier argument seulement, ou seulement sur le second argument, est souhaité, alors le `first` ou le `second` devrait être utilisé (au lieu de `bimap`).

```
first :: Bifunctor f => (a -> c) -> f a b -> f c b
first f = bimap f id

second :: Bifunctor f => (b -> d) -> f a b -> f a d
second g = bimap id g
```

Par exemple,

```
ghci> second (+ 2) (Right 40)
Right 42
ghci> second (+ 2) (Left "uh oh")
Left "uh oh"
```

Définition de Bifunctor

`Bifunctor` est la classe de types avec deux paramètres de type (`f :: * -> * -> *`), les deux pouvant être mappés de manière covariante simultanément.

```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
```

`bimap` peut être considéré comme appliquant une paire d'opérations `fmap` à un type de données.

Une instance correcte de `Bifunctor` pour un type `f` doit satisfaire aux *lois de bifonctor* , qui sont analogues aux *lois de foncteur* :

```
bimap id id = id  -- identity
bimap (f . g) (h . i) = bimap f h . bimap g i  -- composition
```

La classe `Bifunctor` se trouve dans le module `Data.Bifunctor` . Pour les versions GHC> 7.10, ce module est fourni avec le compilateur; pour les versions antérieures, vous devez installer le package `bifunctors` .

Lire Bifunctor en ligne: https://riptutorial.com/fr/haskell/topic/8020/bifunctor

Chapitre 11: Cabale

Syntaxe

- cabal <command> où <command> est l'un des suivants:
- **[global]**
  - mettre à jour
    - Liste des mises à jour des paquets connus
  - installer
    - Installer des paquets
  - Aidez-moi
    - Aide sur les commandes
  - Info
    - Afficher des informations détaillées sur un paquet particulier
  - liste
    - Liste des paquets correspondant à une chaîne de recherche
  - chercher
    - Télécharge les paquets pour une installation ultérieure
  - configuration de l'utilisateur
    - Afficher et mettre à jour la configuration globale de l'utilisateur
- **[paquet]**
  - obtenir
    - Télécharger / Extraire le code source d'un package (référentiel)
  - init
    - Créer un nouveau fichier de package .cabal (interactivement)
  - configurer
    - Préparez-vous à construire le package
  - construire
    - Compiler tous les composants / spécifiques
  - nettoyer
    - Nettoyer après une construction
  - courir
    - Construit et exécute un exécutable
  - repl
    - Ouvrir une session d'interprète pour le composant donné
  - tester
    - Exécuter tous les tests spécifiques dans la suite de tests
  - banc
    - Exécuter tous les benchmarks / spécifiques
  - vérifier
    - Vérifiez le paquet pour les erreurs courantes
  - sdist
    - Générer un fichier de distribution source (.tar.gz)
  - télécharger

        - Télécharge des paquets source ou de la documentation sur Hackage
    - rapport
        - Télécharger des rapports de build sur un serveur distant
    - gel
        - Geler les dépendances
    - limites de la gen
        - Générer des bornes de dépendance
    - églefin
        - Générer de la documentation Haddock HTML
    - hscolour
        - Générer du code coloré HsColour, au format HTML
    - copie
        - Copiez les fichiers dans les emplacements d'installation
    - registre
        - Enregistrez ce paquet avec le compilateur
- **[bac à sable]**
    - bac à sable
        - Créer / modifier / supprimer un bac à sable
            - cabal sandbox init [DRAPEAUX]
            - cabal sandbox supprimer [DRAPEAUX]
            - bac à sable cabal add-source [FLAGS] PATHS
            - cabal sandbox delete-source [FLAGS] PATHS
            - cabal sandbox list-sources [DRAPEAUX]
            - bac à sable cabal hc-pkg [DRAPEAUX] [-] COMMANDE [-] [ARGS]
    - exec
        - Donner un accès de commande au référentiel de packages sandbox
    - repl
        - Interprète ouvert avec accès aux packages sandbox

Examples

Installer des paquets

Pour installer un nouveau paquet, par exemple aeson:

```
cabal install aeson
```

Travailler avec des bacs à sable

Un projet Haskell peut soit utiliser les packages à l'échelle du système, soit utiliser un bac à sable. Un sandbox est une base de données de packages isolés et peut empêcher les conflits de dépendance, par exemple si plusieurs projets Haskell utilisent des versions différentes d'un package.

Pour initialiser un sandbox pour un package Haskell, accédez à son répertoire et exécutez:

```
cabal sandbox init
```

Maintenant, les paquets peuvent être installés en exécutant simplement l' `cabal install` .

Liste des paquets dans un sandbox:

```
cabal sandbox hc-pkg list
```

Supprimer un sandbox:

```
cabal sandbox delete
```

Ajouter une dépendance locale:

```
cabal sandbox add-source /path/to/dependency
```

Lire Cabale en ligne: https://riptutorial.com/fr/haskell/topic/4740/cabale

Chapitre 12: Classes de types

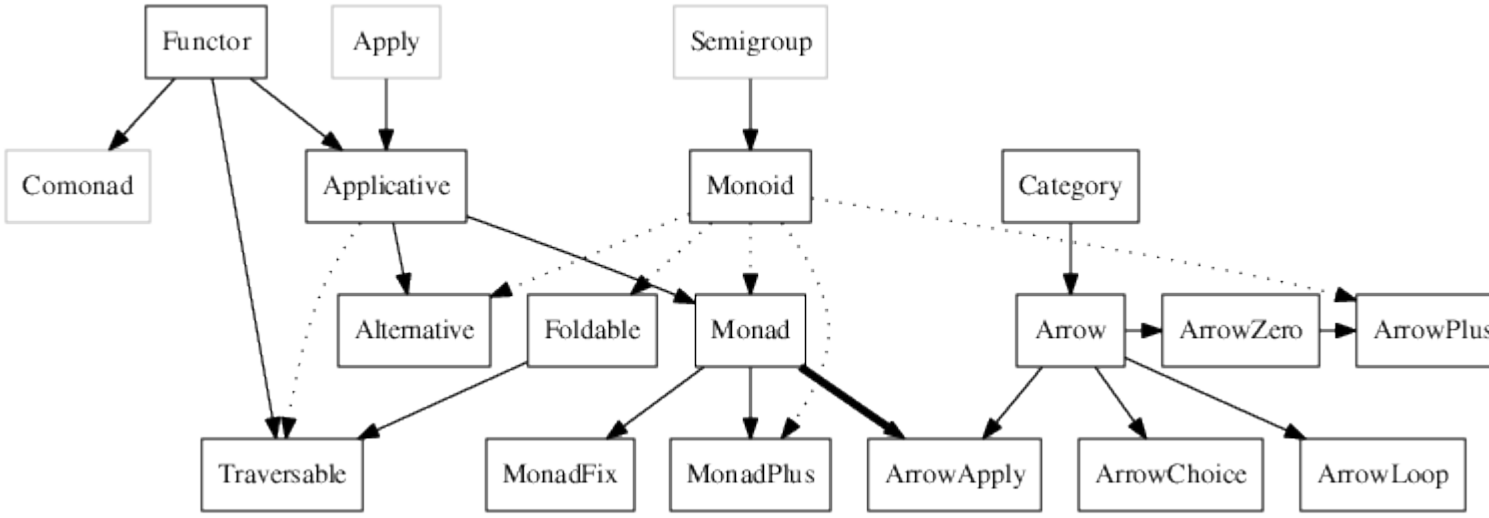
Introduction

Les classes de caractères dans Haskell permettent de définir le comportement associé à un type séparément de la définition de ce type. Alors que, par exemple, en Java, vous définissez le comportement comme faisant partie de la définition du type - c'est-à-dire que dans une interface, une classe abstraite ou une classe concrète - Haskell garde ces deux choses séparées.

Un certain nombre de classes de caractères sont déjà définies dans le package de `base` d'Haskell. La relation entre ceux-ci est illustrée dans la section Remarques ci-dessous.

Remarques

Le diagramme suivant tiré de l'article de Typeclassopedia montre la relation entre les différentes classes de caractères de Haskell.



Examples

Peut-être et la classe Functor

Dans Haskell, les types de données peuvent avoir des arguments comme les fonctions. Prenez le type `Maybe` par exemple.

`Maybe` - `Maybe` est-ce un type très utile qui nous permet de représenter l'idée de l'échec ou de sa possibilité. En d'autres termes, si un calcul risque d'échouer, nous utiliserons le type `Maybe` ici. `Maybe` agit comme un wrapper pour d'autres types, en leur donnant des fonctionnalités supplémentaires.

Sa déclaration actuelle est assez simple.

```
Maybe a = Just a | Nothing
```

Ce que cela raconte, c'est qu'un `Maybe` présente sous deux formes, un `Just` , qui représente le succès, et un `Nothing` , qui représente un échec. `Just` un argument qui détermine le type de l' `Maybe` , et `Nothing` n'en prend aucun. Par exemple, la valeur `Just "foo"` aura le type `Maybe String` , qui est un type de chaîne encapsulé avec la fonctionnalité `Maybe` supplémentaire. La valeur `Nothing` est de type `Maybe a` - `a` `Maybe a` où `a` peut être de tout type.

Cette idée d'encapsuler les types pour leur donner des fonctionnalités supplémentaires est très utile et s'applique à plus que `Maybe` - `Maybe` . D'autres exemples incluent les types `Either` , `IO` et list, chacun offrant des fonctionnalités différentes. Cependant, certaines actions et capacités sont communes à tous ces types d'encapsuleurs. Le plus notable est la possibilité de modifier la valeur encapsulée.

Il est courant de penser à ces types de types comme des boîtes pouvant contenir des valeurs. Différentes boîtes contiennent des valeurs différentes et font des choses différentes, mais aucune n'est utile sans pouvoir accéder au contenu.

Pour encapsuler cette idée, Haskell est livré avec une classe de caractères standard, nommée `Functor` . Il est défini comme suit.

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Comme on peut le voir, la classe a une seule fonction, `fmap` , de deux arguments. Le premier argument est une fonction d'un type, d' `a` à l'autre, `b` . Le second argument est un foncteur (type wrapper) contenant une valeur de type `a` . Il retourne un foncteur (type wrapper) contenant une valeur de type `b` .

En termes simples, `fmap` prend une fonction et s'applique à la valeur à l'intérieur d'un foncteur. C'est la seule fonction nécessaire pour qu'un type soit membre de la classe `Functor` , mais c'est extrêmement utile. Les classes fonctionnant sur des foncteurs ayant des applications plus spécifiques peuvent être trouvées dans les classes de types `Applicative` et `Monad` .

Classe d'héritage: Classe de type Ord

Haskell supporte une notion d'extension de classe. Par exemple, la classe `Ord` hérite de toutes les opérations dans `Eq` , mais a en outre une fonction de `compare` qui renvoie une `Ordering` entre les valeurs. `Ord` peut également contenir les opérateurs de comparaison de commandes courants, ainsi qu'une méthode `min` et une méthode `max` .

La notation `=>` a la même signification que dans une signature de fonction et nécessite le type `a` pour implémenter `Eq` , afin de mettre en œuvre `Ord` .

```
data Ordering = EQ | LT | GT

class Eq a => Ord a where
    compare :: Ord a => a -> a -> Ordering
    (<)     :: Ord a => a -> a -> Bool
```

```
    (<=)    :: Ord a => a -> a -> Bool
    (>)     :: Ord a => a -> a -> Bool
    (>=)    :: Ord a => a -> a -> Bool
    min     :: Ord a => a -> a -> a
    max     :: Ord a => a -> a -> a
```

Toutes les méthodes suivantes peuvent être `compare` de différentes manières:

```
x < y   = compare x y == LT
x <= y  = x < y || x == y -- Note the use of (==) inherited from Eq
x > y   = not (x <= y)
x >= y  = not (x < y)

min x y = case compare x y of
               EQ -> x
               LT -> x
               GT -> y

max x y = case compare x y of
               EQ -> x
               LT -> y
               GT -> x
```

Les classes de type qui étendent elles-mêmes `Ord` doivent implémenter au moins la méthode `compare` ou la méthode `(<=)` elles-mêmes, ce qui crée le réseau d'héritage dirigé.

Eq

Tous les types de données de base (comme `Int` , `String` , `Eq a => [a]`) de Prelude sauf pour les fonctions et `IO` ont des instances de `Eq` . Si un type instancie `Eq` cela signifie que nous savons comparer deux valeurs pour une *valeur* ou *une* égalité *structurelle* .

```
> 3 == 2
False
> 3 == 3
True
```

Méthodes requises

- `(==) :: Eq a => a -> a -> Boolean` ou `(/=) :: Eq a => a -> a -> Boolean` (si un seul est implémenté, l'autre utilise par défaut la négation du défini un)

Définit

- `(==) :: Eq a => a -> a -> Boolean`
- `(/=) :: Eq a => a -> a -> Boolean`

Superclasses directes

Aucun

Sous-classes notables

- `Ord`

Ord

Les types instanciant `Ord` incluent, par exemple, `Int` , `String` et `[a]` (pour les types `a` où il y a une instance `Ord a`). Si un type instancie `Ord` cela signifie que nous connaissons un ordre «naturel» de valeurs de ce type. Notez qu'il y a souvent beaucoup de choix possibles pour un ordre "naturel" d'un type et `Ord` nous oblige à en privilégier un.

`Ord` fournit les opérateurs standard `(<=)` , `(<)` , `(>)` , `(>=)` mais les définit de manière intéressante en utilisant un type de données algébrique personnalisé

```
data Ordering = LT | EQ | GT

compare :: Ord a => a -> a -> Ordering
```

Méthodes requises

- `compare :: Ord a => a -> a -> Ordering` ou `(<=) :: Ord a => a -> a -> Boolean` (la méthode de `compare` par défaut du standard utilise `(<=)` dans son implémentation)

Définit

- `compare :: Ord a => a -> a -> Ordering`
- `(<=) :: Ord a => a -> a -> Boolean`
- `(<) :: Ord a => a -> a -> Boolean`
- `(>=) :: Ord a => a -> a -> Boolean`
- `(>) :: Ord a => a -> a -> Boolean`
- `min :: Ord a => a -> a -> a`
- `max :: Ord a => a -> a -> a`

Superclasses directes

- `Eq`

Monoïde

Les types d'instanciation de `Monoid` incluent, entre autres, des listes, des nombres et des fonctions avec des valeurs de retour `Monoid` . Pour instancier un `Monoid` un type doit prendre en charge une opération binaire associative (`mappend` ou `(<>)`) qui combine ses valeurs et avoir une valeur "zéro"

spéciale (`mempty`) telle que combiner une valeur avec elle ne change pas cette valeur:

```
mempty  <>  x == x
x <>  mempty  == x

x <> (y <> z) == (x <> y) <> z
```

Intuitivement, les types de `Monoid` sont "similaires à la liste" dans la mesure où ils prennent en charge les valeurs ajoutées ensemble. Alternativement, les types `Monoid` peuvent être considérés comme des séquences de valeurs pour lesquelles nous nous préoccupons de l'ordre mais pas du groupement. Par exemple, un arbre binaire est un `Monoid` , mais en utilisant les opérations `Monoid` , nous ne pouvons pas voir sa structure de branchement, seulement une traversée de ses valeurs (voir `Foldable` et `Traversable`).

Méthodes requises

- `mempty :: Monoid m => m`
- `mappend :: Monoid m => m -> m -> m`

Superclasses directes

Aucun

Num

La classe la plus générale pour les types de nombres, plus précisément pour les anneaux , c'est-à-dire les nombres pouvant être ajoutés et soustraits et multipliés au sens habituel, mais pas nécessairement divisés.

Cette classe contient à la fois des types intégraux (`Int` , `Integer` , `Word32` etc.) et des types fractionnaires (`Double` , `Rational` , ainsi que des nombres complexes, etc.). Dans le cas des types finis, la sémantique est généralement comprise comme *une arithmétique modulaire* , c'est-à-dire avec un dépassement et un débordement † .

Notez que les règles pour les classes numériques sont beaucoup moins strictement respectées que les lois monades ou monoïdes, ou celles pour la comparaison d'égalité . En particulier, les nombres à virgule flottante obéissent généralement aux lois uniquement dans un sens approximatif.

Les méthodes

- `fromInteger :: Num a => Integer -> a` . convertir un entier au type de nombre général (en entourant la plage, si nécessaire). Les littéraux numériques de Haskell peuvent être compris comme un littéral `Integer` monomorphe avec la conversion générale qui l'entoure. Vous pouvez donc utiliser le littéral `5` à la fois dans un contexte `Int` et un paramètre `Complex Double` .

- `(+) :: Num a => a -> a -> a` . Ajout standard, généralement compris comme associatif et commutatif, c'est-à-dire

```
a + (b + c) ≡ (a + b) + c
a + b ≡ b + a
```

- `(-) :: Num a => a -> a -> a` . Soustraction, qui est l'inverse de l'addition:

```
(a - b) + b ≡ (a + b) - b ≡ a
```

- `(*) :: Num a => a -> a -> a` . Multiplication, une opération associative distributive par rapport à l'addition:

```
a * (b * c) ≡ (a * b) * c
a * (b + c) ≡ a * b + a * c
```

  pour les cas les plus courants, la multiplication est également commutative, mais ce n'est certainement pas une exigence.

- `negate :: Num a => a -> a` . Le nom complet de l'opérateur de négation unaire. `-1` est le sucre syntaxique pour `negate 1` .

```
-a ≡ negate a ≡ 0 - a
```

- `abs :: Num a => a -> a` . La fonction de valeur absolue donne toujours un résultat non négatif de la même amplitude

```
abs (-a) ≡ abs a
abs (abs a) ≡ abs a
```

  `abs a ≡ 0` ne devrait arriver que si `a ≡ 0` .

  Pour les types réels, il est clair ce que signifie non négatif: vous avez toujours `abs a >= 0` . Les types complexes, etc., n'ont pas un ordre bien défini, mais le résultat des `abs` doit toujours se situer dans le sous-ensemble réel ‡ (c.-à-d. Donner un nombre qui pourrait aussi être écrit sous forme de littéral sans négation).

- `signum :: Num a => a -> a` . La fonction de signe, selon le nom, ne donne que `-1` ou `1` , en fonction du signe de l'argument. En fait, c'est seulement vrai pour les nombres réels non nuls; en général, la `signum` est mieux comprise comme la fonction de *normalisation* :

```
abs (signum a) ≡ 1   -- unless a≡0
signum a * abs a ≡ a -- This is required to be true for all Num instances
```

  Notez que la section 6.4.4 du rapport Haskell 2010 exige explicitement que cette dernière égalité soit valide pour toute instance `Num` valide.

Certaines bibliothèques, notamment les bibliothèques linéaires et hmatrix , ont une compréhension beaucoup plus laxiste de la classe `Num` : elles la traitent simplement comme *un moyen de surcharger les opérateurs arithmétiques* . Bien que ce soit assez simple pour `+` et `-` , cela devient déjà gênant avec `*` et plus encore avec les autres méthodes. Par exemple, *devrait-on dire `*` multiplication par matrice ou multiplication par éléments?*
Il est sans doute une mauvaise idée de définir de telles instances sans nombre; Veuillez considérer des classes dédiées telles que `VectorSpace` .

---

† En particulier, les «négatifs» des types non signés sont entourés d'un grand positif, par exemple `(-4 :: Word32) == 4294967292` .

‡ Ce n'est *pas le cas* : les types de vecteurs n'ont pas de sous-ensemble réel. La controverse `Num` -instances pour ces types définissent généralement `abs` et `signum` élément par élément, qui mathématiquement parlant ne fait pas vraiment de sens.

Lire Classes de types en ligne: https://riptutorial.com/fr/haskell/topic/1879/classes-de-types

Chapitre 13: Composition de fonction

Remarques

L'opérateur de composition de fonctions `(.)` Est défini comme

```
(.) :: (b -> c) -> (a -> b) ->  (a -> c)
(.)       f           g          x =  f (g x)     -- or, equivalently,

(.)       f           g     =   \x -> f (g x)
(.)       f     =     \g    ->  \x -> f (g x)
(.) =     \f    ->    \g    ->  \x -> f (g x)
(.) =     \f    ->   (\g    -> (\x -> f (g x) ) )
```

Le type `(b -> c) -> (a -> b) -> (a -> c)` peut être écrit comme `(b -> c) -> (a -> b) -> a -> c` car le `->` dans les signatures de type "associates" à droite, correspondant à l'application associée à la gauche,

```
 f g x y z ...    ==    (((f g) x) y) z ...
```

Donc le "dataflow" est de droite à gauche: `x` "va" dans `g` , dont le résultat entre dans `f` , produisant le résultat final:

```
(.)       f           g          x =  r
                                      where r = f (g x)
-- g :: a -> b
-- f ::      b -> c
-- x :: a
-- r ::           c

(.)       f           g     =    q
                                 where q = \x -> f (g x)
-- g :: a -> b
-- f ::      b -> c
-- q :: a      -> c

....
```

Syntaxiquement, les éléments suivants sont identiques:

```
(.) f g x  =  (f . g) x  =  (f .) g x  =  (. g) f x
```

ce qui est facile à comprendre comme "les trois règles des sections de l' opérateur ", où "l'argument manquant" va juste dans l'emplacement vide près de l'opérateur:

```
(.) f g    =  (f . g)    =  (f .) g    =  (. g) f
--         1             2             3
```

Le `x` , présent des deux côtés de l'équation, peut être omis. Ceci est connu comme eta-

contraction. Ainsi, la manière la plus simple d'écrire la définition de la composition de fonction est juste

```
(f . g) x   =   f (g x)
```

Cela fait bien sûr référence à "l'argument" `x` ; chaque fois que nous écrivons simplement `(f . g)` sans le `x` il est appelé style sans point.

Examples

Composition de droite à gauche

`(.)` nous permet de composer deux fonctions, alimentant la sortie de l'une en entrée de l'autre:

```
(f . g) x = f (g x)
```

Par exemple, si nous voulons équarrir le successeur d'un nombre d'entrée, nous pouvons écrire

```
((^2) . succ) 1        --    4
```

Il y a aussi `(<<<)` qui est un alias à `(.)` . Alors,

```
(+ 1) <<< sqrt $ 25    --    6
```

Composition de gauche à droite

`Control.Category` définit `(>>>)` , qui, spécialisé dans les fonctions, est

```
-- (>>>) :: Category cat => cat a b -> cat b c -> cat a c
-- (>>>) :: (->) a b -> (->) b c -> (->) a c
-- (>>>) :: (a -> b) -> (b -> c) -> (a -> c)
( f >>> g ) x = g (f x)
```

Exemple:

```
sqrt >>> (+ 1) $ 25    --    6.0
```

Composition avec fonction binaire

La composition régulière fonctionne pour des fonctions unaires. Dans le cas du binaire, on peut définir

```
(f .: g) x y = f (g x y)          -- which is also
             = f ((g x) y)
             = (f . g x) y        -- by definition of (.)
             = (f .) (g x) y
             = ((f .) . g) x y
```

Ainsi, `(f .: g) = ((f .) . g)` par eta-contraction, et de plus,

```
(.:) f g    = ((f .) . g)
            = (.) (f .) g
            = (.) ((.) f) g
            = ((.) . (.)) f g
```

so `(.:) = ((.) . (.))` , une définition semi-célèbre.

Exemples:

```
(map (+1) .: filter) even [1..5]      --  [3,5]
(length   .: filter) even [1..5]      --  2
```

Lire Composition de fonction en ligne: https://riptutorial.com/fr/haskell/topic/4430/composition-de-fonction

Chapitre 14: Concurrence

Remarques

Les bonnes ressources pour apprendre sur la programmation concurrente et parallèle dans Haskell sont:

- Programmation parallèle et simultanée dans Haskell
- le wiki Haskell

Examples

Fils de frai avec `forkIO`

Haskell prend en charge de nombreuses formes d' `forkIO` simultané et la plus évidente `forkIO` à `forkIO` un thread à l'aide de `forkIO` .

La fonction `forkIO :: IO () -> IO ThreadId` prend une action `IO` et retourne son `ThreadId` , tandis que l'action sera exécutée en arrière-plan.

Nous pouvons démontrer cela de manière succincte en utilisant `ghci` :

```
Prelude Control.Concurrent> forkIO $ (print . sum) [1..100000000]
ThreadId 290
Prelude Control.Concurrent> forkIO $ print "hi!"
"hi!"
-- some time later....
Prelude Control.Concurrent> 50000005000000
```

Les deux actions se dérouleront en arrière-plan et la seconde sera presque terminée avant la dernière!

Communication entre les threads avec `MVar`

Il est très facile de transmettre des informations entre les threads en utilisant le type `MVar a` et ses fonctions `Control.Concurrent` dans `Control.Concurrent` :

- `newEmptyMVar :: IO (MVar a)` - crée un nouveau `MVar a`
- `newMVar :: a -> IO (MVar a)` - crée un nouveau `MVar` avec la valeur donnée
- `takeMVar :: MVar a -> IO a` - récupère la valeur du `MVar` donné, ou **bloque** jusqu'à ce qu'une soit disponible
- `putMVar :: MVar a -> a -> IO ()` - met la valeur donnée dans le `MVar` , ou **bloque** jusqu'à ce qu'il soit vide

Sommez les nombres de 1 à 100 millions dans un thread et attendez le résultat:

```
import Control.Concurrent
main = do
  m <- newEmptyMVar
  forkIO $ putMVar m $ sum [1..10000000]
  print =<< takeMVar m  -- takeMVar will block 'til m is non-empty!
```

Une démonstration plus complexe pourrait consister à prendre l'entrée utilisateur et la somme en arrière-plan en attendant d'autres entrées:

```
main2 = loop
  where
    loop = do
        m <- newEmptyMVar
        n <- getLine
        putStrLn "Calculating. Please wait"
        -- In another thread, parse the user input and sum
        forkIO $ putMVar m $ sum [1..(read n :: Int)]
        -- In another thread, wait 'til the sum's complete then print it
        forkIO $ print =<< takeMVar m
        loop
```

Comme indiqué plus haut, si vous appelez `takeMVar` et que le `MVar` est vide, il se bloque jusqu'à ce qu'un autre thread insère quelque chose dans le `MVar` , ce qui pourrait entraîner un problème `takeMVar` aux philosophes de la restauration . La même chose se produit avec `putMVar` : si c'est plein, ça va bloquer jusqu'à ce qu'il soit vide!

Prenez la fonction suivante:

```
concurrent ma mb = do
  a <- takeMVar ma
  b <- takeMVar mb
  putMVar ma a
  putMVar mb b
```

Nous exécutons les deux fonctions avec des `MVar`

```
concurrent ma mb     -- new thread 1
concurrent mb ma     -- new thread 2
```

Ce qui pourrait arriver, c'est que:

1. Le fil 1 lit `ma` et bloque `ma`
2. Le thread 2 lit `mb` et bloque donc `mb`

Maintenant, le thread 1 ne peut pas lire `mb` car le thread 2 l'a bloqué et le thread 2 ne peut pas lire `ma` car le thread 1 l'a bloqué. Une impasse classique!

Blocs atomiques avec mémoire transactionnelle logicielle

Un autre outil concurrentiel puissant et évolué de Haskell est Software Transactional Memory, qui permet à plusieurs threads d'écrire de manière atomique sur une seule variable de type `TVar a` .

`TVar a` est le type principal associé à la monade `STM` et représente la variable transactionnelle. Ils sont utilisés de la même manière que `MVar` mais dans la monade `STM` grâce aux fonctions suivantes:

`atomically :: STM a -> IO a`

Effectuez une série d'actions STM de manière atomique.

`readTVar :: TVar a -> STM a`

Lisez la `TVar` la `TVar` , par exemple:

```
value <- readTVar t
```

`writeTVar :: TVar a -> a -> STM ()`

Ecrivez une valeur à la `TVar` donnée.

```
t <- newTVar Nothing
writeTVar t (Just "Hello")
```

Cet exemple est tiré du wiki Haskell:

```
import Control.Monad
import Control.Concurrent
import Control.Concurrent.STM

main = do
  -- Initialise a new TVar
  shared <- atomically $ newTVar 0
  -- Read the value
  before <- atomRead shared
  putStrLn $ "Before: " ++ show before
  forkIO $ 25 `timesDo` (dispVar shared >> milliSleep 20)
  forkIO $ 10 `timesDo` (appV ((+) 2) shared >> milliSleep 50)
  forkIO $ 20 `timesDo` (appV pred shared >> milliSleep 25)
  milliSleep 800
  after <- atomRead shared
  putStrLn $ "After: " ++ show after
  where timesDo = replicateM_
        milliSleep = threadDelay . (*) 1000

atomRead = atomically . readTVar
dispVar x = atomRead x >>= print
appV fn x = atomically $ readTVar x >>= writeTVar x . fn
```

Lire Concurrence en ligne: https://riptutorial.com/fr/haskell/topic/4426/concurrence

Chapitre 15: Conteneurs - Data.Map

Examples

La construction

Nous pouvons créer une carte à partir d'une liste de tuples comme ceci:

```
Map.fromList [("Alex", 31), ("Bob", 22)]
```

Une carte peut également être construite avec une seule valeur:

```
> Map.singleton "Alex" 31
fromList [("Alex",31)]
```

Il y a aussi la fonction `empty` .

```
empty :: Map k a
```

Data.Map prend également en charge les opérations classiques telles que l' `union` , la `difference` et l' `intersection` .

Vérification si vide

Nous utilisons la fonction `null` pour vérifier si une carte donnée est vide:

```
> Map.null $ Map.fromList [("Alex", 31), ("Bob", 22)]
False

> Map.null $ Map.empty
True
```

Trouver des valeurs

Il existe de nombreuses opérations d'interrogation sur les cartes.

`member :: Ord k => k -> Map ka -> Bool True` si la clé de type `k` est dans la `Map ka` :

```
> Map.member "Alex" $ Map.singleton "Alex" 31
True
> Map.member "Jenny" $ Map.empty
False
```

`notMember` est similaire:

```
> Map.notMember "Alex" $ Map.singleton "Alex" 31
False
```

```
> Map.notMember "Jenny" $ Map.empty
True
```

Vous pouvez également utiliser `findWithDefault :: Ord k => a -> k -> Map ka -> a` pour donner une valeur par défaut si la clé n'est pas présente:

```
Map.findWithDefault 'x' 1 (fromList [(5,'a'), (3,'b')]) == 'x'
Map.findWithDefault 'x' 5 (fromList [(5,'a'), (3,'b')]) == 'a'
```

Insertion d'éléments

L'insertion d' éléments est simple:

```
> let m = Map.singleton "Alex" 31
fromList [("Alex",31)]

> Map.insert "Bob" 99 m
fromList [("Alex",31),("Bob",99)]
```

Supprimer des éléments

```
> let m = Map.fromList [("Alex", 31), ("Bob", 99)]
fromList [("Alex",31),("Bob",99)]

> Map.delete "Bob" m
fromList [("Alex",31)]
```

Importation du module

Le module `Data.Map` dans le package de `containers` fournit une structure `Map` qui possède des implémentations strictes et différées.

Lors de l'utilisation de `Data.Map` , on l'importe habituellement pour éviter les conflits avec les fonctions déjà définies dans Prelude:

```
import qualified Data.Map as Map
```

Donc , nous avions alors précédez `Map` fonction des appels avec `Map.` , par exemple

```
Map.empty -- give me an empty Map
```

Instance monoïde

`Map kv` fournit une instance Monoid avec la sémantique suivante:

- `mempty` est la `Map` vide, c'est-à-dire identique à `Map.empty`
- `m1 <> m2` est la relation biaisée à gauche de `m1` et `m2` , c'est-à-dire que si une clé est présente à la fois dans `m1` et `m2` , la valeur de `m1` est choisie pour `m1 <> m2` . Cette opération est

également disponible en dehors de la `Monoid` par exemple comme `Map.union` .

Lire Conteneurs - Data.Map en ligne: https://riptutorial.com/fr/haskell/topic/4591/conteneurs---data-map

Chapitre 16: Création de types de données personnalisés

Examples

Créer un type de données simple

La méthode la plus simple pour créer un type de données personnalisé dans Haskell consiste à utiliser le mot-clé `data` :

```
data Foo = Bar | Biz
```

Le nom du type est spécifié entre `data` et `=` et est appelé **constructeur de type** . Après `=` nous spécifions tous les **constructeurs** de **valeurs** de notre type de données, délimité par le `|` signe. Il existe une règle dans Haskell selon laquelle tous les constructeurs de types et de valeurs doivent commencer par une majuscule. La déclaration ci-dessus peut se lire comme suit:

> Définissez un type appelé `Foo` , qui a deux valeurs possibles: `Bar` et `Biz` .

Création de variables de notre type personnalisé

```
let x = Bar
```

L'instruction ci-dessus crée une variable nommée `x` de type `Foo` . Vérifions cela en vérifiant son type.

```
:t x
```

estampes

```
x :: Foo
```

Création d'un type de données avec des paramètres de constructeur de valeur

Les constructeurs de valeurs sont des fonctions qui renvoient une valeur d'un type de données. De ce fait, comme toute autre fonction, ils peuvent prendre un ou plusieurs paramètres:

```
data Foo = Bar String Int | Biz String
```

Vérifions le type du constructeur de la valeur `Bar` .

```
:t Bar
```

estampes

```
Bar :: String -> Int -> Foo
```

ce qui prouve que `Bar` est bien une fonction.

Création de variables de notre type personnalisé

```
let x = Bar "Hello" 10
let y = Biz "Goodbye"
```

Création d'un type de données avec des paramètres de type

Les constructeurs de types peuvent prendre un ou plusieurs paramètres de type:

```
data Foo a b = Bar a b | Biz a b
```

Les paramètres de type dans Haskell doivent commencer par une lettre minuscule. Notre type de données personnalisé n'est pas encore un type réel. Afin de créer des valeurs de notre type, nous devons substituer tous les paramètres de type aux types réels. Comme `a` et `b` peuvent être de tout type, nos constructeurs de valeurs sont des fonctions polymorphes.

Création de variables de notre type personnalisé

```
let x = Bar "Hello" 10      -- x :: Foo [Char] Integer
let y = Biz "Goodbye" 6.0   -- y :: Fractional b => Foo [Char] b
let z = Biz True False      -- z :: Foo Bool Bool
```

Type de données personnalisé avec paramètres d'enregistrement

Supposons que nous voulions créer un type de données Personne qui a un prénom et un nom, un âge, un numéro de téléphone, une rue, un code postal et une ville.

Nous pourrions écrire

```
data Person = Person String String Int Int String String String
```

Si nous voulons maintenant obtenir le numéro de téléphone, nous devons créer une fonction

```
getPhone :: Person -> Int
getPhone (Person _ _ _ phone _ _ _) = phone
```

Eh bien, ce n'est pas amusant. Nous pouvons faire mieux en utilisant les paramètres:

```
data Person' = Person' { firstName     :: String
                       , lastName      :: String
```

```
                        , age           :: Int
                        , phone         :: Int
                        , street        :: String
                        , code          :: String
                        , town          :: String }
```

Maintenant, nous obtenons le `phone` fonction où

```
:t phone
phone :: Person' -> Int
```

Nous pouvons maintenant faire ce que nous voulons, par exemple:

```
printPhone :: Person' -> IO ()
printPhone = putStrLn . show . phone
```

Nous pouvons également lier le numéro de téléphone par correspondance de motif :

```
getPhone' :: Person' -> Int
getPhone' (Person {phone = p}) = p
```

Pour une utilisation facile des paramètres, voir RecordWildCards

Lire Création de types de données personnalisés en ligne:
https://riptutorial.com/fr/haskell/topic/4057/creation-de-types-de-donnees-personnalises

Chapitre 17: Data.Aeson - JSON dans Haskell

Examples

Encodage et décodage intelligents à l'aide de génériques

Le moyen le plus simple et le plus rapide d'encoder un type de données Haskell en JSON avec Aeson consiste à utiliser des génériques.

```
{-# LANGUAGE DeriveGeneric #-}

import GHC.Generics
import Data.Text
import Data.Aeson
import Data.ByteString.Lazy
```

Commençons par créer un type de données Personne:

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Int
                     } deriving (Show, Generic)
```

Pour utiliser la fonction d' `encode` et de `decode` du package `Data.Aeson` , nous devons `Data.Aeson` `Person` comme instance de `ToJSON` et `FromJSON` . Puisque nous dérivons `Generic` for `Person` , nous pouvons créer des instances vides pour ces classes. Les définitions par défaut des méthodes sont définies en fonction des méthodes fournies par la classe de type `Generic` .

```
instance ToJSON Person
instance FromJSON Person
```

Terminé! Afin d'améliorer la vitesse d'encodage, nous pouvons modifier légèrement l'instance `ToJSON` :

```
instance ToJSON Person where
    toEncoding = genericToEncoding defaultOptions
```

Maintenant, nous pouvons utiliser la fonction `encode` pour convertir `Person` en un bytestring (paresseux):

```
encodeNewPerson :: Text -> Text -> Int -> ByteString
encodeNewPerson first last age = encode $ Person first last age
```

Et pour décoder, nous pouvons simplement utiliser le `decode` :

```
> encodeNewPerson "Hans" "Wurst" 30
"{\"lastName\":\"Wurst\",\"age\":30,\"firstName\":\"Hans\"}"
```

```
> decode $ encodeNewPerson "Hans" "Wurst" 30
Just (Person {firstName = "Hans", lastName = "Wurst", age = 30})
```

Un moyen rapide de générer un Data.Aeson.Value

```
{-# LANGUAGE OverloadedStrings #-}
module Main where

import Data.Aeson

main :: IO ()
main = do
  let example = Data.Aeson.object [ "key" .= (5 :: Integer), "somethingElse" .= (2 :: Integer)
] :: Value
  print . encode $ example
```

Champs facultatifs

Parfois, nous voulons que certains champs de la chaîne JSON soient facultatifs. Par exemple,

```
data Person = Person { firstName :: Text
                     , lastName  :: Text
                     , age       :: Maybe Int
                     }
```

Cela peut être réalisé par

```
import Data.Aeson.TH

$(deriveJSON defaultOptions{omitNothingFields = True} ''Person)
```

Lire Data.Aeson - JSON dans Haskell en ligne: https://riptutorial.com/fr/haskell/topic/4525/data-aeson---json-dans-haskell

Chapitre 18: Data.Text

Remarques

`Text` est une alternative plus efficace au type de `String` standard d'Haskell. `String` est définie comme une liste liée de caractères dans le Prélude standard, conformément au rapport Haskell :

```
type String = [Char]
```

`Text` est représenté sous la forme d'un tableau condensé de caractères Unicode. Ceci est similaire à la façon dont la plupart des langages de haut niveau représentent des chaînes et offre une efficacité en termes de temps et d'espace bien supérieure à celle de la version de liste.

`Text` doit être préféré à `String` pour toute utilisation en production. Une exception notable dépend d'une bibliothèque qui possède une API `String` , mais même dans ce cas, il peut être avantageux d'utiliser `Text` interne et de le convertir en `String` juste avant d'interfacer avec la bibliothèque.

Tous les exemples de cette rubrique utilisent l'extension de langue `OverloadedStrings` .

Examples

Littéraux de texte

L'extension de langage `OverloadedStrings` permet d'utiliser des littéraux de chaîne normaux pour représenter des valeurs `Text` .

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "overloaded"
```

Supprimer les espaces

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "\n\r\t   leading and trailing whitespace   \t\r\n"
```

`strip` supprime les espaces au début et à la fin d'une valeur de `Text` .

```
ghci> T.strip myText
"leading and trailing whitespace"
```

`stripStart` supprime les espaces que depuis le début.

```
ghci> T.stripStart myText
"leading and trailing whitespace   \t\r\n"
```

`stripEnd` supprime uniquement les espaces à partir de la fin.

```
ghci> T.stripEnd myText
"\n\r\t   leading and trailing whitespace"
```

`filter` peut être utilisé pour supprimer des espaces ou d'autres caractères du milieu.

```
ghci> T.filter /=' ' "spaces in the middle of a text string"
"spacesinthemiddleofatextstring"
```

Fractionnement des valeurs de texte

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text
myText = "mississippi"
```

`splitOn` décompose un `Text` en une liste de `Texts` sur les occurrences d'une sous-chaîne.

```
ghci> T.splitOn "ss" myText
["mi","i","ippi"]
```

`splitOn` est l'inverse de l' `intercalate` .

```
ghci> intercalate "ss" (splitOn "ss" "mississippi")
"mississippi"
```

`split` divise une valeur `Text` en morceaux sur des caractères satisfaisant un prédicat booléen.

```
ghci> T.split (== 'i') myText
["m","ss","ss","pp",""]
```

Texte codant et décodé

Les fonctions d'encodage et de décodage pour divers codages Unicode sont disponibles dans le module `Data.Text.Encoding` .

```
ghci> import Data.Text.Encoding
ghci> decodeUtf8 (encodeUtf8 "my text")
"my text"
```

Notez que `decodeUtf8` lancera une exception sur une entrée non valide. Si vous voulez gérer vous-

même un UTF-8 non valide, utilisez `decodeUtf8With` .

```
ghci> decodeUtf8With (\errorDescription input -> Nothing) messyOutsideData
```

Vérifier si un texte est une sous-chaîne d'un autre texte

```
ghci> :set -XOverloadedStrings
ghci> import Data.Text as T
```

`isInfixOf :: Text -> Text -> Bool` vérifie si un `Text` est contenu n'importe où dans un autre `Text` .

```
ghci> "rum" `T.isInfixOf` "crumble"
True
```

`isPrefixOf :: Text -> Text -> Bool` vérifie si un `Text` apparaît au début d'un autre `Text` .

```
ghci> "crumb" `T.isPrefixOf` "crumble"
True
```

`isSuffixOf :: Text -> Text -> Bool` vérifie si un `Text` apparaît à la fin d'un autre `Text` .

```
ghci> "rumble" `T.isSuffixOf` "crumble"
True
```

Texte d'indexation

```
{-# LANGUAGE OverloadedStrings #-}

import qualified Data.Text as T

myText :: T.Text

myText = "mississippi"
```

Les caractères à des indices spécifiques peuvent être renvoyés par la fonction d' `index` .

```
ghci> T.index myText 2
's'
```

La fonction `findIndex` prend une fonction de type `(Char -> Bool)` et Text et renvoie l'index de la première occurrence d'une chaîne donnée ou Nothing si cela ne se produit pas.

```
ghci> T.findIndex ('s'==) myText
Just 2
ghci> T.findIndex ('c'==) myText
Nothing
```

La fonction `count` renvoie le nombre de fois qu'un `Text` requête apparaît dans un autre `Text` .

```
ghci> count ("miss"::T.Text) myText
1
```

Lire Data.Text en ligne: https://riptutorial.com/fr/haskell/topic/3406/data-text

Chapitre 19: Date et l'heure

Syntaxe

- addDays :: Integer -> Jour -> Jour
- diffDays :: Jour -> Jour -> Entier
- fromGregorian :: Integer -> Int -> Int -> Jour

```
 convert from proleptic Gregorian calendar. First argument is year, second month number
(1-12), third day (1-31). Invalid values will be clipped to the correct range, month
first, then day.
```

- getCurrentTime :: IO UTCTime

Remarques

Le module `Data.Time` de `time` Package fournit un support pour la récupération et la manipulation des valeurs de date et heure:

Examples

Trouver la date du jour

La date et l'heure actuelles peuvent être trouvées avec `getCurrentTime` :

```
import Data.Time

print =<< getCurrentTime
-- 2016-08-02 12:05:08.937169 UTC
```

Alternativement, juste la date est retournée par `fromGregorian` :

```
fromGregorian 1984 11 17  -- yields a Day
```

Ajout, soustraction et comparaison de jours

Étant donné un `Day` , nous pouvons effectuer des calculs et des comparaisons simples, tels que l'ajout:

```
import Data.Time

addDays 1 (fromGregorian 2000 1 1)
-- 2000-01-02
addDays 1 (fromGregorian 2000 12 31)
```

```
-- 2001-01-01
```

Soustraire:

```
addDays (-1) (fromGregorian 2000 1 1)
-- 1999-12-31

addDays (-1) (fromGregorian 0 1 1)
-- -0001-12-31
-- wat
```

et même trouver la différence:

```
diffDays (fromGregorian 2000 12 31) (fromGregorian 2000 1 1)
365
```

notez que l'ordre compte:

```
diffDays (fromGregorian 2000 1 1) (fromGregorian 2000 12 31)
-365
```

Lire Date et l'heure en ligne: https://riptutorial.com/fr/haskell/topic/4950/date-et-l-heure

Chapitre 20: Déclarations de fixité

Syntaxe

1. infixe [entier] ops
2. infixl [entier] ops
3. infixr [entier] ops

Paramètres

Composant de déclaration	Sens
`infixr`	l'opérateur est associé à droite
`infixl`	l'opérateur est associé à gauche
`infix`	l'opérateur est non associatif
chiffre optionnel	priorité de l'opérateur (plage 0 ... 9, valeur par défaut 9)
`op1, ... , opn`	les opérateurs

Remarques

Pour analyser des expressions impliquant des opérateurs et des fonctions, Haskell utilise des déclarations de fixité pour déterminer où vont les parenthèses. Dans l'ordre, il

1. encapsule des applications de fonction dans parens
2. utilise la précaution de liaison pour envelopper des groupes de termes tous séparés par des opérateurs de même priorité
3. utilise l'associativité de ces opérateurs pour déterminer comment ajouter des parens à ces groupes

Notez que nous supposons ici que les opérateurs d'un groupe donné de l'étape 2 doivent tous avoir la même associativité. En fait, Haskell rejettera tout programme où cette condition n'est pas remplie.

À titre d'exemple de l'algorithme ci-dessus, nous pouvons suivre le processus d'ajout de parenthèses à `1 + negate 5 * 2 - 3 * 4 ^ 2 ^ 1`.

```
infixl 6 +
infixl 6 -
infixl 7 *
infixr 8 ^
```

1. `1 + (negate 5) * 2 - 3 * 4 ^ 2 ^ 1`

2. `1 + ((negate 5) * 2) - (3 * (4 ^ 2 ^ 1))`
3. `(1 + ((negate 5) * 2)) - (3 * (4 ^ (2 ^ 1)))`

Plus de détails dans la section 4.4.2 du rapport Haskell 98 .

Examples

Associativité

`infixl` vs `infixr` vs `infix` décrivent de quel côté les parens seront groupés. Par exemple, considérons les déclarations de fixité suivantes (en base)

```
infixl 6 -
infixr 5 :
infix  4 ==
```

L' `infixl` nous dit que `-` a quitté l'associativité, ce qui signifie que `1 - 2 - 3 - 4` est analysé comme

```
((1 - 2) - 3) - 4
```

L' `infixr` nous dit que `:` a l'associativité correcte, ce qui signifie que `1 : 2 : 3 : []` est analysé comme

```
1 : (2 : (3 : []))
```

L' `infix` nous dit que `==` ne peut être utilisé sans nous, y compris les parenthèses, ce qui signifie que `True == False == True` est une erreur de syntaxe. En revanche, `True == (False == True)` ou `(True == False) == True` va.

Les opérateurs sans déclaration de fixité explicite sont `infixl 9` .

Priorité obligatoire

Le nombre qui suit les informations d'associativité décrit dans quel ordre les opérateurs sont appliqués. Il doit toujours être compris entre `0` et `9` inclus. C'est ce que l'on appelle communément le niveau de liaison de l'opérateur. Par exemple, considérons les déclarations de fixité suivantes (en base)

```
infixl 6 +
infixl 7 *
```

Puisque `*` a une préséance de liaison plus élevée que `+` nous lisons `1 * 2 + 3` comme

```
(1 * 2) + 3
```

En résumé, plus le nombre est élevé, plus l'opérateur se rapproche des parens.

Remarques

- L'application de fonction se lie *toujours* plus haut que les opérateurs, donc `fx `op` gy` doit être interprété comme `(fx)` op `(gy)` peu importe ce que l'opérateur` `op` et sa déclaration de fixité sont.
- Si la priorité de liaison est omise dans une déclaration de fixité (par exemple, nous avons `infixl *!?`), la valeur par défaut est `9` .

Exemples de déclarations

- `infixr 5 ++`
- `infixl 4 <*>, <*, *>, <**>`
- `infixl 8 `shift`, `rotate`, `shiftL`, `shiftR`, `rotateL`, `rotateR``
- `infix 4 ==, /=, <, <=, >=, >`
- `infix ??`

Lire Déclarations de fixité en ligne: https://riptutorial.com/fr/haskell/topic/4691/declarations-de-fixite

Chapitre 21: Des listes

Syntaxe

1. constructeur de liste vide

   ```
   [] :: [a]
   ```

2. constructeur de liste non vide

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

3. head - renvoie la première valeur d'une liste

   ```
   head :: [a] -> a
   ```

4. last - renvoie la dernière valeur d'une liste

   ```
   last :: [a] -> a
   ```

5. tail - renvoie une liste sans le premier élément

   ```
   tail :: [a] -> [a]
   ```

6. init - renvoie une liste sans le dernier élément

   ```
   init :: [a] -> [a]
   ```

7. xs !! i - renvoie l'élément à un index i dans la liste xs

   ```
   (!!) :: Int -> [a] -> a
   ```

8. take n xs - retourne une nouvelle liste contenant n premiers éléments de la liste xs

   ```
   take :: Int -> [a] -> [a]
   ```

9. map :: (a -> b) -> [a] -> [b]
10. filter :: (a -> Bool) -> [a] -> [a]
11. (++) :: [a] -> [a]
12. concat :: [[a]] -> [a]

Remarques

1. Le type `[a]` est équivalent à `[] a` .
2. `[]` construit la liste vide.
3. `[]` dans une définition de fonction LHS, par exemple `f [] = ...` , est le modèle de liste vide.
4. `x:xs` construit une liste où un élément `x` est ajouté à la liste `xs`
5. `f (x:xs) = ...` est une correspondance de modèle pour une liste non vide où `x` est la tête et `xs`

la queue.

6. `f (a:b:cs) = ...` et `f (a:(b:cs)) = ...` sont les mêmes. Ils correspondent à une liste d'au moins deux éléments dont le premier élément est `a` , le deuxième élément est `b` et le reste de la liste est `cs` .
7. `f ((a:as):bs) = ...` n'est PAS la même chose que `f (a:(as:bs)) = ...` Le premier correspond à une liste non vide de listes, où `a` est la tête de la tête, de même `as` la queue de la tête, et `bs` la queue.
8. `f (x:[]) = ...` et `f [x] = ...` sont les mêmes. Ils correspondent à un modèle pour une liste d'un seul élément.
9. `f (a:b:[]) = ...` et `f [a,b] = ...` sont les mêmes. Ils correspondent à une liste de deux éléments exactement.
10. `f [a:b] = ...` est une correspondance de modèle pour une liste d'exactement un élément où l'élément est aussi une liste. `a` est la tête de l'élément et `b` est la queue de l'élément.
11. `[a,b,c]` est le même que `(a:b:c:[])` . La notation de liste standard est simplement du sucre syntaxique pour les constructeurs `(:)` et `[]` .
12. Vous pouvez utiliser `all@(x:y:ys)` afin de se référer à la liste tout comme `all` (ou tout autre nom que vous choisissez) au lieu de répéter `(x:y:ys)` à nouveau.

Examples

Liste des littéraux

```
emptyList     = []

singletonList = [0]               -- = 0 : []

listOfNums    = [1, 2, 3]         -- = 1 : 2 : [3]

listOfStrings = ["A", "B", "C"]
```

Concaténation de liste

```
listA      = [1, 2, 3]

listB      = [4, 5, 6]

listAThenB = listA ++ listB       -- [1, 2, 3, 4, 5, 6]

(++) xs     [] = xs
(++) []     ys = ys
(++) (x:xs) ys = x : (xs ++ ys)
```

Liste des bases

Le constructeur de type pour les listes dans le Prelude Haskell est `[]` . La déclaration de type pour une liste contenant des valeurs de type `Int` est écrite comme suit:

```
xs :: [Int]    -- or equivalently, but less conveniently,
xs :: [] Int
```

Les listes dans Haskell sont *des séquences homogènes* , c'est-à-dire que tous les éléments doivent être du même type. Contrairement aux tuples, le type de liste n'est pas affecté par la longueur:

```
[1,2,3]   :: [Int]
[1,2,3,4] :: [Int]
```

Les listes sont construites en utilisant deux constructeurs :

- `[]` construit une liste vide.
- `(:)` , prononcé "contre", ajoute des éléments à une liste. Conserver `x` (une valeur de type `a`) sur `xs` (une liste de valeurs du même type `a`) crée une nouvelle liste dont la *tête* (le premier élément) est `x` et *tail* (le reste des éléments) est `xs` .

Nous pouvons définir des listes simples comme suit:

```
ys :: [a]
ys = []

xs :: [Int]
xs = 12 : (99 : (37 : []))
-- or  = 12 : 99 : 37 : []      -- ((:) is right-associative)
-- or  = [12, 99, 37]           -- (syntactic sugar for lists)
```

Notez que `(++)` , qui peut être utilisé pour construire des listes, est défini récursivement en termes de `(:)` et `[]` .

Listes de traitement

Pour traiter des listes, nous pouvons simplement créer des correspondances sur les constructeurs du type liste:

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

Nous pouvons faire correspondre plus de valeurs en spécifiant un modèle plus élaboré:

```
sumTwoPer :: [Int] -> Int
sumTwoPer [] = 0
sumTwoPer (x1:x2:xs) = x1 + x2 + sumTwoPer xs
sumTwoPer (x:xs) = x + sumTwoPer xs
```

Notez que dans l'exemple ci-dessus, nous avons dû fournir une correspondance de modèle plus exhaustive pour gérer les cas où une liste de longueurs impaires est donnée en argument.

Le Prelude Haskell définit de nombreux éléments intégrés pour gérer les listes, comme `map` , `filter` , etc. Dans la mesure du possible, vous devriez les utiliser au lieu d'écrire vos propres fonctions récursives.

Accéder aux éléments dans les listes

Accéder à la *n* - ième élément d'une liste (base zéro):

```
list = [1 .. 10]

firstElement = list !! 0           -- 1
```

Notez que `!!` est une fonction partielle, de sorte que certaines entrées produisent des erreurs:

```
list !! (-1)     -- *** Exception: Prelude.!!: negative index

list !! 1000     -- *** Exception: Prelude.!!: index too large
```

Il y a aussi `Data.List.genericIndex` , une version surchargée de `!!` , qui accepte toute valeur `Integral` comme index.

```
import Data.List (genericIndex)

list `genericIndex` 4              -- 5
```

Lorsqu'elles sont implémentées sous forme de listes à liens uniques, ces opérations prennent du temps *O (n)* . Si vous accédez fréquemment à des éléments par index, il est probablement préférable d'utiliser `Data.Vector` (à partir du package vectoriel) ou d'autres structures de données.

Gammes

Créer une liste de 1 à 10 est simple en utilisant la notation par intervalle:

```
[1..10]    -- [1,2,3,4,5,6,7,8,9,10]
```

Pour spécifier une étape, ajoutez une virgule et l'élément suivant après l'élément start:

```
[1,3..10]  -- [1,3,5,7,9]
```

Notez que Haskell prend toujours le pas comme différence arithmétique entre les termes et que vous ne pouvez pas spécifier plus que les deux premiers éléments et la limite supérieure:

```
[1,3,5..10] -- error
[1,3,9..20] -- error
```

Pour générer une plage dans l'ordre décroissant, spécifiez toujours l'étape négative:

```
[5..1]     -- []

[5,4..1]   -- [5,4,3,2,1]
```

Comme Haskell est non strict, les éléments de la liste ne sont évalués que s'ils sont nécessaires,

ce qui nous permet d'utiliser des listes infinies. `[1..]` est une liste infinie à partir de 1. Cette liste peut être liée à une variable ou passée en argument de fonction:

```
take 5 [1..]   -- returns [1,2,3,4,5] even though [1..] is infinite
```

Soyez prudent lorsque vous utilisez des plages avec des valeurs à virgule flottante, car il accepte les retombées jusqu'à la moitié du delta, pour éviter les problèmes d'arrondi:

```
[1.0,1.5..2.4]    -- [1.0,1.5,2.0,2.5] , though 2.5 > 2.4

[1.0,1.1..1.2]    -- [1.0,1.1,1.2000000000000002] , though 1.2000000000000002 > 1.2
```

Les plages ne fonctionnent pas uniquement avec des nombres mais avec n'importe quel type qui implémente la classe `Enum` . Étant donné certaines variables énumérables `a` , `b` , `c` , la syntaxe de la plage est équivalente à l'appel de ces méthodes `Enum` :

```
[a..]    == enumFrom a
[a..c]   == enumFromTo a c
[a,b..]  == enumFromThen a b
[a,b..c] == enumFromThenTo a b c
```

Par exemple, avec `Bool` c'est

```
 [False ..]      -- [False,True]
```

Notez l'espace après `False` , pour éviter que cela soit analysé comme une qualification de nom du module (c. -à- `False..` serait analysé comme `.` D'un module `False`).

Fonctions de base sur les listes

```
head [1..10]       --    1

last [1..20]       --    20

tail [1..5]        --    [2, 3, 4, 5]

init [1..5]        --    [1, 2, 3, 4]

length [1 .. 10]   --    10

reverse [1 .. 10]  --    [10, 9 .. 1]

take 5 [1, 2 .. ]  --    [1, 2, 3, 4, 5]

drop 5 [1 .. 10]   --    [6, 7, 8, 9, 10]

concat [[1,2], [], [4]]   --    [1,2,4]
```

plier

C'est comme ça que le pli gauche est implémenté. Notez que l'ordre des arguments dans la

fonction step est `foldr` par rapport à `foldr` (le pli droit):

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

Le pli gauche, `foldl` , s'associe à gauche. C'est:

```
foldl (+) 0 [1, 2, 3]     -- is equivalent to ((0 + 1) + 2) + 3
```

La raison en est que `foldl` est évalué comme ça (regardez le pas inductif de `foldl`):

```
foldl (+) 0 [1, 2, 3]                       --  foldl (+)    0   [ 1,   2,   3 ]
foldl (+) ((+) 0 1) [2, 3]                  --  foldl (+)   (0 + 1)   [ 2,   3 ]
foldl (+) ((+) ((+) 0 1) 2) [3]             --  foldl (+)  ((0 + 1) + 2)   [ 3 ]
foldl (+) ((+) ((+) ((+) 0 1) 2) 3) []      --  foldl (+) (((0 + 1) + 2) + 3) []
((+) ((+) ((+) 0 1) 2) 3)                   --            (((0 + 1) + 2) + 3)
```

La dernière ligne est équivalente à `((0 + 1) + 2) + 3` . C'est parce que `(fab)` est identique à `(a `f` b)` en général, et donc `((+) 0 1)` est le même que `(0 + 1)` en particulier.

foldr

Voici comment le bon pli est mis en œuvre:

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

Le bon pli, `foldr` , s'associe à droite. C'est:

```
foldr (+) 0 [1, 2, 3]      -- is equivalent to 1 + (2 + (3 + 0))
```

La raison en est que `foldr` est évalué comme ça (regardez le pas inductif de `foldr`):

```
foldr (+) 0 [1, 2, 3]                       --          foldr (+) 0  [1,2,3]
(+) 1 (foldr (+) 0 [2, 3])                  -- 1 +        foldr (+) 0  [2,3]
(+) 1 ((+) 2 (foldr (+) 0 [3]))             -- 1 + (2 +     foldr (+) 0  [3])
(+) 1 ((+) 2 ((+) 3 (foldr (+) 0 [])))      -- 1 + (2 + (3 +  foldr (+) 0 []))
(+) 1 ((+) 2 ((+) 3 0))                     -- 1 + (2 + (3 +            0   ))
```

La dernière ligne est équivalente à `1 + (2 + (3 + 0))` , car `((+) 3 0)` est identique à `(3 + 0)` .

Transformer avec `map`

Nous souhaitons souvent convertir ou transformer le contenu d'une collection (une liste ou quelque chose pouvant être traversé). Dans Haskell, nous utilisons la `map` :

```
-- Simple add 1
map (+ 1) [1,2,3]
```

```
[2,3,4]

map odd [1,2,3]
[True,False,True]

data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

-- Extract just the age from a list of people
map (\(Person n g a) -> a) [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
[31,29]
```

Filtrage avec `filter`

Donné une liste:

```
li = [1,2,3,4,5]
```

nous pouvons filtrer une liste avec un prédicat utilisant `filter :: (a -> Bool) -> [a] -> [a]` :

```
filter (== 1) li       -- [1]

filter (even) li       -- [2,4]

filter (odd) li        -- [1,3,5]

-- Something slightly more complicated
comfy i = notTooLarge && isEven
  where
     notTooLarge = (i + 1) < 5
     isEven = even i

filter comfy li        -- [2]
```

Bien sûr, il n'y a pas que des chiffres:

```
data Gender = Male | Female deriving Show
data Person = Person String Gender Int deriving Show

onlyLadies :: [Person] -> Person
onlyLadies x = filter isFemale x
  where
    isFemale (Person _ Female _) = True
    isFemale _ = False

onlyLadies [(Person "Alex" Male 31),(Person "Ellie" Female 29)]
-- [Person "Ellie" Female 29]
```

Listes de décompression et de décompression

zip prend deux listes et renvoie une liste de paires correspondantes:

```
zip []     _      = []
zip _      []     = []
```

```
zip (a:as) (b:bs) = (a,b) : zip as bs

> zip [1,3,5] [2,4,6]
> [(1,2),(3,4),(5,6)]
```

Compacter deux listes avec une fonction:

```
zipWith f  []     _      = []
zipWith f  _      []     = []
zipWith f  (a:as) (b:bs) = f a b : zipWith f as bs

> zipWith (+) [1,3,5] [2,4,6]
> [3,7,11]
```

Décompression d'une liste:

```
unzip = foldr (\(a,b) ~(as,bs) -> (a:as,b:bs)) ([],[])

> unzip [(1,2),(3,4),(5,6)]
> ([1,3,5],[2,4,6])
```

Lire Des listes en ligne: https://riptutorial.com/fr/haskell/topic/2281/des-listes

Chapitre 22: Des procurations

Examples

Utiliser un proxy

Le type `Proxy :: k -> *` , trouvé dans `Data.Proxy` , est utilisé lorsque vous avez besoin de donner des informations de type au compilateur - par exemple, pour choisir une instance de classe de type - qui est néanmoins sans importance à l'exécution.

```
{-# LANGUAGE PolyKinds #-}

data Proxy a = Proxy
```

Les fonctions qui utilisent un `Proxy` utilisent généralement `ScopedTypeVariables` pour choisir une instance de classe de type basée sur `a` type.

Par exemple, l'exemple classique d'une fonction ambiguë,

```
showread :: String -> String
showread = show . read
```

ce qui se traduit par une erreur de type car le constructeur ne sait pas quelle instance de `Show` ou de `Read` à utiliser, peut être résolue avec `Proxy` :

```
{-# LANGUAGE ScopedTypeVariables #-}

import Data.Proxy

showread :: forall a. (Show a, Read a) => Proxy a -> String -> String
showread _ = (show :: a -> String) . read
```

Lorsque vous appelez une fonction avec `Proxy` , vous devez utiliser une annotation de type pour déclarer celle `a` vous vouliez dire.

```
ghci> showread (Proxy :: Proxy Int) "3"
"3"
ghci> showread (Proxy :: Proxy Bool) "'m'"  -- attempt to parse a char literal as a Bool
"*** Exception: Prelude.read: no parse
```

L'idiome "proxy polymorphe"

Étant donné que le `Proxy` ne contient aucune information d'exécution, il n'est jamais nécessaire de faire correspondre le modèle au constructeur du `Proxy` . Un idiome commun consiste donc à faire un résumé sur le type de données `Proxy` utilisant une variable de type.

```
showread :: forall proxy a. (Show a, Read a) => proxy a -> String -> String
```

```
showread _ = (show :: a -> String) . read
```

Maintenant, si vous avez une `fa` dans la portée de certains `f` , vous n'avez pas besoin d'écrire `Proxy :: Proxy a` lorsque vous appelez `f` .

```
ghci> let chars = "foo"  -- chars :: [Char]
ghci> showread chars "'a'"
"'a'"
```

Le proxy est comme ()

Comme `Proxy` ne contient aucune information d'exécution, vous pouvez toujours écrire une transformation naturelle `fa -> Proxy a` pour tout `f` .

```
proxy :: f a -> Proxy a
proxy _ = Proxy
```

C'est comme si une valeur donnée pouvait toujours être effacée sur `()` :

```
unit :: a -> ()
unit _ = ()
```

Techniquement, `Proxy` est l'objet terminal dans la catégorie des foncteurs, tout comme `()` est l'objet terminal dans la catégorie des valeurs.

Lire Des procurations en ligne: https://riptutorial.com/fr/haskell/topic/8025/des-procurations

Chapitre 23: Développement web

Examples

Serviteur

Servant est une bibliothèque pour déclarer des API au niveau du type, puis:

- écrire des serveurs (cette partie de serviteur peut être considérée comme un cadre web),
- obtenir des fonctions client (en haskell),
- générer des fonctions client pour d'autres langages de programmation,
- générer de la documentation pour vos applications Web
- et plus...

Servant possède une API concise mais puissante. Une API simple peut être écrite en très peu de lignes de code:

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE TypeOperators #-}

import Data.Text
import Data.Aeson.Types
import GHC.Generics
import Servant.API

data SortBy = Age | Name

data User = User {
  name :: String,
  age :: Int
} deriving (Eq, Show, Generic)

instance ToJSON User  -- automatically convert User to JSON
```

Maintenant, nous pouvons déclarer notre API:

```
type UserAPI = "users" :> QueryParam "sortby" SortBy :> Get '[JSON] [User]
```

qui indique que nous souhaitons exposer `/users` aux requêtes `GET` avec un `sortby` de requête `sortby` de type `SortBy` et renvoyer JSON de type `User` dans la réponse.

Maintenant, nous pouvons définir notre gestionnaire:

```
-- This is where we'd return our user data, or e.g. do a database lookup
server :: Server UserAPI
server = return [User "Alex" 31]

userAPI :: Proxy UserAPI
userAPI = Proxy
```

```
app1 :: Application
app1 = serve userAPI server
```

Et la méthode principale qui écoute sur le port `8081` et sert notre API utilisateur:

```
main :: IO ()
main = run 8081 app1
```

Remarque: Stack dispose d'un modèle pour générer des API de base dans Servant, ce qui est utile pour démarrer rapidement.

Yessod

Le projet Yesod peut être créé avec la `stack new` utilisant les modèles suivants:

- `yesod-minimal` . Le plus simple échafaudage de Yesod possible.
- `yesod-mongo` . Utilise MongoDB comme moteur de base de données.
- `yesod-mysql` . Utilise MySQL comme moteur de base de données.
- `yesod-postgres` . Utilise PostgreSQL comme moteur de base de données.
- `yesod-postgres-fay` . Utilise PostgreSQL comme moteur de base de données. Utilise le langage Fay pour le front-end.
- `yesod-simple` . Modèle recommandé à utiliser si vous n'avez pas besoin d'une base de données.
- `yesod-sqlite` . Utilise SQlite comme moteur de base de données.

`yesod-bin` paquet `yesod-bin` fournit `yesod` exécutable `yesod` , qui peut être utilisé pour exécuter le serveur de développement. Notez que vous pouvez également exécuter votre application directement, donc l'outil `yesod` est facultatif.

`Application.hs` contient du code qui distribue les requêtes entre les gestionnaires. Il configure également la base de données et les paramètres de journalisation, si vous les avez utilisés.

`Foundation.hs` définit le type d' `App` , qui peut être considéré comme un environnement pour tous les gestionnaires. En étant dans `HandlerT` , vous pouvez obtenir cette valeur en utilisant la fonction `getYesod` .

`Import.hs` est un module qui ne fait que réexporter des éléments couramment utilisés.

`Model.hs` contient Template Haskell qui génère des codes et des types de données utilisés pour l'interaction de la base de données. Présent uniquement si vous utilisez DB.

`config/models` est l'endroit où vous définissez votre schéma de base de données. Utilisé par `Model.hs` .

`config/routes` définit les URI de l'application Web. Pour chaque méthode HTTP de la route, vous devez créer un gestionnaire nommé `{method}{RouteR}` .

`static/` directory contient les ressources statiques du site. Ceux-ci sont compilés en binaire par le

module `Settings/StaticFiles.hs` .

`templates/` directory contient les modèles Shakespeare utilisés lors du traitement des requêtes.

Enfin, `Handler/` directory contient des modules qui définissent des gestionnaires pour les routes.

Chaque gestionnaire est une action `HandlerT` basée sur IO. Vous pouvez inspecter les paramètres de requête, son corps et d'autres informations, effectuer des requêtes sur la base de données avec `runDB` , exécuter des E / S arbitraires et renvoyer différents types de contenu à l'utilisateur. Pour servir HTML, la fonction `defaultLayout` est utilisée pour permettre une composition soignée des modèles shakespeariens.

Lire Développement web en ligne: https://riptutorial.com/fr/haskell/topic/4721/developpement-web

Chapitre 24: Empiler

Examples

Installation de la pile

Mac OS X

Utiliser Homebrew :

```
brew install haskell-stack
```

Créer un projet simple

Pour créer un projet appelé **helloworld** run:

```
stack new helloworld simple
```

Cela créera un répertoire appelé `helloworld` avec les fichiers nécessaires à un projet Stack.

Structure

Structure de fichier

Un projet simple contient les fichiers suivants:

```
➜ helloworld ls
LICENSE        Setup.hs        helloworld.cabal src            stack.yaml
```

Dans le dossier `src` il y a un fichier nommé `Main.hs` C'est le "point de départ" du projet `helloworld` . Par défaut, `Main.hs` contient un simple "Hello, World!" programme.

Main.hs

```
module Main where

main :: IO ()
main = do
  putStrLn "hello world"
```

Lancer le programme

Assurez-vous d'être dans le répertoire `helloworld` et exécutez:

```
stack build # Compile the program
stack exec helloworld # Run the program
# prints "hello world"
```

Stackage Packages et modification de la version LTS (resolver)

Stackage est un référentiel pour les packages Haskell. Nous pouvons ajouter ces paquets à un projet de pile.

Ajout de lentille à un projet.

Dans un projet de pile, il existe un fichier appelé `stack.yaml` . Dans `stack.yaml` il y a un segment qui ressemble à:

```
resolver: lts-6.8
```

Stackage conserve une liste de paquets pour chaque révision de `lts` . Dans notre cas, nous voulons la liste des paquets pour `lts-6.8` Pour trouver ces paquets, visitez:

```
https://www.stackage.org/lts-6.8 # if a different version is used, change 6.8 to the correct
resolver number.
```

En parcourant les paquets, il y a un Lens-4.13 .

Nous pouvons maintenant ajouter le package de langue en modifiant la section de `helloworld.cabal` :

```
  build-depends: base >= 4.7 && < 5
```

à:

```
  build-depends: base >= 4.7 && 5,
                 lens == 4.13
```

Évidemment, si nous voulons changer un LTS plus récent (après sa sortie), nous changeons simplement le numéro du résolveur, par exemple .:

```
resolver: lts-6.9
```

Avec la `stack build` suivante, Stack utilisera la version LTS 6.9 et téléchargera donc de nouvelles dépendances.

Construire et exécuter un projet de pile

Dans cet exemple, le nom de notre projet est "helloworld" qui a été créé avec `stack new helloworld simple`

Nous devons d'abord construire le projet avec la `stack build` et ensuite nous pouvons l'exécuter avec

```
stack exec helloworld-exe
```

Installer pile

En exécutant la commande

```
stack install
```

Stack copiera un fichier exécutable dans le dossier

```
/Users/<yourusername>/.local/bin/
```

Profilage avec pile

Configurez le profilage pour un projet via une `stack` . Construisez d'abord le projet avec l'indicateur `--profile` :

```
stack build --profile
```

Les drapeaux GHC ne sont pas requis dans le fichier cabal pour que cela fonctionne (comme `-prof`). `stack` activera automatiquement le profilage pour la bibliothèque et les exécutables du projet. La prochaine fois qu'un exécutable s'exécute dans le projet, les indicateurs habituels `+RTS` peuvent être utilisés:

```
stack exec -- my-bin +RTS -p
```

Affichage des dépendances

Pour savoir quels paquets dépendent directement de votre projet, vous pouvez simplement utiliser cette commande:

```
stack list-dependencies
```

De cette façon, vous pouvez savoir quelle version de vos dépendances a été réduite par pile.

Les projets Haskell se retrouvent souvent dans de nombreuses bibliothèques indirectement, et parfois ces dépendances externes causent des problèmes que vous devez suivre. Si vous vous trouvez avec une dépendance externe malveillante que vous souhaitez identifier, vous pouvez parcourir l'ensemble du graphique de dépendance et identifier les dépendances qui génèrent finalement le paquet indésirable:

```
stack dot --external | grep template-haskell
```

`stack dot` imprime un graphique de dépendance sous forme de texte pouvant être recherché. Il peut également être consulté:

```
stack dot --external | dot -Tpng -o my-project.png
```

Vous pouvez également définir la profondeur du graphique de dépendance si vous le souhaitez:

```
stack dot --external --depth 3 | dot -Tpng -o my-project.png
```

Lire Empiler en ligne: https://riptutorial.com/fr/haskell/topic/2970/empiler

Chapitre 25: Enregistrement

Introduction

La journalisation dans Haskell se fait généralement via des fonctions dans la monade `IO` , et se limite donc aux fonctions non pures ou aux "actions IO".

Il existe plusieurs manières de consigner des informations dans un programme Haskell: de `putStrLn` (ou `print`) à des bibliothèques telles que hslogger ou `Debug.Trace` .

Examples

Se connecter avec hslogger

Le module `hslogger` fournit une API similaire à la structure de `logging` de Python et prend en charge les enregistreurs, les niveaux et la redirection nommés de manière hiérarchique pour les descripteurs situés en dehors de `stdout` et de `stderr` .

Par défaut, tous les messages de niveau `WARNING` et ci-dessus sont envoyés à stderr et tous les autres niveaux de journalisation sont ignorés.

```
import           System.Log.Logger (Priority (DEBUG), debugM, infoM, setLevel,
                                     updateGlobalLogger, warningM)

main = do
  debugM "MyProgram.main" "This won't be seen"
  infoM "MyProgram.main" "This won't be seen either"
  warningM "MyProgram.main" "This will be seen"
```

Nous pouvons définir le niveau d'un enregistreur par son nom en utilisant `updateGlobalLogger` :

```
  updateGlobalLogger "MyProgram.main" (setLevel DEBUG)

  debugM "MyProgram.main" "This will now be seen"
```

Chaque enregistreur a un nom et ils sont organisés hiérarchiquement. `MyProgram` est donc un parent de `MyParent.Module` .

Lire Enregistrement en ligne: https://riptutorial.com/fr/haskell/topic/9628/enregistrement

Chapitre 26: Etat Monad

Introduction

Les monades d'état sont une sorte de monade qui porte un état qui peut changer pendant chaque calcul effectué dans la monade. Les implémentations sont généralement de la forme `State sa` qui représente un calcul qui porte et modifie potentiellement un état de type `s` et produit un résultat de type `a` , mais le terme "état monade" peut généralement se référer à toute monade qui porte un état. Le `mtl` et `transformers` fournit des implémentations générales de monades d'état.

Remarques

Les nouveaux arrivants à Haskell ont souvent peur de la monade de l' `State` et la traitent comme un tabou. Comme le prétendu avantage de la programmation fonctionnelle est l'évitement de l'état, ne perdez-vous pas cela lorsque vous utilisez `State` ? Une vue plus nuancée est que:

- L'état peut être utile en *petites doses contrôlées* ;
- Le type d' `State` permet de contrôler la dose très précisément.

Les raisons étant que si vous avez l' `action :: State sa` , cela vous dit que:

- `action` est spéciale car elle dépend d'un état;
- L'état a le type `s` , donc l' `action` ne peut pas être influencée par une ancienne valeur de votre programme - seulement une valeur `s` ou une valeur accessible depuis certains `s` ;
- Le `runState :: State sa -> s -> (a, s)` place une "barrière" autour de l'action avec état, de sorte que son efficacité *ne peut pas* être observée en dehors de cette barrière.

Il s'agit donc d'un bon ensemble de critères pour savoir s'il faut utiliser l' `State` dans un scénario particulier. Vous voulez voir que votre code *minimise la portée de l'état* , à la fois en choisissant un type étroit pour `s` et en plaçant `runState` plus près possible du "bas" (afin que vos actions puissent être aussi peu influencées que possible). possible.

Examples

Numérotation des nœuds d'un arbre avec un compteur

Nous avons un type de données arborescent comme ceci:

```
data Tree a = Tree a [Tree a] deriving Show
```

Et nous souhaitons écrire une fonction qui attribue un numéro à chaque nœud de l'arbre, à partir d'un compteur incrémenté:

```
tag :: Tree a -> Tree (a, Int)
```

Le long chemin

D' abord , nous le ferons long chemin, car il illustre l' `State` tout à fait bien la mécanique à faible niveau de monades.

```
import Control.Monad.State

-- Function that numbers the nodes of a `Tree`.
tag :: Tree a -> Tree (a, Int)
tag tree =
    -- tagStep is where the action happens.  This just gets the ball
    -- rolling, with `0` as the initial counter value.
    evalState (tagStep tree) 0

-- This is one monadic "step" of the calculation.  It assumes that
-- it has access to the current counter value implicitly.
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep (Tree a subtrees) = do
    -- The `get :: State s s` action accesses the implicit state
    -- parameter of the State monad.  Here we bind that value to
    -- the variable `counter`.
    counter <- get

    -- The `put :: s -> State s ()` sets the implicit state parameter
    -- of the `State` monad.  The next `get` that we execute will see
    -- the value of `counter + 1` (assuming no other puts in between).
    put (counter + 1)

    -- Recurse into the subtrees.  `mapM` is a utility function
    -- for executing a monadic actions (like `tagStep`) on a list of
    -- elements, and producing the list of results.  Each execution of
    -- `tagStep` will be executed with the counter value that resulted
    -- from the previous list element's execution.
    subtrees' <- mapM tagStep subtrees

    return $ Tree (a, counter) subtrees'
```

Refactoring

Diviser le compteur en une action postIncrement

Le bit où nous `get` le compteur actuel puis `put` compteur + 1 peut être divisé en une action `postIncrement` , similaire à ce que de nombreux langages de style C fournissent:

```
postIncrement :: Enum s => State s s
postIncrement = do
    result <- get
    modify succ
    return result
```

Diviser l'arbre en une fonction d'ordre supérieur

La logique de parcours d'arbre peut être divisée en sa propre fonction, comme ceci:

```
mapTreeM :: Monad m => (a -> m b) -> Tree a -> m (Tree b)
mapTreeM action (Tree a subtrees) = do
    a' <- action a
    subtrees' <- mapM (mapTreeM action) subtrees
    return $ Tree a' subtrees'
```

Avec ceci et la fonction `postIncrement` , nous pouvons réécrire `tagStep` :

```
tagStep :: Tree a -> State Int (Tree (a, Int))
tagStep = mapTreeM step
    where step :: a -> State Int (a, Int)
          step a = do
              counter <- postIncrement
              return (a, counter)
```

Utilisez la classe `Traversable`

La solution `mapTreeM` ci-dessus peut être facilement réécrite dans une instance de la classe `Traversable` :

```
instance Traversable Tree where
    traverse action (Tree a subtrees) =
        Tree <$> action a <*> traverse action subtrees
```

Notez que cela nous a obligé à utiliser `Applicative` (l'opérateur `<*>`) au lieu de `Monad` .

Avec ça, maintenant on peut écrire `tag` comme un pro:

```
tag :: Traversable t => t a -> t (a, Int)
tag init t = evalState (traverse step t) 0
    where step a = do tag <- postIncrement
                      return (a, tag)
```

Notez que cela fonctionne pour n'importe quel type `Traversable` , pas seulement notre type d' `Tree` !

Se débarrasser du passe-partout `Traversable`

GHC a une extension `DeriveTraversable` qui élimine le besoin d'écrire l'instance ci-dessus:

```
{-# LANGUAGE DeriveFunctor, DeriveFoldable, DeriveTraversable #-}

data Tree a = Tree a [Tree a]
            deriving (Show, Functor, Foldable, Traversable)
```

Lire Etat Monad en ligne: https://riptutorial.com/fr/haskell/topic/5740/etat-monad

Chapitre 27: Extensions de langage GHC communes

Remarques

Ces extensions de langage sont généralement disponibles lors de l'utilisation du compilateur Glasgow Haskell (GHC), car elles ne font pas partie du rapport linguistique Haskell 2010 approuvé. Pour utiliser ces extensions, il faut soit informer le compilateur en utilisant un indicateur, soit placer un programme `LANGUAGE` avant le mot-clé du `module` dans un fichier. La documentation officielle se trouve dans la section 7 du guide de l'utilisateur GCH.

Le format du programme `LANGUAGE` est `{-# LANGUAGE ExtensionOne, ExtensionTwo ... #-}` . C'est le littéral `{-#` suivi de `LANGUAGE` suivi d'une liste d'extensions séparées par des virgules, et enfin de la fermeture `#-}` . Plusieurs programmes `LANGUAGE` peuvent être placés dans un fichier.

Examples

MultiParamTypeClasses

C'est une extension très courante qui permet des classes de type avec plusieurs paramètres de type. Vous pouvez considérer MPTC comme une relation entre les types.

```
{-# LANGUAGE MultiParamTypeClasses #-}

class Convertable a b where
    convert :: a -> b

instance Convertable Int Float where
    convert i = fromIntegral i
```

L'ordre des paramètres est important.

Les MPTC peuvent parfois être remplacées par des familles de type.

FlexibleInstances

Les instances régulières nécessitent:

```
All instance types must be of the form (T a1 ... an)
where a1 ... an are *distinct type variables*,
and each type variable appears at most once in the instance head.
```

Cela signifie que, par exemple, alors que vous pouvez créer une instance pour `[a]` vous ne pouvez pas créer d'instance pour spécifiquement `[Int]` . `FlexibleInstances` détend que:

```
class C a where
```

```
-- works out of the box
instance C [a] where

-- requires FlexibleInstances
instance C [Int] where
```

Chaînes surchargées

Normalement, les littéraux de chaîne dans Haskell ont un type de `String` (qui est un alias de type pour `[Char]`). Bien que cela ne pose pas de problème aux petits programmes éducatifs, les applications du monde réel nécessitent souvent un stockage plus efficace, tel que `Text` ou `ByteString` .

`OverloadedStrings` change simplement le type de littéral en

```
"test" :: Data.String.IsString a => a
```

En leur permettant d'être directement transmis aux fonctions qui attendent un tel type. De nombreuses bibliothèques implémentent cette interface pour leurs types de type chaîne, y compris Data.Text et Data.ByteString, qui offrent tous deux des avantages en termes de temps et d'espace sur `[Char]` .

Il y a aussi des utilisations uniques de `OverloadedStrings` comme celles de la bibliothèque Postgresql-simple qui permet d'écrire des requêtes SQL entre guillemets comme une chaîne normale, mais fournit des protections contre la concaténation incorrecte, une source notoire d'attaques par injection SQL.

Pour créer une instance de la classe `IsString` , vous devez `fromString` fonction `fromString` . Exemple † :

```
data Foo = A | B | Other String deriving Show

instance IsString Foo where
  fromString "A" = A
  fromString "B" = B
  fromString xs  = Other xs

tests :: [ Foo ]
tests = [ "A", "B", "Testing" ]
```

---

† Cet exemple est une gracieuseté de Lyndon Maydwell (`sordina` sur GitHub) trouvé ici .

TupleSections

Une extension syntaxique qui permet d'appliquer le constructeur de tuple (qui est un opérateur) de manière sectionnelle:

```
(a,b) == (,) a b
```

```
-- With TupleSections
(a,b) == (,) a b == (a,) b == (,b) a
```

N-tuples

Il fonctionne également pour les tuples avec une arité supérieure à deux

```
(,2,) 1 3 == (1,2,3)
```

Cartographie

Cela peut être utile dans d'autres endroits où des sections sont utilisées:

```
map (,"tag") [1,2,3] == [(1,"tag"), (2, "tag"), (3, "tag")]
```

L'exemple ci-dessus sans cette extension ressemblerait à ceci:

```
map (\a -> (a, "tag")) [1,2,3]
```

UnicodeSyntax

Une extension qui vous permet d'utiliser des caractères Unicode à la place de certains opérateurs et noms intégrés.

ASCII	Unicode	Les usages)
`::`	::	a le type
`->`	→	types de fonctions, lambdas, branches de `case` , etc.
`=>`	⇒	contraintes de classe
`forall`	∀	polymorphisme explicite
`<-`	←	`do` notation
`*`	★	le type (ou type) des types (par exemple, `Int :: ★`)
`>-`	⤛	`proc` pour les `Arrows`
`-<`	⤙	`proc` pour les `Arrows`
`>>-`	⤜	`proc` pour les `Arrows`
`-<<`	⤚	`proc` pour les `Arrows`

Par exemple:

```
runST :: (forall s. ST s a) -> a
```

deviendrait

```
runST :: (∀ s. ST s a) → a
```

Notez que l'exemple `*` vs. `★` est légèrement différent: puisque `*` n'est pas réservé, `★` fonctionne de la même manière que `*` pour la multiplication, ou toute autre fonction nommée `(*)` , et vice-versa. Par exemple:

```
ghci> 2 ★ 3
6
ghci> let (*) = (+) in 2 ★ 3
5
ghci> let (★) = (-) in 2 * 3
-1
```

BinaryLiterals

Standard Haskell vous permet d'écrire des littéraux entiers en décimal (sans préfixe), hexadécimal (précédé de `0x` ou `0X`) et octal (précédé de `0o` ou `0O`). L'extension `BinaryLiterals` ajoute l'option de binary (précédé de `0b` ou `0B`).

```
0b1111 == 15     -- evaluates to: True
```

ExistentialQuantification

Il s'agit d'une extension de type système qui autorise les types quantifiés de manière existentielle ou, en d'autres termes, qui ont des variables de type qui ne sont instanciées qu'à l'exécution † .

Une valeur de type existentiel est similaire à une référence de classe de base abstraite dans les langages OO: vous ne connaissez pas le type exact dans contient, mais vous pouvez contraindre la *classe* de types.

```
data S = forall a. Show a => S a
```

ou de manière équivalente, avec la syntaxe GADT:

```
{-# LANGUAGE GADTs #-}
data S where
   S :: Show a => a -> S
```

Types existentiels ouvrent la porte à des choses comme des conteneurs presque hétérogènes: comme dit plus haut, il peut effectivement être différents types dans une `S` valeur, mais ils peuvent tous être `show` n, vous pouvez donc aussi faire

```
instance Show S where
    show (S a) = show a   -- we rely on (Show a) from the above
```

Maintenant, nous pouvons créer une collection de tels objets:

```
ss = [S 5, S "test", S 3.0]
```

Ce qui nous permet également d'utiliser le comportement polymorphe:

```
mapM_ print ss
```

Les existentiels peuvent être très puissants, mais notez qu'ils ne sont pas vraiment nécessaires dans Haskell. Dans l'exemple ci-dessus, tout ce que vous pouvez réellement faire avec l'instance `Show` est de montrer (duh!) Les valeurs, c.-à-d. Créer une représentation sous forme de chaîne. Le type `S` complet contient donc exactement autant d'informations que la chaîne que vous obtenez lorsque vous la montrez. Par conséquent, il est généralement préférable de simplement stocker cette chaîne tout de suite, d'autant plus que Haskell est paresseux et que, par conséquent, la chaîne ne sera d'abord qu'un non-évalué.

D'autre part, les existentiels causent des problèmes uniques. Par exemple, la manière dont les informations de type sont «cachées» dans un existentiel. Si vous faites correspondre un motif sur une valeur `S` , vous aurez le type contenu dans la portée (plus précisément son instance `Show`), mais cette information ne pourra jamais échapper à sa portée, qui devient donc une «société secrète»: le compilateur ne laisse *rien* échapper à la portée sauf les valeurs dont le type est déjà connu de l'extérieur. Cela peut conduire à des erreurs étranges comme `Couldn't match type 'a0' with '()' 'a0' is untouchable` .

---

† Comparez ceci avec le polymorphisme paramétrique ordinaire, qui est généralement résolu au moment de la compilation (permettant un effacement complet).

Les types existentiels sont différents des types Rank-N - ces extensions sont, pour ainsi dire, doubles les unes aux autres: pour utiliser réellement des valeurs de type existentiel, il faut une fonction polymorphe (éventuellement contrainte), comme `show` dans l'exemple. Une fonction polymorphe est *universellement* quantifiée, c'est-à-dire qu'elle fonctionne *pour tout* type dans une classe donnée, alors que la quantification existentielle signifie qu'elle fonctionne *pour un* type particulier qui est a priori inconnu. Si vous avez une fonction polymorphe, cela suffit, mais pour passer des fonctions polymorphes telles que des arguments, vous avez besoin de `{-# LANGUAGE Rank2Types #-}` :

```
genShowSs :: (∀ x . Show x => x -> String) -> [S] -> [String]
genShowSs f = map (\(S a) -> f a)
```

LambdaCase

Une extension syntaxique qui vous permet d'écrire `\case` à la place de `\arg -> case arg of` .

Considérons la définition de fonction suivante:

```
dayOfTheWeek :: Int -> String
dayOfTheWeek 0 = "Sunday"
dayOfTheWeek 1 = "Monday"
dayOfTheWeek 2 = "Tuesday"
dayOfTheWeek 3 = "Wednesday"
dayOfTheWeek 4 = "Thursday"
dayOfTheWeek 5 = "Friday"
dayOfTheWeek 6 = "Saturday"
```

Si vous voulez éviter de répéter le nom de la fonction, vous pourriez écrire quelque chose comme:

```
dayOfTheWeek :: Int -> String
dayOfTheWeek i = case i of
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

À l'aide de l'extension LambdaCase, vous pouvez écrire cela en tant qu'expression de fonction, sans avoir à nommer l'argument:

```
{-# LANGUAGE LambdaCase #-}

dayOfTheWeek :: Int -> String
dayOfTheWeek = \case
    0 -> "Sunday"
    1 -> "Monday"
    2 -> "Tuesday"
    3 -> "Wednesday"
    4 -> "Thursday"
    5 -> "Friday"
    6 -> "Saturday"
```

RankNTypes

Imaginez la situation suivante:

```
foo :: Show a => (a -> String) -> String -> Int -> IO ()
foo show' string int = do
   putStrLn (show' string)
   putStrLn (show' int)
```

Ici, nous voulons passer une fonction qui convertit une valeur en une chaîne, appliquer cette fonction à la fois à un paramètre de chaîne et à un paramètre int et les imprimer tous les deux. Dans mon esprit, il n'y a aucune raison que cela échoue! Nous avons une fonction qui fonctionne sur les deux types de paramètres que nous transmettons.

Malheureusement, cela ne va pas vérifier GHC déduit `a` type basé sur sa première occurrence

dans le corps de la fonction. C'est-à-dire dès que nous frappons:

```
putStrLn (show' string)
```

GHC déduira que `show' :: String -> String` , puisque `string` est une `String` . Il va exploser en essayant de `show' int` .

`RankNTypes` vous permet d'écrire la signature de type comme suit, en quantifiant toutes les fonctions qui satisfont le type de `show'` :

```
foo :: (forall a. Show a => (a -> String)) -> String -> Int -> IO ()
```

C'est le polymorphisme de rang 2: nous affirmons que la fonction `show'` doit fonctionner pour tous les `a` s *de* notre fonction, et que l'implémentation précédente fonctionne maintenant.

La `RankNTypes` extension permet l' imbrication arbitraire de `forall ...` blocs dans les signatures de type. En d'autres termes, il permet le polymorphisme de rang N.

Listes surchargées

ajouté dans GHC 7.8 .

OverloadedLists, similaire à OverloadedStrings , permet de désabuser les littéraux de liste comme suit:

```
[]          -- fromListN 0 []
[x]         -- fromListN 1 (x : [])
[x .. ]     -- fromList (enumFrom x)
```

Cela est pratique lorsque vous traitez des types tels que `Set` , `Vector` et `Map` s.

```
['0' .. '9']             :: Set Char
[1 .. 10]                :: Vector Int
[("default",0), (k1,v1)] :: Map String Int
['a' .. 'z']             :: Text
```

`IsList` classe `IsList` dans `GHC.Exts` est destinée à être utilisée avec cette extension.

`IsList` est équipé d'une fonction de type, `Item` , et trois fonctions, `fromList :: [Item l] -> l` , `toList :: l -> [Item l]` et `fromListN :: Int -> [Item l] -> l` où `fromListN` est facultatif. Les implémentations typiques sont:

```
instance IsList [a] where
  type Item [a] = a
  fromList = id
  toList   = id

instance (Ord a) => IsList (Set a) where
  type Item (Set a) = a
  fromList = Set.fromList
```

```
  toList   = Set.toList
```

Exemples tirés de OverloadedLists - GHC .

Dépendances fonctionnelles

Si vous avez une classe de type multi-paramètres avec les arguments a, b, c et x, cette extension vous permet d'exprimer que le type x peut être identifié de manière unique à partir de a, b et c:

```
class SomeClass a b c x | a b c -> x where ...
```

Lors de la déclaration d'une instance de cette classe, elle sera vérifiée par rapport à toutes les autres instances pour s'assurer que la dépendance fonctionnelle est maintenue, c'est-à-dire qu'aucune autre instance avec le même `abc` mais un `x` différent n'existe.

Vous pouvez spécifier plusieurs dépendances dans une liste séparée par des virgules:

```
class OtherClass a b c d | a b -> c d, a d -> b where ...
```

Par exemple dans MTL on peut voir:

```
class MonadReader r m| m -> r where ...
instance MonadReader r ((->) r) where ...
```

Maintenant, si vous avez une valeur de type `MonadReader a ((->) Foo) => a` , le compilateur peut en déduire `a ~ Foo` , puisque le second argument détermine complètement le premier et simplifiera le type en conséquence.

La classe `SomeClass` peut être considérée comme une fonction des arguments `abc` qui donnent `x` . De telles classes peuvent être utilisées pour effectuer des calculs dans le système de types.

GADT

Les types de données algébriques conventionnels sont paramétriques dans leurs variables de type. Par exemple, si on définit un ADT comme

```
data Expr a = IntLit Int
            | BoolLit Bool
            | If (Expr Bool) (Expr a) (Expr a)
```

avec l'espoir que cela exclut statiquement les conditions non bien typées, cela ne se comportera pas comme prévu puisque le type d' `IntLit :: Int -> Expr a` est quantifié `IntLit :: Int -> Expr a` : pour *tout* choix de `a` , il produit une valeur de type `Expr a` . En particulier, pour `a ~ Bool` , nous avons `IntLit :: Int -> Expr Bool` , ce qui nous permet de construire quelque chose comme `If (IntLit 1) e1 e2` ce que le type du constructeur `If` essayait d'éliminer.

Les types de données algébriques généralisés nous permettent de contrôler le type résultant d'un constructeur de données afin qu'il ne soit pas simplement paramétrique. Nous pouvons réécrire

notre `Expr` de type comme GADT comme celui - ci:

```
data Expr a where
  IntLit :: Int -> Expr Int
  BoolLit :: Bool -> Expr Bool
  If :: Expr Bool -> Expr a -> Expr a -> Expr a
```

Ici, le type du constructeur `IntLit` est `Int -> Expr Int` , donc `IntLit 1 :: Expr Bool` ne va pas taper.

La correspondance de modèle sur une valeur GADT entraîne un affinement du type du terme renvoyé. Par exemple, il est possible d'écrire un évaluateur pour `Expr a` comme ceci:

```
crazyEval :: Expr a -> a
crazyEval (IntLit x) =
   -- Here we can use `(+)` because x :: Int
   x + 1
crazyEval (BoolLit b) =
   -- Here we can use `not` because b :: Bool
   not b
crazyEval (If b thn els) =
  -- Because b :: Expr Bool, we can use `crazyEval b :: Bool`.
  -- Also, because thn :: Expr a and els :: Expr a, we can pass either to
  -- the recursive call to `crazyEval` and get an a back
  crazyEval $ if crazyEval b then thn else els
```

Notez que nous pouvons utiliser `(+)` dans les définitions ci-dessus parce que, par exemple, si `IntLit x` correspond à un motif, nous apprenons aussi `a ~ Int` (et de même pour `not` et `if_then_else_` pour `a ~ Bool`).

ScopedTypeVariables

`ScopedTypeVariables` vous permet de faire référence à des types universellement quantifiés dans une déclaration. Pour être plus explicite:

```
import Data.Monoid

foo :: forall a b c. (Monoid b, Monoid c) => (a, b, c) -> (b, c) -> (a, b, c)
foo (a, b, c) (b', c') = (a :: a, b'', c'')
    where (b'', c'') = (b <> b', c <> c') :: (b, c)
```

L'important est que nous puissions utiliser `a` , `b` et `c` pour instruire le compilateur dans des sous-expressions de la déclaration (le tuple dans la clause `where` et le premier `a` dans le résultat final). En pratique, `ScopedTypeVariables` aide à écrire des fonctions complexes en tant que somme de parties, ce qui permet au programmeur d'ajouter des signatures de type à des valeurs intermédiaires qui n'ont pas de types concrets.

PatternSynonymous

Les synonymes de motif sont des abstractions de motifs similaires à la façon dont les fonctions sont des abstractions d'expressions.

Pour cet exemple, examinons l'interface que `Data.Sequence` expose et voyons comment il peut être amélioré avec des synonymes de modèle. Le type `Seq` est un type de données qui, en interne, utilise une représentation compliquée pour obtenir une bonne complexité asymptotique pour diverses opérations, notamment O (1) (un) consing et (un) snocing.

Mais cette représentation est lourde et certains de ses invariants ne peuvent pas être exprimés dans le système de types de Haskell. De ce fait, le type `Seq` est exposé aux utilisateurs en tant que type abstrait, avec des fonctions d'accesseur et de constructeur préservant les invariants, parmi lesquelles:

```
empty :: Seq a

(<|) :: a -> Seq a -> Seq a
data ViewL a = EmptyL | a :< (Seq a)
viewl :: Seq a -> ViewL a

(|>) :: Seq a -> a -> Seq a
data ViewR a = EmptyR | (Seq a) :> a
viewr :: Seq a -> ViewR a
```

Mais utiliser cette interface peut être un peu lourd:

```
uncons :: Seq a -> Maybe (a, Seq a)
uncons xs = case viewl xs of
    x :< xs' -> Just (x, xs')
    EmptyL -> Nothing
```

Nous pouvons utiliser des patterns de vue pour le nettoyer quelque peu:

```
{-# LANGUAGE ViewPatterns #-}

uncons :: Seq a -> Maybe (a, Seq a)
uncons (viewl -> x :< xs) = Just (x, xs)
uncons _ = Nothing
```

En utilisant l'extension de langage `PatternSynonyms` , nous pouvons donner une interface encore plus belle en autorisant l'appariement de motifs à prétendre que nous avons une liste de cons- ou de snoc:

```
{-# LANGUAGE PatternSynonyms #-}
import Data.Sequence (Seq)
import qualified Data.Sequence as Seq

pattern Empty :: Seq a
pattern Empty <- (Seq.viewl -> Seq.EmptyL)

pattern (:<) :: a -> Seq a -> Seq a
pattern x :< xs <- (Seq.viewl -> x Seq.:< xs)

pattern (:>) :: Seq a -> a -> Seq a
pattern xs :> x <- (Seq.viewr -> xs Seq.:> x)
```

Cela nous permet d'écrire des `uncons` dans un style très naturel:

```
uncons :: Seq a -> Maybe (a, Seq a)
uncons (x :< xs) = Just (x, xs)
uncons _ = Nothing
```

RecordWildCards

Voir RecordWildCards

Lire Extensions de langage GHC communes en ligne:
https://riptutorial.com/fr/haskell/topic/1274/extensions-de-langage-ghc-communes

Chapitre 28: Flèches

Examples

Composition de fonction avec plusieurs canaux

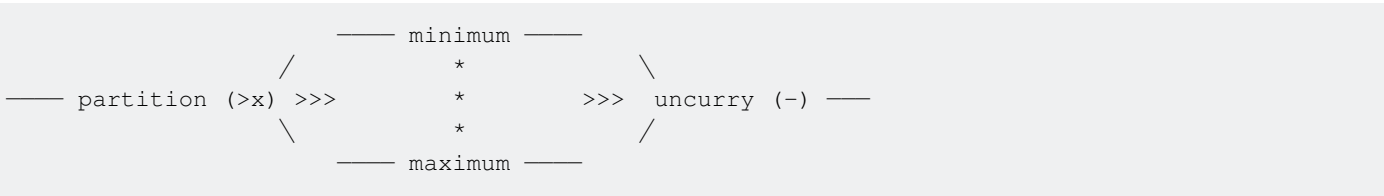
`Arrow` est, en gros, la classe des morphismes qui composent des fonctions semblables, avec à la fois une composition en série et une «composition parallèle». Bien qu'il soit intéressant de *généraliser les* fonctions, l'instance `Arrow (->)` elle-même est déjà très utile. Par exemple, la fonction suivante:

```
spaceAround :: Double -> [Double] -> Double
spaceAround x ys = minimum greater - maximum smaller
 where (greater, smaller) = partition (>x) ys
```

peut aussi être écrit avec des combinateurs de flèches:

```
spaceAround x = partition (>x) >>> minimum *** maximum >>> uncurry (-)
```

Ce type de composition peut être visualisé avec un diagramme:



Ici,

- L' opérateur `>>>` n'est qu'une version inversée de l'ordinaire `.` opérateur de composition (il y a aussi une version `<<<` qui compose de droite à gauche). Il achemine les données d'une étape de traitement à l'autre.
- les going `/ \` indiquent que le flux de données est divisé en deux «canaux». En termes de types Haskell, ceci est réalisé avec des tuples:

  ```
  partition (>x) :: [Double] -> ([Double], [Double])
  ```

  divise le flux en deux canaux `[Double]` , tandis que

  ```
  uncurry (-) :: (Double,Double) -> Double
  ```

  fusionne deux canaux `Double` .
- `***` est l'opérateur de composition parallèle † . Il permet au `maximum` et au `minimum` fonctionner indépendamment sur différents canaux des données. Pour les fonctions, la signature de cet opérateur est

```
(***) :: (b->c) -> (β->γ) -> (b,β)->(c,γ)
```

---

† Au moins dans la catégorie **Hask** (c'est-à-dire dans l'instance `Arrow (->)`), `f***g` ne calcule pas réellement `f` et `g` en parallèle comme dans, sur des threads différents. Cela serait théoriquement possible, cependant.

Lire Flèches en ligne: https://riptutorial.com/fr/haskell/topic/4912/fleches

Chapitre 29: Foncteur applicatif

Introduction

`Applicative` est la classe de types `f :: * -> *` qui permet une application de fonction soulevée sur une structure où la fonction est également incorporée dans cette structure.

Remarques

Définition

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

Notez la contrainte `Functor` sur `f` . La fonction `pure` renvoie son argument incorporé dans la structure `Applicative` . La fonction infixe `<*>` (prononcée "apply") est très similaire à `fmap` sauf avec la fonction intégrée dans la structure `Applicative` .

Une instance correcte de `Applicative` devrait satisfaire aux *lois applicatives* , bien que celles-ci ne soient pas appliquées par le compilateur:

```
pure id <*> a = a                              -- identity
pure (.) <*> a <*> b <*> c = a <*> (b <*> c)   -- composition
pure f <*> pure a = pure (f a)                 -- homomorphism
a <*> pure b = pure ($ b) <*> a                -- interchange
```

Examples

Définition alternative

Comme chaque foncteur applicatif est un foncteur , `fmap` peut toujours y être utilisé; Ainsi, l'essence de Applicative est l'appariement des contenus transportés, ainsi que la possibilité de le créer:

```
class Functor f => PairingFunctor f where
  funit :: f ()                  -- create a context, carrying nothing of import
  fpair :: (f a,f b) -> f (a,b)  -- collapse a pair of contexts into a pair-carrying context
```

Cette classe est isomorphe à `Applicative` .

```
pure a = const a <$> funit = a <$ funit

fa <*> fb = (\(a,b) -> a b) <$> fpair (fa, fb) = uncurry ($) <$> fpair (fa, fb)
```

Inversement,

```
funit = pure ()

fpair (fa, fb) = (,) <$> fa <*> fb
```

Instances courantes d'application

Peut être

`Maybe` - `Maybe` est-ce un foncteur applicatif contenant une valeur éventuellement absente.

```
instance Applicative Maybe where
    pure = Just

    Just f <*> Just x = Just $ f x
    _ <*> _ = Nothing
```

`pure` lève la valeur donnée dans `Maybe` en lui appliquant `Just` . La fonction `(<*>)` applique une fonction enveloppée dans un `Maybe` à une valeur dans `Maybe` . Si la fonction et la valeur sont présentes (construites avec `Just`), la fonction est appliquée à la valeur et le résultat encapsulé est renvoyé. Si l'un ou l'autre est manquant, le calcul ne peut pas continuer et `Nothing` n'est renvoyé à la place.

Des listes

Une façon pour les listes de correspondre à la signature de type `<*> :: [a -> b] -> [a] -> [b]` consiste à prendre le produit cartésien des deux listes, en associant chaque élément de la première liste à chaque élément du second:

```
fs <*> xs = [f x | f <- fs, x <- xs]
          -- = do { f <- fs; x <- xs; return (f x) }

pure x = [x]
```

Ceci est généralement interprété comme une émulation du non-déterminisme, avec une liste de valeurs représentant une valeur non-déterministe dont les valeurs possibles sont comprises dans cette liste; ainsi, une combinaison de deux valeurs non déterministes se situe sur toutes les combinaisons possibles des valeurs des deux listes:

```
ghci> [(+1),(+2)] <*> [3,30,300]
[4,31,301,5,32,302]
```

Flux infinis et listes zip

Il y a une classe de `Applicative` qui "compressent" leurs deux entrées ensemble. Un exemple simple est celui des flux infinis:

```
data Stream a = Stream { headS :: a, tailS :: Stream a }
```

`Applicative` instance `Applicative Stream` applique un flux de fonctions à un flux d'arguments au niveau des points, en associant les valeurs des deux flux par position. `pure` renvoie un flux constant - une liste infinie d'une seule valeur fixe:

```
instance Applicative Stream where
    pure x = let s = Stream x s in s
    Stream f fs <*> Stream x xs = Stream (f x) (fs <*> xs)
```

Les listes admettent également une instance `Applicative` "zippy", pour laquelle il existe le newtype `ZipList` :

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    ZipList xs <*> ZipList ys = ZipList $ zipWith ($) xs ys
```

Puisque `zip` ajuste son résultat en fonction de l'entrée la plus courte, la seule implémentation de `pure` satisfaisant aux lois `Applicative` est celle qui renvoie une liste infinie:

```
    pure a = ZipList (repeat a)   -- ZipList (fix (a:)) = ZipList [a,a,a,a,...
```

Par exemple:

```
ghci> getZipList $ ZipList [(+1),(+2)] <*> ZipList [3,30,300]
[4,32]
```

Les deux possibilités nous rappellent le produit externe et interne, similaire à la multiplication d'une matrice à 1 colonne (`nx 1`) avec une matrice à 1 rangée (`1 xm`) dans le premier cas, obtenant la matrice `nxm` (mais aplatie); ou en multipliant les matrices à une ligne et à une colonne (mais sans les additionner) dans le second cas.

Les fonctions

Lorsqu'elles sont spécialisées dans les fonctions `(->) r` , les signatures de type `pure` et `<*>` correspondent respectivement à celles des combinateurs `K` et `S` :

```
pure :: a -> (r -> a)
<*> :: (r -> (a -> b)) -> (r -> a) -> (r -> b)
```

`pure` doit être `const` , et `<*>` prend une paire de fonctions et les applique à un argument fixe, en appliquant les deux résultats suivants:

```
instance Applicative ((->) r) where
    pure = const
    f <*> g = \x -> f x (g x)
```

Les fonctions sont le prototypique "zippy" applicative. Par exemple, puisque les flux infinis sont isomorphes à `(->) Nat` , ...

```
-- | Index into a stream
to :: Stream a -> (Nat -> a)
to (Stream x xs) Zero = x
to (Stream x xs) (Suc n) = to xs n

-- | List all the return values of the function in order
from :: (Nat -> a) -> Stream a
from f = from' Zero
    where from' n = Stream (f n) (from' (Suc n))
```

... la représentation des flux de manière plus ordonnée produit automatiquement l'instance `Applicative` zippy.

Lire Foncteur applicatif en ligne: https://riptutorial.com/fr/haskell/topic/8162/foncteur-applicatif

Chapitre 30: Fonctions d'ordre supérieur

Remarques

Les fonctions d'ordre supérieur sont des fonctions qui prennent des fonctions comme paramètres et / ou des fonctions de retour comme valeurs de retour.

Examples

Principes de base des fonctions d'ordre supérieur

Passez en revue l' application partielle avant de continuer.

Dans Haskell, une fonction pouvant prendre d'autres fonctions comme arguments ou fonctions de retour est appelée *fonction d'ordre supérieur* .

Toutes *les fonctions d'ordre supérieur* sont les *suivantes* :

```
map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
takeWhile :: (a -> Bool) -> [a] -> [a]
dropWhile :: (a -> Bool) -> [a] -> [a]
iterate   :: (a -> a) -> a -> [a]
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
scanr     :: (a -> b -> b) -> b -> [a] -> [b]
scanl     :: (b -> a -> b) -> b -> [a] -> [b]
```

Celles-ci sont particulièrement utiles car elles nous permettent de créer de nouvelles fonctions par-dessus celles que nous avons déjà, en passant des fonctions comme arguments à d'autres fonctions. D'où le nom, *les fonctions d'ordre supérieur* .

Considérer:

```
Prelude> :t (map (+3))
(map (+3)) :: Num b => [b] -> [b]

Prelude> :t (map (=='c'))
(map (=='c')) :: [Char] -> [Bool]

Prelude> :t (map zipWith)
(map zipWith) :: [a -> b -> c] -> [[a] -> [b] -> [c]]
```

Cette capacité à créer facilement des fonctions (comme par exemple une application partielle telle qu'elle est utilisée ici) est l'une des caractéristiques qui rendent la programmation fonctionnelle particulièrement puissante et nous permet de dériver des solutions courtes et élégantes qui prendraient des dizaines de lignes dans d'autres langues. Par exemple, la fonction suivante nous donne le nombre d'éléments alignés dans deux listes.

```
aligned :: [a] ->  [a] -> Int
aligned xs ys = length (filter id (zipWith (==) xs ys))
```

Expressions lambda

Les expressions Lambda sont similaires aux *fonctions anonymes* dans d'autres langues.

Les expressions lambda sont des formules ouvertes qui spécifient également les variables à lier. L'évaluation (recherche de la valeur d'un appel de fonction) est alors réalisée en substituant les variables liées dans le corps de l'expression lambda aux arguments fournis par l'utilisateur. En termes simples, les expressions lambda nous permettent d'exprimer des fonctions au moyen de la liaison de variables et de la substitution .

Les expressions lambda ressemblent à

```
\x -> let {y = ...x...} in y
```

Dans une expression lambda, les variables du côté gauche de la flèche sont considérées comme liées dans la partie droite, c'est-à-dire le corps de la fonction.

Considérons la fonction mathématique

```
f(x) = x^2
```

En tant que définition de Haskell, c'est

```
f    x =  x^2

f = \x -> x^2
```

ce qui signifie que la fonction `f` est équivalente à l'expression lambda `\x -> x^2` .

Considérons le paramètre de la `map` fonction d'ordre supérieur, qui est une fonction du type `a -> b` . Dans le cas où il n'est utilisé qu'une fois dans un appel à `map` et nulle part ailleurs dans le programme, il convient de le spécifier comme une expression lambda au lieu de nommer une telle fonction. Écrit comme une expression lambda,

```
\x -> let {y = ...x...} in y
```

`x` contient une valeur de type `a` , `...x...` est une expression Haskell qui fait référence à la variable `x` et `y` détient une valeur de type `b` . Ainsi, par exemple, nous pourrions écrire ce qui suit

```
map (\x -> x + 3)

map (\(x,y) -> x * y)

map (\xs -> 'c':xs) ["apples", "oranges", "mangos"]

map (\f -> zipWith f [1..5] [1..5]) [(+), (*), (-)]
```

Currying

Dans Haskell, toutes les fonctions sont considérées comme curry: c'est-à-dire que toutes les fonctions de Haskell ne prennent qu'un *seul* argument.

Prenons la fonction `div` :

```
div :: Int -> Int -> Int
```

Si nous appelons cette fonction avec 6 et 2, nous obtenons sans surprise 3:

```
Prelude> div 6 2
3
```

Cependant, cela ne se comporte pas comme nous pourrions le penser. Le premier `div 6` est évalué et **renvoie une fonction** de type `Int -> Int` . Cette fonction résultante est ensuite appliquée à la valeur 2 qui donne 3.

Lorsque nous regardons la signature de type d'une fonction, nous pouvons changer notre façon de penser de "prend deux arguments de type `Int` " pour "prend un `Int` et retourne une fonction qui prend un `Int` ". Ceci est réaffirmé si l'on considère que les flèches dans la notation de type associent *à la droite* , donc `div` peut en fait être lu ainsi:

```
div :: Int -> (Int -> Int)
```

En général, la plupart des programmeurs peuvent ignorer ce comportement au moins pendant qu'ils apprennent la langue. D'un point de vue théorique , "les preuves formelles sont plus faciles lorsque toutes les fonctions sont traitées de manière uniforme (un argument, un résultat)."

Lire Fonctions d'ordre supérieur en ligne: https://riptutorial.com/fr/haskell/topic/4539/fonctions-d-ordre-superieur

Chapitre 31: Functor

Introduction

`Functor` est la classe de types `f :: * -> *` qui peuvent être *mis* en *correspondance* sur covariante. Le mappage d'une fonction sur une structure de données applique la fonction à tous les éléments de la structure sans modifier la structure elle-même.

Remarques

Un foncteur peut être considéré comme un conteneur pour une valeur ou un contexte de calcul. Les exemples sont `Maybe a` - `Maybe a` ou `[a]` . L'article de Typeclassopedia présente un bon résumé des concepts de Functors.

Pour être considéré comme un vrai foncteur, une instance doit respecter les 2 lois suivantes:

Identité

```
fmap id == id
```

Composition

```
fmap (f . g) = (fmap f) . (fmap g)
```

Examples

Instances courantes de Functor

Peut être

`Maybe` - `Maybe` un `Functor` contenant une valeur éventuellement absente:

```
instance Functor Maybe where
    fmap f Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

`Maybe` l'instance de `Functor` applique une fonction à une valeur enveloppée dans un `Just` . Si le calcul a déjà échoué (donc la valeur `Maybe` est un `Nothing`), alors il n'y a pas de valeur à appliquer la fonction, donc `fmap` est un no-op.

```
> fmap (+ 3) (Just 3)
Just 6
> fmap length (Just "mousetrap")
```

```
Just 9
> fmap sqrt Nothing
Nothing
```

Nous pouvons vérifier les lois du foncteur pour cette instance en utilisant le raisonnement équationnel. Pour la loi d'identité,

```
fmap id Nothing
Nothing  -- definition of fmap
id Nothing  -- definition of id

fmap id (Just x)
Just (id x)  -- definition of fmap
Just x  -- definition of id
id (Just x)  -- definition of id
```

Pour la loi de composition,

```
(fmap f . fmap g) Nothing
fmap f (fmap g Nothing)  -- definition of (.)
fmap f Nothing  -- definition of fmap
Nothing  -- definition of fmap
fmap (f . g) Nothing  -- because Nothing = fmap f Nothing, for all f

(fmap f . fmap g) (Just x)
fmap f (fmap g (Just x))  -- definition of (.)
fmap f (Just (g x))  -- definition of fmap
Just (f (g x))  -- definition of fmap
Just ((f . g) x)  -- definition of (.)
fmap (f . g) (Just x)  -- definition of fmap
```

Des listes

L'instance de `Functor` de `Functor` de `Functor` applique la fonction à toutes les valeurs de la liste en place.

```
instance Functor [] where
    fmap f [] = []
    fmap f (x:xs) = f x : fmap f xs
```

Cela pourrait aussi être écrit comme une compréhension de la liste: `fmap f xs = [fx | x <- xs]` .

Cet exemple montre que `fmap` généralise la `map` . `map` opère uniquement sur les listes, alors que `fmap` fonctionne sur un `Functor` arbitraire.

La loi d'identité peut être démontrée par induction:

```
-- base case
fmap id []
[]  -- definition of fmap
id []  -- definition of id
```

```
-- inductive step
fmap id (x:xs)
id x : fmap id xs  -- definition of fmap
x : fmap id xs  -- definition of id
x : id xs  -- by the inductive hypothesis
x : xs  -- definition of id
id (x : xs)  -- definition of id
```

et de même, la loi de composition:

```
-- base case
(fmap f . fmap g) []
fmap f (fmap g [])  -- definition of (.)
fmap f []  -- definition of fmap
[]  -- definition of fmap
fmap (f . g) []  -- because [] = fmap f [], for all f

-- inductive step
(fmap f . fmap g) (x:xs)
fmap f (fmap g (x:xs))  -- definition of (.)
fmap f (g x : fmap g xs)  -- definition of fmap
f (g x) : fmap f (fmap g xs)  -- definition of fmap
(f . g) x : fmap f (fmap g xs)  -- definition of (.)
(f . g) x : fmap (f . g) xs  -- by the inductive hypothesis
fmap (f . g) xs  -- definition of fmap
```

Les fonctions

Tous les `Functor` ressemblent pas à un conteneur. Les instances de `Functor` des `Functor` appliquent une fonction à la valeur de retour d'une autre fonction.

```
instance Functor ((->) r) where
    fmap f g = \x -> f (g x)
```

Notez que cette définition est équivalente à `fmap = (.)` . Donc `fmap` généralise la composition des fonctions.

Vérification de la loi d'identité:

```
fmap id g
\x -> id (g x)  -- definition of fmap
\x -> g x  -- definition of id
g  -- eta-reduction
id g  -- definition of id
```

et la loi de composition:

```
(fmap f . fmap g) h
fmap f (fmap g h)  -- definition of (.)
fmap f (\x -> g (h x))  -- definition of fmap
\y -> f ((\x -> g (h x)) y)  -- definition of fmap
\y -> f (g (h y))  -- beta-reduction
\y -> (f . g) (h y)  -- definition of (.)
```

```
fmap (f . g) h  -- definition of fmap
```

Définition de classe du foncteur et des lois

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Une façon de le voir est que `fmap` *soulève* une fonction de valeurs dans une fonction de valeurs dans un contexte `f` .

Une instance correcte de `Functor` devrait satisfaire aux *lois* du *foncteur* , bien que celles-ci ne soient pas appliquées par le compilateur:

```
fmap id = id                    -- identity
fmap f . fmap g = fmap (f . g)  -- composition
```

Il existe un alias infix couramment utilisé pour `fmap` appelé `<$>` .

```
infixl 4 <$>
(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap
```

Remplacement de tous les éléments d'un foncteur par une valeur unique

Le module `Data.Functor` contient deux combinateurs, `<$` et `$>` , qui ignorent toutes les valeurs contenues dans un foncteur, les remplaçant toutes par une seule valeur constante.

```
infixl 4 <$, $>

<$ :: Functor f => a -> f b -> f a
(<$) = fmap . const

$> :: Functor f => f a -> b -> f b
($>) = flip (<$)
```

`void` ignore la valeur de retour d'un calcul.

```
void :: Functor f => f a -> f ()
void = (() <$)
```

Foncteurs polynomiaux

Il existe un ensemble utile de combinateurs de types pour la construction de gros `Functor` parmi les plus petits. Ce sont des exemples instructifs d'exemples de `Functor` , et ils sont également utiles en tant que technique de programmation générique, car ils peuvent être utilisés pour représenter une grande classe de foncteurs communs.

Le foncteur d'identité

Le foncteur d'identité encapsule simplement son argument. C'est une implémentation au niveau du combinateur `I` du calcul SKI.

```
newtype I a = I a

instance Functor I where
    fmap f (I x) = I (f x)
```

`I` peux être trouvé, sous le nom d' `Identity` , dans le module `Data.Functor.Identity` .

Le foncteur constant

Le foncteur constant ignore son second argument, ne contenant qu'une valeur constante. C'est un analogue au niveau du type de `const` , le combinateur `K` du calcul SKI.

```
newtype K c a = K c
```

Notez que `K ca` ne contient aucune valeur `a` ; `K ()` est isomorphe au `Proxy` . Cela signifie que `K` mise en œuvre de l » de `fmap` ne fait aucun mappage du tout!

```
instance Functor (K c) where
    fmap _ (K c) = K c
```

`K` est également appelé `Const` , à partir de `Data.Functor.Const` .

Les autres foncteurs dans cet exemple combinent des foncteurs plus petits en plus gros.

Produits Functor

Le foncteur prend une paire de foncteurs et les emballe. C'est analogue à un tuple, sauf que while `(,) :: * -> * -> *` opère sur les `types` `* , (:*:) :: (* -> *) -> (* -> *) -> (* -> *)` opère sur les `functors` `* -> * .`

```
infixl 7 :*:
data (f :*: g) a = f a :*: g a

instance (Functor f, Functor g) => Functor (f :*: g) where
    fmap f (fx :*: gy) = fmap f fx :*: fmap f gy
```

Ce type peut être trouvé, sous le nom de `Product` , dans le module `Data.Functor.Product` .

Coproduits Functor

Tout comme `:*:` est analogue à `(,)` `:+:` est l'analogue de niveau fonctionnel de `Either` .

```
infixl 6 :+:
data (f :+: g) a = InL (f a) | InR (g a)

instance (Functor f, Functor g) => Functor (f :+: g) where
    fmap f (InL fx) = InL (fmap f fx)
    fmap f (InR gy) = InR (fmap f gy)
```

`:+:` peut être trouvé sous le nom `Sum` , dans le module `Data.Functor.Sum` .

Composition de foncteur

Enfin, `:.:` Fonctionne comme un type de niveau `(.)` , En prenant la sortie d'un foncteur et la plomberie dans l'entrée d'un autre.

```
infixr 9 :.:
newtype (f :.: g) a = Cmp (f (g a))

instance (Functor f, Functor g) => Functor (f :.: g) where
    fmap f (Cmp fgx) = Cmp (fmap (fmap f) fgx)
```

Le type de `Compose` se trouve dans `Data.Functor.Compose`

Functeurs polynomiaux pour la programmation générique

`I` , `K` `:*:` , `:+:` et `:.:` Peut être considéré comme un kit de blocs de construction pour une certaine classe de types de données simples. Le kit devient particulièrement puissant lorsque vous le combinez avec des points fixes, car les types de données `Functor` avec ces combinateurs sont automatiquement des instances de `Functor` . Vous utilisez le kit pour créer un type de modèle, en marquant des points récursifs à l'aide de `I` , puis en le `Fix` dans `Fix` pour obtenir un type pouvant être utilisé avec le zoo standard des schémas de récurrence.

prénom	En tant que type de données	Utiliser le kit de foncteur
Paires de valeurs	`data Pair a = Pair aa`	`type Pair = I :*: I`
Deux par deux grilles	`type Grid a = Pair (Pair a)`	`type Grid = Pair :.: Pair`
Nombres naturels	`data Nat = Zero | Succ Nat`	`type Nat = Fix (K () :+: I)`
Des listes	`data List a = Nil | Cons a (List a)`	`type List a = Fix (K () :+: K a :*: I)`
Arbres binaires	`data Tree a = Leaf | Node (Tree a) a (Tree a)`	`type Tree a = Fix (K () :+: I :*: K a :*: I)`

prénom	En tant que type de données	Utiliser le kit de foncteur
Rosiers	`data Rose a = Rose a (List (Rose a))`	`type Rose a = Fix (K a :*: List :.: I)`

Cette approche "kit" de la conception des types de données est l'idée derrière *les* bibliothèques de *programmation génériques* telles que les `generics-sop` . L'idée est d'écrire des opérations génériques en utilisant un kit comme celui présenté ci-dessus, puis d'utiliser une classe de type pour convertir des types de données arbitraires vers et à partir de leur représentation générique:

```
class Generic a where
    type Rep a  -- a generic representation built using a kit
    to :: a -> Rep a
    from :: Rep a -> a
```

Les foncteurs dans la théorie des catégories

Un foncteur est défini dans la théorie des catégories comme une carte de préservation de la structure (un «homomorphisme») entre les catégories. Plus précisément, tous les objets sont mappés à des objets et toutes les flèches sont associées à des flèches, de sorte que les lois de la catégorie sont préservées.

La catégorie dans laquelle les objets sont des types Haskell et les morphismes sont des fonctions Haskell appelée **Hask** . Ainsi, un foncteur de **Hask** à **Hask** consisterait en un mappage des types en types et en un mappage des fonctions vers les fonctions.

La relation que cette notion théorique de catégorie a avec la construction de programmation de Haskell `Functor` est plutôt directe. Le mappage des types aux types prend la forme d'un type `f :: * -> *` , et le mappage des fonctions vers les fonctions prend la forme d'une fonction `fmap :: (a -> b) -> (fa -> fb)` . Assembler ces éléments dans une classe,

```
class Functor (f :: * -> *) where
    fmap :: (a -> b) -> f a -> f b
```

`fmap` est une opération qui prend une fonction (un type de morphisme), `:: a -> b` , et la mappe à une autre fonction, `:: fa -> fb` . On suppose (mais c'est au programmeur de s'assurer) que les instances de `Functor` sont bien des foncteurs mathématiques, préservant la structure catégorielle de **Hask** :

```
fmap (id {- :: a -> a -})  ==  id {- :: f a -> f a -}
fmap (h . g)               ==  fmap h . fmap g
```

`fmap` lève une fonction `:: a -> b` dans une sous-catégorie de **Hask** d'une manière qui préserve à la fois l'existence de flèches d'identité et l'associativité de la composition.

La classe `Functor` ne code que les *endo* **functors** sur **Hask** . Mais en mathématiques, les foncteurs peuvent mapper entre des catégories arbitraires. Un encodage plus fidèle de ce concept ressemblerait à ceci:

```
class Category c where
    id  :: c i i
    (.) :: c j k -> c i j -> c i k

class (Category c1, Category c2) => CFunctor c1 c2 f where
    cfmap :: c1 a b -> c2 (f a) (f b)
```

La classe standard Functor est un cas particulier de cette classe dans lequel les catégories source et cible sont toutes deux **Hask** . Par exemple,

```
instance Category (->) where        -- Hask
    id    = \x -> x
    f . g = \x -> f (g x)

instance CFunctor (->) (->) [] where
    cfmap = fmap
```

Functor dérivant

L'extension de langage `DeriveFunctor` permet à GHC de générer automatiquement des instances de `Functor` .

```
{-# LANGUAGE DeriveFunctor #-}

data List a = Nil | Cons a (List a) deriving Functor

-- instance Functor List where            -- automatically defined
--   fmap f Nil = Nil
--   fmap f (Cons x xs) = Cons (f x) (fmap f xs)

map :: (a -> b) -> List a -> List b
map = fmap
```

Lire Functor en ligne: https://riptutorial.com/fr/haskell/topic/3800/functor

Chapitre 32: GHCJS

Introduction

GHCJS est un compilateur Haskell to JavaScript qui utilise l'API GHC.

Examples

Courir "Hello World!" avec Node.js

`ghcjs` peut être `ghcjs` avec les mêmes arguments de ligne de commande que `ghc` . Les programmes générés peuvent être exécutés directement depuis le shell avec Node.js et SpiderMonkey jsshell . par exemple:

```
$ ghcjs -o helloWorld helloWorld.hs
$ node helloWorld.jsexe/all.js
Hello world!
```

Lire GHCJS en ligne: https://riptutorial.com/fr/haskell/topic/9260/ghcjs

Chapitre 33: Graphisme avec Gloss

Examples

Installation de Gloss

Gloss est facilement installé à l'aide de l'outil Cabal. Après avoir installé Cabal, il est possible de faire passer le `cabal install gloss` d'installation de la cabine.

Alternativement, le paquet peut être construit à partir de sources, en téléchargeant le source depuis Hackage ou GitHub , et en procédant comme suit:

1. Entrez le répertoire `gloss/gloss-rendering/` et `cabal install`
2. Entrez le répertoire `gloss/gloss/` et encore une fois `cabal install`

Obtenir quelque chose à l'écran

Dans Gloss, vous pouvez utiliser la fonction d' `display` pour créer des graphiques statiques très simples.

Pour l'utiliser, il faut d'abord `import Graphics.Gloss` . Ensuite, dans le code, il devrait y avoir ce qui suit:

```
main :: IO ()
main = display window background drawing
```

`window` est de type `Display` qui peut être construite de deux manières:

```
-- Defines window as an actual window with a given name and size
window = InWindow name (width, height) (0,0)

-- Defines window as a fullscreen window
window = FullScreen
```

Ici, le dernier argument `(0,0)` dans `InWindow` marque l'emplacement du coin supérieur gauche.

Pour les versions antérieures à la version 1.11: Dans les anciennes versions de Gloss, `FullScreen` prend un autre argument qui correspond à la taille de l'image qui est dessinée et qui est ensuite étendue à la taille plein écran, par exemple: `FullScreen (1024,768)`

`background` est de type `Color` . Il définit la couleur d'arrière-plan, c'est aussi simple que:

```
background = white
```

Ensuite, nous arrivons au dessin lui-même. Les dessins peuvent être très complexes. Comment spécifier ceux-ci seront couverts ailleurs ([on peut s'y référer pour le moment] [1]), mais cela peut être aussi simple que le cercle suivant avec un rayon de 80:

```
drawing = Circle 80
```

Exemple de résumé

Comme plus ou moins indiqué dans la documentation de Hackage, obtenir quelque chose à l'écran est aussi simple que:

```
import Graphics.Gloss

main :: IO ()
main = display window background drawing
    where
      window = InWindow "Nice Window" (200, 200) (0, 0)
      background = white
      drawing = Circle 80
```

Lire Graphisme avec Gloss en ligne: https://riptutorial.com/fr/haskell/topic/5570/graphisme-avec-gloss

Chapitre 34: Gtk3

Syntaxe

- obj <- <widgetName> Nouveau - Comment les widgets (par exemple Windows, boutons, grilles) sont créés
- set <widget> [<attributes>] - Définit les attributs tels que définis dans la documentation du widget (par exemple, buttonLabel)
- sur <widget> <event> <action IO> - Ajouter une action IO à un widget Signal self (par exemple, buttonActivated)

Remarques

Sur de nombreuses distributions Linux, la bibliothèque Haskell Gtk3 est disponible sous forme de package dans le gestionnaire de packages système (par exemple, `libghc-gtk` dans l'APT Ubuntu). Cependant, pour certains développeurs , il pourrait être préférable d'utiliser un outil comme `stack` pour gérer des environnements isolés, et ont gtk3 installé via `cabal` au lieu de via une installation globale par le gestionnaire de paquets de systèmes. Pour cette option, `gtk2hs-buildtools` est requis. Exécutez `cabal install gtk2hs-buildtools` avant d'ajouter `gtk` , `gtk3` ou toute autre bibliothèque Haskell basée sur Gtk à vos projets. L'entrée `build-depends` de votre fichier cabal.

Examples

Bonjour tout le monde en GTK

Cet exemple montre comment créer un simple "Hello World" dans Gtk3, en configurant une fenêtre et des boutons. L'exemple de code montrera également comment définir différents attributs et actions sur les widgets.

```
module Main (Main.main) where

import Graphics.UI.Gtk

main :: IO ()
main = do
  initGUI
  window <- windowNew
  on window objectDestroy mainQuit
  set window [ containerBorderWidth := 10, windowTitle := "Hello World" ]
  button <- buttonNew
  set button [ buttonLabel := "Hello World" ]
  on button buttonActivated $ do
    putStrLn "A \"clicked\"-handler to say \"destroy\""
    widgetDestroy window
  set window [ containerChild := button ]
  widgetShowAll window
```

```
  mainGUI -- main loop
```

Lire Gtk3 en ligne: https://riptutorial.com/fr/haskell/topic/7342/gtk3

Chapitre 35: Interface de fonction étrangère

Syntaxe

- import étranger ccall unsafe "foo" hFoo :: Int32 -> IO Int32 {- Importe une fonction nommée `foo` dans un fichier objet et définit le symbole `hFoo` qui peut être appelé avec le code Haskell. -}

Remarques

Bien que cabal prenne en charge l'inclusion de bibliothèques C et C ++ dans un package Haskell, il existe quelques bogues. D'abord, si vous avez des données (plutôt qu'une fonction) définies dans `bo` qui sont utilisées dans `ao` , et répertoriez les `C-sources: ac, bc` , alors cabal ne pourra pas trouver les données. Ceci est documenté dans # 12152 . Une solution de contournement lors de l'utilisation de cabal est de réorganiser la liste de `C-sources` `C-sources: bc, ac` pour qu'elle soit `C-sources: bc, ac` . Cela peut ne pas fonctionner lorsque vous utilisez la pile, car la pile lie toujours les `C-sources` ordre alphabétique, quel que soit l'ordre dans lequel vous les répertoriez.

Un autre problème est que vous devez entourer tout code C ++ dans les fichiers d'en-tête (.h) avec les gardes `#ifdef __cplusplus` . C'est parce que GHC ne comprend pas le code C ++ dans les fichiers d'en-tête. Vous pouvez toujours écrire du code C ++ dans les fichiers d'en-tête, mais vous devez l'entourer de gardes.

`ccall` fait référence à la *convention d'appel* ; actuellement `ccall` et `stdcall` (convention Pascal) sont supportés. Le mot clé `unsafe` est facultatif; Cela réduit la surcharge pour les fonctions simples, mais peut provoquer des blocages si la fonction étrangère bloque indéfiniment ou si les autorisations d'exécution sont insuffisantes 1 .

Examples

Appeler C de Haskell

Pour des raisons de performances, ou en raison de l'existence de bibliothèques C matures, vous souhaiterez peut-être appeler du code C à partir d'un programme Haskell. Voici un exemple simple de la façon dont vous pouvez transmettre des données à une bibliothèque C et obtenir une réponse.

foo.c:

```
#include <inttypes.h>

int32_t foo(int32_t a) {
  return a+1;
}
```

Foo.hs:

```
import Data.Int

main :: IO ()
main = print =<< hFoo 41

foreign import ccall unsafe "foo" hFoo :: Int32 -> IO Int32
```

Le mot-clé `unsafe` génère un appel plus efficace que 'safe', mais requiert que le code C ne rappelle jamais le système Haskell. Puisque `foo` est complètement en C et n'appellera jamais Haskell, nous pouvons utiliser `unsafe` .

Nous devons également demander à cabal de compiler et de lier le code source C.

foo.cabal:

```
name:                foo
version:             0.0.0.1
build-type:          Simple
extra-source-files:  *.c
cabal-version:       >= 1.10

executable foo
  default-language: Haskell2010
  main-is:       Foo.hs
  C-sources:     foo.c
  build-depends: base
```

Ensuite, vous pouvez exécuter:

```
> cabal configure
> cabal build foo
> ./dist/build/foo/foo
42
```

Passer Haskell fonctionne comme des rappels au code C.

Il est très courant que les fonctions C acceptent des pointeurs vers d'autres fonctions en tant qu'arguments. L'exemple le plus courant consiste à définir une action à exécuter lorsqu'un bouton est cliqué dans une bibliothèque de toolkit GUI. Il est possible de passer des fonctions Haskell en tant que rappels C.

Pour appeler cette fonction C:

```
void event_callback_add (Object *obj, Object_Event_Cb func, const void *data)
```

nous l'importons d'abord dans le code Haskell:

```
foreign import ccall "header.h event_callback_add"
    callbackAdd :: Ptr () -> FunPtr Callback -> Ptr () -> IO ()
```

Maintenant, en regardant comment `Object_Event_Cb` est défini dans l'en-tête C, définissez ce que `Callback` est dans Haskell:

```
type Callback = Ptr () -> Ptr () -> IO ()
```

Enfin, créez une fonction spéciale qui `FunPtr Callback` fonction Haskell de type `Callback` dans un pointeur `FunPtr Callback` :

```
foreign import ccall "wrapper"
    mkCallback :: Callback -> IO (FunPtr Callback)
```

Maintenant, nous pouvons enregistrer le rappel avec le code C:

```
cbPtr <- mkCallback $ \objPtr dataPtr -> do
    -- callback code
    return ()
callbackAdd cpPtr
```

Il est important de libérer `FunPtr` alloué une fois que vous `FunPtr` le rappel:

```
freeHaskellFunPtr cbPtr
```

Lire Interface de fonction étrangère en ligne: https://riptutorial.com/fr/haskell/topic/7256/interface-de-fonction-etrangere

Chapitre 36: IO

Examples

Lecture de tous les contenus d'une entrée standard dans une chaîne

```
main = do
    input <- getContents
    putStr input
```

Contribution:

```
This is an example sentence.
And this one is, too!
```

Sortie:

```
This is an example sentence.
And this one is, too!
```

Remarque: Ce programme imprimera en fait des parties de la sortie avant que toutes les entrées aient été entièrement lues. Par exemple, si vous utilisez `getContents` sur un fichier de 50MiB, l'évaluation paresseuse et le ramasse-miettes de Haskell garantissent que seul le fichier les parties du fichier actuellement nécessaires (lire: indispensable pour une exécution ultérieure) seront chargées en mémoire. Ainsi, le fichier 50MiB ne sera pas chargé en mémoire en même temps.

Lecture d'une ligne à partir d'une entrée standard

```
main = do
    line <- getLine
    putStrLn line
```

Contribution:

```
This is an example.
```

Sortie:

```
This is an example.
```

Analyse et construction d'un objet à partir d'une entrée standard

```
readFloat :: IO Float
readFloat =
    fmap read getLine
```

```
main :: IO ()
main = do
    putStr "Type the first number: "
    first <- readFloat

    putStr "Type the second number: "
    second <- readFloat

    putStrLn $ show first ++ " + " ++ show second ++ " = " ++ show ( first + second )
```

Contribution:

```
Type the first number: 9.5
Type the second number: -2.02
```

Sortie:

```
9.5 + -2.02 = 7.48
```

Lecture des descripteurs de fichiers

Comme dans plusieurs autres parties de la bibliothèque d'E / S, les fonctions qui utilisent implicitement un flux standard ont un équivalent dans `System.IO` qui effectue le même travail, mais avec un paramètre supplémentaire de type `Handle` , représentant le flux en cours. manipulé Par exemple, `getLine :: IO String` a un équivalent `hGetLine :: Handle -> IO String` .

```
import System.IO( Handle, FilePath, IOMode( ReadMode ),
                  openFile, hGetLine, hPutStr, hClose, hIsEOF, stderr )

import Control.Monad( when )


dumpFile :: Handle -> FilePath -> Integer -> IO ()

dumpFile handle filename lineNumber = do      -- show file contents line by line
    end <- hIsEOF handle
    when ( not end ) $ do
        line <- hGetLine handle
        putStrLn $ filename ++ ":" ++ show lineNumber ++ ": " ++ line
        dumpFile handle filename $ lineNumber + 1


main :: IO ()

main = do
    hPutStr stderr "Type a filename: "
    filename <- getLine
    handle <- openFile filename ReadMode
    dumpFile handle filename 1
    hClose handle
```

Contenu du fichier `example.txt` :

```
This is an example.
Hello, world!
This is another example.
```

Contribution:

```
Type a filename: example.txt
```

Sortie:

```
example.txt:1: This is an example.
example.txt:2: Hello, world!
example.txt:3: This is another example
```

Vérification des conditions de fin de fichier

Un peu contre-intuitif par rapport à la manière dont les bibliothèques d'E / S standard de la plupart des autres langages le font, `isEOF` d'Haskell ne vous oblige pas à effectuer une opération de lecture avant de vérifier une condition EOF; le runtime le fera pour vous.

```
import System.IO( isEOF )


eofTest :: Int -> IO ()
eofTest line = do
    end <- isEOF
    if end then
        putStrLn $ "End-of-file reached at line " ++ show line ++ "."
    else do
        getLine
        eofTest $ line + 1


main :: IO ()
main =
    eofTest 1
```

Contribution:

```
Line #1.
Line #2.
Line #3.
```

Sortie:

```
End-of-file reached at line 4.
```

Lire des mots d'un fichier entier

Dans Haskell, il est souvent judicieux de *ne pas se préoccuper des descripteurs de fichiers* , mais simplement de lire ou d'écrire un fichier entier directement du disque vers la mémoire † , et

d'effectuer tout le partitionnement / traitement du texte avec la structure de données pure. Cela évite de mélanger IO et la logique du programme, ce qui peut grandement aider à éviter les bogues.

```
-- | The interesting part of the program, which actually processes data
--   but doesn't do any IO!
reverseWords :: String -> [String]
reverseWords = reverse . words

-- | A simple wrapper that only fetches the data from disk, lets
--   'reverseWords' do its job, and puts the result to stdout.
main :: IO ()
main = do
   content <- readFile "loremipsum.txt"
   mapM_ putStrLn $ reverseWords content
```

Si `loremipsum.txt` contient

```
Lorem ipsum dolor sit amet,
consectetur adipiscing elit
```

alors le programme sortira

```
elit
adipiscing
consectetur
amet,
sit
dolor
ipsum
Lorem
```

Ici, `mapM_` a parcouru la liste de tous les mots du fichier et les a imprimés sur une ligne distincte avec `putStrLn` .

---

† Si vous pensez que cela vous fait perdre de la mémoire, vous avez raison. En fait, la paresse de Haskell évite souvent que le fichier entier doive résider simultanément dans la mémoire… mais attention, ce type d'OE paresseux provoque ses propres problèmes. Pour les applications critiques en termes de performances, il est souvent judicieux de forcer la lecture intégrale du fichier en une seule fois; vous pouvez le faire avec la `Data.Text` version de `readFile` .

IO définit l'action `main` de votre programme

Pour rendre un programme Haskell exécutable, vous devez fournir un fichier avec une fonction `main` de type `IO ()`

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

Lorsque Haskell est compilé, il examine les données d' `IO` et les transforme en un exécutable. Lorsque nous exécuterons ce programme, il imprimera `Hello world!` .

Si vous avez des valeurs de type `IO a` autre que `main` elles ne feront rien.

```
other :: IO ()
other = putStrLn "I won't get printed"

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

Compiler ce programme et le lancer aura le même effet que le dernier exemple. Le code dans l' `other` est ignoré.

Afin de rendre le code dans d' `other` effets d'exécution, vous devez le *composer* en `main` . Aucune valeur d' `IO` / `IO` n'est pas finalement composée dans `main` aura un effet d'exécution. Pour composer deux valeurs d' `IO` manière séquentielle, vous pouvez utiliser `do` -notation:

```
other :: IO ()
other = putStrLn "I will get printed... but only at the point where I'm composed into main"

main :: IO ()
main = do
  putStrLn "Hello world!"
  other
```

Lorsque vous compilez et exécutez *ce* programme, il affiche

```
Hello world!
I will get printed... but only at the point where I'm composed into main
```

Notez que l'ordre des opérations est décrit par la façon dont l' `other` été composé dans l'ordre `main` et non dans l'ordre de définition.

Rôle et objet des opérations d'information

Haskell est un langage pur, ce qui signifie que les expressions ne peuvent avoir d'effets secondaires. Un effet secondaire est tout ce que l'expression ou la fonction fait d'autre que produire une valeur, par exemple modifier un compteur global ou imprimer en sortie standard.

Dans Haskell, les calculs à effets secondaires (en particulier ceux qui peuvent avoir un effet sur le monde réel) sont modélisés en utilisant `IO` . Strictement parlant, `IO` est un constructeur de type, prenant un type et produisant un type. Par exemple, `IO Int` est le type d'un calcul d'E / S produisant une valeur `Int` . Le type `IO` est *abstrait* et l'interface fournie pour `IO` garantit que certaines valeurs illégales (c'est-à-dire des fonctions avec des types non sensibles) ne peuvent exister, en veillant à ce que toutes les fonctions intégrées exécutant IO aient un type de retour compris dans `IO` .

Lorsqu'un programme Haskell est exécuté, le calcul représenté par la valeur Haskell nommée `main` , dont le type peut être `IO x` pour tout type `x` , est exécuté.

Manipulation des valeurs IO

Il existe de nombreuses fonctions dans la bibliothèque standard fournissant des actions d' `IO` typiques qu'un langage de programmation général doit exécuter, telles que la lecture et l'écriture sur des descripteurs de fichiers. Les actions `IO` générales sont créées et combinées principalement avec deux fonctions:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

Cette fonction (généralement appelée *bind*) prend une action `IO` et une fonction qui renvoie une action `IO` et produit l'action `IO` qui résulte de l'application de la fonction à la valeur produite par la première action `IO` .

```
return :: a -> IO a
```

Cette fonction prend n'importe quelle valeur (c'est-à-dire une valeur pure) et retourne le calcul IO qui ne fait pas IO et produit la valeur donnée. En d'autres termes, il s'agit d'une action d'E / S sans opération.

Il y a des fonctions générales supplémentaires qui sont souvent utilisées, mais toutes peuvent être écrites en fonction des deux précédentes. Par exemple, `(>>) :: IO a -> IO b -> IO b` est similaire à `(>>=)` mais le résultat de la première action est ignoré.

Un programme simple accueillant l'utilisateur en utilisant ces fonctions:

```
main :: IO ()
main =
  putStrLn "What is your name?" >>
  getLine >>= \name ->
  putStrLn ("Hello " ++ name ++ "!")
```

Ce programme utilise également `putStrLn :: String -> IO ()` et `getLine :: IO String` .

Note: les types de certaines fonctions ci-dessus sont en réalité plus généraux que ceux donnés (à savoir `>>=` , `>>` et `return`).

Sémantique IO

Le type `IO` dans Haskell a une sémantique très similaire à celle des langages de programmation impératifs. Par exemple, quand on écrit `s1 ; s2` dans un langage impératif pour indiquer l'instruction d'exécution `s1` , puis l'instruction `s2` , on peut écrire `s1 >> s2` pour modéliser la même chose dans Haskell.

Cependant, la sémantique d' `IO` divergent légèrement de ce que l'on pourrait attendre d'un contexte impératif. La fonction de `return` n'interrompt *pas* le flux de contrôle - elle n'a aucun effet sur le programme si une autre action `IO` est exécutée en séquence. Par exemple, `return () >> putStrLn "boom"` imprime correctement "boom" sur la sortie standard.

La sémantique formelle de `IO` peut être donnée en termes d'égalités simples impliquant les fonctions de la section précédente:

```
return x >>= f ≡ f x, ∀ f x
y >>= return ≡ return y, ∀ y
(m >>= f) >>= g ≡ m >>= (\x -> (f x >>= g)), ∀ m f g
```

Ces lois sont généralement appelées identité de gauche, identité correcte et composition, respectivement. Ils peuvent être énoncés plus naturellement en termes de fonction

```
(>=>) :: (a -> IO b) -> (b -> IO c) -> a -> IO c
(f >=> g) x = (f x) >>= g
```

comme suit:

```
return >=> f ≡ f, ∀ f
f >=> return ≡ f, ∀ f
(f >=> g) >=> h ≡ f >=> (g >=> h), ∀ f g h
```

Paresseux IO

Les fonctions exécutant des calculs d'E / S sont généralement strictes, ce qui signifie que toutes les actions précédentes d'une séquence d'actions doivent être terminées avant que l'action suivante ne commence. Typiquement, c'est un comportement utile et attendu - `putStrLn "X" >> putStrLn "Y"` devrait imprimer "XY". Cependant, certaines fonctions de bibliothèque effectuent des E / S paresseuses, ce qui signifie que les actions d'E / S requises pour produire la valeur ne sont effectuées que lorsque la valeur est réellement consommée. Les exemples de telles fonctions sont `getContents` et `readFile` . Les E / S différées peuvent réduire considérablement les performances d'un programme Haskell. Par conséquent, lors de l'utilisation des fonctions de bibliothèque, il convient de noter les fonctions paresseuses.

IO et `do` notation

Haskell fournit une méthode plus simple pour combiner différentes valeurs d'E / S en plus grandes valeurs d'E / S. Cette syntaxe spéciale est connu comme `do` notation * et est tout simplement le sucre syntaxique pour des usages de la `>>=` , `>>` et `return` fonctions.

Le programme de la section précédente peut être écrit de deux manières différentes en utilisant la notation `do` , la première étant sensible à la mise en page et la seconde insensible à la mise en page:

```
main = do
  putStrLn "What is your name?"
  name <- getLine
  putStrLn ("Hello " ++ name ++ "!")
```

```
main = do {
  putStrLn "What is your name?" ;
  name <- getLine ;
  putStrLn ("Hello " ++ name ++ "!")
  }
```

Les trois programmes sont exactement équivalents.

---

* Notez que `do` notation est également applicable à une classe plus large de constructeurs de types appelés *monads* .

Obtenir le "un" de "IO a"

Une question commune est «J'ai une valeur de `IO a` , mais je veux faire quelque chose à ce `a` valeur: comment puis-je y avoir accès » Comment peut-on opérer sur des données provenant du monde extérieur (par exemple, incrémenter un numéro saisi par l'utilisateur)?

Le fait est que si vous utilisez une fonction pure sur des données obtenues de manière impeccable, le résultat est encore impur. Cela dépend de ce que l'utilisateur a fait! Une valeur de type `IO a` signifie un "calcul avec effet secondaire entraînant une valeur de type `a` " qui ne *peut* être exécuté qu'en (a) le composant en `main` et (b) en compilant et en exécutant votre programme. Pour cette raison, il n'y a aucun moyen au sein du monde pur Haskell à « obtenir le `a` sur ».

Au lieu de cela, nous voulons construire un nouveau calcul, une nouvelle valeur d' `IO` / `IO` , qui utilise `a` valeur *à l'exécution* . Ceci est une autre façon de *composer les* valeurs des entrées- `IO` et, encore une fois, nous pouvons utiliser `do` -notation:

```
-- assuming
myComputation :: IO Int

getMessage :: Int -> String
getMessage int = "My computation resulted in: " ++ show int

newComputation :: IO ()
newComputation = do
  int <- myComputation       -- we "bind" the result of myComputation to a name, 'int'
  putStrLn $ getMessage int   -- 'int' holds a value of type Int
```

Ici, nous utilisons une fonction pure (`getMessage`) pour transformer un `Int` en `String` , mais nous utilisons la notation `do` pour l'appliquer au résultat d'un calcul `IO myComputation` *lorsque* (après) le calcul s'exécute. Le résultat est un calcul `IO` plus important, `newComputation` . Cette technique consistant à utiliser des fonctions pures dans un contexte impur est appelée *lever* .

Ecrire à stdout

Conformément à la spécification de langage Haskell 2010, les fonctions d'E / S standard disponibles dans Prelude sont les suivantes: aucune importation n'est requise pour les utiliser.

`putChar :: Char -> IO ()` - écrit un `char` dans `stdout`

```
Prelude> putChar 'a'
aPrelude>  -- Note, no new line
```

`putStr :: String -> IO ()` - écrit une `String` dans `stdout`

```
Prelude> putStr "This is a string!"
This is a string!Prelude>  -- Note, no new line
```

`putStrLn :: String -> IO ()` - écrit une `String` dans `stdout` et ajoute une nouvelle ligne

```
Prelude> putStrLn "Hi there, this is another String!"
Hi there, this is another String!
```

`print :: Show a => a -> IO ()` - écrit `a` instance de `Show` à `stdout`

```
Prelude> print "hi"
"hi"
Prelude> print 1
1
Prelude> print 'a'
'a'
Prelude> print (Just 'a')  -- Maybe is an instance of the `Show` type class
Just 'a'
Prelude> print Nothing
Nothing
```

Rappelez-vous que vous pouvez instancier `Show` pour vos propres types en utilisant la `deriving` :

```
-- In ex.hs
data Person = Person { name :: String } deriving Show
main = print (Person "Alex")  -- Person is an instance of `Show`, thanks to `deriving`
```

Chargement et exécution dans GHCi:

```
Prelude> :load ex.hs
[1 of 1] Compiling ex             ( ex.hs, interpreted )
Ok, modules loaded: ex.
*Main> main  -- from ex.hs
Person {name = "Alex"}
*Main>
```

Lecture de `stdin`

Conformément à la spécification de langage Haskell 2010 , les fonctions d'E / S standard disponibles dans Prelude sont les suivantes: aucune importation n'est requise pour les utiliser.

`getChar :: IO Char` - lire un `Char` de `stdin`

```
-- MyChar.hs
```

```
main = do
  myChar <- getChar
  print myChar

-- In your shell

runhaskell MyChar.hs
a -- you enter a and press enter
'a'  -- the program prints 'a'
```

`getLine :: IO String` - `getLine :: IO String` une `String` de `stdin` , sans nouvelle ligne

```
Prelude> getLine
Hello there!  -- user enters some text and presses enter
"Hello there!"
```

`read :: Read a => String -> a` - convertit une String en une valeur

```
Prelude> read "1" :: Int
1
Prelude> read "1" :: Float
1.0
Prelude> read "True" :: Bool
True
```

D'autres fonctions moins courantes sont:

- `getContents :: IO String` - Retourne toutes les entrées utilisateur sous la forme d'une chaîne unique, qui est lue paresseusement au fur et à mesure des besoins
- `interact :: (String -> String) -> IO ()` - prend une fonction de type String-> String comme argument. L'intégralité de l'entrée du périphérique d'entrée standard est transmise à cette fonction en tant qu'argument

Lire IO en ligne: https://riptutorial.com/fr/haskell/topic/1904/io

Chapitre 37: Lecteur / lecteur

Introduction

Reader fournit des fonctionnalités pour transmettre une valeur à chaque fonction. Un guide utile avec quelques diagrammes peut être trouvé ici: http://adit.io/posts/2013-06-10-three-useful-monads.html

Examples

Démonstration simple

Un élément clé du lecteur monad est la fonction `ask` (https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-Reader.html#v:ask) , définie pour illustrative fins:

```
import Control.Monad.Trans.Reader hiding (ask)
import Control.Monad.Trans

ask :: Monad m => ReaderT r m r
ask = reader id

main :: IO ()
main = do
  let f = (runReaderT $ readerExample) :: Integer -> IO String
  x <- f 100
  print x
  --
  let fIO = (runReaderT $ readerExampleIO) :: Integer -> IO String
  y <- fIO 200
  print y

readerExample :: ReaderT Integer IO String
readerExample = do
  x <- ask
  return $ "The value is: " ++ show x

liftAnnotated :: IO a -> ReaderT Integer IO a
liftAnnotated = lift

readerExampleIO :: ReaderT Integer IO String
readerExampleIO = do
  x <- reader id
  lift $ print "Hello from within"
  liftAnnotated $ print "Hello from within..."
  return $ "The value is: " ++ show x
```

Le ci-dessus imprimera:

```
"The value is: 100"
"Hello from within"
"Hello from within..."
"The value is: 200"
```

Lire Lecteur / lecteur en ligne: https://riptutorial.com/fr/haskell/topic/9320/lecteur---lecteur

Chapitre 38: Lentille

Introduction

Lens est une bibliothèque pour Haskell qui fournit des objectifs, des isomorphismes, des plis, des traversées, des getters et des setters, exposant une interface uniforme pour interroger et manipuler des structures arbitraires, contrairement aux concepts d'accesseur et de mutateur de Java.

Remarques

Qu'est-ce qu'un objectif?

Les objectifs (et autres optiques) nous permettent de séparer la description de la *manière dont* nous voulons accéder à certaines données de *ce que* nous voulons en faire. Il est important de faire la distinction entre la notion abstraite d'une lentille et la mise en œuvre concrète. Comprendre de manière abstraite facilite beaucoup la programmation avec des `lens` à long terme. Il existe de nombreuses représentations isomorphes des lentilles. Par conséquent, pour cette discussion, nous éviterons toute discussion concrète sur la mise en œuvre et donnerons une vue d'ensemble de haut niveau des concepts.

Mise au point

Un concept important dans la compréhension abstraite est la notion de *focalisation* . Les optiques importantes se *concentrent* sur une partie spécifique d'une structure de données plus grande sans oublier le contexte plus large. Par exemple, l'objectif `_1` se concentre sur le premier élément d'un tuple mais n'oublie pas ce qui se trouvait dans le second champ.

Une fois que nous avons le focus, nous pouvons alors parler des opérations que nous sommes autorisés à effectuer avec un objectif. Étant donné un `Lens sa` qui , lorsqu'il est donné un type de données de type `s` se concentre sur un particulier `a` , nous pouvons soit

1. Extraire le `a` en oubliant le contexte supplémentaire ou
2. Remplace le `a` en fournissant une nouvelle valeur

Celles-ci correspondent aux opérations `get` et `set` bien connues qui sont généralement utilisées pour caractériser une lentille.

Autres optiques

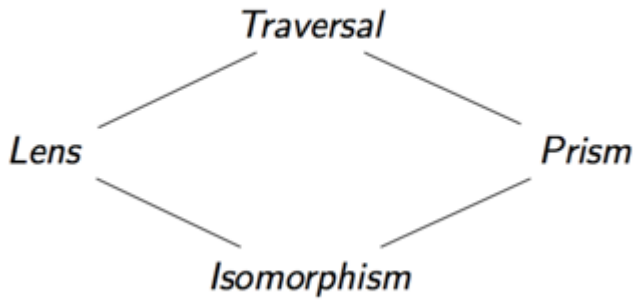
Nous pouvons parler d'autres optiques d'une manière similaire.

Optique	Met l'accent sur...
Lentille	Une partie d'un produit
Prisme	Une partie d'une somme
Traversal	Zéro ou plusieurs parties d'une structure de données
Isomorphisme	...

Chaque optique se concentre différemment, en tant que tel, selon le type d'optique, nous pouvons effectuer différentes opérations.

Composition

De plus, nous pouvons composer n'importe laquelle des deux optiques que nous avons déjà discutées afin de spécifier des accès de données complexes. Les quatre types d'optiques dont nous avons discuté forment un réseau, le résultat de la composition de deux optiques ensemble est leur limite supérieure.



Par exemple, si nous composons ensemble une lentille et un prisme, nous obtenons une traversée. La raison en est que, par leur composition (verticale), nous nous concentrons d'abord sur une partie d'un produit, puis sur une partie d'une somme. Le résultat étant une optique qui se concentre précisément sur zéro ou une partie de nos données, ce qui est un cas particulier d'une traversée. (Ceci est aussi parfois appelé une traversée affine).

À Haskell

La popularité de Haskell s'explique par la représentation très succincte de l'optique. Toutes les optiques ne sont que des fonctions d'une certaine forme qui peuvent être composées ensemble en utilisant la composition de fonctions. Cela conduit à une intégration très légère, ce qui facilite l'intégration de l'optique dans vos programmes. De plus, en raison des particularités de l'encodage, la composition de la fonction calcule automatiquement la limite supérieure de deux optiques que nous composons. Cela signifie que nous pouvons réutiliser les mêmes combinateurs pour différentes optiques sans coulée explicite.

Examples

Manipulation de tuples avec lentille

Obtenir

```
("a", 1) ^. _1 -- returns "a"
("a", 1) ^. _2 -- returns 1
```

Réglage

```
("a", 1) & _1 .~ "b" -- returns ("b", 1)
```

Modifier

```
("a", 1) & _2 %~ (+1) -- returns ("a", 2)
```

`both` Traversal

```
(1, 2) & both *~ 2 -- returns (2, 4)
```

Objectifs pour disques

Enregistrement simple

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Lens

data Point = Point {
    _x :: Float,
    _y :: Float
}
makeLenses ''Point
```

Les objectifs `x` et `y` sont créés.

```
let p = Point 5.0 6.0
p ^. x     -- returns 5.0
set x 10 p -- returns Point { _x = 10.0, _y = 6.0 }
p & x +~ 1 -- returns Point { _x = 6.0, _y = 6.0 }
```

Gestion des enregistrements avec des noms de champs répétés

```
data Person = Person { _personName :: String }
makeFields ''Person
```

Crée une classe de type `HasName` , le `name` objectif pour `Person` et fait de `Person` une instance de `HasName` . Les enregistrements suivants seront également ajoutés à la classe:

```
data Entity = Entity { _entityName :: String }
makeFields ''Entity
```

L'extension Template Haskell est requise pour que `makeFields` fonctionne. Techniquement, il est tout à fait possible de créer les objectifs de cette manière par d'autres moyens, par exemple à la main.

Verres Stateful

Les opérateurs de lentilles ont des variantes utiles qui fonctionnent dans des contextes avec état. Ils sont obtenus en remplaçant `~` avec `=` dans le nom de l'opérateur.

```
(+~) :: Num a => ASetter s t a a -> a -> s -> t
(+=) :: (MonadState s m, Num a) => ASetter' s a -> a -> m ()
```

> Remarque: les variantes avec état ne sont pas censées modifier le type, elles ont donc les signatures d' `Lens'` ou d' `Simple Lens'` .

Se débarrasser de `&` chaînes

Si les opérations d'objectif doivent être enchaînées, cela ressemble souvent à ceci:

```
change :: A -> A
change a = a & lensA %~ operationA
             & lensB %~ operationB
             & lensC %~ operationC
```

Cela fonctionne grâce à l'associativité de `&` . La version avec état est plus claire, cependant.

```
change a = flip execState a $ do
    lensA %= operationA
    lensB %= operationB
    lensC %= operationC
```

Si `lensX` est réellement `id` , l'opération entière peut bien sûr être exécutée directement en la soulevant simplement avec la `modify` .

Code impératif avec état structuré

En supposant que cet exemple indique:

```
data Point = Point { _x :: Float, _y :: Float }
data Entity = Entity { _position :: Point, _direction :: Float }
data World = World { _entities :: [Entity] }

makeLenses ''Point
makeLenses ''Entity
makeLenses ''World
```

Nous pouvons écrire du code qui ressemble aux langages impératifs classiques, tout en nous permettant d'utiliser les avantages de Haskell:

```
updateWorld :: MonadState World m => m ()
updateWorld = do
    -- move the first entity
    entities . ix 0 . position . x += 1

    -- do some operation on all of them
    entities . traversed . position %= \p -> p `pointAdd` ...

    -- or only on a subset
    entities . traversed . filtered (\e -> e ^. position.x > 100) %= ...
```

Écrire une lentille sans gabarit Haskell

Pour démystifier Template Haskell, supposez que vous avez

```
data Example a = Example { _foo :: Int, _bar :: a }
```

puis

```
makeLenses 'Example
```

produit (plus ou moins)

```
foo :: Lens' (Example a) Int
bar :: Lens (Example a) (Example b) a b
```

Il n'y a rien de particulièrement magique, cependant. Vous pouvez les écrire vous-même:

```
foo :: Lens' (Example a) Int
--  :: Functor f => (Int -> f Int) -> (Example a -> f (Example a))    ;; expand the alias
foo wrap (Example foo bar) = fmap (\newFoo -> Example newFoo bar) (wrap foo)

bar :: Lens (Example a) (Example b) a b
--  :: Functor f => (a -> f b) -> (Example a -> f (Example b))    ;; expand the alias
bar wrap (Example foo bar) = fmap (\newBar -> Example foo newBar) (wrap bar)
```

Essentiellement, vous voulez "visiter" le "focus" de votre objectif avec la fonction `wrap` , puis reconstruire le type "entier".

Lens et Prism

Une `Lens' sa` signifie que vous pouvez *toujours* trouver un `a` dans n'importe quel `s` . Un `Prism' sa` signifie que vous pouvez *parfois* trouver que `s` *est* juste `a` mais parfois c'est autre chose.

Pour être plus clair, nous avons `_1 :: Lens' (a, b) a` parce que tout tuple a *toujours* un premier élément. Nous avons `_Just :: Prism' (Maybe a) a` parce que *parfois* `Maybe a` est `a` valeur enveloppée dans `Just` mais *parfois* c'est `Nothing` .

Avec cette intuition, certains combinateurs standard peuvent être interprétés parallèlement les uns aux autres

- `view :: Lens' sa -> (s -> a)` "obtient" le `a` de `s`
- `set :: Lens' sa -> (a -> s -> s)` "définit" `a` emplacement dans `s`
- `review :: Prism' sa -> (a -> s)` "réalise" qu'un `a` pourrait être un `s`
- `preview :: Prism' sa -> (s -> Maybe a)` "tente" de transformer un `s` en `a` .

Une autre façon d'y penser est qu'une valeur de type `Lens' sa` démontre que `s` a la même structure que `(r, a)` pour un `r` inconnu. Par contre, `Prism' sa` démontre que `s` a la même structure que `r Either ra` pour certains `r` . Nous pouvons écrire ces quatre fonctions ci-dessus avec cette connaissance:

```
-- `Lens' s a` is no longer supplied, instead we just *know* that `s ~ (r, a)`

view :: (r, a) -> a
view (r, a) = a

set :: a -> (r, a) -> (r, a)
set a (r, _) = (r, a)

-- `Prism' s a` is no longer supplied, instead we just *know* that `s ~ Either r a`

review :: a -> Either r a
review a = Right a

preview :: Either r a -> Maybe a
preview (Left _) = Nothing
preview (Right a) = Just a
```

Les traversées

Un `Traversal' sa` montre que `s` a 0 à beaucoup `a` s à l' intérieur de celui - ci.

```
toListOf :: Traversal' s a -> (s -> [a])
```

Tout type `t` qui est `Traversable` automatiquement ce `traverse :: Traversal (ta) a` .

Nous pouvons utiliser un `Traversal` pour définir ou carte sur tous ces `a` valeur

```
> set traverse 1 [1..10]
[1,1,1,1,1,1,1,1,1,1]

> over traverse (+1) [1..10]
[2,3,4,5,6,7,8,9,10,11]
```

Un `f :: Lens' sa` dit qu'il y a exactement `a` intérieur de `s` . A `g :: Prism' ab` dit qu'il y a 0 ou 1 `b` s dans `a` . Composer `f . g` nous donne un `Traversal' sb` car suite à `f` puis `g` montre comment il y a 0 à 1 `b` s dans `s` .

Les lentilles composent

Si vous avez un `f :: Lens' ab` et un `g :: Lens' bc` alors `f . g` est une `Lens' ac` obtenue en suivant `f` abord et ensuite `g` . Notamment:

- Les lentilles composent des fonctions (ce *sont* vraiment des fonctions)
- Si vous pensez à la fonctionnalité de `view` de `Lens` , il semble que les flux de données "de gauche à droite" - cela peut sembler en arrière à votre intuition normale pour la composition de la fonction. Par contre, vous devriez vous sentir naturel si vous pensez `.` -notation comme cela se passe dans les langages OO.

Plus que la simple composition de l' `Lens` avec l' `Lens` , `(.)` Peut être utilisé pour composer presque tous les types de type " `Lens` ". Il n'est pas toujours facile de voir le résultat puisque le type devient plus difficile à suivre, mais vous pouvez utiliser le tableau des `lens` pour le comprendre. La composition `x . y` a le type de la limite inférieure des types de `x` et `y` dans ce graphique.

Lentilles de classe

Outre la fonction standard `makeLenses` pour générer des `Lens` , `Control.Lens.TH` offre également la fonction `makeClassy` . `makeClassy` a le même type et fonctionne de la même manière que `makeLenses` , avec une différence essentielle. En plus de générer les objectifs standard et les traversées, si le type n'a pas d'arguments, il créera également une classe décrivant tous les types de données qui possèdent le type en tant que champ. Par exemple

```
data Foo = Foo { _fooX, _fooY :: Int }
  makeClassy ''Foo
```

créera

```
class HasFoo t where
   foo :: Simple Lens t Foo

instance HasFoo Foo where foo = id

fooX, fooY :: HasFoo t => Simple Lens t Int
```

Champs avec makeFields

(Cet exemple copié à partir de cette réponse StackOverflow)

Disons que vous avez un certain nombre de types de données différents que tous devraient avoir une lentille du même nom, en l'occurrence la `capacity` . La tranche `makeFields` va créer une classe qui accomplit cela sans conflits d'espace de noms.

```
{-# LANGUAGE FunctionalDependencies
           , MultiParamTypeClasses
           , TemplateHaskell
  #-}

module Foo
where

import Control.Lens

data Foo
  = Foo { fooCapacity :: Int }
  deriving (Eq, Show)
$(makeFields ''Foo)

data Bar
  = Bar { barCapacity :: Double }
  deriving (Eq, Show)
$(makeFields ''Bar)
```

Puis en ghci:

```
*Foo
λ let f = Foo 3
|     b = Bar 7
|
b :: Bar
f :: Foo

*Foo
λ fooCapacity f
3
it :: Int

*Foo
λ barCapacity b
7.0
it :: Double

*Foo
λ f ^. capacity
3
it :: Int

*Foo
λ b ^. capacity
7.0
it :: Double

λ :info HasCapacity
class HasCapacity s a | s -> a where
  capacity :: Lens' s a
    -- Defined at Foo.hs:14:3
instance HasCapacity Foo Int -- Defined at Foo.hs:14:3
instance HasCapacity Bar Double -- Defined at Foo.hs:19:3
```

Donc, ce qui est réellement fait est déclaré une classe `HasCapacity sa` , où la capacité est une `Lens'` de `s` à `a` (`a` est fixé une fois que s est connu). Il a compris le nom "capacity" en supprimant le nom (en minuscules) du type de données du champ; Je trouve agréable de ne pas avoir à utiliser

un trait de soulignement sur le nom du champ ou sur le nom de l'objectif, car parfois la syntaxe d'enregistrement est ce que vous voulez. Vous pouvez utiliser makeFieldsWith et les diverses lentilles pour avoir différentes options pour calculer les noms d'objectif.

Au cas où cela vous aiderait, utilisez ghci -ddump-splices Foo.hs:

```
[1 of 1] Compiling Foo              ( Foo.hs, interpreted )
Foo.hs:14:3-18: Splicing declarations
    makeFields ''Foo
  ======>
    class HasCapacity s a | s -> a where
      capacity :: Lens' s a
    instance HasCapacity Foo Int where
      {-# INLINE capacity #-}
      capacity = iso (\ (Foo x_a7fG) -> x_a7fG) Foo
Foo.hs:19:3-18: Splicing declarations
    makeFields ''Bar
  ======>
    instance HasCapacity Bar Double where
      {-# INLINE capacity #-}
      capacity = iso (\ (Bar x_a7ne) -> x_a7ne) Bar
Ok, modules loaded: Foo.
```

La première épissure a donc fait de la classe `HasCapcity` et ajouté une instance pour Foo; le second utilisait la classe existante et créait une instance pour Bar.

Cela fonctionne également si vous importez la classe `HasCapcity` d'un autre module; `makeFields` peut ajouter plus d'instances à la classe existante et répartir vos types sur plusieurs modules. Mais si vous l'utilisez à nouveau dans un autre module où vous n'avez pas importé la classe, cela créera une nouvelle classe (avec le même nom) et vous aurez deux objectifs de capacité surchargés distincts qui ne sont pas compatibles.

Lire Lentille en ligne: https://riptutorial.com/fr/haskell/topic/891/lentille

Chapitre 39: Les foncteurs communs comme base des comonades cofree

Examples

Cofree Vide ~~ Vide

Donné

```
data Empty a
```

nous avons

```
data Cofree Empty a
   --  = a :< ...  not possible!
```

Cofree (Const c) ~~ Writer c

Donné

```
data Const c a = Const c
```

nous avons

```
data Cofree (Const c) a
     = a :< Const c
```

qui est isomorphe à

```
data Writer c a = Writer c a
```

Identité Cofree ~~ Stream

Donné

```
data Identity a = Identity a
```

nous avons

```
data Cofree Identity a
     = a :< Identity (Cofree Identity a)
```

qui est isomorphe à

```
data Stream a = Stream a (Stream a)
```

Cofree Peut-être ~~ NonEmpty

Donné

```
data Maybe a = Just a
             | Nothing
```

nous avons

```
data Cofree Maybe a
     = a :< Just (Cofree Maybe a)
     | a :< Nothing
```

qui est isomorphe à

```
data NonEmpty a
     = NECons a (NonEmpty a)
     | NESingle a
```

Cofree (Writer w) ~~ WriterT w Stream

Donné

```
data Writer w a = Writer w a
```

nous avons

```
data Cofree (Writer w) a
     = a :< (w, Cofree (Writer w) a)
```

ce qui équivaut à

```
data Stream (w,a)
     = Stream (w,a) (Stream (w,a))
```

qui peut correctement être écrit comme `WriterT w Stream` avec

```
data WriterT w m a = WriterT (m (w,a))
```

Cofree (Soit e) ~~ NonEmptyT (Writer e)

Donné

```
data Either e a = Left e
                | Right a
```

nous avons

```
data Cofree (Either e) a
     = a :< Left e
     | a :< Right (Cofree (Either e) a)
```

qui est isomorphe à

```
data Hospitable e a
     = Sorry_AllIHaveIsThis_Here'sWhy a e
     | EatThis a (Hospitable e a)
```

ou, si vous vous engagez à n'évaluer le journal qu'après le résultat complet, `NonEmptyT (Writer e) a` avec

```
data NonEmptyT (Writer e) a = NonEmptyT (e,a,[a])
```

Cofree (Reader x) ~~ Moore x

Donné

```
data Reader x a = Reader (x -> a)
```

nous avons

```
data Cofree (Reader x) a
     = a :< (x -> Cofree (Reader x) a)
```

qui est isomorphe à

```
data Plant x a
     = Plant a (x -> Plant x a)
```

aka machine Moore .

Lire Les foncteurs communs comme base des comonades cofree en ligne: https://riptutorial.com/fr/haskell/topic/8258/les-foncteurs-communs-comme-base-des-comonades-cofree

Chapitre 40: Liste des compréhensions

Examples

Compréhension de la liste de base

Haskell a des compréhensions de listes , qui ressemblent beaucoup à des compréhensions d'ensemble en mathématiques et à des implémentations similaires dans des langages impératifs tels que Python et JavaScript. Dans leur forme la plus simple, les compréhensions de liste prennent la forme suivante.

```
[ x | x <- someList ]
```

Par exemple

```
[ x | x <- [1..4] ]    -- [1,2,3,4]
```

Les fonctions peuvent aussi être appliquées directement à x:

```
[ f x | x <- someList ]
```

Ceci est équivalent à:

```
map f someList
```

Exemple:

```
[ x+1 | x <- [1..4]]    -- [2,3,4,5]
```

Motifs dans les expressions du générateur

Cependant, `x` dans l'expression du générateur n'est pas seulement variable, mais peut être n'importe quel modèle. En cas de non concordance de modèle, l'élément généré est ignoré et le traitement de la liste continue avec l'élément suivant, agissant ainsi comme un filtre:

```
[x | Just x <- [Just 1, Nothing, Just 3]]     -- [1, 3]
```

Un générateur avec une variable `x` dans son modèle crée une nouvelle étendue contenant toutes les expressions à sa droite, où `x` est défini comme étant l'élément généré.

Cela signifie que les gardes peuvent être codés comme

```
[ x | x <- [1..4], even x] ==
[ x | x <- [1..4], () <- [() | even x]] ==
[ x | x <- [1..4], () <- if even x then [()] else []]
```

Gardes

Une autre caractéristique de la compréhension de liste est celle des gardes, qui servent également de filtres. Les gardes sont des expressions booléennes et apparaissent sur le côté droit de la barre dans une liste de compréhension.

Leur utilisation la plus élémentaire est

```
[x    | p x]   ===   if p x then [x] else []
```

Toute variable utilisée dans un garde doit apparaître à sa gauche dans la compréhension ou être dans la portée. Alors,

```
[ f x | x <- list, pred1 x y, pred2 x]     -- `y` must be defined in outer scope
```

ce qui équivaut à

```
map f (filter pred2 (filter (\x -> pred1 x y) list))          -- or,

-- ($ list) (filter (`pred1` y) >>> filter pred2 >>> map f)

-- list >>= (\x-> [x | pred1 x y]) >>= (\x-> [x | pred2 x]) >>= (\x -> [f x])
```

(l'opérateur `>>=` est `infixl 1` , c'est-à-dire qu'il est associé (est mis entre parenthèses) à gauche). Exemples:

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

[ x^2 + 1 | x <- [1..100], even x ]        -- map (\x -> x^2 + 1) (filter even [1..100])
```

Générateurs imbriqués

Les compréhensions de listes peuvent également dessiner des éléments de plusieurs listes, auquel cas le résultat sera la liste de toutes les combinaisons possibles des deux éléments, comme si les deux listes étaient traitées de manière *imbriquée* . Par exemple,

```
[ (a,b) | a <- [1,2,3], b <- ['a','b'] ]

-- [(1,'a'), (1,'b'), (2,'a'), (2,'b'), (3,'a'), (3,'b')]
```

Compréhensions parallèles

Avec l'extension de langage Parallel List Comprehensions ,

```
[(x,y) | x <- xs | y <- ys]
```

est équivalent à

```
zip xs ys
```

Exemple:

```
[(x,y) | x <- [1,2,3] | y <- [10,20]]

-- [(1,10),(2,20)]
```

Liaisons locales

Les compréhensions de liste peuvent introduire des liaisons locales pour les variables afin de contenir certaines valeurs intermédiaires:

```
[(x,y) | x <- [1..4], let y=x*x+1, even y]    -- [(1,2),(3,10)]
```

Le même effet peut être obtenu avec une astuce,

```
[(x,y) | x <- [1..4], y <- [x*x+1], even y]   -- [(1,2),(3,10)]
```

La `let` dans la liste compréhensions est récursive, comme d' habitude. Mais les liaisons de générateur ne le sont pas, ce qui permet l' *observation* :

```
[x | x <- [1..4], x <- [x*x+1], even x]       -- [2,10]
```

Notation

Toute compréhension de liste peut être en correspondance codé avec la liste monade de `do` notation .

```
[f x | x <- xs]                f  <$> xs         do { x <- xs ; return (f x) }

[f x | f <- fs, x <- xs]       fs <*> xs         do { f <- fs ; x <- xs ; return (f x) }

[y   | x <- xs, y <- f x]      f  =<< xs         do { x <- xs ; y <- f x ; return y }
```

Les gardes peuvent être manipulés en utilisant `Control.Monad.guard` :

```
[x   | x <- xs, even x]                         do { x <- xs ; guard (even x) ; return x }
```

Lire Liste des compréhensions en ligne: https://riptutorial.com/fr/haskell/topic/4970/liste-des-comprehensions

Chapitre 41: Littéraux surchargés

Remarques

Littéraux entiers

est un chiffre **sans** point décimal

par exemple `0` , `1` , `42` , ...

est implicitement appliqué à `fromInteger` qui fait partie de la classe de type `Num` donc il a en effet le type `Num a => a` - c'est-à-dire qu'il peut avoir n'importe quel type qui est une instance de `Num`

Littéraux fractionnaires

est un chiffre **avec** un point décimal

par exemple `0.0` , `-0.1111` , ...

est implicitement appliqué à `fromRational` qui fait partie de la classe de type `Fractional` , il a donc bien le type `a => a` - c'est-à-dire qu'il peut avoir n'importe quel type qui est une instance de `Fractional`

Littéraux de chaîne

Si vous ajoutez l'extension de langue `OverloadedStrings` à *GHC,* vous pouvez avoir la même chose pour les listes `String` qui sont ensuite appliquées à `fromString` partir de la classe de type `Data.String.IsString` .

Ceci est souvent utilisé pour remplacer la `String` avec du `Text` ou `ByteString` .

Liste des littéraux

Les listes peuvent être définies avec la syntaxe littérale `[1, 2, 3]` . Dans GHC 7.8 et au-delà, cela peut également être utilisé pour définir d'autres structures de type liste avec l'extension `OverloadedLists` .

Par défaut, le type de `[]` est:

```
> :t []
[] :: [t]
```

Avec `OverloadedLists` , cela devient:

```
[] :: GHC.Exts.IsList l => l
```

Examples

Nombre entier

Le type du littéral

```
Prelude> :t 1
1 :: Num a => a
```

choisir un type de béton avec des annotations

Vous pouvez spécifier le type tant que le type de cible est `Num` avec une *annotation* :

```
Prelude> 1 :: Int
1
it :: Int
Prelude> 1 :: Double
1.0
it :: Double
Prelude> 1 :: Word
1
it :: Word
```

sinon le compilateur se plaindra

Prelude> 1 :: String

```
<interactive>:
    No instance for (Num String) arising from the literal `1'
    In the expression: 1 :: String
    In an equation for `it': it = 1 :: String
```

Chiffre flottant

Le type du littéral

```
Prelude> :t 1.0
1.0 :: Fractional a => a
```

Choisir un type concret avec des annotations

Vous pouvez spécifier le type avec une *annotation de type* . La seule exigence est que le type doit

avoir une instance `Fractional` .

```
Prelude> 1.0 :: Double
1.0
it :: Double
Prelude> 1.0 :: Data.Ratio.Ratio Int
1 % 1
it :: GHC.Real.Ratio Int
```

sinon le compilateur se plaindra

```
Prelude> 1.0 :: Int
<interactive>:
    No instance for (Fractional Int) arising from the literal `1.0'
    In the expression: 1.0 :: Int
    In an equation for `it': it = 1.0 :: Int
```

Cordes

Le type du littéral

Sans aucune extension, le type d'une chaîne littérale - c'est-à-dire quelque chose entre les guillemets - est juste une chaîne, c'est-à-dire une liste de caractères:

```
Prelude> :t "foo"
"foo" :: [Char]
```

Cependant, lorsque l'extension `OverloadedStrings` est activée, les littéraux de chaîne deviennent polymorphes, similaires aux littéraux numériques :

```
Prelude> :set -XOverloadedStrings
Prelude> :t "foo"
"foo" :: Data.String.IsString t => t
```

Cela nous permet de définir des valeurs de type chaîne sans nécessiter de conversion explicite. Essentiellement, l'extension `OverloadedStrings` ne fait qu'emballer chaque littéral de chaîne dans la fonction de conversion `fromString` générique, donc si le contexte demande, par exemple, le `Text` le plus efficace au lieu de `String` , vous n'avez pas à vous en soucier.

Utiliser des littéraux de chaîne

```
{-# LANGUAGE OverloadedStrings #-}

import Data.Text (Text, pack)
import Data.ByteString (ByteString, pack)


withString :: String
withString = "Hello String"
```

```
-- The following two examples are only allowed with OverloadedStrings

withText :: Text
withText = "Hello Text"      -- instead of: withText = Data.Text.pack "Hello Text"

withBS :: ByteString
withBS = "Hello ByteString"  -- instead of: withBS = Data.ByteString.pack "Hello ByteString"
```

Remarquez comment nous avons pu construire des valeurs de `Text` et `ByteString` de la même manière que nous construisons des valeurs `String` (ou `[Char]`) ordinaires, plutôt que d'utiliser la fonction de `pack` types pour encoder explicitement la chaîne.

Pour plus d'informations sur l'extension de langue `OverloadedStrings` , consultez la documentation de l'extension .

Liste des littéraux

L'extension OverloadedLists de GHC vous permet de construire des structures de données de type liste avec la syntaxe littérale de liste.

Cela vous permet de Data.Map comme ceci:

```
> :set -XOverloadedLists
> import qualified Data.Map as M
> M.lookup "foo" [("foo", 1), ("bar", 2)]
Just 1
```

Au lieu de cela (notez l'utilisation de la M.fromList supplémentaire):

```
> import Data.Map as M
> M.lookup "foo" (M.fromList [("foo", 1), ("bar", 2)])
Just 1
```

Lire Littéraux surchargés en ligne: https://riptutorial.com/fr/haskell/topic/369/litteraux-surcharges

Chapitre 42: Modules

Syntaxe

- Nom du module où - exporte tous les noms déclarés dans ce fichier
- Nom du module (functionOne, Type (..)) où - exporte uniquement les constructeurs functionOne, Type et Type
- import module - Importer tous les noms exportés du module
- importer un module qualifié en tant que MN - importation qualifiée
- import Module (justThisFunction) - Importer uniquement certains noms d'un module
- import Module masquant (functionName, Type) - Importer tous les noms d'un module à l'exception de NomFonction et Type

Remarques

Haskell prend en charge les modules:

- un module peut exporter tout ou un sous-ensemble de ses types et fonctions membres
- un module peut "réexporter" les noms importés d'autres modules

Du côté consommateur d'un module, on peut:

- importer tout ou un sous-ensemble des membres du module
- masquer les importations d'un membre particulier ou d'un ensemble de membres

haskell.org a un excellent chapitre sur la définition des modules.

Examples

Définir votre propre module

Si nous avons un fichier appelé `Business.hs` , nous pouvons définir un module `Business` pouvant être `import` , comme ceci:

```
module Business (
    Person (..), -- ^ Export the Person type and all its constructors and field names
    employees   -- ^ Export the employees function
) where
-- begin types, function definitions, etc
```

Une hiérarchie plus profonde est bien sûr possible; voir l'exemple des noms de modules hiérarchiques .

Exportateurs Constructeurs

Pour exporter le type et tous ses constructeurs, il faut utiliser la syntaxe suivante:

```
module X (Person (..)) where
```

Donc, pour les définitions de niveau supérieur suivantes dans un fichier appelé `People.hs` :

```
data Person = Friend String | Foe deriving (Show, Eq, Ord)

isFoe Foe = True
isFoe _   = False
```

Cette déclaration de module en haut:

```
module People (Person (..)) where
```

ne ferait qu'exporter `Person` et ses constructeurs `Friend` and `Foe` .

Si la liste d'export suivant le mot-clé module est omise, tous les noms liés au niveau supérieur du module seront exportés:

```
module People where
```

exporterait `Person` , ses constructeurs et la fonction `isFoe` .

Importation de membres spécifiques d'un module

Haskell prend en charge l'importation d'un sous-ensemble d'éléments à partir d'un module.

```
import qualified Data.Stream (map) as D
```

importerait uniquement la `map` de `Data.Stream` , et les appels à cette fonction nécessiteraient `D.` .:

```
D.map odd [1..]
```

Sinon, le compilateur essaiera d'utiliser la fonction de `map` `Prelude` .

Cacher les importations

Le prélude définit souvent des fonctions dont les noms sont utilisés ailleurs. Le fait de ne pas cacher ces importations (ou d'utiliser des importations qualifiées en cas de conflit) provoquera des erreurs de compilation.

`Data.Stream` définit les fonctions nommées `map` , `head` et `tail` qui sont normalement incompatibles

avec celles définies dans Prelude. On peut cacher ces importations de Prelude en se `hiding` :

```
import Data.Stream -- everything from Data.Stream
import Prelude hiding (map, head, tail, scan, foldl, foldr, filter, dropWhile, take) -- etc
```

En réalité, cela nécessiterait trop de code pour masquer des conflits de type Prelude, de sorte que vous utiliseriez en fait une importation `qualified` de `Data.Stream` .

Importations admissibles

Lorsque plusieurs modules définissent les mêmes fonctions par leur nom, le compilateur se plaindra. Dans de tels cas (ou pour améliorer la lisibilité), nous pouvons utiliser une importation `qualified` :

```
import qualified Data.Stream as D
```

Maintenant, nous pouvons éviter les erreurs d'ambiguïté du compilateur lorsque nous utilisons la `map` , qui est définie dans `Prelude` et `Data.Stream` :

```
map (== 1) [1,2,3] -- will use Prelude.map
D.map (odd) (fromList [1..]) -- will use Data.Stream.map
```

Il est également possible d'importer un module avec seulement les noms en `import Data.Text as T` qualifiés via l' `import Data.Text as T` , ce qui permet d'avoir `Text` au lieu de `T.Text` etc.

Noms de modules hiérarchiques

Les noms des modules suivent la structure hiérarchique du système de fichiers. Avec la disposition de fichier suivante:

```
Foo/
├── Baz/
│   └── Quux.hs
└── Bar.hs
Foo.hs
Bar.hs
```

les en-têtes de module ressembleraient à ceci:

```
-- file Foo.hs
module Foo where

-- file Bar.hs
module Bar where

-- file Foo/Bar.hs
module Foo.Bar where

-- file Foo/Baz/Quux.hs
module Foo.Baz.Quux where
```

Notez que:

- le nom du module est basé sur le chemin du fichier déclarant le module
- Les dossiers peuvent partager un nom avec un module, ce qui donne une structure de nommage naturellement hiérarchique aux modules.

Lire Modules en ligne: https://riptutorial.com/fr/haskell/topic/5234/modules

Chapitre 43: Monad Transformers

Examples

Un compteur monadique

Un exemple sur la façon de composer le lecteur, l'écrivain et la monade d'état en utilisant des transformateurs monad. Le code source se trouve dans ce référentiel

Nous voulons implémenter un compteur qui incrémente sa valeur par une constante donnée.

Nous commençons par définir certains types et fonctions:

```
newtype Counter = MkCounter {cValue :: Int}
  deriving (Show)

-- | 'inc c n' increments the counter by 'n' units.
inc :: Counter -> Int -> Counter
inc (MkCounter c) n = MkCounter (c + n)
```

Supposons que nous voulions effectuer le calcul suivant en utilisant le compteur:

- mettre le compteur à 0
- définir la constante d'incrément à 3
- incrémenter le compteur 3 fois
- définir la constante d'incrément à 5
- incrémenter le compteur 2 fois

La monade d'état fournit des abstractions pour faire passer l'état. Nous pouvons utiliser la monade d'état et définir notre fonction d'incrément comme un transformateur d'état.

```
-- | CounterS is a monad.
type CounterS = State Counter

-- | Increment the counter by 'n' units.
incS :: Int-> CounterS ()
incS n = modify (\c -> inc c n)
```

Cela nous permet déjà d'exprimer un calcul de manière plus claire et succincte:

```
-- | The computation we want to run, with the state monad.
mComputationS :: CounterS ()
mComputationS = do
  incS 3
  incS 3
  incS 3
  incS 5
  incS 5
```

Mais il faut encore passer l'incrément constant à chaque invocation. Nous aimerions éviter cela.

Ajouter un environnement

Le lecteur monad offre un moyen pratique de faire passer un environnement. Cette monade est utilisée dans la programmation fonctionnelle pour réaliser ce que l'on appelle l' *injection de dépendance* dans le monde OO.

Dans sa version la plus simple, le monad du lecteur nécessite deux types:

- le type de la valeur en cours de lecture (c.-à-d. notre environnement, `r` ci-dessous),
- la valeur renvoyée par le lecteur monad (`a` ci `a` dessous).

  Lecteur ra

Cependant, nous devons également utiliser la monade d'état. Il faut donc utiliser le transformateur `ReaderT` :

```
newtype ReaderT r m a :: * -> (* -> *) -> * -> *
```

En utilisant `ReaderT` , nous pouvons définir notre compteur avec l'environnement et indiquer comme suit:

```
type CounterRS = ReaderT Int CounterS
```

Nous définissons une fonction `incR` qui prend la constante d'incrément de l'environnement (en utilisant `ask`), et pour définir notre fonction d'incrémentation en fonction de notre monade `CounterS` nous utilisons la fonction `lift` (qui appartient à la classe du transformateur monad).

```
-- | Increment the counter by the amount of units specified by the environment.
incR :: CounterRS ()
incR = ask >>= lift . incS
```

En utilisant le lecteur monad on peut définir notre calcul comme suit:

```
-- | The computation we want to run, using reader and state monads.
mComputationRS :: CounterRS ()
mComputationRS = do
  local (const 3) $ do
    incR
    incR
    incR
    local (const 5) $ do
      incR
      incR
```

Les exigences ont changé: nous avons

besoin de journalisation!

Supposons maintenant que nous souhaitons ajouter la journalisation à notre calcul, afin que nous puissions voir l'évolution de notre compteur dans le temps.

Nous avons également une monade pour effectuer cette tâche, l' écrivain monad . Comme avec le lecteur monad, puisque nous les composons, nous devons utiliser le transformateur monad du lecteur:

```
newtype WriterT w m a :: * -> (* -> *) -> * -> *
```

Ici , `w` représente le type de sortie pour accumuler (qui doit être un monoid, qui nous permettent d'accumuler cette valeur), `m` est la monade intérieure, et `a` type de calcul.

Nous pouvons alors définir notre compteur avec logging, environment et state comme suit:

```
type CounterWRS = WriterT [Int] CounterRS
```

Et en utilisant `lift` nous pouvons définir la version de la fonction d'incrément qui enregistre la valeur du compteur après chaque incrément:

```
incW :: CounterWRS ()
incW = lift incR >> get >>= tell . (:[]) . cValue
```

Maintenant, le calcul qui contient la journalisation peut être écrit comme suit:

```
mComputationWRS :: CounterWRS ()
mComputationWRS = do
  local (const 3) $ do
    incW
    incW
    incW
    local (const 5) $ do
      incW
      incW
```

Faire tout en une fois

Cet exemple a pour but de montrer les transformateurs monad au travail. Cependant, nous pouvons obtenir le même effet en composant tous les aspects (environnement, état et journalisation) en une seule opération d'incrémentation.

Pour ce faire, nous utilisons des contraintes de type:

```
inc' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
inc' = ask >>= modify . (flip inc) >> get >>= tell . (:[]) . cValue
```

Nous arrivons ici à une solution qui fonctionnera pour n'importe quelle monade répondant aux contraintes ci-dessus. La fonction de calcul est définie avec le type:

```
mComputation' :: (MonadReader Int m, MonadState Counter m, MonadWriter [Int] m) => m ()
```

puisque dans son corps, nous utilisons inc.

Nous pourrions exécuter ce calcul, dans la `ghci` REPL par exemple, comme suit:

```
runState ( runReaderT ( runWriterT mComputation' ) 15 )  (MkCounter 0)
```

Lire Monad Transformers en ligne: https://riptutorial.com/fr/haskell/topic/7752/monad-transformers

Chapitre 44: Monades

Introduction

Une monade est un type de données d'actions composables. `Monad` est la classe des constructeurs de types dont les valeurs représentent de telles actions. Peut-être `IO` est le plus reconnaissable d' un: une valeur de `IO a` une « recette pour récupérer une `a` valeur du monde réel ».

Nous disons qu'un constructeur de type `m` (tel que `[]` ou `Maybe`) *forme une monade* s'il existe une `instance Monad m` satisfaisant certaines lois sur la composition des actions. On peut alors raisonner sur `ma` comme une "action dont le résultat a tapé `a` ".

Examples

La monade peut-être

`Maybe` - `Maybe` est-il utilisé pour représenter des valeurs éventuellement vides - similaires à `null` dans les autres langues. Habituellement, il est utilisé comme type de sortie des fonctions qui peuvent échouer d'une certaine manière.

Considérons la fonction suivante:

```
halve :: Int -> Maybe Int
halve x
  | even x = Just (x `div` 2)
  | odd x  = Nothing
```

Pensez à `halve` de `halve` en tant qu'action, en fonction d'un `Int` , qui tente de réduire de moitié l'entier, en cas d'échec s'il est impair.

Comment `halve` nous par un entier trois fois?

```
takeOneEighth :: Int -> Maybe Int            -- (after you read the 'do' sub-section:)
takeOneEighth x =
  case halve x of                               --  do {
    Nothing -> Nothing
    Just oneHalf ->                             --     oneHalf    <- halve x
      case halve oneHalf of
        Nothing -> Nothing
        Just oneQuarter ->                      --     oneQuarter <- halve oneHalf
          case halve oneQuarter of
            Nothing -> Nothing                  --     oneEighth  <- halve oneQuarter
            Just oneEighth ->
              Just oneEighth                    --     return oneEighth }
```

- `takeOneEighth` est une *séquence* de trois étapes de `halve` `takeOneEighth` `halve` .
- Si une `halve` étape échoue, nous voulons que toute la composition `takeOneEighth` échoue.
- Si une `halve` étape réussit, nous voulons que son résultat soit reporté.

```
instance Monad Maybe where
  -- (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
  Nothing >>= f  = Nothing                          -- infixl 1 >>=
  Just x  >>= f  = Just (f x)                       -- also, f =<< m = m >>= f

  -- return :: a -> Maybe a
  return x       = Just x
```

et maintenant nous pouvons écrire:

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth x = halve x >>= halve >>= halve             -- or,
    -- return x >>= halve >>= halve >>= halve             -- which is parsed as
    -- (((return x) >>= halve) >>= halve) >>= halve       -- which can also be written as
    -- (halve =<<) . (halve =<<) . (halve =<<) $ return x    -- or, equivalently, as
    --  halve <=<     halve <=<     halve      $        x
```

La composition de Kleisli `<=<` est définie comme `(g <=< f) x = g =<< fx` ou, de manière équivalente, comme `(f >=> g) x = fx >>= g` . Avec elle la définition ci-dessus devient juste

```
takeOneEighth :: Int -> Maybe Int
takeOneEighth = halve <=< halve <=< halve              -- infixr 1 <=<
        -- or, equivalently,
        --      halve >=> halve >=> halve              -- infixr 1 >=>
```

Chaque monade doit obéir à trois lois de la monade, c'est-à-dire à tout type qui est une instance de la classe `Monad` :

```
1.  return x >>= f  =  f x
2.    m >>= return  =  m
3. (m >>= g) >>= h  =  m >>= (\y -> g y >>= h)
```

où `m` est une monade, `f` a le type `a -> mb` et `g` a le type `b -> mc` .

Ou de manière équivalente, en utilisant l'opérateur de composition `>=>` Kleisli défini ci-dessus:

```
1.    return >=> g  =  g                  -- do { y <- return x ; g y } == g x
2.    f >=> return  =  f                  -- do { y <- f x ; return y } == f x
3. (f >=> g) >=> h  =  f >=> (g >=> h)    -- do { z <- do { y <- f x; g y } ; h z }
                                          --  == do { y <- f x ; do { z <- g y; h z } }
```

L'obéissance à ces lois rend la raison de la monade beaucoup plus facile à comprendre, car elle garantit que l'utilisation de fonctions monadiques et leur composition se comportent de manière raisonnable, comme d'autres monades.

Vérifions si la monade `Maybe` obéit aux trois lois de la monade.

1. **La loi d'identité de gauche -** `return x >>= f = fx`

```
return z >>= f
= (Just z) >>= f
```

```
= f z
```

2. **La bonne loi d'identité -** `m >>= return = m`

- `Just` constructeur de données

```
Just z >>= return
= return z
= Just z
```

- Constructeur de données `Nothing`

```
Nothing >>= return
= Nothing
```

3. **La loi d'associativité -** `(m >>= f) >>= g = m >>= (\x -> fx >>= g)`

- `Just` constructeur de données

```
-- Left-hand side
((Just z) >>= f) >>= g
= f z >>= g

-- Right-hand side
(Just z) >>= (\x -> f x >>= g)
(\x -> f x >>= g) z
= f z >>= g
```

- Constructeur de données `Nothing`

```
-- Left-hand side
(Nothing >>= f) >>= g
= Nothing >>= g
= Nothing

-- Right-hand side
Nothing >>= (\x -> f x >>= g)
= Nothing
```

IO monade

Il n'y a aucun moyen d'obtenir une valeur de type `a` sur une expression de type `IO a` et il ne devrait pas y en avoir. C'est en fait une grande partie de la raison pour laquelle les monades sont utilisées pour modéliser les `IO` .

Une expression de type `IO a` peut être considérée comme représentant une action pouvant interagir avec le monde réel et, si elle est exécutée, entraînerait quelque chose de type `a` . Par exemple, la fonction `getLine :: IO String` du prélude ne signifie pas que sous `getLine` il existe une chaîne spécifique que je peux extraire - cela signifie que `getLine` représente l'action `getLine` obtenir une ligne à partir d'une entrée standard.

Sans surprise, `main :: IO ()` depuis un programme Haskell représente un calcul / une action qui interagit avec le monde réel.

Les choses que vous *pouvez* faire pour les expressions de type `IO a` car `IO` est une monade:

- Séquence de deux actions en utilisant `(>>)` pour produire une nouvelle action qui exécute la première action, rejette la valeur produite, puis exécute la seconde action.

```
-- print the lines "Hello" then "World" to stdout
putStrLn "Hello" >> putStrLn "World"
```

- Parfois, vous ne voulez pas ignorer la valeur qui a été produite lors de la première action. En fait, vous souhaitez qu'elle soit ajoutée à une seconde action. Pour cela, nous avons `>>=` . Pour `IO` , il a le type `(>>=) :: IO a -> (a -> IO b) -> IO b` .

```
-- get a line from stdin and print it back out
getLine >>= putStrLn
```

- Prenez une valeur normale et convertissez-la en une action qui retourne immédiatement la valeur que vous lui avez donnée. Cette fonction est moins utile jusqu'à ce que vous commenciez à utiliser la notation `do` .

```
-- make an action that just returns 5
return 5
```

Plus d'informations sur le wiki Haskell sur la monade IO ici .

Liste Monad

Les listes forment une monade. Ils ont une instanciation de monade équivalente à celle-ci:

```
instance Monad [] where
  return x = [x]
  xs >>= f = concat (map f xs)
```

Nous pouvons les utiliser pour émuler le non-déterminisme dans nos calculs. Lorsque nous utilisons `xs >>= f` , la fonction `f :: a -> [b]` est mappée sur la liste `xs` , en obtenant une liste de résultats de chaque application de `f` sur chaque élément de `xs` , et toutes les listes de les résultats sont ensuite concaténés dans une liste de tous les résultats. A titre d'exemple, nous calculons une somme de deux nombres non déterministes en utilisant la **notation do** , la somme étant représentée par la liste des sommes de toutes les paires d'entiers de deux listes, chaque liste représentant toutes les valeurs possibles d'un nombre non déterministe:

```
sumnd xs ys = do
  x <- xs
  y <- ys
  return (x + y)
```

Ou, de manière équivalente, en utilisant `liftM2` dans `Control.Monad` :

```
sumnd = liftM2 (+)
```

on obtient:

```
> sumnd [1,2,3] [0,10]
[1,11,2,12,3,13]
```

Monad comme sous-classe d'application

A partir de GHC 7.10, `Applicative` est une super-classe de `Monad` (c'est-à-dire que chaque type qui est une `Monad` doit également être un `Applicative`). Toutes les méthodes de `Applicative` (`pure` , `<*>`) peuvent être implémentées en termes de méthodes de `Monad` (`return` , `>>=`).

Il est évident que le `pure` et le `return` servent des objectifs équivalents, donc `pure = return` . La définition de `<*>` est trop claire:

```
mf <*> mx = do { f <- mf; x <- mx; return (f x) }
       -- = mf >>= (\f -> mx >>= (\x -> return (f x)))
       -- = [r   | f <- mf, x <- mx, r <- return (f x)]   -- with MonadComprehensions
       -- = [f x | f <- mf, x <- mx]
```

Cette fonction est définie comme `ap` dans les bibliothèques standard.

Ainsi, si vous avez déjà défini une instance de `Monad` pour un type, vous pouvez effectivement obtenir une instance de `Applicative` "gratuitement" en définissant

```
instance Applicative < type > where
    pure  = return
    (<*>) = ap
```

Comme pour les lois monades, ces équivalences ne sont pas appliquées, mais les développeurs doivent s'assurer qu'elles sont toujours respectées.

Pas de moyen général d'extraire de la valeur d'un calcul monadique

Vous pouvez envelopper des valeurs dans des actions et transférer le résultat d'un calcul dans un autre:

```
return :: Monad m => a -> m a
(>>=)  :: Monad m => m a -> (a -> m b) -> m b
```

Cependant, la définition d'une Monade ne garantit pas l'existence d'une fonction de type `Monad m => ma -> a` .

Cela signifie qu'il n'existe en général **aucun moyen d'extraire une valeur d'un calcul** (par exemple, le «déplier»). C'est le cas pour de nombreuses instances:

```
extract :: Maybe a -> a
extract (Just x) = x          -- Sure, this works, but...
```

```
extract Nothing  = undefined  -- We can't extract a value from failure.
```

Plus précisément, il n'y a pas de fonction `IO a -> a` , qui confond souvent les débutants; voir cet exemple .

do-notation

`do` -notation est le sucre syntaxique pour les monades. Voici les règles:

```
do x <- mx                              do x <- mx
   y <- my       is equivalent to           do y <- my
   ...                                         ...
```

```
do let a = b                            let a = b in
   ...           is equivalent to         do ...
```

```
do m                                    m >> (
   e             is equivalent to         e)
```

```
do x <- m                               m >>= (\x ->
   e             is equivalent to         e)
```

```
do m             is equivalent to       m
```

Par exemple, ces définitions sont équivalentes:

```
example :: IO Integer
example =
  putStrLn "What's your name?" >> (
    getLine >>= (\name ->
      putStrLn ("Hello, " ++ name ++ ".") >> (
        putStrLn "What should we return?" >> (
          getLine >>= (\line ->
            let n = (read line :: Integer) in
              return (n + n))))))
```

```
example :: IO Integer
example = do
  putStrLn "What's your name?"
  name <- getLine
  putStrLn ("Hello, " ++ name ++ ".")
  putStrLn "What should we return?"
  line <- getLine
  let n = (read line :: Integer)
  return (n + n)
```

Définition de Monade

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

La fonction la plus importante pour traiter les monades est l' **opérateur de liaison** `>>=` :

```
(>>=) :: m a -> (a -> m b) -> m b
```

- Pensez à `ma` comme *"une action avec* `a` *résultat"* .
- Considérez `a -> mb` comme *«une action (dépendant d'* `a` *paramètre) avec un résultat* `b` *.»* .

`>>=` **séquence deux actions en canalisant le résultat de la première action à la seconde.**

L'autre fonction définie par `Monad` est:

```
return :: a -> m a
```

Son nom est regrettable: ce `return` n'a rien à voir avec le mot-clé de `return` trouvé dans les langages de programmation impératifs.

`return x` **est l'action triviale donnant** `x` **comme résultat.** (C'est trivial dans le sens suivant :)

```
return x >>= f       ≡  f x     --  “left identity” monad law
       x >>= return  ≡  x       -- “right identity” monad law
```

Lire Monades en ligne: https://riptutorial.com/fr/haskell/topic/2968/monades

Chapitre 45: Monades communes comme monades gratuites

Examples

Vide libre ~~ Identité

Donné

```
data Empty a
```

nous avons

```
data Free Empty a
     = Pure a
-- the Free constructor is impossible!
```

qui est isomorphe à

```
data Identity a
     = Identity a
```

Identité Libre ~~ (Nat,) ~~ Écrivain Nat

Donné

```
data Identity a = Identity a
```

nous avons

```
data Free Identity a
     = Pure a
     | Free (Identity (Free Identity a))
```

qui est isomorphe à

```
data Deferred a
     = Now a
     | Later (Deferred a)
```

ou de manière équivalente (si vous promettez d'évaluer l'élément fst en premier) `(Nat, a)` , alias `Writer Nat a` , avec

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
```

Gratuit Peut-être ~~ MaybeT (écrivain Nat)

Donné

```
data Maybe a = Just a
             | Nothing
```

nous avons

```
data Free Maybe a
     = Pure a
     | Free (Just (Free Maybe a))
     | Free Nothing
```

ce qui équivaut à

```
data Hopes a
     = Confirmed a
     | Possible (Hopes a)
     | Failed
```

ou de manière équivalente (si vous vous engagez à évaluer l'élément fst en premier) `(Nat, Maybe a)` , alias `MaybeT (Writer Nat) a` with

```
data Nat = Z | S Nat
data Writer Nat a = Writer Nat a
data MaybeT (Writer Nat) a = MaybeT (Nat, Maybe a)
```

Free (Writer w) ~~ Writer [w]

Donné

```
data Writer w a = Writer w a
```

nous avons

```
data Free (Writer w) a
     = Pure a
     | Free (Writer w (Free (Writer w) a))
```

qui est isomorphe à

```
data ProgLog w a
     = Done a
     | After w (ProgLog w a)
```

ou, de manière équivalente (si vous vous engagez à évaluer le journal en premier), `Writer [w] a` .

Gratuit (Const c) ~~ Soit c

Donné

```
data Const c a = Const c
```

nous avons

```
data Free (Const c) a
     = Pure a
     | Free (Const c)
```

qui est isomorphe à

```
data Either c a
     = Right a
     | Left c
```

Gratuit (Reader x) ~~ Reader (Stream x)

Donné

```
data Reader x a = Reader (x -> a)
```

nous avons

```
data Free (Reader x) a
     = Pure a
     | Free (x -> Free (Reader x) a)
```

qui est isomorphe à

```
data Demand x a
     = Satisfied a
     | Hungry (x -> Demand x a)
```

ou équivalent `Stream x -> a` avec

```
data Stream x = Stream x (Stream x)
```

Lire Monades communes comme monades gratuites en ligne:
https://riptutorial.com/fr/haskell/topic/8256/monades-communes-comme-monades-gratuites

Chapitre 46: Monades gratuites

Examples

Monades gratuites divisent les calculs monadiques en structures de données et interprètes

Par exemple, un calcul impliquant des commandes à lire et à écrire à l'invite:

Nous décrivons tout d'abord les "commandes" de notre calcul en tant que type de données Functor

```
{-# LANGUAGE DeriveFunctor #-}

data TeletypeF next
    = PrintLine String next
    | ReadLine (String -> next)
    deriving Functor
```

Ensuite, nous utilisons `Free` pour créer "Free Monad over `TeletypeF` " et créer des opérations de base.

```
import Control.Monad.Free (Free, liftF, iterM)

type Teletype = Free TeletypeF

printLine :: String -> Teletype ()
printLine str = liftF (PrintLine str ())

readLine :: Teletype String
readLine = liftF (ReadLine id)
```

Comme `Free f` est une `Monad` lorsque `f` est un `Functor` , nous pouvons utiliser les combinateurs `Monad` standard (y compris la notation `do`) pour construire des calculs `Teletype` .

```
import Control.Monad -- we can use the standard combinators

echo :: Teletype ()
echo = readLine >>= printLine

mockingbird :: Teletype a
mockingbird = forever echo
```

Enfin, nous écrivons un "interprète" qui transforme les valeurs `Teletype a` en quelque chose que nous savons utiliser avec `IO a`

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype = foldFree run where
  run :: TeletypeF a -> IO a
  run (PrintLine str x) = putStrLn *> return x
```

```
  run (ReadLine f) = fmap f getLine
```

Que nous pouvons utiliser pour "exécuter" le `Teletype a` calcul en `IO`

```
> interpretTeletype mockingbird
hello
hello
goodbye
goodbye
this will go on forever
this will go on forever
```

Les Monades gratuites sont comme des points fixes

Comparez la définition de `Free` à celle de `Fix` :

```
data Free f a = Return a
              | Free (f (Free f a))

newtype Fix f = Fix { unFix :: f (Fix f) }
```

En particulier, comparez le type du constructeur `Free` avec le type du constructeur `Fix` . `Free` couches `Free` montent un foncteur comme `Fix` , sauf que `Free` a un `Return a` supplémentaire.

Comment foldFree et iterM fonctionnent-ils?

Il y a quelques fonctions pour aider à décomposer `Free` calculs `Free` en les interprétant dans une autre monade `m` : `iterM :: (Functor f, Monad m) => (f (ma) -> ma) -> (Free fa -> ma)` et `foldFree :: Monad m => (forall x. fx -> mx) -> (Free fa -> ma)` . Que font-ils?

Tout d'abord, voyons ce qu'il faudrait pour `IO` manuellement `Teletype a` fonction de `Teletype a` dans `IO` . On peut voir `Free fa` comme étant défini

```
data Free f a
    = Pure a
    | Free (f (Free f a))
```

Le cas `Pure` est facile:

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = _
```

Maintenant, comment interpréter un calcul de `Teletype` construit avec le constructeur `Free` ? Nous aimerions obtenir une valeur de type `IO a` en examinant `teletypeF :: TeletypeF (Teletype a)` . Pour commencer, nous allons écrire une fonction `runIO :: TeletypeF a -> IO a` qui mappe une seule couche de monad libre sur une action `IO` :

```
runIO :: TeletypeF a -> IO a
runIO (PrintLine msg x) = putStrLn msg *> return x
```

```
runIO (ReadLine k) = fmap k getLine
```

Maintenant, nous pouvons utiliser `runIO` pour remplir le reste de `interpretTeletype` . Rappelons que `teletypeF :: TeletypeF (Teletype a)` est une couche du foncteur `TeletypeF` qui contient le reste du calcul `Free` . Nous utiliserons `runIO` pour interpréter la couche la plus externe (nous avons donc `runIO teletypeF :: IO (Teletype a)`) et utilisons ensuite le combinateur `>>=` monade `IO` pour interpréter le `Teletype a` renvoyé `Teletype a` .

```
interpretTeletype :: Teletype a -> IO a
interpretTeletype (Pure x) = return x
interpretTeletype (Free teletypeF) = runIO teletypeF >>= interpretTeletype
```

La définition de `foldFree` est juste celle de `interpretTeletype` , sauf que la fonction `runIO` a été `runIO` . Par conséquent, `foldFree` fonctionne indépendamment de tout foncteur de base particulier et de toute monade cible.

```
foldFree :: Monad m => (forall x. f x -> m x) -> Free f a -> m a
foldFree eta (Pure x) = return x
foldFree eta (Free fa) = eta fa >>= foldFree eta
```

`foldFree` a un type rank-2: `eta` est une transformation naturelle. On aurait pu donner à `foldFree` un type de `Monad m => (f (Free fa) -> m (Free fa)) -> Free fa -> ma` , mais cela donne à `eta` la liberté d'inspecter le calcul `Free` dans le calque `f` . Donner à `foldFree` ce type plus restrictif garantit que `eta` ne peut traiter qu'un seul calque à la fois.

`iterM` donne à la fonction de pliage la possibilité d'examiner le sous-calcul. Le résultat (monadique) de l'itération précédente est disponible pour le suivant, à l'intérieur du paramètre de `f` . `iterM` est analogue à un *paramorphisme,* alors que `foldFree` est comme un *catamorphisme* .

```
iterM :: (Monad m, Functor f) => (f (m a) -> m a) -> Free f a -> m a
iterM phi (Pure x) = return x
iterM phi (Free fa) = phi (fmap (iterM phi) fa)
```

La monade plus libre

Il existe une autre formulation de la monade libre appelée la monade Freer (ou Prompt ou Operational). Le Monad Freer ne nécessite pas d'instance Functor pour son jeu d'instructions sous-jacent, et sa structure ressemble à celle d'une monade libre standard.

La monade Freer représente les programmes sous la forme d'une séquence d' *instructions* atomiques appartenant au jeu d'instructions `i :: * -> *` . Chaque instruction utilise son paramètre pour déclarer son type de retour. Par exemple, les instructions de base pour la monade d' `State` sont les suivantes:

```
data StateI s a where
    Get :: StateI s s  -- the Get instruction returns a value of type 's'
    Put :: s -> StateI s ()  -- the Put instruction contains an 's' as an argument and returns
()
```

Le séquençage de ces instructions s'effectue avec le constructeur `:>>=` . `:>>=` prend une seule instruction renvoyant un `a` et le ajoute au reste du programme, introduisant sa valeur de retour dans la suite. En d'autres termes, étant donné une instruction renvoyant un `a` , et une fonction pour transformer `a` en un programme renvoyant un `b` `:>>=` produira un programme renvoyant un `b` .

```
data Freer i a where
    Return :: a -> Freer i a
    (:>>=) :: i a -> (a -> Freer i b) -> Freer i b
```

Notez que `a` est quantifié existentiellement dans le constructeur `:>>=` . La seule manière pour un interprète d'apprendre ce qu'est `a` est de faire correspondre un motif sur le GADT `i` .

> **À part** : Le lemme co-Yoneda nous dit que `Freer` est isomorphe à `Free` . Rappelons la définition du foncteur `CoYoneda` :
>
> ```
> data CoYoneda i b where
>   CoYoneda :: i a -> (a -> b) -> CoYoneda i b
> ```
>
> `Freer i` est équivalent à `Free (CoYoneda i)` . Si vous prenez les constructeurs de `Free` et définissez `f ~ CoYoneda i` , vous obtenez:
>
> ```
> Pure :: a -> Free (CoYoneda i) a
> Free :: CoYoneda i (Free (CoYoneda i) b) -> Free (CoYonda i) b ~
>         i a -> (a -> Free (CoYoneda i) b) -> Free (CoYoneda i) b
> ```
>
> à partir de laquelle nous pouvons récupérer `Freer i` constructeurs d » en tout réglage `Freer i ~ Free (CoYoneda i)` .

Parce que `CoYoneda i` est un `Functor` pour tout `i` , `Freer` est une `Monad` pour tout `i` , même si `i` ne `i` pas un `Functor` .

```
instance Monad (Freer i) where
    return = Return
    Return x >>= f = f x
    (i :>>= g) >>= f = i :>>= fmap (>>= f) g  -- using `(->) r`'s instance of Functor, so fmap
= (.)
```

Des interpréteurs peuvent être construits pour `Freer` en mappant des instructions à certains gestionnaires de monade.

```
foldFreer :: Monad m => (forall x. i x -> m x) -> Freer i a -> m a
foldFreer eta (Return x) = return x
foldFreer eta (i :>>= f) = eta i >>= (foldFreer eta . f)
```

Par exemple, nous pouvons interpréter la monade `Freer (StateI s)` utilisant la monade de l' `State s` normal comme gestionnaire:

```
runFreerState :: Freer (StateI s) a -> s -> (a, s)
runFreerState = State.runState . foldFreer toState
    where toState :: StateI s a -> State s a
```

```
        toState Get = State.get
        toState (Put x) = State.put x
```

Lire Monades gratuites en ligne: https://riptutorial.com/fr/haskell/topic/1290/monades-gratuites

Chapitre 47: Monoïde

Examples

Une instance de Monoid pour les listes

```
instance Monoid [a] where
    mempty  = []
    mappend = (++)
```

Vérification des lois du `Monoid` pour cette instance:

```
mempty `mappend` x = x   <->   [] ++ xs = xs  -- prepending an empty list is a no-op

x `mappend` mempty = x   <->   xs ++ [] = xs  -- appending an empty list is a no-op

x `mappend` (y `mappend` z) = (x `mappend` y) `mappend` z
    <->
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs           -- appending lists is associative
```

Réduire une liste de monoïdes en une seule valeur

`mconcat :: [a] -> a` est `mconcat :: [a] -> a` autre méthode de la classe de `Monoid` :

```
ghci> mconcat [Sum 1, Sum 2, Sum 3]
Sum {getSum = 6}
ghci> mconcat ["concat", "enate"]
"concatenate"
```

Sa définition par défaut est `mconcat = foldr mappend mempty` .

Monoïdes Numériques

Les nombres sont monoïdaux de deux manières: *addition* avec 0 comme unité et *multiplication* avec 1 comme unité. Les deux sont également valables et utiles dans différentes circonstances. Donc, plutôt que de choisir une instance préférée pour les nombres, il y a deux `newtypes` , `Sum` et `Product` pour les marquer pour les différentes fonctionnalités.

```
newtype Sum n = Sum { getSum :: n }

instance Num n => Monoid (Sum n) where
    mempty = Sum 0
    Sum x `mappend` Sum y = Sum (x + y)

newtype Product n = Product { getProduct :: n }

instance Num n => Monoid (Product n) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

Cela permet effectivement au développeur de choisir la fonctionnalité à utiliser en encapsulant la valeur dans le `newtype` approprié.

```
Sum 3     <> Sum 5     == Sum 8
Product 3 <> Product 5 == Product 15
```

Une instance de Monoid pour ()

`()` est un `Monoid` . Comme il n'y a qu'une seule valeur de type `()` , il n'y a qu'une seule chose que `mempty` et `mappend` pourraient faire:

```
instance Monoid () where
    mempty = ()
    () `mappend` () = ()
```

Lire Monoïde en ligne: https://riptutorial.com/fr/haskell/topic/2211/monoide

Chapitre 48: Opérateurs Infix

Remarques

La plupart des fonctions Haskell sont appelées avec le nom de la fonction suivi par des arguments (notation préfixe). Pour les fonctions qui acceptent deux arguments comme (+), il est parfois judicieux de fournir un argument avant et après la fonction (infix).

Examples

Prélude

Logique

`&&` est logique ET, `||` est logique OU.

`==` est égalité, `/=` non-égalité, `<` / `<=` moindre et `>` / `>=` opérateurs plus grands.

Opérateurs arithmétiques

Les opérateurs numériques `+` , `-` et `/` se comportent largement comme prévu. (Division ne fonctionne que sur des nombres fractionnaires pour éviter les problèmes d'arrondi - la division entière doit être effectuée avec `quot` ou `div`). Les trois opérateurs d'exponentiation de Haskell sont plus inhabituels:

- `^` prend une base de n'importe quel type de nombre à une puissance intégrale non négative. Cela fonctionne simplement par une multiplication itérée (rapide). Par exemple

  ```
  4^5  ≡  (4*4)*(4*4)*4
  ```

- `^^` fait de même dans le cas positif, mais fonctionne également pour les exposants négatifs. Par exemple

  ```
  3^^(-2)  ≡  1 / (2*2)
  ```

  Contrairement à `^` , ceci nécessite un type de base fractionnaire (c.-à-d. `4^^5 :: Int` ne fonctionnera pas, seulement `4^5 :: Int` ou `4^^5 :: Rational`).

- `**` implémente une exponentiation en nombres réels. Cela fonctionne pour des arguments très généraux, mais est plus coûteux que `^` ou `^^` , et engendre généralement de petites erreurs en virgule flottante.

```
2**pi  ≡  exp (pi * log 2)
```

Des listes

Il y a deux opérateurs de concaténation:

- `:` (prononcé contre) ajoute un seul argument avant une liste. Cet opérateur est en réalité un constructeur et peut donc aussi être utilisé pour *créer une correspondance* («construction inverse») dans une liste.
- `++` concatène des listes entières.

```
[1,2] ++ [3,4]  ≡  1 : 2 : [3,4]  ≡  1 : [2,3,4]  ≡  [1,2,3,4]
```

`!!` est un opérateur d'indexation.

```
[0, 10, 20, 30, 40] !! 3  ≡  30
```

Notez que les listes d'indexation sont inefficaces (complexité *O* (*n*) au lieu de *O* (1) pour les tableaux ou *O* (log *n*) pour les cartes); Il est généralement préférable dans Haskell de déconstruire les listes en repliant la correspondance de motifs au lieu de l'indexation.

Flux de contrôle

- `$` est un opérateur d'application de fonction.

  ```
  f $ x  ≡  f x
         ≡  f(x)  -- disapproved style
  ```

  Cet opérateur est principalement utilisé pour éviter les parenthèses. Il a aussi une version stricte `$!` , qui force l'évaluation de l'argument avant d'appliquer la fonction.

- `.` compose des fonctions.

  ```
  (f . g) x  ≡  f (g x)  ≡  f $ g x
  ```

- `>>` séquences d'actions monadiques. Par exemple, `writeFile "foo.txt" "bla" >> putStrLn "Done."` va d'abord écrire dans un fichier, puis imprimer un message à l'écran.
- `>>=` fait de même, tout en acceptant qu'un argument soit transmis de la première action à la suivante. `readLn >>= \x -> print (x^2)` attend que l'utilisateur saisisse un nombre, puis affiche le carré de ce numéro sur l'écran.

Opérateurs personnalisés

Dans Haskell, vous pouvez définir n'importe quel opérateur d'infixe que vous aimez. Par exemple, je pourrais définir l'opérateur d'enveloppes de liste comme

```
(>+<) :: [a] -> [a] -> [a]
env >+< l = env ++ l ++ env

GHCi> "**">+<"emphasis"
"**emphasis**"
```

Vous devriez toujours donner à ces opérateurs une déclaration de fixité , comme

```
infixr 5 >+<
```

(ce qui voudrait dire `>+<` lie aussi étroitement que `++` et `:` do).

Recherche d'informations sur les opérateurs infixes

Comme les infixes sont si courants chez Haskell, vous devrez régulièrement rechercher leur signature, etc. Heureusement, c'est aussi simple que pour toute autre fonction:

- Les moteurs de recherche Haskell Hayoo et Hoogle peuvent être utilisés pour les opérateurs infixes, comme pour tout ce qui est défini dans une bibliothèque.
- Dans GHCi ou IHaskell, vous pouvez utiliser les directives `:i` et `:t` (**i** nfo et **t** ype) pour apprendre les propriétés de base d'un opérateur. Par exemple,

  ```
  Prelude> :i +
  class Num a where
    (+) :: a -> a -> a
    ...
        -- Defined in ‘GHC.Num’
  infixl 6 +
  Prelude> :i ^^
  (^^) :: (Fractional a, Integral b) => a -> b -> a
        -- Defined in ‘GHC.Real’
  infixr 8 ^^
  ```

  Cela me dit que `^^` se lie plus étroitement que `+` , les deux prennent des types numériques comme éléments, mais `^^` exige que l'exposant soit intégral et la base fractionnaire.
  Le moins verbeux `:t` nécessite l'opérateur entre parenthèses, comme

  ```
  Prelude> :t (==)
  (==) :: Eq a => a -> a -> Bool
  ```

Lire Opérateurs Infix en ligne: https://riptutorial.com/fr/haskell/topic/6792/operateurs-infix

Chapitre 49: Optimisation

Examples

Compiler votre programme pour le profilage

Le compilateur GHC prend en charge la compilation des annotations de profilage.

L'utilisation des `-prof` et `-fprof-auto` lors de la compilation ajoutera un support à votre fichier binaire pour le profilage des indicateurs à utiliser lors de l'exécution.

Supposons que nous ayons ce programme:

```
main = print (fib 30)
fib n = if n < 2 then 1 else fib (n-1) + fib (n-2)
```

Compilé comme ça:

```
ghc -prof -fprof-auto -rtsopts Main.hs
```

Puis l'a exécuté avec les options du système d'exécution pour le profilage:

```
./Main +RTS -p
```

Nous verrons un fichier `main.prof` créé après exécution (une fois le programme terminé), ce qui nous fournira toutes sortes d'informations de profilage telles que les centres de coûts, ce qui nous donnera une ventilation des coûts associés à l'exécution des différentes parties du code. :

```
    Wed Oct 12 16:14 2011 Time and Allocation Profiling Report  (Final)

           Main +RTS -p -RTS

        total time  =        0.68 secs   (34 ticks @ 20 ms)
        total alloc = 204,677,844 bytes  (excludes profiling overheads)

COST CENTRE MODULE  %time %alloc

fib         Main    100.0  100.0


                                                 individual     inherited
COST CENTRE MODULE                  no.     entries  %time %alloc   %time %alloc

MAIN        MAIN                    102           0    0.0    0.0   100.0  100.0
 CAF        GHC.IO.Handle.FD        128           0    0.0    0.0     0.0    0.0
 CAF        GHC.IO.Encoding.Iconv   120           0    0.0    0.0     0.0    0.0
 CAF        GHC.Conc.Signal         110           0    0.0    0.0     0.0    0.0
 CAF        Main                    108           0    0.0    0.0   100.0  100.0
  main      Main                    204           1    0.0    0.0   100.0  100.0
   fib      Main                    205     2692537  100.0  100.0   100.0  100.0
```

Centres de coûts

Les centres de coûts sont des annotations sur un programme Haskell qui peuvent être ajoutées automatiquement par le compilateur GHC - en utilisant `-fprot-auto` - ou par un programmeur utilisant `{-# SCC "name" #-} <expression>` , où "name" est n'importe quel nom que vous souhaitez et `<expression>` est une expression Haskell valide:

```
-- Main.hs
main :: IO ()
main = do let l = [1..9999999]
          print $ {-# SCC "print_list" #-} (length l)
```

Compiler avec `-fprof` et exécuter avec `+RTS -p` par exemple `ghc -prof -rtsopts Main.hs && ./Main.hs +RTS -p` produirait `Main.prof` une fois le programme `Main.prof` .

Lire Optimisation en ligne: https://riptutorial.com/fr/haskell/topic/4342/optimisation

Chapitre 50: Parallélisme

Paramètres

Type / Fonction	Détail
`data Eval a`	Eval est une Monade qui facilite la définition de stratégies parallèles
`type Strategy a = a -> Eval a`	une fonction qui incarne une stratégie d'évaluation parallèle. La fonction parcourt son argument en évaluant les sous-expressions en parallèle ou en séquence
`rpar :: Strategy a`	déclenche son argument (pour une évaluation en parallèle)
`rseq :: Strategy a`	évalue son argument à la forme normale de la tête faible
`force :: NFData a => a -> a`	évalue la structure entière de son argument en le réduisant à la forme normale avant de renvoyer l'argument lui-même. Il est fourni par le module Control.DeepSeq

Remarques

Le livre de Simon Marlow , Programmation simultanée et parallèle à Haskell, est remarquable et couvre une multitude de concepts. Il est également très accessible même pour le plus récent programmeur Haskell. Il est fortement recommandé et disponible en PDF ou en ligne gratuitement.

Parallèle vs Concurrent

Simon Marlow le dit mieux :

> Un programme parallèle est un programme qui utilise une multiplicité de matériel informatique (par exemple, plusieurs cœurs de processeur) pour effectuer un calcul plus rapidement. L'objectif est d'arriver à la réponse plus tôt, en déléguant différentes parties du calcul à différents processeurs exécutant en même temps.
>
> En revanche, la concurrence est une technique de structuration de programme dans laquelle il existe plusieurs threads de contrôle. Conceptuellement, les threads de contrôle s'exécutent «en même temps»; c'est-à-dire que l'utilisateur voit ses effets entrelacés. Qu'elles soient exécutées en même temps ou non, c'est un détail d'implémentation. un programme simultané peut s'exécuter sur un seul processeur par exécution entrelacée ou sur plusieurs processeurs physiques.

Forme normale de tête faible

Il est important de savoir comment fonctionne l'évaluation paresseuse. La première section de ce chapitre présentera une introduction solide sur le WHNF et sa relation avec la programmation parallèle et simultanée.

Examples

L'éval monade

Le parallélisme dans Haskell peut être exprimé à l'aide de l' `Eval` Monad de `Control.Parallel.Strategies` , en utilisant les fonctions `rpar` et `rseq` (entre autres).

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  return (a,b)
```

En cours d'exécution au `main` dessus s'exécutera et "retournera" immédiatement, tandis que les deux valeurs, `a` et `b` seront calculées en arrière-plan via `rpar` .

Remarque: assurez-vous de compiler avec `-threaded` pour que l'exécution en parallèle se produise.

rpar

`rpar :: Strategy a` exécute la stratégie donnée (rappel: `type Strategy a = a -> Eval a`) en parallèle:

```
import Control.Concurrent
import Control.DeepSeq
import Control.Parallel.Strategies
import Data.List.Ordered

main = loop
  where
    loop = do
      putStrLn "Enter a number"
      n <- getLine

      let lim = read n :: Int
          hf  = quot lim 2
          result = runEval $ do
            -- we split the computation in half, so we can concurrently calculate primes
            as <- rpar (force (primesBtwn 2 hf))
            bs <- rpar (force (primesBtwn (hf + 1) lim))
            return (as ++ bs)

      forkIO $ putStrLn ("\nPrimes are: " ++ (show result) ++ " for " ++ n ++ "\n")
```

```
      loop

-- Compute primes between two integers
-- Deliberately inefficient for demonstration purposes
primesBtwn n m = eratos [n..m]
  where
    eratos []     = []
    eratos (p:xs) = p : eratos (xs `minus` [p, p+p..])
```

L'exécution de cette opération démontrera le comportement simultané:

```
Enter a number
12
Enter a number

Primes are: [2,3,5,7,8,9,10,11,12] for 12

100
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,7
for 100

200000000
Enter a number
-- waiting for 200000000
200
Enter a number

Primes are:
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,101,102,103,104,105,106,107,108
for 200

-- still waiting for 200000000
```

rseq

On peut utiliser `rseq :: Strategy a` pour forcer un argument à Weak Head Normal Form:

```
f1 :: [Int]
f1 = [1..100000000]

f2 :: [Int]
f2 = [1..200000000]

main = runEval $ do
  a <- rpar (f1) -- this'll take a while...
  b <- rpar (f2) -- this'll take a while and then some...
  rseq a
  return (a,b)
```

Cela modifie subtilement la sémantique de l'exemple `rpar` ; alors que celui - ci retourneraient immédiatement tout en calculant les valeurs en arrière - plan, cet exemple attendre `a` peut être évalué à WHNF.

Lire Parallélisme en ligne: https://riptutorial.com/fr/haskell/topic/6887/parallelisme

Chapitre 51: Pipes

Remarques

Comme la page de piratage décrit:

> pipes est une bibliothèque de traitement de flux propre et puissante qui vous permet de créer et de connecter des composants de streaming réutilisables

Les programmes mis en œuvre par le biais de la diffusion en continu peuvent souvent être succincts et composables, avec des fonctions simples et courtes vous permettant d'intégrer ou non des fonctions facilement grâce au système de type Haskell.

`await :: Monad m => Consumer' ama`

Extrait une valeur de l'amont, où `a` est notre type d'entrée.

`yield :: Monad m => a -> Producer' am ()`

Produire une valeur, où `a` est le type de sortie.

Il est fortement recommandé de lire le paquet `Pipes.Tutorial` intégré qui donne un excellent aperçu des concepts de base de Pipes et de la manière dont `Producer` , `Consumer` et `Effect` interagissent.

Examples

Producteurs

Un `Producer` est une action monadique qui peut `yield` valeurs pour la consommation en aval:

```
type Producer b = Proxy X () () b
yield :: Monad m => a -> Producer a m ()
```

Par exemple:

```
naturals :: Monad m => Producer Int m ()
naturals = each [1..] -- each is a utility function exported by Pipes
```

On peut bien sûr avoir aussi des `Producer` qui sont des fonctions d'autres valeurs:

```
naturalsUntil :: Monad m => Int -> Producer Int m ()
naturalsUntil n = each [1..n]
```

Les consommateurs

Un `Consumer` ne peut `await` valeurs en amont.

```
type Consumer a = Proxy () a () X
await :: Monad m => Consumer a m a
```

Par exemple:

```
fancyPrint :: MonadIO m => Consumer String m ()
fancyPrint = forever $ do
  numStr <- await
  liftIO $ putStrLn ("I received: " ++ numStr)
```

Pipes

Les tuyaux peuvent à la fois `await` et `yield` .

```
type Pipe a b = Proxy () a () b
```

Ce Pipe attend un `Int` et le convertit en `String` :

```
intToStr :: Monad m => Pipe Int String m ()
intToStr = forever $ await >>= (yield . show)
```

Running Pipes avec runEffect

Nous utilisons `runEffect` pour faire fonctionner notre `Pipe` :

```
main :: IO ()
main = do
  runEffect $ naturalsUntil 10 >-> intToStr >-> fancyPrint
```

Notez que `runEffect` nécessite un `Effect` , qui est un `Proxy` autonome sans entrées ni sorties:

```
runEffect :: Monad m => Effect m r -> m r
type Effect = Proxy X () () X
```

(où `X` est le type vide, également appelé `Void`).

Tuyaux de raccordement

Utilisez `>->` pour connecter les `Producer` , les `Consumer` et les `Pipe` pour composer des fonctions de `Pipe` plus larges.

```
printNaturals :: MonadIO m => Effect m ()
printNaturals = naturalsUntil 10 >-> intToStr >-> fancyPrint
```

`Producer` types `Producer` , `Consumer` , `Pipe` et `Effect` sont tous définis en termes de type `Proxy` général. Par conséquent, `>->` peut être utilisé à diverses fins. Les types définis par l'argument de gauche doivent correspondre au type consommé par le bon argument:

```
(>->) :: Monad m => Producer b m r -> Consumer b   m r -> Effect       m r
(>->) :: Monad m => Producer b m r -> Pipe     b c m r -> Producer   c m r
(>->) :: Monad m => Pipe   a b m r -> Consumer b   m r -> Consumer a   m r
(>->) :: Monad m => Pipe   a b m r -> Pipe     b c m r -> Pipe     a c m r
```

Le transformateur Proxy Monad

Le type de données de base de `pipes` est le transformateur de monade de `Proxy` . `Pipe` , `Producer` , `Consumer` , etc. sont définis en termes de `Proxy` .

Comme `Proxy` est un transformateur monad, les définitions de `Pipe` s prennent la forme de scripts monadiques qui `await` et `yield` valeurs, effectuant en outre des effets à partir de la monade de base `m` .

Combinaison de tuyaux et de communication réseau

Pipes prend en charge la communication binaire simple entre un client et un serveur

Dans cet exemple:

1. un client se connecte et envoie un `FirstMessage`
2. le serveur reçoit et répond à `DoSomething 0`
3. le client reçoit et réponses `DoNothing`
4. les étapes 2 et 3 sont répétées indéfiniment

Le type de données de commande échangé sur le réseau:

```
-- Command.hs
{-# LANGUAGE DeriveGeneric #-}
module Command where
import Data.Binary
import GHC.Generics (Generic)

data Command = FirstMessage
             | DoNothing
             | DoSomething Int
             deriving (Show,Generic)

instance Binary Command
```

Ici, le serveur attend qu'un client se connecte:

```
module Server where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Command as C
import qualified Pipes.Parse as PP
import qualified Pipes.Prelude as PipesPrelude

pageSize :: Int
pageSize = 4096
```

```
-- pure handler, to be used with PipesPrelude.map
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answers the same command that we have receveid

-- impure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "received message = " ++ (show c)
  return $ C.DoSomething 0
  -- whatever incoming command `c` from the client, answer DoSomething 0

main :: IO ()
main = PNT.serve (PNT.Host "127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
                putStrLn $ "Remote connection from ip = " ++ (show remoteAddress)
                _ <- runEffect $ do
                  let bytesReceiver = PNT.fromSocket connectionSocket pageSize
                  let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
                  commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat
PipesBinary.encode >-> PNT.toSocket connectionSocket
                  -- if we want to use the pureHandler
                  --commandDecoder >-> PipesPrelude.map pureHandler >-> for cat
PipesBinary.Encode >-> PNT.toSocket connectionSocket
                return ()
```

Le client se connecte ainsi:

```
module Client where

import Pipes
import qualified Pipes.Binary as PipesBinary
import qualified Pipes.Network.TCP as PNT
import qualified Pipes.Prelude as PipesPrelude
import qualified Pipes.Parse as PP
import qualified Command as C

pageSize :: Int
pageSize = 4096

-- pure handler, to be used with PipesPrelude.amp
pureHandler :: C.Command -> C.Command
pureHandler c = c  -- answer the same command received from the server

-- inpure handler, to be used with PipesPremude.mapM
sideffectHandler :: MonadIO m => C.Command -> m C.Command
sideffectHandler c = do
  liftIO $ putStrLn $ "Received: " ++ (show c)
  return C.DoNothing  -- whatever is received from server, answer DoNothing

main :: IO ()
main = PNT.connect ("127.0.0.1") "23456" $
  \(connectionSocket, remoteAddress) -> do
    putStrLn $ "Connected to distant server ip = " ++ (show remoteAddress)
    sendFirstMessage connectionSocket
    _ <- runEffect $ do
      let bytesReceiver = PNT.fromSocket connectionSocket pageSize
      let commandDecoder = PP.parsed PipesBinary.decode bytesReceiver
      commandDecoder >-> PipesPrelude.mapM sideffectHandler >-> for cat PipesBinary.encode >->
PNT.toSocket connectionSocket
```

```
    return ()

sendFirstMessage :: PNT.Socket -> IO ()
sendFirstMessage s = do
  _ <- runEffect $ do
    let encodedProducer = PipesBinary.encode C.FirstMessage
    encodedProducer >-> PNT.toSocket s
  return ()
```

Lire Pipes en ligne: https://riptutorial.com/fr/haskell/topic/6768/pipes

Chapitre 52: Pliable

Introduction

`Foldable` est la classe des types `t :: * -> *` qui admettent une opération de *pliage* . Un pli agrège les éléments d'une structure dans un ordre bien défini, en utilisant une fonction de combinaison.

Remarques

Si `t` est `Foldable` cela signifie que pour toute valeur `ta` nous savons comment accéder à tous les éléments d' `a` « de l' intérieur » de `ta` dans un ordre linéaire fixe. C'est la signification de `foldMap :: Monoid m => (a -> m) -> (ta -> m)` : nous "visitons" chaque élément avec une fonction récapitulative et cassons tous les résumés. Ordre de respect du `Monoid` (mais invariant pour différents groupes).

Examples

Compter les éléments d'une structure pliable

`length` compte les occurrences des éléments `a` dans une structure pliable `ta` .

```
ghci> length [7, 2, 9]  -- t ~ []
3
ghci> length (Right 'a')  -- t ~ Either e
1  -- 'Either e a' may contain zero or one 'a'
ghci> length (Left "foo")  -- t ~ Either String
0
ghci> length (3, True)  -- t ~ (,) Int
1  -- '(c, a)' always contains exactly one 'a'
```

`length` est définie comme étant équivalente à:

```
class Foldable t where
    -- ...
    length :: t a -> Int
    length = foldl' (\c _ -> c+1) 0
```

Notez que ce type de retour `Int` limite les opérations pouvant être effectuées sur les valeurs obtenues par les appels à la fonction `length` . `fromIntegral` est une fonction utile qui nous permet de résoudre ce problème.

Plier une structure en sens inverse

Tout pli peut être exécuté dans la direction opposée à l'aide du monoïde `Dual` , qui fait basculer un monoïde existant de sorte que l'agrégation retourne en arrière.

```
newtype Dual a = Dual { getDual :: a }

instance Monoid m => Monoid (Dual m) where
    mempty = Dual mempty
    (Dual x) `mappend` (Dual y) = Dual (y `mappend` x)
```

Lorsque le monoïde sous-jacent d'un appel `foldMap` est retourné avec `Dual` , le pli est en arrière; le type d' `Reverse` suivant est défini dans `Data.Functor.Reverse` :

```
newtype Reverse t a = Reverse { getReverse :: t a }

instance Foldable t => Foldable (Reverse t) where
    foldMap f = getDual . foldMap (Dual . f) . getReverse
```

Nous pouvons utiliser cette machinerie pour écrire un `reverse` laconique pour les listes:

```
reverse :: [a] -> [a]
reverse = toList . Reverse
```

Une instance de Pliable pour un arbre binaire

Pour instancier `Foldable` vous devez fournir une définition d'au moins `foldMap` ou `foldr` .

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

instance Foldable Tree where
    foldMap f Leaf = mempty
    foldMap f (Node l x r) = foldMap f l `mappend` f x `mappend` foldMap f r

    foldr f acc Leaf = acc
    foldr f acc (Node l x r) = foldr f (f x (foldr f acc r)) l
```

Cette implémentation effectue une traversée dans l'ordre de l'arborescence.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> toList myTree
"abc"
```

La `DeriveFoldable` extension permet de générer GHC `Foldable` cas en fonction de la structure du type. Nous pouvons modifier l'ordre du parcours écrit en ajustant la disposition du constructeur de `Node` .

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving Foldable
```

```
data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving Foldable

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving Foldable

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x

ghci> toList (inorder myTree)
"abc"
ghci> toList (preorder myTree)
"bac"
ghci> toList (postorder myTree)
"acb"
```

Aplatir une structure pliable en une liste

`toList` aplatit une `Foldable` structure de `ta` dans une liste d' `a` s.

```
ghci> toList [7, 2, 9]  -- t ~ []
[7, 2, 9]
ghci> toList (Right 'a')  -- t ~ Either e
"a"
ghci> toList (Left "foo")  -- t ~ Either String
[]
ghci> toList (3, True)  -- t ~ (,) Int
[True]
```

`toList` est défini comme étant équivalent à:

```
class Foldable t where
    -- ...
    toList :: t a -> [a]
    toList = foldr (:) []
```

Effectuer un effet secondaire pour chaque élément d'une structure pliable

`traverse_` exécute une action `Applicative` pour chaque élément d'une structure `Foldable` . Il ignore le résultat de l'action, ne gardant que les effets secondaires. (Pour une version qui ne supprime pas les résultats, utilisez `Traversable` .)

```
-- using the Writer applicative functor (and the Sum monoid)
ghci> runWriter $ traverse_ (\x -> tell (Sum x)) [1,2,3]
((),Sum {getSum = 6})
-- using the IO applicative functor
ghci> traverse_ putStrLn (Right "traversing")
traversing
ghci> traverse_ putStrLn (Left False)
-- nothing printed
```

`for_` est `traverse_` avec les arguments retournés. Cela ressemble à une boucle `foreach` dans un langage impératif.

```
ghci> let greetings = ["Hello", "Bonjour", "Hola"]
ghci> :{
ghci|     for_ greetings $ \greeting -> do
ghci|         print (greeting ++ " Stack Overflow!")
ghci| :}
"Hello Stack Overflow!"
"Bonjour Stack Overflow!"
"Hola Stack Overflow!"
```

`sequenceA_` réduit un `Foldable` rempli d'actions `Applicative` en une seule action, en ignorant le résultat.

```
ghci> let actions = [putStrLn "one", putStLn "two"]
ghci> sequenceA_ actions
one
two
```

`traverse_` est défini comme étant équivalent à:

```
traverse_ :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f ()
traverse_ f = foldr (\x action -> f x *> action) (pure ())
```

`sequenceA_` est défini comme suit:

```
sequenceA_ :: (Foldable t, Applicative f) -> t (f a) -> f ()
sequenceA_ = traverse_ id
```

En outre, lorsque le `Foldable` est également un `Functor` , `traverse_` et `sequenceA_` ont la relation suivante:

```
traverse_ f = sequenceA_ . fmap f
```

Aplatir une structure pliable en un monoïde

`foldMap` mappe chaque élément de la structure pliable sur un `Monoid` , puis les combine en une seule valeur.

`foldMap` et `foldr` peuvent être définis les uns par rapport aux autres, ce qui signifie que les instances de `Foldable` ne doivent donner qu'une définition pour l'une d'entre elles.

```
class Foldable t where
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty
```

Exemple d'utilisation avec le monoïde de `Product` :

```
product :: (Num n, Foldable t) => t n -> n
product = getProduct . foldMap Product
```

Définition de pliable

```
class Foldable t where
    {-# MINIMAL foldMap | foldr #-}

    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldMap f = foldr (mappend . f) mempty

    foldr :: (a -> b -> b) -> b -> t a -> b
    foldr f z t = appEndo (foldMap (Endo #. f) t) z

    -- and a number of optional methods
```

Intuitivement (mais pas techniquement), `Foldable` structures sont des conteneurs d'éléments d' `a` permettant d' accéder à leurs éléments dans un ordre bien défini. L'opération `foldMap` mappe chaque élément du conteneur sur un `Monoid` et les réduit en utilisant la structure `Monoid` .

Vérifier si une structure pliable est vide

`null` renvoie `True` s'il n'y a pas d'éléments `a` dans une structure pliable `ta` , et `False` s'il en existe un ou plusieurs. Les structures pour lesquelles `null` est `True` ont une `length` de 0.

```
ghci> null []
True
ghci> null [14, 29]
False
ghci> null Nothing
True
ghci> null (Right 'a')
False
ghci> null ('x', 3)
False
```

`null` est défini comme étant équivalent à:

```
class Foldable t where
    -- ...
    null :: t a -> Bool
    null = foldr (\_ _ -> False) True
```

Lire Pliable en ligne: https://riptutorial.com/fr/haskell/topic/753/pliable

Chapitre 53: Polymorphisme de rang arbitraire avec RankNTypes

Introduction

Le système de type GHC prend en charge la quantification universelle explicite de rang arbitraire dans les types grâce à l'utilisation des extensions de langage `Rank2Types` et `RankNTypes` .

Syntaxe

- La quantification de rang arbitraire est activée avec l'extension de langage `Rank2Types` ou `RankNTypes` .
- Avec cette extension activée, le mot clé `forall` peut être utilisé pour ajouter une quantification de rang supérieur.

Examples

RankNTypes

StackOverflow me force à avoir un exemple. Si ce sujet est approuvé, nous devrions déplacer cet exemple ici.

Lire Polymorphisme de rang arbitraire avec RankNTypes en ligne: https://riptutorial.com/fr/haskell/topic/8984/polymorphisme-de-rang-arbitraire-avec-rankntypes

Chapitre 54: Profunctor

Introduction

`Profunctor` est une classe de types fournis par le `profunctors` paquet dans `Data.Profunctor` .

Voir la section "Remarques" pour une explication complète.

Syntaxe

- dimap :: Profunctor p => (a -> b) -> (c -> d) -> pbc -> pad
- lmap :: Profunctor p => (a -> b) -> pbc -> pac
- rmap :: Profunctor p => (b -> c) -> pab -> pac
- id de dimap id = id
- lmap id = id
- rmap id = id
- dimap fg = lmap f. rmap g
- lmap f = dimap f id
- rmap f = id dimap f

Remarques

Les descripteurs sont, comme décrit par les documents sur Hackage, "un bifunctor où le premier argument est contravariant et le second argument est covariant".

Qu'est-ce que cela signifie? Eh bien, un bifunctor est comme un foncteur normal, sauf qu'il a deux paramètres au lieu d'un, chacun avec sa propre fonction de type `fmap` pour le mapper.

Etre "covariant" signifie que le second argument de la source est comme un foncteur normal: sa fonction de mappage (`rmap`) a un type signature de `Profunctor p => (b -> c) -> pab -> pac` . Il ne fait que mapper la fonction sur le second argument.

Être "contravariant" rend le premier argument un peu plus étrange. Au lieu de mapper comme un foncteur normal, sa fonction de mappage (`lmap`) a une signature de type `Profunctor p => (a -> b) -> pbc -> pac` . Cette correspondance apparemment en arrière est très utile pour les entrées d'une fonction: vous exécuteriez `a -> b` sur l'entrée, puis votre autre fonction, en laissant la nouvelle entrée sous la forme d' `a` .

Note: La dénomination des foncteurs normaux à un argument est un peu trompeuse: la `Functor` Functor implémente des foncteurs "covariants", tandis que les foncteurs "contravariant" sont implémentés dans la classe `Contravariant` dans `Data.Functor.Contravariant` , et précédemment la (dénomination trompeuse) `Cofunctor` cofunctor dans `Data.Cofunctor` .

Examples

(->) Profunctor

(->) est un exemple simple de source: l'argument de gauche est l'entrée d'une fonction et l'argument de droite est le même que l'instance de foncteur de lecteur.

```
instance Profunctor (->) where
    lmap f g = g . f
    rmap f g = g . g
```

Lire Profunctor en ligne: https://riptutorial.com/fr/haskell/topic/9694/profunctor

Chapitre 55: Règles de réécriture (GHC)

Examples

Utilisation de règles de réécriture sur les fonctions surchargées

Dans cette question , @Viclib a demandé comment utiliser les règles de réécriture pour exploiter les lois de la classe de polices afin d'éliminer certains appels de fonctions surchargés:

> Attention à la classe suivante:
>
> ```
> class ListIsomorphic l where
>     toList    :: l a -> [a]
>     fromList  :: [a] -> l a
> ```
>
> Je demande aussi cette `toList . fromList == id` . Comment écrire des règles de réécriture pour dire à GHC de faire cette substitution?

Ceci est un cas d'utilisation quelque peu délicat pour le mécanisme de règles de réécriture de GHC, car les fonctions surchargées sont réécrites dans leurs méthodes d'instance spécifiques par des règles implicitement créées en arrière-plan par GHC (quelque chose comme `fromList :: Seq a -> [a]` serait réécrit en `Seq$fromList` etc.).

Cependant, en réécrivant d'abord `toList` et `fromList` dans des méthodes non-typées non incorporées, nous pouvons les protéger contre une réécriture prématurée et les conserver jusqu'à ce que la règle de la composition puisse s'exécuter:

```
{-# RULES
  "protect toList"   toList = toList';
  "protect fromList" fromList = fromList';
  "fromList/toList"  forall x . fromList' (toList' x) = x; #-}

{-# NOINLINE [0] fromList' #-}
fromList' :: (ListIsomorphic l) => [a] -> l a
fromList' = fromList

{-# NOINLINE [0] toList' #-}
toList' :: (ListIsomorphic l) => l a -> [a]
toList' = toList
```

Lire Règles de réécriture (GHC) en ligne: https://riptutorial.com/fr/haskell/topic/4914/regles-de-reecriture--ghc-

Chapitre 56: Rigueur

Examples

Modèles de bang

Les motifs annotés par un bang (`!`) Sont évalués strictement plutôt que paresseusement.

```
foo (!x, y) !z = [x, y, z]
```

Dans cet exemple, `x` et `z` seront tous deux évalués à la forme normale de la tête faible avant de renvoyer la liste. C'est équivalent à:

```
foo (x, y) z = x `seq` z `seq` [x, y, z]
```

Les patterns Bang sont activés en utilisant l'extension de langage Haskell 2010 `BangPatterns` .

Formes normales

Cet exemple fournit un bref aperçu - pour une explication plus détaillée des *formes* et des exemples *normaux* , consultez cette question .

Forme normale réduite

La forme normale réduite (ou simplement la forme normale, lorsque le contexte est clair) d'une expression est le résultat de l'évaluation de toutes les sous-expressions réductibles dans l'expression donnée. En raison de la sémantique non stricte de Haskell (généralement appelée la *paresse*), une sous-expression n'est pas réductible si elle est sous un classeur (c'est-à-dire une abstraction lambda - `\x -> ..`). La forme normale d'une expression a la propriété que si elle existe, elle est unique.

En d'autres termes, l'ordre de réduction des sous-expressions n'est pas important (en termes de sémantique dénotationnelle). Cependant, la clé de l'écriture de programmes Haskell performants consiste souvent à s'assurer que la bonne expression est évaluée au bon moment, c'est-à-dire à comprendre la sémantique opérationnelle.

Une expression dont la forme normale est elle-même est dite *sous forme normale* .

Certaines expressions, par exemple, `let x = 1:x in x` , n'ont pas de forme normale, mais sont toujours productives. L'expression d'exemple a toujours une *valeur* , si l'on admet des valeurs infinies, qui est la liste `[1,1, ...]` . D'autres expressions, telles que `let y = 1+y in y` , n'ont aucune valeur ou leur valeur est `undefined` .

Faible tête forme normale

Le RNF correspond à une évaluation complète d'une expression - de même, la forme normale de la tête faible (WHNF) correspond à l'évaluation de la *tête* de l'expression. La tête d'une expression `e` est entièrement évaluée si `e` est une application `Con e1 e2 .. en` et `Con` est un constructeur; ou une abstraction `\x -> e1` ; ou une application partielle `f e1 e2 .. en` , où une application partielle signifie que `f` prend plus de `n` arguments (ou, de manière équivalente, le type de `e` est un type de fonction). Dans tous les cas, les sous-expressions `e1..en` peuvent être évaluées ou non pour que l'expression soit dans WHNF - elles peuvent même être `undefined` .

La sémantique d'évaluation de Haskell peut être décrite en termes de WHNF - pour évaluer une expression `e` , évaluez-la d'abord à WHNF, puis évaluez récursivement toutes ses sous-expressions de gauche à droite.

La fonction `seq` primitive est utilisée pour évaluer une expression en WHNF. `seq xy` est dénotationnellement égal à `y` (la valeur de `seq xy` est précisément `y`); de plus, `x` est évalué à WHNF lorsque `y` est évalué à WHNF. Une expression peut également être évaluée avec WHNF avec un motif bang (activé par l'extension `-XBangPatterns`), dont la syntaxe est la suivante:

```
f !x y = ...
```

Dans lequel `x` sera évalué à WHNF lorsque `f` est évalué, alors que `y` n'est pas (nécessairement) évalué. Un motif de bang peut également apparaître dans un constructeur, par exemple

```
data X = Con A !B C .. N
```

Dans ce cas, le constructeur `Con` est dit strict dans le champ `B` , ce qui signifie que le champ `B` est évalué à WHNF lorsque le constructeur est appliqué à des arguments suffisants (ici deux).

Motifs paresseux

Les patterns paresseux ou *irréfutables* (désignés par la syntaxe `~pat`) sont des patterns qui correspondent toujours, sans même regarder la valeur correspondante. Cela signifie que les motifs paresseux correspondent aux valeurs les plus basses. Cependant, les utilisations ultérieures de variables liées à des sous-modèles d'un modèle irréfutable forceront la correspondance du modèle, en évaluant vers le bas à moins que la correspondance ne réussisse.

La fonction suivante est paresseuse dans son argument:

```
f1 :: Either e Int -> Int
f1 ~(Right 1) = 42
```

et ainsi nous obtenons

```
λ» f1 (Right 1)
42
λ» f1 (Right 2)
```

```
42
λ» f1 (Left "foo")
42
λ» f1 (error "oops!")
42
λ» f1 "oops!"
*** type mismatch ***
```

La fonction suivante est écrite avec un pattern paresseux mais utilise en fait la variable du pattern qui force la correspondance, donc échouera pour les arguments `Left` :

```
f2 :: Either e Int -> Int
f2 ~(Right x) = x + 1

λ» f2 (Right 1)
2
λ» f2 (Right 2)
3
λ» f2 (Right (error "oops!"))
*** Exception: oops!
λ» f2 (Left "foo")
*** Exception: lazypat.hs:5:1-21: Irrefutable pattern failed for pattern (Right x)
λ» f2 (error "oops!")
*** Exception: oops!
```

`let` liaisons soient paresseuses, se comportent comme des modèles irréfutables:

```
act1 :: IO ()
act1 = do
    ss <- readLn
    let [s1, s2] = ss :: [String]
    putStrLn "Done"

act2 :: IO ()
act2 = do
    ss <- readLn
    let [s1, s2] = ss
    putStrLn s1
```

Ici, `act1` travaille sur des entrées qui analysent n'importe quelle liste de chaînes, alors que dans `act2` `putStrLn s1` besoin de la valeur de `s1` qui force la correspondance de modèle pour `[s1, s2]` .

```
λ» act1
> ["foo"]
Done
λ» act2
> ["foo"]
*** readIO: no parse ***
```

Champs stricts

Dans une déclaration de `data` , le fait de préfixer un type avec un bang (`!`) Fait du champ un *champ strict* . Lorsque le constructeur de données est appliqué, ces champs seront évalués à la forme normale de tête faible, de sorte que les données dans les champs sont garanties pour être

toujours sous la forme normale de tête faible.

Les champs stricts peuvent être utilisés dans les types d'enregistrement et de non-enregistrement:

```
data User = User
    { identifier :: !Int
    , firstName :: !Text
    , lastName :: !Text
    }

data T = MkT !Int !Int
```

Lire Rigueur en ligne: https://riptutorial.com/fr/haskell/topic/3798/rigueur

Chapitre 57: Rôle

Introduction

L'extension de langage `TypeFamilies` permet au programmeur de définir des fonctions au niveau du type. Ce qui distingue les fonctions de type des constructeurs de type non-GADT, c'est que les paramètres des fonctions de type peuvent être non paramétriques alors que les paramètres des constructeurs de types sont toujours paramétriques. Cette distinction est importante pour l'exactitude de l'extension `GeneralizedNewTypeDeriving` . Pour expliquer cette distinction, les rôles sont introduits dans Haskell.

Remarques

Voir aussi `SafeNewtypeDeriving` .

Examples

Rôle nominal

Haskell Wiki a un exemple de paramètre non paramétrique d'une fonction de type:

```
type family Inspect x
type instance Inspect Age = Int
type instance Inspect Int = Bool
```

Ici, `x` est non paramétrique car pour déterminer le résultat de l'application d' `Inspect` à un argument de type, la fonction de type doit inspecter `x` .

Dans ce cas, le rôle de `x` est nominal. Nous pouvons déclarer le rôle explicitement avec l'extension `RoleAnnotations` :

```
type role Inspect nominal
```

Rôle de représentation

Un exemple de paramètre paramétrique d'une fonction de type:

```
data List a = Nil | Cons a (List a)

type family DoNotInspect x
type instance DoNotInspect x = List x
```

Ici, `x` est paramétrique car, pour déterminer le résultat de l'application de `DoNotInspect` à un argument de type, la fonction de type n'a pas besoin d'inspecter `x` .

Dans ce cas, le rôle de x est figuratif. Nous pouvons déclarer le rôle explicitement avec l'extension `RoleAnnotations` :

```
type role DoNotInspect representational
```

Rôle fantôme

Un paramètre de type fantôme a un rôle fantôme. Les rôles fantômes ne peuvent pas être déclarés explicitement.

Lire Rôle en ligne: https://riptutorial.com/fr/haskell/topic/8753/role

Chapitre 58: Schémas de récursivité

Remarques

Les fonctions mentionnées ici dans les exemples sont définies avec des degrés d'abstraction variables dans plusieurs packages, par exemple les `recursion-schemes` `data-fix` et de `recursion-schemes` (plus de fonctions ici). Vous pouvez afficher une liste plus complète en effectuant une recherche sur Hayoo .

Examples

Points fixes

`Fix` prend un type "template" et lie le nœud récursif, superposant le gabarit comme une lasagne.

```
newtype Fix f = Fix { unFix :: f (Fix f) }
```

Dans un `Fix f` on trouve une couche du gabarit `f` . Pour remplir `f` paramètre d », `Fix f` bouchons en *soi.* Alors , quand vous regardez à l' intérieur du modèle `f` vous trouvez un événement récurrent de `Fix f` .

Voici comment un type de données récursif typique peut être traduit dans notre structure de modèles et de points fixes. Nous supprimons les occurrences récursives du type et marquons leurs positions en utilisant le paramètre `r` .

```
{-# LANGUAGE DeriveFunctor #-}

-- natural numbers
-- data Nat = Zero | Suc Nat
data NatF r = Zero_ | Suc_ r deriving Functor
type Nat = Fix NatF

zero :: Nat
zero = Fix Zero_
suc :: Nat -> Nat
suc n = Fix (Suc_ n)


-- lists: note the additional type parameter a
-- data List a = Nil | Cons a (List a)
data ListF a r = Nil_ | Cons_ a r deriving Functor
type List a = Fix (ListF a)

nil :: List a
nil = Fix Nil_
cons :: a -> List a -> List a
cons x xs = Fix (Cons_ x xs)


-- binary trees: note two recursive occurrences
```

```
-- data Tree a = Leaf | Node (Tree a) a (Tree a)
data TreeF a r = Leaf_ | Node_ r a r deriving Functor
type Tree a = Fix (TreeF a)

leaf :: Tree a
leaf = Fix Leaf_
node :: Tree a -> a -> Tree a -> Tree a
node l x r = Fix (Node_ l x r)
```

Plier une structure une couche à la fois

Catamorphismes , ou *replis* , modèle *récursif* primitif. `cata` déchire un point de repère couche par couche, en utilisant une fonction d' *algèbre* (ou une *fonction de pliage*) pour traiter chaque couche. `cata` nécessite une instance `Functor` pour le type de modèle `f` .

```
cata :: Functor f => (f a -> a) -> Fix f -> a
cata f = f . fmap (cata f) . unFix

-- list example
foldr :: (a -> b -> b) -> b -> List a -> b
foldr f z = cata alg
    where alg Nil_ = z
          alg (Cons_ x acc) = f x acc
```

Déplier une structure une couche à la fois

Les anamorphismes , ou *se* déploient, modélisent la corécursion primitive. `ana` construit un point de repère couche par couche, en utilisant une fonction *Coalgebra* (ou une *fonction de déploiement*) pour produire chaque nouveau calque. `ana` nécessite une instance `Functor` pour le type de modèle `f` .

```
ana :: Functor f => (a -> f a) -> a -> Fix f
ana f = Fix . fmap (ana f) . f

-- list example
unfoldr :: (b -> Maybe (a, b)) -> b -> List a
unfoldr f = ana coalg
    where coalg x = case f x of
                        Nothing -> Nil_
                        Just (x, y) -> Cons_ x y
```

Notez que `ana` et `cata` sont *doubles* . Les types et les implémentations sont des images en miroir les uns des autres.

Dépliage puis pliage, fusionné

Il est courant de structurer un programme en créant une structure de données, puis en la réduisant à une valeur unique. Cela s'appelle un *hylomorphisme* ou un *repli* . Il est possible d'éliminer complètement la structure intermédiaire pour améliorer l'efficacité.

```
hylo :: Functor f => (a -> f a) -> (f b -> b) -> a -> b
hylo f g = g . fmap (hylo f g) . f  -- no mention of Fix!
```

Dérivation:

```
hylo f g = cata g . ana f
         = g . fmap (cata g) . unFix . Fix . fmap (ana f) . f  -- definition of cata and ana
         = g . fmap (cata g) . fmap (ana f) . f  -- unfix . Fix = id
         = g . fmap (cata g . ana f) . f  -- Functor law
         = g . fmap (hylo f g) . f  -- definition of hylo
```

Récursivité primitive

Les paramorphismes modélisent la récursivité primitive. A chaque itération du pli, la fonction de pliage reçoit le sous-arbre pour un traitement ultérieur.

```
para :: Functor f => (f (Fix f, a) -> a) -> Fix f -> a
para f = f . fmap (\x -> (x, para f x)) . unFix
```

Les `tails` du prélude peuvent être modélisées en tant que paramorphisme.

```
tails :: List a -> List (List a)
tails = para alg
    where alg Nil_ = cons nil nil  -- [[]]
          alg (Cons_ x (xs, xss)) = cons (cons x xs) xss  -- (x:xs):xss
```

Corecursion primitive

Les apomorphismes modélisent la corécursion primitive. A chaque itération du dépliage, la fonction dépliante peut renvoyer une nouvelle graine ou un sous-arbre entier.

```
apo :: Functor f => (a -> f (Either (Fix f) a)) -> a -> Fix f
apo f = Fix . fmap (either id (apo f)) . f
```

Notez que `apo` et `para` sont *doubles* . Les flèches du type sont retournées; le tuple dans `para` est double à l'un `Either` à l' `Either` dans l' `apo` , et les implémentations sont des images miroir l'une de l'autre.

Lire Schémas de récursivité en ligne: https://riptutorial.com/fr/haskell/topic/2984/schemas-de-recursivite

Chapitre 59: Streaming IO

Examples

Streaming IO

`io-streams` est une bibliothèque basée sur un flux qui se concentre sur l'abstraction du flux mais pour IO. Il expose deux types:

- `InputStream` : une poignée intelligente en lecture seule
- `OutputStream` : une poignée intelligente en écriture seule

Nous pouvons créer un flux avec `makeInputStream :: IO (Maybe a) -> IO (InputStream a)` . La lecture d'un flux est effectuée en utilisant `read :: InputStream a -> IO (Maybe a)` , où `Nothing` dénote un EOF:

```
import Control.Monad (forever)
import qualified System.IO.Streams as S
import System.Random (randomRIO)

main :: IO ()
main = do
  is <- S.makeInputStream $ randomInt  -- create an InputStream
  forever $ printStream =<< S.read is  -- forever read from that stream
  return ()

randomInt :: IO (Maybe Int)
randomInt = do
  r <- randomRIO (1, 100)
  return $ Just r

printStream :: Maybe Int -> IO ()
printStream Nothing  = print "Nada!"
printStream (Just a) = putStrLn $ show a
```

Lire Streaming IO en ligne: https://riptutorial.com/fr/haskell/topic/4984/streaming-io

Chapitre 60: Syntaxe d'appel de fonction

Introduction

La syntaxe d'appel de la fonction de Haskell, expliquée par des comparaisons avec les langages de style C, le cas échéant. Ceci est destiné aux personnes qui viennent à Haskell à partir d'une formation en langage C.

Remarques

En général, la règle pour convertir un appel de fonction de style C en Haskell, dans n'importe quel contexte (affectation, retour ou incorporé dans un autre appel), consiste à remplacer les virgules de la liste d'arguments de style C par des espaces et à déplacer l'ouverture. parenthèses de l'appel de style C pour contenir le nom de la fonction et ses paramètres.

Si des expressions sont placées entièrement entre parenthèses, ces paires de parenthèses (externes) peuvent être supprimées pour des raisons de lisibilité, car elles n'affectent pas la signification de l'expression.
Il existe d'autres circonstances dans lesquelles les parenthèses peuvent être supprimées, mais cela n'affecte que la lisibilité et la maintenabilité.

Examples

Parenthèses dans un appel de fonction de base

Pour un appel de fonction de style C, par exemple

```
plus(a, b); // Parentheses surrounding only the arguments, comma separated
```

Alors le code Haskell équivalent sera

```
(plus a b) -- Parentheses surrounding the function and the arguments, no commas
```

Dans Haskell, les parenthèses ne sont pas explicitement requises pour l'application de fonction et ne sont utilisées que pour désambiguïser des expressions, comme en mathématiques; ainsi, dans les cas où les crochets entourent tout le texte de l'expression, les parenthèses ne sont en fait pas nécessaires et les suivantes sont également équivalentes:

```
plus a b -- no parentheses are needed here!
```

Il est important de se rappeler que dans les langages de style C, la fonction

Parenthèses dans les appels de fonctions incorporés

Dans l'exemple précédent, nous n'avons pas eu besoin des parenthèses, car elles n'affectaient pas le sens de l'instruction. Cependant, ils sont souvent nécessaires dans une expression plus complexe, comme celle ci-dessous.
En C:

```
plus(a, take(b, c));
```

En Haskell, cela devient:

```
(plus a (take b c))
-- or equivalently, omitting the outermost parentheses
plus a (take b c)
```

Notez que ce n'est pas équivalent à:

```
plus a take b c -- Not what we want!
```

On pourrait penser que le compilateur sachant que `take` est une fonction, il pourrait savoir que vous voulez l'appliquer aux arguments `b` et `c` et transmettre son résultat à `plus` .
Cependant, dans Haskell, les fonctions prennent souvent d'autres fonctions comme arguments et peu de distinction est faite entre les fonctions et les autres valeurs. et donc le compilateur ne peut pas assumer votre intention simplement parce que `take` est une fonction.

Et ainsi, le dernier exemple est analogue à l'appel de fonction C suivant:

```
plus(a, take, b, c); // Not what we want!
```

Application partielle - Partie 1

Dans Haskell, les fonctions peuvent être partiellement appliquées; Nous pouvons considérer toutes les fonctions comme prenant un seul argument et renvoyer une fonction modifiée pour laquelle cet argument est constant. Pour illustrer cela, nous pouvons encadrer les fonctions comme suit:

```
(((plus) 1) 2)
```

Ici, la fonction `(plus)` est appliquée à `1` donnant la fonction `((plus) 1)` , qui est appliquée à `2` , donnant la fonction `(((plus) 1) 2)` . Parce que `plus 1 2` est une fonction qui ne prend aucun argument, vous pouvez la considérer comme une valeur simple; Cependant, dans Haskell, il y a peu de distinction entre les fonctions et les valeurs.

Pour aller plus en détail, la fonction `plus` est une fonction qui ajoute ses arguments.
La fonction `plus 1` est une fonction qui ajoute `1` à son argument.
La fonction `plus 1 2` est une fonction qui ajoute `1` à `2` , qui est toujours la valeur `3` .

Application partielle - Partie 2

Comme autre exemple, nous avons la fonction `map` , qui prend une fonction et une liste de valeurs, et applique la fonction à chaque valeur de la liste:

```
map :: (a -> b) -> [a] -> [b]
```

Disons que nous voulons incrémenter chaque valeur dans une liste. Vous pouvez décider de définir votre propre fonction, ce qui en ajoute une à son argument, et `map` cette fonction sur votre liste.

```
addOne x = plus 1 x
map addOne [1,2,3]
```

mais si vous avez un autre regard sur la définition de `addOne` , avec des parenthèses pour les mettre en évidence:

```
(addOne) x = ((plus) 1) x
```

La fonction `addOne` , appliquée à toute valeur `x` , est identique à la fonction partiellement appliquée `plus 1` appliquée à `x` . Cela signifie que les fonctions `addOne` et `plus 1` sont identiques, et nous pouvons éviter de définir une nouvelle fonction en remplaçant simplement `addOne` par `plus 1` , sans oublier d'utiliser des parenthèses pour isoler `plus 1` tant que sous-expression:

```
map (plus 1) [1,2,3]
```

Lire Syntaxe d'appel de fonction en ligne: https://riptutorial.com/fr/haskell/topic/9615/syntaxe-d-appel-de-fonction

Chapitre 61: Syntaxe d'enregistrement

Examples

Syntaxe de base

Les enregistrements sont une extension du type de `data` algébrique somme qui permet de nommer les champs:

```
data StandardType = StandardType String Int Bool --standard way to create a sum type

data RecordType = RecordType { -- the same sum type with record syntax
    aString :: String
  , aNumber :: Int
  , isTrue  :: Bool
  }
```

Les noms de champs peuvent alors être utilisés pour extraire le champ nommé

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> :t r
  r :: RecordType
> :t aString
  aString :: RecordType -> String
> aString r
  "Foobar"
```

Les enregistrements peuvent être associés à un motif

```
case r of
  RecordType{aNumber = x, aString=str} -> ... -- x = 42, str = "Foobar"
```

Notez que tous les champs ne doivent pas être nommés

Les enregistrements sont créés en nommant leurs champs, mais peuvent également être créés en tant que types de somme ordinaires (souvent utiles lorsque le nombre de champs est petit et peu susceptible de changer)

```
r  = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
r' = RecordType  "Foobar" 42 True
```

Si un enregistrement est créé sans champ nommé, le compilateur émettra un avertissement et la valeur résultante sera `undefined` .

```
> let r = RecordType {aString = "Foobar", aNumber= 42}
  <interactive>:1:9: Warning:
     Fields of RecordType not initialized: isTrue
> isTrue r
  Error 'undefined'
```

Un champ d'un enregistrement peut être mis à jour en définissant sa valeur. Les champs non mentionnés ne changent pas.

```
> let r = RecordType {aString = "Foobar", aNumber= 42, isTrue = True}
> let r' = r{aNumber=117}
    -- r'{aString = "Foobar", aNumber= 117, isTrue = True}
```

Il est souvent utile de créer des objectifs pour des types d'enregistrements complexes.

Copier des enregistrements en changeant les valeurs de champs

Supposons que vous ayez ce type:

```
data Person = Person { name :: String, age:: Int } deriving (Show, Eq)
```

et deux valeurs:

```
alex = Person { name = "Alex", age = 21 }
jenny = Person { name = "Jenny", age = 36 }
```

Une nouvelle valeur de type `Person` peut être créée en copiant depuis `alex` , en spécifiant les valeurs à modifier:

```
anotherAlex = alex { age = 31 }
```

Les valeurs de `alex` et `anotherAlex` seront maintenant:

```
Person {name = "Alex", age = 21}

Person {name = "Alex", age = 31}
```

Records avec newtype

La syntaxe d'enregistrement peut être utilisée avec `newtype` avec la restriction qu'il existe exactement un constructeur avec un seul champ. L'avantage ici est la création automatique d'une fonction pour dérouler le nouveau type. Ces champs sont souvent nommés en commençant par `run` pour les monads, `get` pour les monoids et `un` pour les autres types.

```
newtype State s a = State { runState :: s -> (s, a) }

newtype Product a = Product { getProduct :: a }

newtype Fancy = Fancy { unfancy :: String }
  -- a fancy string that wants to avoid concatenation with ordinary strings
```

Il est important de noter que la syntaxe de l'enregistrement n'est généralement pas utilisée pour former des valeurs et que le nom du champ est strictement utilisé pour le dépliage.

```
getProduct $ mconcat [Product 7, Product 9, Product 12]
```

```
-- > 756
```

RecordWildCards

```
{-# LANGUAGE RecordWildCards #-}

data Client = Client { firstName     :: String
                     , lastName      :: String
                     , clientID      :: String
                     } deriving (Show)

printClientName :: Client -> IO ()
printClientName Client{..} = do
    putStrLn firstName
    putStrLn lastName
    putStrLn clientID
```

Le modèle `Client{..}` apporte la portée à tous les champs du constructeur `Client` , et est équivalent au pattern

```
Client{ firstName = firstName, lastName = lastName, clientID = clientID }
```

Il peut également être combiné avec d'autres agents de terrain, comme ceci:

```
Client { firstName = "Joe", .. }
```

Ceci est équivalent à

```
Client{ firstName = "Joe", lastName = lastName, clientID = clientID }
```

Définition d'un type de données avec des étiquettes de champ

Il est possible de définir un type de données avec des étiquettes de champ.

```
data Person = Person { age :: Int, name :: String }
```

Cette définition diffère d'une définition d'enregistrement normale car elle définit également * *les accesseurs d'enregistrement* qui peuvent être utilisés pour accéder à des parties d'un type de données.

Dans cet exemple, deux accesseurs d’enregistrement sont définis: `age` et `name` , ce qui nous permet d’accéder respectivement aux champs `age` et `name` .

```
age :: Person -> Int
name :: Person -> String
```

Les accesseurs d'enregistrements ne sont que des fonctions Haskell générées automatiquement par le compilateur. En tant que tels, ils sont utilisés comme des fonctions Haskell ordinaires.

En nommant les champs, nous pouvons également utiliser les étiquettes de champs dans plusieurs autres contextes afin de rendre notre code plus lisible.

Correspondance de motif

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name = x }) = map toLower x
```

Nous pouvons lier la valeur située à la position du libellé du champ concerné tout en faisant correspondre le modèle à une nouvelle valeur (dans ce cas `x`) qui peut être utilisée sur le RHS d'une définition.

Correspondance de modèle avec `NamedFieldPuns`

```
lowerCaseName :: Person -> String
lowerCaseName (Person { name }) = map toLower name
```

L'extension `NamedFieldPuns` nous permet à la place de simplement spécifier le libellé du champ sur lequel nous voulons faire correspondre, ce nom est alors ombré sur le RHS d'une définition, ainsi la référence au `name` fait référence à la valeur plutôt qu'à l'accesseur d'enregistrement.

Correspondance de motifs avec `RecordWildcards`

```
lowerCaseName :: Person -> String
lowerCaseName (Person { .. }) = map toLower name
```

Lors de la correspondance à l'aide de `RecordWildCards` , toutes les étiquettes de champs sont intégrées à la portée. (Dans cet exemple spécifique, `name` et `age`)

Cette extension est légèrement controversée car on ne sait pas comment les valeurs sont intégrées si vous n'êtes pas certain de la définition de `Person` .

Mises à jour des enregistrements

```
setName :: String -> Person -> Person
setName newName person = person { name = newName }
```

Il existe également une syntaxe spéciale pour la mise à jour des types de données avec des étiquettes de champ.

Lire Syntaxe d'enregistrement en ligne: https://riptutorial.com/fr/haskell/topic/1950/syntaxe-d-enregistrement

Chapitre 62: Syntaxe dans les fonctions

Examples

Gardes

Une fonction peut être définie à l'aide de gardes, qui peuvent être considérés comme classant le comportement en fonction de l'entrée.

Prenez la définition de fonction suivante:

```
absolute :: Int -> Int  -- definition restricted to Ints for simplicity
absolute n = if (n < 0) then (-n) else n
```

Nous pouvons le réorganiser en utilisant des gardes:

```
absolute :: Int -> Int
absolute n
  | n < 0 = -n
  | otherwise = n
```

Dans ce contexte, `otherwise` est un alias significatif pour `True` , donc il devrait toujours être le dernier gardien.

Correspondance de motif

Haskell prend en charge les expressions de correspondance de modèle à la fois dans la définition des fonctions et dans les déclarations de `case` .

Une instruction de cas ressemble beaucoup à un commutateur dans d'autres langues, sauf qu'elle prend en charge tous les types de Haskell.

Commençons simple:

```
longName :: String -> String
longName name = case name of
                  "Alex"  -> "Alexander"
                  "Jenny" -> "Jennifer"
                  _       -> "Unknown"  -- the "default" case, if you like
```

Ou, nous pourrions définir notre fonction comme une équation qui serait une correspondance de modèle, sans utiliser une déclaration de `case` :

```
longName "Alex"  = "Alexander"
longName "Jenny" = "Jennifer"
longName _       = "Unknown"
```

Un exemple plus courant concerne le type `Maybe` :

```
data Person = Person { name :: String, petName :: (Maybe String) }

hasPet :: Person -> Bool
hasPet (Person _ Nothing) = False
hasPet _ = True  -- Maybe can only take `Just a` or `Nothing`, so this wildcard suffices
```

La correspondance de modèle peut également être utilisée sur des listes:

```
isEmptyList :: [a] -> Bool
isEmptyList [] = True
isEmptyList _  = False

addFirstTwoItems :: [Int] -> [Int]
addFirstTwoItems []        = []
addFirstTwoItems (x:[])    = [x]
addFirstTwoItems (x:y:ys)  = (x + y) : ys
```

En fait, la correspondance de modèle peut être utilisée sur n'importe quel constructeur pour n'importe quelle classe de type. Par exemple, le constructeur des listes est `:` et pour les tuples `,`

En utilisant où et les gardes

Compte tenu de cette fonction:

```
annualSalaryCalc :: (RealFloat a) => a -> a -> String
annualSalaryCalc hourlyRate weekHoursOfWork
  | hourlyRate * (weekHoursOfWork * 52) <= 40000  = "Poor child, try to get another job"
  | hourlyRate * (weekHoursOfWork * 52) <= 120000 = "Money, Money, Money!"
  | hourlyRate * (weekHoursOfWork * 52) <= 200000 = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
```

Nous pouvons utiliser `where` éviter la répétition et rendre notre code plus lisible. Voir la fonction alternative ci-dessous, en utilisant `where` :

```
annualSalaryCalc' :: (RealFloat a) => a -> a -> String
annualSalaryCalc' hourlyRate weekHoursOfWork
  | annualSalary <= smallSalary  = "Poor child, try to get another job"
  | annualSalary <= mediumSalary = "Money, Money, Money!"
  | annualSalary <= highSalary   = "Ri¢hie Ri¢h"
  | otherwise = "Hello Elon Musk!"
  where
      annualSalary = hourlyRate * (weekHoursOfWork * 52)
      (smallSalary, mediumSalary, highSalary)  = (40000, 120000, 200000)
```

Comme l'a observé, nous avons utilisé le `where` à la fin du corps de la fonction éliminant la répétition du calcul (`hourlyRate * (weekHoursOfWork * 52)`) et nous avons également utilisé `where` d'organiser l'échelle salariale.

La dénomination des sous-expressions communes peut également être obtenue avec `let` expressions `let` , mais uniquement la syntaxe `where` permet aux *gardes de* faire référence à ces sous-expressions nommées.

Lire Syntaxe dans les fonctions en ligne: https://riptutorial.com/fr/haskell/topic/3799/syntaxe-dans-

les-fonctions

Chapitre 63: Tampons de protocole Google

Remarques

Pour utiliser les tampons de protocole avec Haskell, vous devez installer le package `htprotoc` :

1. Cloner le projet depuis Github
2. Utilisez Stack pour construire et installer

Vous devriez maintenant trouver l'exécutable `hprotoc` dans `$HOME/.local/bin/` .

Examples

Créer, construire et utiliser un simple fichier .proto

Laissez - nous d' abord créer un simple `.proto` fichier `person.proto`

```
package Protocol;

message Person {
    required string firstName = 1;
    required string lastName  = 2;
    optional int32  age       = 3;
}
```

Après la sauvegarde, nous pouvons maintenant créer les fichiers Haskell que nous pouvons utiliser dans notre projet en exécutant

```
$HOME/.local/bin/hprotoc --proto_path=. --haskell_out=. person.proto
```

Nous devrions obtenir une sortie similaire à celle-ci:

```
Loading filepath: "/<path-to-project>/person.proto"
All proto files loaded
Haskell name mangling done
Recursive modules resolved
./Protocol/Person.hs
./Protocol.hs
Processing complete, have a nice day.
```

`hprotoc` va créer un nouveau dossier `Protocol` dans le répertoire courant avec `Person.hs` que nous pouvons simplement importer dans notre projet haskell:

```
import Protocol (Person)
```

Comme étape suivante, si vous utilisez Stack, ajoutez

```
    protocol-buffers
  , protocol-buffers-descriptor
```

`build-depends:` et

```
Protocol
```

aux `exposed-modules` dans votre fichier `.cabal` .

Si nous recevons maintenant un message entrant d'un flux, le message aura le type `ByteString` .

Pour transformer le `ByteString` (qui doit évidemment contenir des données "Person" encodées) dans notre type de données Haskell, nous devons appeler la fonction `messageGet` que nous importons par

```
import Text.ProtocolBuffers (messageGet)
```

qui permet de créer une valeur de type `Person` utilisant:

```
transformRawPerson :: ByteString -> Maybe Person
transformRawPerson raw = case messageGet raw of
    Left   _           -> Nothing
    Right (person, _)  -> Just person
```

Lire Tampons de protocole Google en ligne: https://riptutorial.com/fr/haskell/topic/5018/tampons-de-protocole-google

Chapitre 64: Template Haskell & QuasiQuotes

Remarques

Qu'est-ce que Template Haskell?

Template Haskell fait référence aux fonctions de méta-programmation de modèles intégrées à GHC Haskell. Le document décrivant la mise en œuvre originale peut être trouvé ici .

Quelles sont les étapes? (Ou quelle est la restriction de la scène?)

Les étapes se réfèrent au *moment où le* code est exécuté. Normalement, le code n'est exécuté qu'au moment de l'exécution, mais avec Template Haskell, le code peut être exécuté au moment de la compilation. Le code "normal" est l'étape 0 et le code de compilation est l'étape 1.

La restriction d'étape fait référence au fait qu'un programme d'étape 0 peut ne pas être exécuté à l'étape 1 - cela équivaudrait à être capable d'exécuter un programme *régulier* (pas seulement un méta-programme) au moment de la compilation.

Par convention (et dans un souci de simplicité d'implémentation), le code du module en cours est toujours l'étape 0 et le code importé de tous les autres modules est l'étape 1. Pour cette raison, seules les expressions d'autres modules peuvent être épissées.

Notez qu'un programme d'étape 1 est une expression d'étape 0 du type `Q Exp` , `Q Type` , etc. mais l'inverse n'est pas vrai - toutes les valeurs (programme de l'étape 0) de type `Q Exp` sont pas des programmes de l'étape 1,

De plus, puisque les épissures peuvent être imbriquées, les identificateurs peuvent avoir des stades supérieurs à 1. La restriction d'étape peut alors être généralisée - un programme d'étape *n* peut ne pas être exécuté à n'importe quelle étape *m> n* . Par exemple, on peut voir des références à ces étapes supérieures à 1 dans certains messages d'erreur:

```
>:t [| \x -> $x |]

<interactive>:1:10: error:
    * Stage error: `x' is bound at stage 2 but used at stage 1
    * In the untyped splice: $x
      In the Template Haskell quotation [| \ x -> $x |]
```

Utilisation de Template Haskell provoque des

erreurs non liées à la portée des identificateurs sans lien?

Normalement, toutes les déclarations dans un seul module Haskell peuvent être considérées comme toutes récursives. En d'autres termes, chaque déclaration de niveau supérieur est dans le cadre de chaque autre dans un seul module. Lorsque Template Haskell est activé, les règles de portée changent - le module est plutôt divisé en groupes de code séparés par TH épissures, et chaque groupe est mutuellement récursif, et chaque groupe est dans la portée de tous les autres groupes.

Examples

Le type Q

Le constructeur de type `Q :: * -> *` défini dans `Language.Haskell.TH.Syntax` est un type abstrait représentant des calculs ayant accès à l'environnement de compilation du module dans lequel le calcul est exécuté. Le type `Q` gère également la substitution variable, appelée *capture de nom* par TH (et discuté ici). Toutes les épissures ont le type `QX` pour certains `X`

L'environnement de compilation comprend:

- des identificateurs dans la portée et des informations sur lesdits identificateurs,
    - types de fonctions
    - types et types de données source des constructeurs
    - spécification complète des déclarations de type (classes, familles de types)
- l'emplacement dans le code source (ligne, colonne, module, package) où l'épissure se produit
- fixités des fonctions (GHC 7.10)
- extensions GHC activées (GHC 8.0)

Le type `Q` a également la capacité de générer de nouveaux noms, avec la fonction `newName :: String -> Q Name` . Notez que le nom n'est lié de manière implicite, de sorte que l'utilisateur doit le lier lui-même et que l'utilisateur doit donc s'assurer que l'utilisation du nom qui en résulte est bien définie.

`Q` a des instances pour `Functor,Monad,Applicative` et c'est l'interface principale pour manipuler les valeurs `Q` , ainsi que les combinateurs fournis dans `Language.Haskell.TH.Lib` , qui définissent une fonction d'assistance pour chaque constructeur de la th TH du formulaire:

```
LitE :: Lit -> Exp
litE :: Lit -> ExpQ

AppE :: Exp -> Exp -> Exp
appE :: ExpQ -> ExpQ -> ExpQ
```

Notez que `ExpQ` , `TypeQ` , `DecsQ` et `PatQ` sont des synonymes pour les types AST qui sont

généralement stockés dans le type `Q`

La bibliothèque TH fournit une fonction `runQ :: Quasi m => Q a -> ma` , et il y a une instance `Quasi IO` , il semblerait donc que le type `Q` ne soit qu'un `IO` sophistiqué. Cependant, l'utilisation de `runQ :: Q a -> IO a` produit une `IO` action n'a *pas* accès à tout environnement de compilation - ce n'est disponible dans le réel `Q` de type. Ces actions `IO` échoueront lors de l'exécution si vous essayez d'accéder à cet environnement.

Un curry n-arité

Le familier

```
curry :: ((a,b) -> c) -> a -> b -> c
curry = \f a b -> f (a,b)
```

la fonction peut être généralisée à des tuples d'arité arbitraire, par exemple:

```
curry3 :: ((a, b, c) -> d) -> a -> b -> c -> d
curry4 :: ((a, b, c, d) -> e) -> a -> b -> c -> d -> e
```

Cependant, écrire de telles fonctions pour des tuples d'arity 2 à (par exemple) 20 à la main serait fastidieux (et ignorer le fait que la présence de 20 tuples dans votre programme signalent presque certainement des problèmes de conception qui devraient être corrigés avec des enregistrements).

Nous pouvons utiliser Template Haskell pour produire de telles fonctions `curryN` pour `n` arbitraire:

```
{-# LANGUAGE TemplateHaskell #-}
import Control.Monad (replicateM)
import Language.Haskell.TH (ExpQ, newName, Exp(..), Pat(..))
import Numeric.Natural (Natural)

curryN :: Natural -> Q Exp
```

La fonction `curryN` prend un nombre naturel et produit la fonction curry de cette arité, en tant que AST Haskell.

```
curryN n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
```

Nous produisons d'abord des variables de type *fraîches* pour chacun des arguments de la fonction - une pour la fonction d'entrée et une pour chacun des arguments de cette fonction.

```
  let args = map VarP (f:xs)
```

L'expression `args` représente le motif `f x1 x2 .. xn` . Notez qu'un modèle est une entité syntaxique distincte - nous pourrions prendre ce même modèle et le placer dans un lambda, ou une liaison de fonction, ou même le LHS d'une liaison let (ce qui serait une erreur).

```
      ntup = TupE (map VarE xs)
```

La fonction doit générer l'argument tuple à partir de la séquence d'arguments, ce que nous avons fait ici. Notez la distinction entre les variables de modèle (`VarP`) et les variables d'expression (`VarE`).

```
  return $ LamE args (AppE (VarE f) ntup)
```

Enfin, la valeur que nous produisons est l'AST `\f x1 x2 .. xn -> f (x1, x2, .. , xn)` .

Nous aurions aussi pu écrire cette fonction en utilisant des citations et des constructeurs 'levés':

```
...
import Language.Haskell.TH.Lib

curryN' :: Natural -> ExpQ
curryN' n = do
  f  <- newName "f"
  xs <- replicateM (fromIntegral n) (newName "x")
  lamE (map varP (f:xs))
       [| $(varE f) $(tupE (map varE xs)) |]
```

Notez que les citations doivent être syntaxiquement valides, donc `[| \ $(map varP (f:xs)) -> .. |]` n'est pas valide, car il n'existe aucun moyen pour Haskell de déclarer une liste de motifs - ce qui précède est interprété comme `\ var -> ..` et le l'expression `PatQ` devrait avoir le type `PatQ` , c'est-à-dire un seul motif, pas une liste de motifs.

Enfin, nous pouvons charger cette fonction TH dans GHCi:

```
>:set -XTemplateHaskell
>:t $(curryN 5)
$(curryN 5)
  :: ((t1, t2, t3, t4, t5) -> t) -> t1 -> t2 -> t3 -> t4 -> t5 -> t
>$(curryN 5) (\(a,b,c,d,e) -> a+b+c+d+e) 1 2 3 4 5
15
```

Cet exemple est adapté principalement à partir d' ici .

Syntaxe du modèle Haskell et Quasiquotes

Template Haskell est activé par l'extension `-XTemplateHaskell` GHC. Cette extension active toutes les fonctionnalités syntaxiques détaillées dans cette section. Les détails complets sur Template Haskell sont donnés dans le guide de l' utilisateur .

Épissures

- Une épissure est une nouvelle entité syntaxique activée par Template Haskell, écrite sous la forme `$(...)` , où `(...)` est une expression.

- Il ne doit pas y avoir d'espace entre `$` et le premier caractère de l'expression; et Template Haskell remplace l'analyse syntaxique de l'opérateur `$` - par exemple, `f$g` est normalement analysé en tant que `($) fg` alors qu'avec Template Haskell activé, il est analysé en tant qu'épissure.
- Lorsqu'une épissure apparaît au niveau supérieur, `$` peut être omis. Dans ce cas, l'expression épissée est la ligne entière.
- Une épissure représente le code qui est exécuté au moment de la compilation pour produire un Haskell AST, et que AST est compilé en tant que code Haskell et inséré dans le programme.
- Les épissures peuvent apparaître à la place de: expressions, modèles, types et déclarations de niveau supérieur. Le type de l'expression épissée, dans chaque cas respectivement, est `Q Exp` , `Q Pat` , `Q Type` , `Q [Decl]` . Notez que les épissures de déclaration *ne* peuvent apparaître qu'au niveau supérieur, tandis que les autres peuvent se trouver dans d'autres expressions, modèles ou types, respectivement.

Citations d'expression (note: *pas* une QuasiQuotation)

- Une citation d'expression est une nouvelle entité syntaxique écrite comme l'une des suivantes:
  - `[e|..|]` ou `[|..|]` - `..` est une expression et la citation a le type `Q Exp` ;
  - `[p|..|]` - `..` est un motif et la citation a le type `Q Pat` ;
  - `[t|..|]` - `..` est un type et la citation a le type `Q Type` ;
  - `[d|..|]` - `..` est une liste de déclarations et la citation a le type `Q [Dec]` .
- Une citation d'expression prend un programme de compilation et produit l'AST représentée par ce programme.
- L'utilisation d'une valeur dans une citation (par exemple, `\x -> [| x |]`) sans épissure correspond au sucre syntaxique pour `\x -> [| $(lift x) |]` , où `lift :: Lift t => t -> Q Exp` vient de la classe

```
class Lift t where
  lift :: t -> Q Exp
  default lift :: Data t => t -> Q Exp
```

Épissures et citations typées

- Les épissures typées sont similaires aux épissures (non typées) mentionnées précédemment et sont écrites sous la forme `$$(..)` où `(..)` est une expression.

- Si `e` a le type `Q (TExp a)` alors `$$e` a le type `a` .
- Les citations typées prennent la forme `[||..||]` où `..` est une expression de type `a` ; la citation résultante a le type `Q (TExp a)` .
- Les expressions typées peuvent être converties en expressions non typées: `unType :: TExp a -> Exp` .

QuasiQuotes

- QuasiQuotes généraliser les citations d'expression - auparavant, l'analyseur utilisé par l'expression entre guillemets fait partie d'un ensemble fixe (`e,p,t,d`), mais QuasiQuotes permet de définir un analyseur personnalisé et de le générer au moment de la compilation. Les quasi-citations peuvent apparaître dans tous les mêmes contextes que les citations régulières.
- Une quasi-citation est écrite comme `[iden|...|]` , où `iden` est un identifiant de type `Language.Haskell.TH.Quote.QuasiQuoter` .
- Un `QuasiQuoter` est simplement composé de quatre analyseurs, un pour chacun des différents contextes dans lesquels les citations peuvent apparaître:

```
data QuasiQuoter = QuasiQuoter { quoteExp  :: String -> Q Exp,
                                 quotePat  :: String -> Q Pat,
                                 quoteType :: String -> Q Type,
                                 quoteDec  :: String -> Q [Dec] }
```

Des noms

- Les identificateurs Haskell sont représentés par le type `Language.Haskell.TH.Syntax.Name` . Les noms forment les feuilles des arbres de syntaxe abstraites représentant les programmes Haskell dans Template Haskell.
- Un identifiant qui est actuellement dans la portée peut être transformé en un nom avec: `'e` ou `'T` Dans le premier cas, `e` est interprété dans la portée de l'expression, tandis que dans le second cas, `T` est dans le type scope (en rappelant que les constructeurs de types et de valeurs peuvent partager le nom sans amiguité dans Haskell).

Lire Template Haskell & QuasiQuotes en ligne:
https://riptutorial.com/fr/haskell/topic/5216/template-haskell--amp--quasiquotes

Chapitre 65: Test avec savoureux

Examples

SmallCheck, QuickCheck et HUnit

```
import Test.Tasty
import Test.Tasty.SmallCheck as SC
import Test.Tasty.QuickCheck as QC
import Test.Tasty.HUnit


main :: IO ()
main = defaultMain tests

tests :: TestTree
tests = testGroup "Tests" [smallCheckTests, quickCheckTests, unitTests]

smallCheckTests :: TestTree
smallCheckTests = testGroup "SmallCheck Tests"
  [ SC.testProperty "String length <= 3" $
      \s -> length (take 3 (s :: String)) <= 3
  , SC.testProperty "String length <= 2" $  -- should fail
      \s -> length (take 3 (s :: String)) <= 2
  ]

quickCheckTests :: TestTree
quickCheckTests = testGroup "QuickCheck Tests"
  [ QC.testProperty "String length <= 5" $
      \s -> length (take 5 (s :: String)) <= 5
  , QC.testProperty "String length <= 4" $  -- should fail
      \s -> length (take 5 (s :: String)) <= 4
  ]

unitTests :: TestTree
unitTests = testGroup "Unit Tests"
  [ testCase "String comparison 1" $
      assertEqual "description" "OK" "OK"

  , testCase "String comparison 2" $  -- should fail
      assertEqual "description" "fail" "fail!"
  ]
```

Installer des paquets:

```
cabal install tasty-smallcheck tasty-quickcheck tasty-hunit
```

Courir avec cabale:

```
cabal exec runhaskell test.hs
```

Lire Test avec savoureux en ligne: https://riptutorial.com/fr/haskell/topic/3816/test-avec-savoureux

Chapitre 66: Théorie des catégories

Examples

La théorie des catégories comme système d'organisation de l'abstraction

La théorie des catégories est une théorie mathématique moderne et une branche de l'algèbre abstraite axée sur la nature de la connectivité et de la relation. Il est utile pour donner des bases solides et un langage commun à de nombreuses abstractions de programmation hautement réutilisables. Haskell utilise la théorie des catégories comme source d'inspiration pour certaines des classes de base disponibles à la fois dans la bibliothèque standard et dans plusieurs bibliothèques tierces populaires.

Un exemple

La `Functor` Functor dit que si un type `F` instancie `Functor` (pour lequel on écrit `Functor F`), alors nous avons une opération générique

```
fmap :: (a -> b) -> (F a -> F b)
```

ce qui nous permet de "carte" sur `F` L'intuition standard (mais imparfaite) est que `F a` est un conteneur plein de valeurs de type `a` et `fmap` nous permet d'appliquer une transformation à chacun de ces éléments contenus. Un exemple est `Maybe` - `Maybe`

```
instance Functor Maybe where
  fmap f Nothing = Nothing     -- if there are no values contained, do nothing
  fmap f (Just a) = Just (f a) -- else, apply our transformation
```

Compte tenu de cette intuition, une question commune est "pourquoi ne pas appeler `Functor` quelque chose d'évident comme `Mappable` ?".

Un soupçon de théorie des catégories

La raison en est que Functor s'intègre dans un ensemble de structures communes à la théorie des catégories et donc, en appelant `Functor` "Functor", nous pouvons voir comment il se connecte à ce corpus de connaissances plus profond.

En particulier, la théorie des catégories est très concernée par l'idée des flèches d'un endroit à un autre. Dans Haskell, les flèches les plus importantes sont les flèches de fonction `a -> b` . Une chose courante à étudier dans la théorie des catégories est la relation entre un ensemble de flèches et un autre ensemble. En particulier, pour tout constructeur de type `F` , l'ensemble des flèches de la forme `F a -> F b` est également intéressant.

Ainsi , un Functor est une `F` telle qu'il ya une connexion entre Haskell normale flèches `a -> b` et `F` flèches Spécifiques `F a -> F b` . La connexion est définie par `fmap` et nous reconnaissons également quelques lois qui doivent tenir

```
forall (x :: F a) . fmap id x == x

forall (f :: a -> b) (g :: b -> c) . fmap g . fmap f = fmap (g . f)
```

Toutes ces lois découlent naturellement de l'interprétation théorique de `Functor` et ne seraient évidemment pas nécessaires si l'on pensait seulement à `Functor` comme se rapportant à la "cartographie des éléments".

Définition d'une catégorie

Une catégorie `C` comprend:

- Une collection d'objets appelée `Obj(C)` ;
- Une collection (appelée `Hom(C)`) de morphismes entre ces objets. Si `a` et `b` sont dans `Obj(C)` , alors un morphisme `f` dans `Hom(C)` est typiquement noté `f : a -> b` , et la collection de tout morphisme entre `a` et `b` est notée `hom(a,b)` ;
- Un morphisme spécial appelé morphisme de l' *identité* - pour tout `a : Obj(C)` il existe un `id : a -> a` morphisme `id : a -> a` ;
- Un opérateur de composition (`.`), Prenant deux morphismes `f : a -> b` , `g : b -> c` et produisant un morphisme `a -> c`

qui obéissent aux lois suivantes:

```
For all f : a -> x, g : x -> b, then id . f = f and g . id = g
```

```
For all f : a -> b, g : b -> c and h : c -> d, then h . (g . f) = (h . g) . f
```

En d'autres termes, la composition avec le morphisme identitaire (à gauche ou à droite) ne change pas l'autre morphisme, et la composition est associative.

Dans Haskell, la `Category` est définie comme une classe de type dans Control.Category :

```
-- | A class for categories.
--   id and (.) must form a monoid.
class Category cat where
    -- | the identity morphism
    id :: cat a a

    -- | morphism composition
    (.) :: cat b c -> cat a b -> cat a c
```

Dans ce cas, `cat :: k -> k -> *` objective la relation de morphisme - il existe un morphisme `cat ab` si et seulement si `cat ab` est habité (c'est-à-dire a une valeur). `a` , `b` et `c` sont tous dans `Obj(C)` . `Obj(C)` lui-même est représenté par le *genre* `k` - par exemple, lorsque `k ~ *` , comme c'est généralement le cas, les objets sont des types.

L'exemple canonique d'une catégorie dans Haskell est la catégorie de fonction:

```
instance Category (->) where
  id = Prelude.id
  (.) = Prelude..
```

Un autre exemple courant est la `Category` des flèches de `Kleisli` pour une `Monad` :

```
newtype Kleisli m a b = Kleisli (a -> m b)

class Monad m => Category (Kleisli m) where
  id = Kleisli return
  Kleisli f . Kleisli g = Kleisli (f >=> g)
```

Les types Haskell en tant que catégorie

Définition de la catégorie

Les types Haskell ainsi que les fonctions entre les types forment (presque †) une catégorie. Nous avons un morphisme d'identité (fonction) (`id :: a -> a`) pour chaque objet (type) `a` ; et composition des morphismes (`(.) :: (b -> c) -> (a -> b) -> a -> c`), qui obéissent aux lois de la catégorie:

```
f . id = f = id . f
h . (g . f) = (h . g) . f
```

Nous appelons généralement cette catégorie **Hask** .

Isomorphismes

En théorie des catégories, nous avons un isomorphisme lorsque nous avons un morphisme qui a un inverse, en d'autres termes, il existe un morphisme qui peut être composé avec lui pour créer l'identité. Dans **Hask,** cela revient à avoir une paire de morphismes `f` , `g` tels que:

```
 f . g == id == g . f
```

Si nous trouvons une paire de tels morphismes entre deux types, nous les appelons *isomorphes les uns aux autres* .

Un exemple de deux types isomorphes serait `((),a)` et `a` pour certains `a` . Nous pouvons construire les deux morphismes:

```
f :: ((),a) -> a
f ((),x) = x

g :: a -> ((),a)
g x = ((),x)
```

Et on peut vérifier ça `f . g == id == g . f` .

Les foncteurs

Un foncteur, en théorie des catégories, passe d'une catégorie à une autre, mappant des objets et des morphismes. Nous travaillons uniquement sur une catégorie, la catégorie **Hask** de types Haskell, nous allons donc voir uniquement les foncteurs de **Hask** à **Hask** , ces foncteurs, dont l'origine et la catégorie de destination sont les mêmes, sont appelés **endofuncteurs** . Nos endofuncteurs seront les types polymorphes prenant un type et en renvoyant un autre:

```
F :: * -> *
```

Obéir aux lois des foncteurs catégoriels (préserver les identités et la composition) équivaut à obéir aux lois du foncteur Haskell:

```
fmap (f . g) = (fmap f) . (fmap g)
fmap id = id
```

Nous avons, par exemple, que `[]` , `Maybe` ou `(-> r)` sont des foncteurs dans **Hask** .

Monades

Une monade dans la théorie des catégories est un monoïde sur la **catégorie des endofuncteurs** . Cette catégorie a des endofuncteurs en tant qu'objets `F :: * -> *` et des transformations naturelles (les transformations entre eux pour tout `forall a . F a -> G a`) en tant que morphismes.

Un objet monoïde peut être défini sur une catégorie monoïdale et est un type ayant deux morphismes:

```
zero :: () -> M
mappend :: (M,M) -> M
```

Nous pouvons traduire cela grossièrement dans la catégorie des endofuncteurs Hask comme:

```
return :: a -> m a
join :: m (m a) -> m a
```

Et obéir aux lois de la monade équivaut à obéir aux lois catégoriques des objets monoïdes.

---

† En fait, la classe de tous les types ainsi que la classe des fonctions entre les types ne forment *pas* strictement une catégorie dans Haskell, en raison de l'existence de `undefined` . En règle générale, cela est résolu en définissant simplement les objets de la catégorie **Hask en** tant que types sans valeurs inférieures, ce qui exclut les fonctions non terminantes et les valeurs infinies (codata). Pour une discussion détaillée de ce sujet, voir ici .

Produits catégoriels

En théorie des catégories, le produit de deux objets **X** , **Y** est un autre objet **Z** avec deux projections: $\pi_1: Z \to X$ et $\pi_2: Z \to Y$; de sorte que tous les deux autres morphismes d'un autre objet se décomposent uniquement à travers ces projections. En d'autres termes, s'il existe $f: W \to X$ et $f_2: W \to Y$, il existe un morphisme unique $g: W \to Z$ tel que $\pi_1 \circ g = f_1$ et $\pi_2 \circ g = f_2$.

Produits en Hask

Cela se traduit dans la catégorie **Hask** des types Haskell comme suit, `Z` est le produit de `A` , `B` lorsque:

```
-- if there are two functions
f1 :: W -> A
f2 :: W -> B
-- we can construct a unique function
g  :: W -> Z
-- and we have two projections
p1 :: Z -> A
p2 :: Z -> B
-- such that the other two functions decompose using g
p1 . g == f1
p2 . g == f2
```

Le **type de produit de deux types** `A` , `B` , qui suit la loi indiquée ci-dessus, **est le tuple** des deux types `(A,B)` , et les deux projections sont `fst` et `snd` . On peut vérifier qu'il suit la règle ci-dessus, si on a deux fonctions `f1 :: W -> A` et `f2 :: W -> B` on peut les décomposer de la manière suivante:

```
decompose :: (W -> A) -> (W -> B) -> (W -> (A,B))
decompose f1 f2 = (\x -> (f1 x, f2 x))
```

Et on peut vérifier que la décomposition est correcte:

```
fst . (decompose f1 f2) = f1
snd . (decompose f1 f2) = f2
```

Unicité jusqu'à l'isomorphisme

Le choix de `(A,B)` comme produit de `A` et `B` n'est pas unique. Un autre choix logique et équivalent aurait été:

```
data Pair a b = Pair a b
```

De plus, nous aurions pu aussi choisir `(B,A)` comme produit, ou même `(B,A,())` , et nous pourrions trouver une fonction de décomposition comme ci-dessus en suivant les règles:

```
decompose2 :: (W -> A) -> (W -> B) -> (W -> (B,A,()))
decompose2 f1 f2 = (\x -> (f2 x, f1 x, ()))
```

C'est parce que le produit n'est pas unique mais *unique jusqu'à l'isomorphisme* . Tous les deux produits de `A` et `B` ne doivent pas nécessairement être égaux, mais ils doivent être isomorphes. Par exemple, les deux produits que nous venons de définir `(A,B)` et `(B,A,())` sont isomorphes:

```
iso1 :: (A,B) -> (B,A,())
iso1 (x,y) = (y,x,())

iso2 :: (B,A,()) -> (A,B)
iso2 (y,x,()) = (x,y)
```

Unicité de la décomposition

Il est important de noter que la fonction de décomposition doit également être unique. Il existe des types qui suivent toutes les règles requises pour être produit, mais la décomposition n'est pas unique. Par exemple, nous pouvons essayer d'utiliser `(A,(B,Bool))` avec des projections du `fst fst . snd` tant que produit de `A` et `B` :

```
decompose3 :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3 f1 f2 = (\x -> (f1 x, (f2 x, True)))
```

Nous pouvons vérifier que cela fonctionne:

```
fst         . (decompose3 f1 f2) = f1 x
(fst . snd) . (decompose3 f1 f2) = f2 x
```

Mais le problème ici est que nous aurions pu écrire une autre décomposition, à savoir:

```
decompose3' :: (W -> A) -> (W -> B) -> (W -> (A,(B,Bool)))
decompose3' f1 f2 = (\x -> (f1 x, (f2 x, False)))
```

Et, comme la décomposition n'est **pas unique** , `(A,(B,Bool))` n'est **pas** le produit de `A` et `B` dans **Hask**

Coproduit de types en Hask

Intuition

Le produit catégoriel de deux types **A** et **B** doit contenir les informations minimales nécessaires pour contenir une instance de type **A** ou **B.** Nous pouvons voir maintenant que le coproduit intuitif de deux types devrait être l'un `Either ab` l' `Either ab` . D'autres candidats, tels que `Either a (b,Bool)` , contiendraient une partie des informations inutiles et ne seraient pas minimes.

La définition formelle est dérivée de la définition catégorique du coproduit.

Coproduits catégoriels

Un coproduit catégorique est la double notion de produit catégorique. Il est obtenu directement en inversant toutes les flèches dans la définition du produit. Le coproduit de deux objets **X** , **Y** est un autre objet **Z** avec deux inclusions: **i_1: X ⟶ Z** et **i_2: Y ⟶ Z** ; de telle sorte que tous les deux morphismes de **X** et **Y** à un autre objet se décomposent uniquement à travers ces inclusions. En d'autres termes, s'il y a deux morphismes $\mathbf{f_1}$**: X ⟶ W** et $\mathbf{f_2}$**: Y ⟶ W** , il existe un morphisme unique **g: Z ⟶ W** tel que $\mathbf{g} \circ \mathbf{i_1} = \mathbf{f_1}$ et $\mathbf{g} \circ \mathbf{i_2} = \mathbf{f_2}$

Coproduits en Hask

La traduction dans la catégorie **Hask** est similaire à la traduction du produit:

```
-- if there are two functions
f1 :: A -> W
f2 :: B -> W
-- and we have a coproduct with two inclusions
i1 :: A -> Z
i2 :: B -> Z
-- we can construct a unique function
g  :: Z -> W
-- such that the other two functions decompose using g
g . i1 == f1
g . i2 == f2
```

Le type de coproduit de deux types `A` et `B` dans **Hask** est `Either ab` ou tout autre type isomorphe à celui-ci:

```
-- Coproduct
-- The two inclusions are Left and Right
data Either a b = Left a | Right b

-- If we have those functions, we can decompose them through the coproduct
decompose :: (A -> W) -> (B -> W) -> (Either A B -> W)
decompose f1 f2 (Left x)  = f1 x
decompose f1 f2 (Right y) = f2 y
```

Haskell Applicative en termes de théorie des catégories

Un `Functor` de Haskell permet de mapper n'importe quel type `a` (un objet de **Hask**) à un type `F a` et de mapper une fonction `a -> b` (un morphisme de **Hask**) sur une fonction de type `F a -> F b` . Cela correspond à une définition de théorie de catégorie dans un sens que functor préserve la structure de catégorie de base.

Une **catégorie monoïdale** est une catégorie qui a une structure *supplémentaire* :

- Un produit tenseur (voir Produit de types dans Hask)
- Une unité de tenseur (objet unitaire)

En prenant une paire comme notre produit, cette définition peut être traduite en Haskell de la manière suivante:

```
class Functor f => Monoidal f where
    mcat :: f a -> f b -> f (a,b)
    munit :: f ()
```

La classe `Applicative` est équivalente à celle du `Monoidal` et peut donc être implémentée en termes de:

```
instance Monoidal f => Applicative f where
    pure x = fmap (const x) munit
    f <*> fa = (\(f, a) -> f a) <$> (mcat f fa)
```

Lire Théorie des catégories en ligne: https://riptutorial.com/fr/haskell/topic/2261/theorie-des-categories

Chapitre 67: Traversable

Introduction

La classe `Traversable` généralise la fonction anciennement appelée `mapM :: Monad m => (a -> mb) -> [a] -> m [b]` pour travailler avec des effets `Applicative` sur des structures autres que des listes.

Examples

Functor Instanciation et Pliable pour une structure Traversable

```
import Data.Traversable as Traversable

data MyType a =  -- ...
instance Traversable MyType where
    traverse = -- ...
```

Chaque `Traversable` structure peut être un `Foldable` `Functor` en utilisant les `fmapDefault` et `foldMapDefault` fonctions trouvées dans `Data.Traversable` .

```
instance Functor MyType where
    fmap = Traversable.fmapDefault

instance Foldable MyType where
    foldMap = Traversable.foldMapDefault
```

`fmapDefault` est défini en exécutant `traverse` dans le foncteur applicatif d' `Identity` .

```
newtype Identity a = Identity { runIdentity :: a }

instance Applicative Identity where
    pure = Identity
    Identity f <*> Identity x = Identity (f x)

fmapDefault :: Traversable t => (a -> b) -> t a -> t b
fmapDefault f = runIdentity . traverse (Identity . f)
```

`foldMapDefault` est défini à l'aide du foncteur applicatif `Const` , qui ignore son paramètre tout en accumulant une valeur monoïdale.

```
newtype Const c a = Const { getConst :: c }

instance Monoid m => Applicative (Const m) where
    pure _ = Const mempty
    Const x <*> Const y = Const (x `mappend` y)

foldMapDefault :: (Traversable t, Monoid m) => (a -> m) -> t a -> m
foldMapDefault f = getConst . traverse (Const . f)
```

Une instance de Traversable pour un arbre binaire

Les implémentations de `traverse` ressemblent généralement à une implémentation de `fmap` soulevée dans un contexte `Applicative` .

```
data Tree a = Leaf
            | Node (Tree a) a (Tree a)

instance Traversable Tree where
    traverse f Leaf = pure Leaf
    traverse f (Node l x r) = Node <$> traverse f l <*> f x <*> traverse f r
```

Cette implémentation effectue une traversée dans l'ordre de l'arborescence.

```
ghci> let myTree = Node (Node Leaf 'a' Leaf) 'b' (Node Leaf 'c' Leaf)

--    +--'b'--+
--    |       |
-- +-'a'-+ +-'c'-+
-- |     | |     |
-- *     * *     *

ghci> traverse print myTree
'a'
'b'
'c'
```

La `DeriveTraversable` extension permet de générer GHC `Traversable` cas en fonction de la structure du type. Nous pouvons modifier l'ordre du parcours écrit en ajustant la disposition du constructeur de `Node` .

```
data Inorder a = ILeaf
               | INode (Inorder a) a (Inorder a)  -- as before
               deriving (Functor, Foldable, Traversable)  -- also using DeriveFunctor and
DeriveFoldable

data Preorder a = PrLeaf
                | PrNode a (Preorder a) (Preorder a)
                deriving (Functor, Foldable, Traversable)

data Postorder a = PoLeaf
                 | PoNode (Postorder a) (Postorder a) a
                 deriving (Functor, Foldable, Traversable)

-- injections from the earlier Tree type
inorder :: Tree a -> Inorder a
inorder Leaf = ILeaf
inorder (Node l x r) = INode (inorder l) x (inorder r)

preorder :: Tree a -> Preorder a
preorder Leaf = PrLeaf
preorder (Node l x r) = PrNode x (preorder l) (preorder r)

postorder :: Tree a -> Postorder a
postorder Leaf = PoLeaf
postorder (Node l x r) = PoNode (postorder l) (postorder r) x
```

```
ghci> traverse print (inorder myTree)
'a'
'b'
'c'
ghci> traverse print (preorder myTree)
'b'
'a'
'c'
ghci> traverse print (postorder myTree)
'a'
'c'
'b'
```

Traverser une structure en sens inverse

Un parcours peut être exécuté dans le sens opposé à l'aide du foncteur applicatif `Backwards` , qui retourne une application existante afin que les effets composés se produisent dans l'ordre inverse.

```
newtype Backwards f a = Backwards { forwards :: f a }

instance Applicative f => Applicative (Backwards f) where
    pure = Backwards . pure
    Backwards ff <*> Backwards fx = Backwards ((\x f -> f x) <$> fx <*> ff)
```

`Backwards` peuvent être utilisés dans une " `traverse` inversée". Lorsque le sous-jacent d'un applicatif `traverse` appel est retourné avec `Backwards` , l'effet résultant se produit dans l' ordre inverse.

```
newtype Reverse t a = Reverse { getReverse :: t a }

instance Traversable t => Traversable (Reverse t) where
    traverse f = fmap Reverse . forwards . traverse (Backwards . f) . getReverse

ghci> traverse print (Reverse "abc")
'c'
'b'
'a'
```

Le newtype `Reverse` se trouve sous Data.Functor.Reverse.

Définition de Traversable

```
class (Functor t, Foldable t) => Traversable t where
    {-# MINIMAL traverse | sequenceA #-}

    traverse :: Applicative f => (a -> f b) -> t a -> f (t b)
    traverse f = sequenceA . fmap f

    sequenceA :: Applicative f => t (f a) -> f (t a)
    sequenceA = traverse id

    mapM :: Monad m => (a -> m b) -> t a -> m (t b)
    mapM = traverse

    sequence :: Monad m => t (m a) -> m (t a)
```

```
    sequence = sequenceA
```

`Traversable` structures `t` sont des conteneurs finitaires d'éléments `a` qui peuvent être exploités avec une effectful opération « visiteur ». La fonction visiteur `f :: a -> fb` exerce un effet secondaire sur chaque élément de la structure et `traverse` les effets secondaires en utilisant `Applicative` . Une autre façon de regarder ce que `sequenceA` dit `Traversable` structures font la navette avec `Applicative` s.

Transformer une structure traversable à l'aide d'un paramètre d'accumulation

Les deux fonctions `mapAccum` combinent les opérations de pliage et de mappage.

```
--                                                    A Traversable structure
--                                                                  |
--                                                 A seed value     |
--                                                         |        |
--                                                        |-|   |---|
mapAccumL, mapAccumR :: Traversable t => (a -> b -> (a, c)) -> a -> t b -> (a, t c)
--                                       |------------------|              |--------|
--                                                  |                           |
--                        A folding function which produces a new mapped        |
--                          element 'c' and a new accumulator value 'a'         |
--                                                                              |
--                                                          Final accumulator value
--                                                           and mapped structure
```

Ces fonctions généralisent `fmap` en ce sens qu'elles permettent aux valeurs mappées de dépendre de ce qui s'est passé plus tôt dans le pli. Ils généralisent `foldl` / `foldr` en ce qu'ils cartographient la structure en place et la réduisent à une valeur.

Par exemple, les `tails` peuvent être implémentées en utilisant `mapAccumR` et ses `inits` sœurs peuvent être implémentées en utilisant `mapAccumL` .

```
tails, inits :: [a] -> [[a]]
tails = uncurry (:) . mapAccumR (\xs x -> (x:xs, xs)) []
inits = uncurry snoc . mapAccumL (\xs x -> (x `snoc` xs, xs)) []
    where snoc x xs = xs ++ [x]

ghci> tails "abc"
["abc", "bc", "c", ""]
ghci> inits "abc"
["", "a", "ab", "abc"]
```

`mapAccumL` est implémenté en traversant le foncteur applicatif d' `State` .

```
{-# LANGUAGE DeriveFunctor #-}

newtype State s a = State { runState :: s -> (s, a) } deriving Functor
instance Applicative (State s) where
    pure x = State $ \s -> (s, x)
    State ff <*> State fx = State $ \s -> let (t, f) = ff s
                                              (u, x) = fx t
                                          in (u, f x)
```

```
mapAccumL f z t = runState (traverse (State . flip f) t) z
```

`mapAccumR` fonctionne en exécutant `mapAccumL` en sens inverse .

```
mapAccumR f z = fmap getReverse . mapAccumL f z . Reverse
```

Structures traversables en tant que formes avec contenu

Si un type `t` est `Traversable` alors les valeurs de `ta` peuvent être divisées en deux parties: leur "forme" et leur "contenu":

```
data Traversed t a = Traversed { shape :: t (), contents :: [a] }
```

où le "contenu" est le même que celui que vous "visitiez" en utilisant une instance `Foldable` .

Franchir une direction, de `ta` à `Traversed ta` ne nécessite rien mais `Functor` et `Foldable`

```
break :: (Functor t, Foldable t) => t a -> Traversed t a
break ta = Traversed (fmap (const ()) ta) (toList ta)
```

mais revenir en arrière utilise la fonction de `traverse` cruciale

```
import Control.Monad.State

-- invariant: state is non-empty
pop :: State [a] a
pop = state $ \(a:as) -> (a, as)

recombine :: Traversable t => Traversed t a -> t a
recombine (Traversed s c) = evalState (traverse (const pop) s) c
```

Les lois `Traversable` exigent cette `break . recombine` et `recombine . break` est à la fois identité. Notamment, cela signifie qu'il y a exactement le bon nombre d'éléments de `contents` dans le `contents` pour remplir complètement la `shape` sans aucun reste.

`Traversed t` est `Traversable` lui-même. L'implémentation de `traverse` fonctionne en visitant les éléments en utilisant l'instance de `Traversable` de la liste, puis en rattachant la forme inerte au résultat.

```
instance Traversable (Traversed t) where
    traverse f (Traversed s c) = fmap (Traversed s) (traverse f c)
```

Transposer une liste de listes

Notant que `zip` transpose un tuple de listes dans une liste de tuples,

```
ghci> uncurry zip ([1,2],[3,4])
[(1,3), (2,4)]
```

et la similitude entre les types de `transpose` et de `sequenceA` ,

```
-- transpose exchanges the inner list with the outer list
--           +---+-->--+-+
--           |   |     | |
transpose :: [[a]] -> [[a]]
--            | |     |   |
--            +-+-->--+---+

-- sequenceA exchanges the inner Applicative with the outer Traversable
--                                            +------>------+
--                                            |             |
sequenceA :: (Traversable t, Applicative f) => t (f a) -> f (t a)
--                                              |       |
--                                              +--->---+
```

L'idée est d'utiliser la structure `Traversable` et `Applicative []` pour déployer `sequenceA` comme une sorte de *`zip` n-aire* , en regroupant toutes les listes internes ensemble de manière ponctuelle.

`[]` De » par défaut « choix prioritaire » `Applicative` exemple ne convient pas à notre utilisation - nous avons besoin d' un « Zippy » `Applicative` . Pour cela, nous utilisons le nouveau type `ZipList` , trouvé dans `Control.Applicative` .

```
newtype ZipList a = ZipList { getZipList :: [a] }

instance Applicative ZipList where
    pure x = ZipList (repeat x)
    ZipList fs <*> ZipList xs = ZipList (zipWith ($) fs xs)
```

Maintenant, nous obtenons gratuitement la `transpose` en parcourant l' `Applicative ZipList` .

```
transpose :: [[a]] -> [[a]]
transpose = getZipList . traverse ZipList

ghci> let myMatrix = [[1,2,3],[4,5,6],[7,8,9]]
ghci> transpose myMatrix
[[1,4,7],[2,5,8],[3,6,9]]
```

Lire Traversable en ligne: https://riptutorial.com/fr/haskell/topic/754/traversable

Chapitre 68: Trous dactylographiés

Remarques

L'un des points forts de Haskell est la possibilité de tirer parti du système de types pour modéliser des parties de votre domaine problématique dans le système de types. Ce faisant, on rencontre souvent des types très complexes. Lorsque vous écrivez des programmes avec ces types (c.-à-d. Avec des valeurs ayant ces types), il devient parfois presque impossible de jongler avec tous les types. À partir du GHC 7.8, il existe une nouvelle fonctionnalité syntaxique appelée trous typés. Les trous typés ne modifient pas la sémantique de la langue principale; ils sont uniquement destinés à aider à écrire des programmes.

Pour une explication détaillée des trous dactylographiés, ainsi qu'une discussion sur la conception des trous tapés, voir le wiki Haskell .

Section du guide de l'utilisateur du GHC sur les trous dactylographiés .

Examples

Syntaxe des trous tapés

Un trou dactylographié est un seul trait de soulignement (_) ou un identifiant Haskell valide qui n'est pas dans la portée, dans un contexte d'expression. Avant l'existence de trous typés, ces deux éléments déclencheraient une erreur. La nouvelle syntaxe n'interfère donc pas avec une ancienne syntaxe.

Contrôle du comportement des trous tapés

Le comportement par défaut des trous typés consiste à générer une erreur de compilation lorsqu'un trou est détecté. Cependant, il existe plusieurs indicateurs pour affiner leur comportement. Ces indicateurs sont résumés comme suit (trac GHC):

> Par défaut, GHC a activé les trous de saisie et génère une erreur de compilation lorsqu'il rencontre un trou tapé.
>
> Lorsque l' `-fdefer-type-errors` **ou** `-fdefer-typed-holes` est activée, les erreurs de trous sont converties en avertissements et entraînent des erreurs d'exécution lors de l'évaluation.
>
> Le drapeau d'avertissement `-fwarn-typed-holes` est `-fwarn-typed-holes` par défaut. Sans `-fdefer-type-errors` ou `-fdefer-typed-holes` cet indicateur est un no-op, car les trous tapés sont une erreur dans ces conditions. Si l'un des indicateurs de report est activé (la conversion des erreurs de trous typées en avertissements), l' `-fno-warn-typed-holes` désactive les avertissements. Cela signifie que la compilation réussit silencieusement

et que l'évaluation d'un trou produira une erreur d'exécution.

Sémantique des trous tapés

La valeur d'un trou de type peut simplement être `undefined` comme `undefined` , bien qu'un trou typé déclenche une erreur de compilation, il n'est donc pas strictement nécessaire de lui attribuer une valeur. Cependant, un trou de frappe (lorsqu'ils sont activés) produit une erreur de compilation (ou un avertissement avec des erreurs de type différées) qui indique le nom du trou typé, son type le *plus général* inféré et les types de liaisons locales. Par exemple:

```
Prelude> \x -> _var + length (drop 1 x)

<interactive>:19:7: Warning:
    Found hole `_var' with type: Int
    Relevant bindings include
      x :: [a] (bound at <interactive>:19:2)
      it :: [a] -> Int (bound at <interactive>:19:1)
    In the first argument of `(+)', namely `_var'
    In the expression: _var + length (drop 1 x)
    In the expression: \ x -> _var + length (drop 1 x)
```

Notez que dans le cas des trous typés dans les expressions entrées dans le rempl GHCi (comme ci - dessus), le type de l'expression saisie a également signalé, comme `it` (ici de type `[a] -> Int`).

Utilisation de trous tapés pour définir une instance de classe

Les trous typés peuvent faciliter la définition des fonctions grâce à un processus interactif.

Supposons que vous souhaitiez définir une instance de classe `Foo Bar` (pour votre type de `Bar` personnalisé, afin de pouvoir l'utiliser avec une fonction de bibliothèque polymorphe nécessitant une instance `Foo`). Vous devriez maintenant rechercher la documentation de `Foo` , déterminer quelles méthodes vous devez définir, examiner leurs types, etc. - mais avec les trous tapés, vous pouvez en fait passer outre!

Tout d'abord, définissez une instance factice:

```
instance Foo Bar where
```

Le compilateur va maintenant se plaindre

```
Bar.hs:13:10: Warning:
No explicit implementation for
  ‘foom’ and ‘quun’
In the instance declaration for ‘Foo Bar’
```

Ok, nous devons donc définir `foom` pour `Bar` . Mais qu'est - ce que c'est même supposé être? Encore une fois, nous sommes trop paresseux pour regarder dans la documentation, et il suffit de demander au compilateur:

```
instance Foo Bar where
```

```
  foom = _
```

Ici, nous avons utilisé un trou dactylographié comme une simple «requête de documentation». Les sorties du compilateur

```
Bar.hs:14:10:
    Found hole ‘_’ with type: Bar -> Gronk Bar
    Relevant bindings include
      foom :: Bar -> Gronk Bar (bound at Foo.hs:4:28)
    In the expression: _
    In an equation for ‘foom’: foom = _
    In the instance declaration for ‘Foo Bar’
```

Notez comment le compilateur a déjà rempli la variable de type de classe avec le type de béton `Bar` lequel nous voulons l'instancier. Cela peut rendre la signature beaucoup plus facile à comprendre que celle polymorphe trouvée dans la documentation de la classe, surtout si vous utilisez une méthode plus compliquée, par exemple une classe de type multi-paramètres.

Mais qu'est-ce que c'est que `Gronk` ? À ce stade, c'est probablement une bonne idée de demander à Hayoo . Cependant , nous pouvons encore sortir sans cela: comme une supposition aveugle, nous supposons que cela est non seulement un constructeur de type , mais aussi le constructeur de valeur unique, à savoir qu'il peut être utilisé en fonction qui produira en quelque sorte une `Gronk a` valeur. Nous essayons donc

```
instance Foo Bar where
  foom bar = _ Gronk
```

Si nous avons de la chance, `Gronk` est en fait une valeur, et le compilateur va maintenant dire

```
    Found hole ‘_’
      with type: (Int -> [(Int, b0)] -> Gronk b0) -> Gronk Bar
    Where: ‘b0’ is an ambiguous type variable
```

Ok, c'est moche - tout d'abord notez que `Gronk` a deux arguments, donc nous pouvons affiner notre tentative:

```
instance Foo Bar where
  foom bar = Gronk _ _
```

Et maintenant c'est assez clair:

```
    Found hole ‘_’ with type: [(Int, Bar)]
    Relevant bindings include
      bar :: Bar (bound at Bar.hs:14:29)
      foom :: Bar -> Gronk Bar (bound at Foo.hs:15:24)
    In the second argument of ‘Gronk’, namely ‘_’
    In the expression: Gronk _ _
    In an equation for ‘foom’: foom bar = Gronk _ _
```

Vous pouvez maintenant continuer à progresser, par exemple en déconstruisant la valeur de la

`bar` (les composants apparaîtront alors, avec les types, dans la section `Relevant bindings`). Souvent, il est tout à fait évident que la définition sera correcte, car vous voyez tous les arguments disponibles et les types s'emboîtent comme un puzzle. Ou bien, vous pouvez voir que la définition est *impossible* et pourquoi.

Tout cela fonctionne mieux dans un éditeur avec une compilation interactive, par exemple Emacs en mode haskell. Vous pouvez ensuite utiliser des trous tapés de la même manière que les requêtes de valeur de souris dans un IDE pour un langage impératif dynamique interprété, mais sans toutes les limitations.

Lire Trous dactylographiés en ligne: https://riptutorial.com/fr/haskell/topic/4913/trous-dactylographies

Chapitre 69: Tuples (paires, triples, ...)

Remarques

- Haskell ne prend pas en charge les tuples avec un composant natif.
- Les unités (écrit `()`) peuvent être comprises comme des tuples avec des composants nuls.
- Il n'y a pas de fonctions prédéfinies pour extraire les composants de tuples avec plus de deux composants. Si vous estimez avoir besoin de telles fonctions, envisagez d'utiliser un type de données personnalisé avec des étiquettes d'enregistrement au lieu du type de tuple. Vous pouvez ensuite utiliser les étiquettes d'enregistrement pour extraire les composants.

Examples

Construire des valeurs de tuple

Utilisez des parenthèses et des virgules pour créer des tuples. Utilisez une virgule pour créer une paire.

```
(1, 2)
```

Utilisez plus de virgules pour créer des tuples avec plus de composants.

```
(1, 2, 3)

(1, 2, 3, 4)
```

Notez qu'il est également possible de déclarer les tuples sous leur forme non gravée.

```
(,) 1 2     -- equivalent to (1,2)
(,,) 1 2 3  -- equivalent to (1,2,3)
```

Les tuples peuvent contenir des valeurs de différents types.

```
("answer", 42, '?')
```

Les tuples peuvent contenir des valeurs complexes telles que des listes ou plusieurs tuples.

```
([1, 2, 3], "hello", ('A', 65))

(1, (2, (3, 4), 5), 6)
```

Écrire des types de tuple

Utilisez des parenthèses et des virgules pour écrire les types de tuple. Utilisez une virgule pour

écrire un type de paire.

```
(Int, Int)
```

Utilisez plus de virgules pour écrire des types de tuple avec plus de composants.

```
(Int, Int, Int)

(Int, Int, Int, Int)
```

Les tuples peuvent contenir des valeurs de différents types.

```
(String, Int, Char)
```

Les tuples peuvent contenir des valeurs complexes telles que des listes ou plusieurs tuples.

```
([Int], String, (Char, Int))

(Int, (Int, (Int, Int), Int), Int)
```

Match de motif sur les tuples

La correspondance de motif sur les tuples utilise les constructeurs de tuple. Pour correspondre à une paire par exemple, nous utiliserions le constructeur `(,)` :

```
myFunction1 (a, b) = ...
```

Nous utilisons plus de virgules pour faire correspondre les tuples avec plus de composants:

```
myFunction2 (a, b, c) = ...

myFunction3 (a, b, c, d) = ...
```

Les modèles de tuples peuvent contenir des motifs complexes tels que des motifs de liste ou plusieurs motifs de tuples.

```
myFunction4 ([a, b, c], d, e) = ...

myFunction5 (a, (b, (c, d), e), f) = ...
```

Extraire les composants du tuple

Utilisez les fonctions `fst` et `snd` (de `Prelude` ou `Data.Tuple`) pour extraire le premier et le second composant des paires.

```
fst (1, 2) -- evaluates to 1

snd (1, 2) -- evaluates to 2
```

Ou utilisez la correspondance de motif.

```
case (1, 2) of (result, _) => result -- evaluates to 1

case (1, 2) of (_, result) => result -- evaluates to 2
```

La correspondance de motif fonctionne également pour les tuples comportant plus de deux composants.

```
case (1, 2, 3) of (result, _, _) => result -- evaluates to 1

case (1, 2, 3) of (_, result, _) => result -- evaluates to 2

case (1, 2, 3) of (_, _, result) => result -- evaluates to 3
```

Haskell ne fournit pas de fonctions standard telles que `fst` ou `snd` pour les tuples avec plus de deux composants. La bibliothèque de `tuple` sur Hackage fournit de telles fonctions dans le module `Data.Tuple.Select` .

Appliquer une fonction binaire à un tuple

Utilisez la fonction `uncurry` (à partir de `Prelude` ou `Data.Tuple`) pour convertir une fonction binaire en fonction sur des tuples.

```
uncurry (+) (1, 2) -- computes 3

uncurry map (negate, [1, 2, 3]) -- computes [-1, -2, -3]

uncurry uncurry ((+), (1, 2)) -- computes 3

map (uncurry (+)) [(1, 2), (3, 4), (5, 6)] -- computes [3, 7, 11]

uncurry (curry f) -- computes the same as f
```

Appliquer une fonction tuple à deux arguments (currying)

Utilisez la fonction `curry` (de `Prelude` ou `Data.Tuple`) pour convertir une fonction qui prend des tuples en une fonction qui prend deux arguments.

```
curry fst 1 2 -- computes 1

curry snd 1 2 -- computes 2

curry (uncurry f) -- computes the same as f

import Data.Tuple (swap)
curry swap 1 2 -- computes (2, 1)
```

Composants de la paire de swap

Utilisez `swap` (à partir de `Data.Tuple`) pour échanger les composants d'une paire.

```
import Data.Tuple (swap)
swap (1, 2) -- evaluates to (2, 1)
```

Ou utilisez la correspondance de motif.

```
case (1, 2) of (x, y) => (y, x) -- evaluates to (2, 1)
```

Strict de la correspondance d'un tuple

Le modèle `(p1, p2)` est strict dans le constructeur de tuple le plus externe, ce qui peut conduire à un comportement de rigidité inattendu . Par exemple, l'expression suivante diverge (à l'aide de `Data.Function.fix`):

```
fix $ \(x, y) -> (1, 2)
```

puisque la correspondance à `(x, y)` est stricte dans le constructeur de tuple. Cependant, l'expression suivante, utilisant un modèle irréfutable , est évaluée à `(1, 2)` comme prévu:

```
fix $ \ ~(x, y) -> (1, 2)
```

Lire Tuples (paires, triples, ...) en ligne: https://riptutorial.com/fr/haskell/topic/5342/tuples--paires--triples------

Chapitre 70: Type d'application

Introduction

`TypeApplications` est une alternative à la `TypeApplications` des *annotations* lorsque le compilateur a du mal à déduire des types pour une expression donnée.

Cette série d'exemples expliquera le but de l'extension `TypeApplications` et son utilisation

N'oubliez pas d'activer l'extension en plaçant `{-# LANGUAGE TypeApplications #-}` en haut de votre fichier source.

Examples

Éviter les annotations de type

Nous utilisons des annotations de type pour éviter toute ambiguïté. Les applications de type peuvent être utilisées dans le même but. Par exemple

```
x :: Num a => a
x = 5

main :: IO ()
main = print x
```

Ce code a une erreur d'ambiguïté. Nous savons que `a` a un `Num` exemple, et pour l' imprimer nous savons qu'il a besoin d' un `Show` par exemple. Cela pourrait fonctionner si `a` était, par exemple, un `Int` , afin de corriger l'erreur, nous pouvons ajouter une annotation de type

```
main = print (x :: Int)
```

Une autre solution utilisant des applications de type ressemblerait à ceci

```
main = print @Int x
```

Pour comprendre ce que cela signifie, nous devons examiner le type de signature d' `print` .

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

La fonction prend un paramètre de type `a` , mais une autre façon de le voir est de prendre deux paramètres. Le premier est un paramètre de *type* , le second est une valeur dont le type est le premier paramètre.

La principale différence entre les paramètres de valeur et les paramètres de type est que ces derniers sont implicitement fournis aux fonctions lorsque nous les appelons. Qui les fournit? L'algorithme d'inférence de type! Ce que `TypeApplications` nous permet de faire est de donner

explicitement ces paramètres de type. Ceci est particulièrement utile lorsque l'inférence de type ne peut pas déterminer le type correct.

Donc, pour briser l'exemple ci-dessus

```
print :: Show a => a -> IO ()
print @Int :: Int -> IO ()
print @Int x :: IO ()
```

Tapez les applications dans d'autres langues

Si vous êtes familier avec des langages comme Java, C # ou C ++ et le concept de génériques / modèles, cette comparaison pourrait vous être utile.

Disons que nous avons une fonction générique en C #

```
public static T DoNothing<T>(T in) { return in; }
```

Pour appeler cette fonction avec un `float` nous pouvons faire `DoNothing(5.0f)` ou si nous voulons être explicites, nous pouvons dire `DoNothing<float>(5.0f)` . Cette partie à l'intérieur des crochets est l'application de type.

Dans Haskell, c'est la même chose, sauf que les paramètres de type ne sont pas seulement implicites aux sites d'appel, mais également aux sites de définition.

```
doNothing :: a -> a
doNothing x = x
```

Cela peut également être rendu explicite en utilisant soit les `ScopedTypeVariables` , `Rank2Types` ou `RankNTypes` comme ceci.

```
doNothing :: forall a. a -> a
doNothing x = x
```

Ensuite, sur le site d'appel, nous pouvons à nouveau écrire `doNothing 5.0` ou `doNothing @Float 5.0`

Ordre des paramètres

Le problème des arguments de type implicites devient évident lorsque nous en avons plus d'un. De quel ordre viennent-ils?

```
const :: a -> b -> a
```

Est-ce que l'écriture de `const @Int` signifie `a` est égal à `Int` ou est-ce `b` ? Dans le cas où nous déclarons explicitement les paramètres de type en utilisant un `forall` comme `const :: forall a b. a -> b -> a` alors l'ordre est comme écrit: `a` , alors `b` .

Si nous ne le faisons pas, l'ordre des variables est de gauche à droite. La première variable à

mentionner est le premier paramètre de type, le second est le paramètre de deuxième type, etc.

Que faire si nous voulons spécifier la variable de deuxième type, mais pas la première? Nous pouvons utiliser un caractère générique pour la première variable comme celle-ci

```
const @_ @Int
```

Le type de cette expression est

```
const @_ @Int :: a -> Int -> a
```

Interaction avec des types ambigus

Disons que vous introduisez une classe de types qui ont une taille en octets.

```
class SizeOf a where
    sizeOf :: a -> Int
```

Le problème est que la taille doit être constante pour chaque valeur de ce type. Nous ne voulons pas réellement que la fonction `sizeOf` dépende de `a` , mais seulement de son type.

Sans les applications de type, la meilleure solution était le type de `Proxy` défini comme ceci

```
data Proxy a = Proxy
```

Le but de ce type est de transporter des informations de type, mais aucune information de valeur. Alors notre classe pourrait ressembler à ceci

```
class SizeOf a where
    sizeOf :: Proxy a -> Int
```

Maintenant, vous vous demandez peut-être pourquoi ne pas laisser tomber le premier argument? Le type de notre fonction serait alors simplement `sizeOf :: Int` ou, pour être plus précis, parce que c'est une méthode d'une classe, `sizeOf :: SizeOf a => Int` ou pour être encore plus explicite `sizeOf :: forall a. SizeOf a => Int` .

Le problème est l'inférence de type. Si j'écris quelque part `sizeOf` , l'algorithme d'inférence sait seulement que j'attends un `Int` . Il n'a aucune idée de quel type je veux substituer à `a` . De ce fait, la définition est rejetée par le compilateur à *moins* que l'extension `{-# LANGUAGE AllowAmbiguousTypes #-}` activée. Dans ce cas, la définition est compilée, elle ne peut être utilisée nulle part sans erreur d'ambiguïté.

Heureusement, l'introduction d'applications de type sauve la journée! Maintenant, nous pouvons écrire `sizeOf @Int` , en disant explicitement que `a` est `Int` . Les applications de type nous permettent de fournir un paramètre de type, même s'il n'apparaît pas dans les *paramètres réels de la fonction* !

Lire Type d'application en ligne: https://riptutorial.com/fr/haskell/topic/10767/type-d-application

Chapitre 71: Type de familles

Examples

Type Synonyme Familles

Les familles de types synonyme ne sont que des fonctions de type: elles associent les types de paramètres aux types de résultats. Ceux-ci viennent dans trois variétés différentes.

Familles de type synonyme fermé

Celles-ci fonctionnent beaucoup comme les fonctions Haskell ordinaires au niveau de la valeur: vous spécifiez des clauses, en mappant certains types à d'autres:

```
{-# LANGUAGE TypeFamilies #-}
type family Vanquisher a where
    Vanquisher Rock = Paper
    Vanquisher Paper = Scissors
    Vanquisher Scissors = Rock

data Rock=Rock; data Paper=Paper; data Scissors=Scissors
```

Familles ouvertes de type synonyme

Celles-ci fonctionnent plus comme des instances de type typeclass: tout le monde peut ajouter d'autres clauses dans d'autres modules.

```
type family DoubledSize w

type instance DoubledSize Word16 = Word32
type instance DoubledSize Word32 = Word64
-- Other instances might appear in other modules, but two instances cannot overlap
-- in a way that would produce different results.
```

Synonymes de type associé à la classe

Une famille de type ouvert peut également être combinée avec une classe réelle. Cela se fait généralement lorsque, comme avec les familles de données associées , une méthode de classe nécessite des objets auxiliaires supplémentaires, et que ces objets *peuvent* être différents pour différentes instances, mais peuvent également être partagés. Un bon exemple est la classe `VectorSpace` :

```
class VectorSpace v where
  type Scalar v :: *
```

```
  (*^) :: Scalar v -> v -> v

instance VectorSpace Double where
  type Scalar Double = Double
  μ *^ n = μ * n

instance VectorSpace (Double,Double) where
  type Scalar (Double,Double) = Double
  μ *^ (n,m) = (μ*n, μ*m)

instance VectorSpace (Complex Double) where
  type Scalar (Complex Double) = Complex Double
  μ *^ n = μ*n
```

Notez que dans les deux premières instances, l'implémentation de `Scalar` est la même. Cela ne serait pas possible avec une famille de données associée: les familles de données sont injectives , les familles de type synonyme ne le sont pas.

Bien que la non-injectivité ouvre certaines possibilités comme ci-dessus, cela rend aussi l'inférence de type plus difficile. Par exemple, ce qui suit ne sera pas typique:

```
class Foo a where
  type Bar a :: *
  bar :: a -> Bar a
instance Foo Int where
  type Bar Int = String
  bar = show
instance Foo Double where
  type Bar Double = Bool
  bar = (>0)

main = putStrLn (bar 1)
```

Dans ce cas, le compilateur ne peut pas savoir quelle instance utiliser, car l'argument de `bar` est lui-même un littéral polymorphe `Num` . Et la fonction de type `Bar` ne peut pas être résolue en «sens inverse», précisément parce qu'elle n'est pas injective † et n'est donc pas inversible (il peut y avoir plus d'un type avec `Bar a = String`).

---

† Avec seulement ces deux instances, il *est en* réalité injectif, mais le compilateur ne peut pas savoir que quelqu'un ajoutera plus d'instances ultérieurement et rompra ainsi le comportement.

Familles de types de données

Les familles de données peuvent être utilisées pour créer des types de données ayant des implémentations différentes basées sur leurs arguments de type.

Familles de données autonomes

```
{-# LANGUAGE TypeFamilies #-}
data family List a
data instance List Char = Nil | Cons Char (List Char)
```

```
data instance List () = UnitList Int
```

Dans la déclaration ci-dessus, `Nil :: List Char` et `UnitList :: Int -> List ()`

Familles de données associées

Les familles de données peuvent également être associées à des classes de caractères. Ceci est souvent utile pour les types avec des «objets auxiliaires», qui sont requis pour les méthodes génériques de typeclass mais qui doivent contenir des informations différentes selon l'instance concrète. Par exemple, l'indexation des emplacements dans une liste ne nécessite qu'un seul numéro, tandis que dans un arbre, vous avez besoin d'un numéro pour indiquer le chemin d'accès à chaque noeud:

```
class Container f where
  data Location f
  get :: Location f -> f a -> Maybe a

instance Container [] where
  data Location [] = ListLoc Int
  get (ListLoc i) xs
    | i < length xs  = Just $ xs!!i
    | otherwise      = Nothing

instance Container Tree where
  data Location Tree = ThisNode | NodePath Int (Location Tree)
  get ThisNode (Node x _) = Just x
  get (NodePath i path) (Node _ sfo) = get path =<< get i sfo
```

L'injectivité

Type Les familles ne sont pas nécessairement injectables. Par conséquent, nous ne pouvons pas déduire le paramètre d'une application. Par exemple, dans `servant` , étant donné un type `Server a` nous ne pouvons pas déduire le type `a` . Pour résoudre ce problème, nous pouvons utiliser `Proxy` . Par exemple, en `servant` , le `serve` la fonction est de type `... Proxy a -> Server a -> ...` . Nous pouvons en déduire `a` de `Proxy a` car `Proxy` est défini par des `data` qui sont injectives.

Lire Type de familles en ligne: https://riptutorial.com/fr/haskell/topic/2955/type-de-familles

Chapitre 72: Types de données algébriques généralisées

Examples

Utilisation de base

Lorsque l'extension `GADTs` est activée, outre les déclarations de données régulières, vous pouvez également déclarer les types de données algébriques généralisés comme suit:

```
data DataType a where
    Constr1 :: Int -> a -> Foo a -> DataType a
    Constr2 :: Show a => a -> DataType a
    Constr3 :: DataType Int
```

Une déclaration GADT répertorie les types de tous les constructeurs d'un type de données, explicitement. Contrairement aux déclarations de type de données habituelles, le type de constructeur peut être toute fonction N-aire (y compris la fonction null) qui aboutit au type de données appliqué à certains arguments.

Dans ce cas, nous avons déclaré que le type `DataType` a trois constructeurs: `Constr1` , `Constr2` et `Constr3` .

Le constructeur `Constr1` n'est pas différent de celui déclaré en utilisant une déclaration de données régulière: `data DataType a = Constr1 Int a (Foo a) | ...`

`Constr2` exige cependant que d' `a` a une instance de `Show` , et ainsi lors de l' utilisation du constructeur l'instance aurait besoin d'exister. D'un autre côté, lors de la mise en correspondance d'un motif, le fait que `a` soit une instance de `Show` entre dans le champ d'application, vous pouvez donc écrire:

```
foo :: DataType a -> String
foo val = case val of
    Constr2 x -> show x
    ...
```

Notez que l' `Show a` contrainte n'apparaît pas dans le type de la fonction et n'est visible que dans le code situé à droite de `->` .

`Constr3` a le type `DataType Int` , ce qui signifie que chaque fois qu'une valeur de type `DataType a` est une `Constr3` , il est connu que `a ~ Int` . Cette information peut également être récupérée avec une correspondance de modèle.

Lire Types de données algébriques généralisées en ligne:
https://riptutorial.com/fr/haskell/topic/2971/types-de-donnees-algebriques-generalisees

Chapitre 73: Types fantômes

Examples

Cas d'utilisation pour les types fantômes: devises

Les types fantômes sont utiles pour traiter des données, qui ont des représentations identiques mais qui ne sont pas logiquement du même type.

Un bon exemple concerne les devises. Si vous travaillez avec des devises, vous ne devez absolument jamais ajouter deux quantités de devises différentes. Quelle serait la devise de résultat de `5.32€ + 2.94$` ? Ce n'est pas défini et il n'y a aucune raison de le faire.

Une solution pourrait ressembler à ceci:

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

data USD
data EUR

newtype Amount a = Amount Double
                 deriving (Show, Eq, Ord, Num)
```

L'extension `GeneralisedNewtypeDeriving` nous permet de dériver `Num` pour le type `Amount` . GHC réutilise l'instance `Double` 's `Num` .

Maintenant, si vous représentez des montants en euros avec par exemple `(5.0 :: Amount EUR)` vous avez résolu le problème de la séparation des montants doubles au niveau du type sans introduire de frais généraux. Stuff comme `(1.13 :: Amount EUR) + (5.30 :: Amount USD)` entraînera une erreur de type et vous obligera à gérer la conversion de devise de manière appropriée.

Une documentation plus complète peut être trouvée dans l' article wiki de haskell

Lire Types fantômes en ligne: https://riptutorial.com/fr/haskell/topic/5227/types-fantomes

Chapitre 74: Utiliser GHCi

Remarques

GHCI est le REPL interactif fourni avec GHC.

Examples

Démarrer GHCi

Tapez `ghci` à l'invite du shell pour démarrer GHCI.

```
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Prelude>
```

Modification de l'invite par défaut GHCi

Par défaut, l'invite de GHCI affiche tous les modules que vous avez chargés dans votre session interactive. Si vous avez beaucoup de modules chargés, cela peut être long:

```
Prelude Data.List Control.Monad> -- etc
```

La commande `:set prompt` modifie l'invite pour cette session interactive.

```
Prelude Data.List Control.Monad> :set prompt "foo> "
foo>
```

Pour modifier l'invite de manière permanente, ajoutez `:set prompt "foo> "` dans le fichier de configuration GHCi .

Le fichier de configuration GHCi

GHCi utilise un fichier de configuration dans `~/.ghci` . Un fichier de configuration consiste en une séquence de commandes que GHCi exécutera au démarrage.

```
$ echo ":set prompt \"foo> \"" > ~/.ghci
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
Loaded GHCi configuration from ~/.ghci
foo>
```

Chargement d'un fichier

La commande `:l` ou `:load` vérifie et charge un fichier.

```
$ echo "f = putStrLn \"example\"" > example.hs
$ ghci
GHCi, version 8.0.1: http://www.haskell.org/ghc/  :? for help
ghci> :l example.hs
[1 of 1] Compiling Main               ( example.hs, interpreted )
Ok, modules loaded: Main.
ghci> f
example
```

Quitter GHCi

Vous pouvez quitter GHCi simplement avec `:q` ou `:quit`

```
ghci> :q
Leaving GHCi.

ghci> :quit
Leaving GHCi.
```

Sinon, le raccourci `CTRL` + `D` (`Cmd` + `D` pour OSX) a le même effet que `:q` .

Rechargement d'un fichier déjà chargé

Si vous avez chargé un fichier dans GHCi (par exemple en utilisant `:l filename.hs`) et que vous avez modifié le fichier dans un éditeur en dehors de GHCi vous devez recharger le fichier avec `:r` ou `:reload` afin d'utiliser des changements, d' où vous n'avez pas besoin de saisir à nouveau le nom du fichier.

```
ghci> :r
OK, modules loaded: Main.

ghci> :reload
OK, modules loaded: Main.
```

Points d'arrêt avec GHCi

GHCi prend en charge les points d'arrêt de style impératif en dehors de la boîte avec du code interprété (code qui a été `:loaded`).

Avec le programme suivant:

```
-- mySum.hs
doSum n = do
  putStrLn ("Counting to " ++ (show n))
  let v = sum [1..n]
  putStrLn ("sum to " ++ (show n) ++ " = " ++ (show v))
```

chargé dans GHCi:

```
Prelude> :load mySum.hs
[1 of 1] Compiling Main             ( mySum.hs, interpreted )
```

```
Ok, modules loaded: Main.
*Main>
```

Nous pouvons maintenant définir des points d'arrêt en utilisant des numéros de ligne:

```
*Main> :break 2
Breakpoint 0 activated at mySum.hs:2:3-39
```

et GHCi s'arrêtera à la ligne correspondante lorsque nous exécuterons la fonction:

```
*Main> doSum 12
Stopped at mySum.hs:2:3-39
_result :: IO () = _
n :: Integer = 12
[mySum.hs:2:3-39] *Main>
```

Il peut être déroutant de savoir où nous en sommes dans le programme, nous pouvons donc utiliser `:list` pour clarifier:

```
[mySum.hs:2:3-39] *Main> :list
1  doSum n = do
2    putStrLn ("Counting to " ++ (show n))   -- GHCi will emphasise this line, as that's where
we've stopped
3    let v = sum [1..n]
```

Nous pouvons imprimer des variables et continuer l'exécution aussi:

```
[mySum.hs:2:3-39] *Main> n
12
:continue
Counting to 12
sum to 12 = 78
*Main>
```

Déclarations multilignes

L'instruction `:{` commence le mode *multiligne* et `:}` termine. En mode multiligne, GHCi interprétera les nouvelles lignes comme des points-virgules et non comme la fin d'une instruction.

```
ghci> :{
ghci| myFoldr f z [] = z
ghci| myFoldr f z (y:ys) = f y (myFoldr f z ys)
ghci| :}
ghci> :t myFoldr
myFoldr :: (a -> b -> b) -> b -> [a] -> b
```

Lire Utiliser GHCi en ligne: https://riptutorial.com/fr/haskell/topic/3407/utiliser-ghci

Chapitre 75: Vecteurs

Remarques

> Il [Data.Vector] met l'accent sur les très hautes performances grâce à la fusion en boucle, tout en conservant une interface riche. Les principaux types de données sont les tableaux encadrés et non encadrés, et les tableaux peuvent être immuables (purs) ou mutables. Les tableaux peuvent contenir des éléments stockables, adaptés au passage vers et depuis C, et vous pouvez convertir entre les types de tableau. Les tableaux sont indexés par des valeurs Int non négatives.

Le Wiki Haskell a ces recommandations :

> En général:
>
> - Les utilisateurs finaux doivent utiliser Data.Vector.Unboxed pour la plupart des cas
> - Si vous avez besoin de stocker des structures plus complexes, utilisez Data.Vector
> - Si vous devez passer à C, utilisez Data.Vector.Strable
>
> Pour les écrivains de bibliothèque;
>
> - Utilisez l'interface générique pour vous assurer que votre bibliothèque est extrêmement flexible: Data.Vector.Generic

Examples

Le module Data.Vector

Le module Data.Vector fourni par le vecteur est une bibliothèque hautes performances pour travailler avec des tableaux.

Une fois que vous avez importé `Data.Vector` , il est facile de commencer à utiliser un `Vector` :

```
Prelude> import Data.Vector
Prelude Data.Vector> let a = fromList [2,3,4]

Prelude Data.Vector> a
fromList [2,3,4] :: Data.Vector.Vector

Prelude Data.Vector> :t a
a :: Vector Integer
```

Vous pouvez même avoir un tableau multidimensionnel:

```
Prelude Data.Vector> let x = fromList [ fromList [1 .. x] | x <- [1..10] ]
```

```
Prelude Data.Vector> :t x
x :: Vector (Vector Integer)
```

Filtrage d'un vecteur

Filtrer les éléments impairs:

```
Prelude Data.Vector> Data.Vector.filter odd y
fromList [1,3,5,7,9,11] :: Data.Vector.Vector
```

Mapping (`map`) et Réduire (` fold`) un vecteur

Les vecteurs peuvent être `map` et `fold'd`, filtrés `'d and` compressés:

```
Prelude Data.Vector> Data.Vector.map (^2) y
fromList [0,1,4,9,16,25,36,49,64,81,100,121] :: Data.Vector.Vector
```

Réduire à une seule valeur:

```
Prelude Data.Vector> Data.Vector.foldl (+) 0 y
66
```

Travailler sur plusieurs vecteurs

Zip deux tableaux dans un tableau de paires:

```
Prelude Data.Vector> Data.Vector.zip y y
fromList [(0,0),(1,1),(2,2),(3,3),(4,4),(5,5),(6,6),(7,7),(8,8),(9,9),(10,10),(11,11)] ::
Data.Vector.Vector
```

Lire Vecteurs en ligne: https://riptutorial.com/fr/haskell/topic/4738/vecteurs

Chapitre 76: Vérification rapide

Examples

Déclarer une propriété

Dans sa forme la plus simple, une *propriété* est une fonction qui renvoie un `Bool` .

```
prop_reverseDoesNotChangeLength xs = length (reverse xs) == length xs
```

Une propriété déclare un invariant de haut niveau d'un programme. Le programme de test QuickCheck évalue la fonction avec 100 entrées aléatoires et vérifie que le résultat est toujours `True` .

Par convention, les fonctions qui sont des propriétés ont des noms qui commencent par `prop_` .

Vérification d'une seule propriété

La fonction `quickCheck` teste une propriété sur 100 entrées aléatoires.

```
ghci> quickCheck prop_reverseDoesNotChangeLength
+++ OK, passed 100 tests.
```

Si une propriété échoue pour une entrée, `quickCheck` un contre-exemple.

```
prop_reverseIsAlwaysEmpty xs = reverse xs == []  -- plainly not true for all xs

ghci> quickCheck prop_reverseIsAlwaysEmpty
*** Failed! Falsifiable (after 2 tests):
[()]
```

Vérification de toutes les propriétés dans un fichier

`quickCheckAll` est un assistant Template Haskell qui trouve toutes les définitions du fichier en cours dont le nom commence par `prop_` et les teste.

```
{-# LANGUAGE TemplateHaskell #-}

import Test.QuickCheck (quickCheckAll)
import Data.List (sort)

idempotent :: Eq a => (a -> a) -> a -> Bool
idempotent f x = f (f x) == f x

prop_sortIdempotent = idempotent sort

-- does not begin with prop_, will not be picked up by the test runner
sortDoesNotChangeLength xs = length (sort xs) == length xs
```

```
return []
main = $quickCheckAll
```

Notez que la ligne de `return []` est requise. Il rend les définitions textuellement au-dessus de cette ligne visibles pour Template Haskell.

```
$ runhaskell QuickCheckAllExample.hs
=== prop_sortIdempotent from tree.hs:7 ===
+++ OK, passed 100 tests.
```

Génération aléatoire de données pour les types personnalisés

La classe `Arbitrary` est pour les types pouvant être générés de manière aléatoire par QuickCheck.

L'implémentation minimale d' `Arbitrary` est la méthode `arbitrary` , qui s'exécute dans la monade `Gen` pour produire une valeur aléatoire.

Voici une instance de `Arbitrary` pour le type de données suivant des listes non vides.

```
import Test.QuickCheck.Arbitrary (Arbitrary(..))
import Test.QuickCheck.Gen (oneof)
import Control.Applicative ((<$>), (<*>))

data NonEmpty a = End a | Cons a (NonEmpty a)

instance Arbitrary a => Arbitrary (NonEmpty a) where
    arbitrary = oneof [  -- randomly select one of the cases from the list
        End <$> arbitrary,  -- call a's instance of Arbitrary
        Cons <$>
            arbitrary <*>  -- call a's instance of Arbitrary
            arbitrary  -- recursively call NonEmpty's instance of Arbitrary
        ]
```

Utilisation de l'implication (==>) pour vérifier les propriétés avec des conditions préalables

```
prop_evenNumberPlusOneIsOdd :: Integer -> Property
prop_evenNumberPlusOneIsOdd x = even x ==> odd (x + 1)
```

Si vous voulez vérifier qu'une propriété est conservée si une précondition est vérifiée, vous pouvez utiliser l'opérateur `==>` . Notez que s'il est très improbable que des entrées arbitraires correspondent à la condition préalable, QuickCheck peut abandonner tôt.

```
prop_overlySpecific x y = x == 0 ==> x * y == 0

ghci> quickCheck prop_overlySpecific
*** Gave up! Passed only 31 tests.
```

Limiter la taille des données de test

Il peut être difficile de tester des fonctions avec une complexité asymptotique médiocre en utilisant la vérification rapide car les entrées aléatoires ne sont généralement pas limitées par la taille. En ajoutant une limite supérieure à la taille de l'entrée, nous pouvons toujours tester ces fonctions coûteuses.

```
import Data.List(permutations)
import Test.QuickCheck

longRunningFunction :: [a] -> Int
longRunningFunction xs = length (permutations xs)

factorial :: Integral a => a -> a
factorial n = product [1..n]

prop_numberOfPermutations xs =
    longRunningFunction xs == factorial (length xs)

ghci> quickCheckWith (stdArgs { maxSize = 10}) prop_numberOfPermutations
```

En utilisant `quickCheckWith` avec une version modifiée de `stdArgs` nous pouvons limiter la taille des entrées à 10 au maximum. Dans ce cas, comme nous générons des listes, cela signifie que nous générons des listes allant jusqu'à la taille 10. Notre fonction de permutations ne prendre trop de temps pour courir pour ces listes restreintes, mais nous pouvons toujours être raisonnablement convaincus que notre définition est correcte.

Lire Vérification rapide en ligne: https://riptutorial.com/fr/haskell/topic/1156/verification-rapide

Chapitre 77: XML

Introduction

Codage et décodage de documents XML.

Examples

Encoder un enregistrement en utilisant la bibliothèque `xml`

```
{-# LANGUAGE RecordWildCards #-}
import Text.XML.Light

data Package = Package
  { pOrderNo  :: String
  , pOrderPos :: String
  , pBarcode  :: String
  , pNumber   :: String
  }

-- | Create XML from a Package
instance Node Package where
  node qn Package {..} =
    node qn
      [ unode "package_number" pNumber
      , unode "package_barcode" pBarcode
      , unode "order_number" pOrderNo
      , unode "order_position" pOrderPos
      ]
```

Lire XML en ligne: https://riptutorial.com/fr/haskell/topic/9264/xml

Chapitre 78: zipWithM

Introduction

`zipWithM` est `zipWith` comme `mapM` est à la `map` : il vous permet de combiner deux listes en utilisant une fonction monadique.

A partir du module `Control.Monad`

Syntaxe

- zipWithM :: Applicatif m => (a -> b -> mc) -> [a] -> [b] -> m [c]

Examples

Calcul des prix de vente

Supposons que vous vouliez voir si un certain ensemble de prix de vente est approprié pour un magasin.

À l'origine, les articles coûtaient 5 dollars, vous ne voulez donc pas accepter la vente si le prix de vente est inférieur pour l'un d'entre eux, mais vous voulez savoir quel est le nouveau prix.

Calculer un prix est facile: vous calculez le prix de vente et vous retournez `Nothing` si vous ne faites pas de profit:

```
calculateOne :: Double -> Double -> Maybe Double
calculateOne price percent = let newPrice = price*(percent/100)
                             in if newPrice < 5 then Nothing else Just newPrice
```

Pour le calculer pour toute la vente, `zipWithM` rend très simple:

```
calculateAllPrices :: [Double] -> [Double] -> Maybe [Double]
calculateAllPrices prices percents = zipWithM calculateOne prices percents
```

Cela ne rapportera `Nothing` si l'un des prix de vente est inférieur à 5 $.

Lire zipWithM en ligne: https://riptutorial.com/fr/haskell/topic/9685/zipwithm

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec le langage Haskell	4444, Adam Wagner, alejosocorro, Amitay Stern, arseniiv, baxbaxwalanuksiwe, Benjamin Hodgson, Benjamin Kovach, Burkhard, Carsten, colinro, Community, Daniel Jour, Echo Nolan, erisco, gdziadkiewicz, HBu, J Atkin, Jan Hrcek, Jules, Kwarrtz, leftaroundabout, M. Barbieri, mb21, mnoronha, Mr Tsjolder, ocharles, pouya, R B, Ryan Hilbert, Sebastian Graf, Shoe, Stephen Leppik, Steve Trout, Tim E. Lord, Turion, user239558, Will Ness, zbw, λlεx
2	Algèbre de type	Mario Román
3	Algorithmes de tri	Brian Min, pouya, Romildo, Shoe, Vektorweg, Will Ness
4	Analyse HTML avec lentille taggy et lentille	Kostiantyn Rybnikov, λlεx
5	Application partielle	arseniiv, Benjamin Hodgson, CiscoIPPhone, Dair, Matthew Pickering, Will Ness
6	Arithmétique	ocramz
7	Attoparsec	λlεx
8	Banane réactive	arrowd, Dair, Undreren
9	Bases de données	λlεx
10	Bifunctor	Benjamin Hodgson, liminalisht
11	Cabale	tlo
12	Classes de types	arjanen, Benjamin Hodgson, Billy Brown, Cactus, Dair, gdziadkiewicz, J. Abrahamson, Kwarrtz, leftaroundabout, mnoronha, RamenChef, Will Ness, zeronone, λlεx
13	Composition de fonction	arseniiv, Will Ness
14	Concurrence	Janos Potecki, λlεx
15	Conteneurs -	Cactus, Itbot, Will Ness, λlεx

	Data.Map	
16	Création de types de données personnalisés	Janos Potecki, Kapol
17	Data.Aeson - JSON dans Haskell	Chris Stryczynski, Janos Potecki, Matthew Pickering, rob, xuh
18	Data.Text	Benjamin Hodgson, dkasak, Janos Potecki, jkeuhlen, mnoronha , unhammer
19	Date et l'heure	mnoronha, Will Ness, λlεx
20	Déclarations de fixité	Alec, Will Ness
21	Des listes	Benjamin Hodgson, erisco, Firas Moalla, Janos Potecki, Kwarrtz , Lynn, Matthew Pickering, Matthew Walton, Mirzhan Irkegulov, mnoronha, Tim E. Lord, user2407038, Will Ness, Yosh, λlεx
22	Des procurations	Benjamin Hodgson
23	Développement web	arrowd, λlεx
24	Empiler	4444, curbyourdogma, Dair, Janos Potecki
25	Enregistrement	λlεx
26	Etat Monad	Apoorv Ingle, Kritzefitz, Luis Casillas
27	Extensions de langage GHC communes	Antal Spector-Zabusky, Bartek Banachewicz, Benjamin Hodgson, Benjamin Kovach, Cactus, charlag, Doomjunky, Janos Potecki, John F. Miller, K48, Kwarrtz, leftaroundabout, Mads Buch, Matthew Pickering, mkovacs, mniip, phadej, Yosh, λlεx
28	Flèches	artcorpse, leftaroundabout
29	Foncteur applicatif	Benjamin Hodgson, Kritzefitz, mnoronha, Will Ness, λlεx
30	Fonctions d'ordre supérieur	Community, Doruk, Matthew Pickering, mnoronha, Will Ness, λl εx
31	Functor	Benjamin Hodgson, Delapouite, Janos Potecki, liminalisht, mathk, mnoronha, Will Ness, λlεx
32	GHCJS	Mikkel
33	Graphisme avec Gloss	Wysaard, Zoey Hewll

34	Gtk3	bleakgadfly
35	Interface de fonction étrangère	arrowd, bleakgadfly, crockeea
36	IO	3442, Benjamin Hodgson, David Grayson, J. Abrahamson, Jan Hrcek, John F. Miller, leftaroundabout, mnoronha, Sarah, user2407038, Will Ness, λlεx
37	Lecteur / lecteur	Chris Stryczynski
38	Lentille	Bartek Banachewicz, bennofs, chamini2, dfordivam, dsign, Hjulle, J. Abrahamson, John F. Miller, Kwarrtz, Matthew Pickering, λlεx
39	Les foncteurs communs comme base des comonades cofree	leftaroundabout
40	Liste des compréhensions	Cactus, Kwarrtz, mnoronha, Will Ness
41	Littéraux surchargés	Adam Wagner, Carsten, Kapol, leftaroundabout, pdexter
42	Modules	Benjamin Hodgson, Kapol, Will Ness, λlεx
43	Monad Transformers	Damian Nadales
44	Monades	Alec, Benjamin Hodgson, Cactus, fgb, Kapol, Kwarrtz, Lynn, Mario Román, Matthew Pickering, Will Ness, λlεx
45	Monades communes comme monades gratuites	leftaroundabout
46	Monades gratuites	Benjamin Hodgson, Cactus, J. Abrahamson, pyon, sid-kap
47	Monoïde	Benjamin Hodgson, Kwarrtz, mnoronha, Will Ness
48	Opérateurs Infix	leftaroundabout, mnoronha
49	Optimisation	λlεx
50	Parallélisme	λlεx
51	Pipes	4444, Benjamin Hodgson, Stephane Rolland, λlεx
52	Pliable	Benjamin Hodgson, Cactus, Dair, David Grayson, J. Abrahamson, Jan Hrcek, mnoronha

53	Polymorphisme de rang arbitraire avec RankNTypes	ocharles
54	Profunctor	zbw
55	Règles de réécriture (GHC)	Cactus
56	Rigueur	Benjamin Hodgson, Benjamin Kovach, Cactus, user2407038, Will Ness
57	Rôle	xuh
58	Schémas de récursivité	arseniiv, Benjamin Hodgson
59	Streaming IO	λlεx
60	Syntaxe d'appel de fonction	Zoey Hewll
61	Syntaxe d'enregistrement	Cactus, Janos Potecki, John F. Miller, Mario, Matthew Pickering, Will Ness, λlεx
62	Syntaxe dans les fonctions	Delapouite, James, Janos Potecki, Will Ness, λlεx
63	Tampons de protocole Google	Janos Potecki, λlεx
64	Template Haskell & QuasiQuotes	user2407038
65	Test avec savoureux	tlo
66	Théorie des catégories	arrowd, Benjamin Hodgson, Benjamin Kovach, J. Abrahamson, Mario Román, mmlab, user2407038
67	Traversable	Benjamin Hodgson, ErikR, J. Abrahamson
68	Trous dactylographiés	Cactus, leftaroundabout, user2407038, Will Ness
69	Tuples (paires, triples, ...)	Cactus, mnoronha, Toxaris, λlεx
70	Type d'application	Luka Horvat, λlεx
71	Type de familles	leftaroundabout, mniip, xuh

72	Types de données algébriques généralisées	mniip
73	Types fantômes	Benjamin Hodgson, Christof Schramm
74	Utiliser GHCi	Benjamin Hodgson, James, Janos Potecki, mnoronha, RamenChef, wizzup, λlεx
75	Vecteurs	Benjamin Hodgson, λlεx
76	Vérification rapide	Benjamin Hodgson, gdziadkiewicz, Matthew Pickering, Steve Trout
77	XML	Mikkel
78	zipWithM	zbw