Cycle ingénieur - 2ème année

Programmation fonctionnelle

Types avancés

2023-2024

Plan

Types génériques ≃ types à paramètres

Origine possible : inférence incomplète de type

- Type exact pas toujours déterminable
- Tente de déterminer au moins les liens entre les types des différents éléments de la signature (paramètres, résultat)
- Type réel inféré lors d'évaluations ultérieures

Expression	**Type**
`id = \x -> x`	`a -> a`
`fst = \(x, y) -> x`	`(a, b) -> a`
`snd = \(x, y) -> y`	`(a, b) -> b`

Classes de types (*typeclasses*)

Classe de types

Exemple : `inc n = n + 1`

Plusieurs déclarations de type sont possibles :

- `Int -> Int`, `Integer -> Integer`
- `Double -> Double`, `Float -> Float`
- ou `a -> a` où a est muni de l'opérateur `(+)`

... *et si on veut `inc` pour tous ces types ?*

Haskell sait faire le tri, donc plusieurs versions sont inutiles.

- Avec cette seule définition, `inc 3` et `inc 3.0` sont valides.
- Haskell infère le type de `inc` comme étant `Num a => a -> a` :

  « Si `a` *est une instance de* `Num`*, le type de `inc` est* `a -> a` *».*

Classe de types

`Num` est une classe de types (*typeclass*)

- Une classe de type définit les constantes (fonctionnelles ou non) devant être définies par tout type qui en est une instance.
- Elle peut proposer une implémentation par défaut de ces constantes :
  - chaque instance peut redéfinir cette implémentation ;
  - toute constante non implémentée par défaut doit l'être par toute instance.
- *Peut être rapproché du principe de l'interface en Java*

Déclaration d'une classe

```
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  abs           :: a -> a
  signum        :: a -> a
  fromInteger   :: Integer -> a
  x - y = x + (negate y)
  negate x = 0 - x
```

Déclaration d'une instance

```
instance Num Int where
   x + y = ...
   fromInteger n = ...
   ...
```

`(+)`, `(*)`, `abs`, `signum`, `fromInteger` doivent être implémentés.

Au moins `negate` ou `(-)` (au choix) doit être implémenté.

Classes de types usuelles

Classes générales (liste complète)

- `Eq` : types munis d'une égalité (`==`, `/=`)
- `Ord` : types munis d'un ordre *total* (`<=`, `<`, etc...)
- etc...

Classes numériques (liste complète)

- `Num` : types avec les opérateurs de base (`+`, `-`, `*`)
- `Integral` : types avec division euclidienne (`div`, `mod`)
- `Fractional` : types avec division (`/`)
- `Floating` : types avec fonctions usuelles (sin, cos, ...)
- etc...

Types union

Type union

Type $T_1 \mid T_2 \mid \ldots \mid T_n$

Type des éléments de type T_1 ou T_2 ou ... ou T_n

Potentielle ambigüité sur le typage

Un élément de type T_1 est aussi un élément de type :

- $T_1 \mid T_2$;
- $T_1 \mid T_3$;
- $T_1 \mid T_2 \mid T_3$;
- ...

$\Longrightarrow$ Nécessité de différencier les types union dès leur déclaration
$\Longrightarrow$ Encapsulation de chaque sous-type dans un **constructeur**

Déclaration d'un type union : `data`

Constructeur

- commence toujours par une MAJUSCULE
- constant ou paramétré : si paramétré, **c'est une fonction**
- paramètres nommables (*record syntax*)

  $\Longrightarrow$ les noms deviennent des fonctions d'accès direct

  (ATTENTION aux conflits de noms !)

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

NB: Le type `Maybe` est un type prédéfini en Haskell.

Expression	**Type**
`Nothing`	`Maybe a`
`Nothing 42`	`Couldn't match expected type`
`Just 42`	`Maybe Integer`
`Just`	`a -> Maybe a`

Cas particuliers

Énumération : constructeurs constants uniquement

```
data Bool = False | True
```

Expression	Résultat
`True`	`True`
`TrueAndFalse`	`Data constructor not in scope`

Enregistrement : un seul terme

- Souvent utilisé avec des paramètres nommés

```
data Date = Date {year :: Int, month :: Int, day :: Int}
epoch = Date {day = 01, month = 01, year = 1970}
```

Expression	Résultat
`epoch`	`Date {year = 1970, month = 1, day = 1}`
`Date 1970 01 01`	`Date {year = 1970, month = 1, day = 1}`
`year epoch`	`1970`

Type union : filtrage

Tuples/enregistrements

```
(x1, x2) = (1, 2)
Date y m d = epoch
```

Expression	Résultat
x1	1
y	1970

Applicable quel que soit le tuple ou l'enregistrement

Type union « vrai » (i.e. avec au moins deux termes)

```
Just x = Just 42
Just z = Nothing
```

Pattern match(es) non-exhaustive

Expression	Résultat
x	42
z	Ambiguous type

Non applicable pour tout élément de type `Maybe`

Type union : filtrage

Tuples/enregistrements

- Un unique motif peut suffire pour garantir l'exhaustivité.

« Vrais » types union

- Plusieurs constructeurs distincts

  $\Longrightarrow$ aucun motif n'est exhaustif seul

Pour un « vrai » type union, seul le filtrage par reconnaissance de motifs peut être exhaustif.

Filtrage par reconnaissance de motifs dans une fonction

Cas d'application

- Filtrage par reconnaissance de motifs *directement sur les paramètres de la fonction*
- **Les mêmes règles (exhaustivité, séquentialité, unicité) s'appliquent.**

```
answer :: Maybe Int -> Bool
answer Nothing  = False
answer (Just a) = a == 42
```

Maybe

Maybe (documentation complète)

```haskell
data Maybe a = Nothing | Just a
```

« Vrai » type union ⟹ filtrage par motifs uniquement

- Règles inchangées : exhaustivité, séquentialité, unicité

Fonctions de test prédéfinies

```haskell
isNothing :: Maybe a -> Bool
isNothing Nothing  = True
isNothing (Just _) = False

isJust :: Maybe a -> Bool
isJust Nothing  = False
isJust (Just _) = True
```

Fonctions de transformation prédéfinies

```haskell
fromMaybe :: a -> Maybe a -> a
fromMaybe z Nothing  = z
fromMaybe _ (Just x) = x

join :: Maybe (Maybe a) -> Maybe a
join Nothing  = Nothing
join (Just x) = x

(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b -- bind
Nothing  >>= _ = Nothing
(Just x) >>= f = f x

fmap :: (a -> b) -> Maybe a -> Maybe b
fmap _ Nothing  = Nothing
fmap f (Just x) = Just (f x)

maybe :: b -> (a -> b) -> Maybe a -> b
maybe z _ Nothing  -> z
maybe _ f (Just x) -> f x
```

Fonctions à resultat de type `Maybe a`

Motivation

- Renvoie un résultat au lieu de provoquer une erreur

  ⇒ Fonctionnement normal non interrompu

  ⇒ Pas de `try` … `catch`
- `Nothing` ≡ absence de résultat

Exemple : Recherche d'un élément dans une liste

L'absence de résultat devient un résultat en soi.

- Possibilité de transmettre l'« erreur » à la fonction appelante

  ⇒ traitement optimisé puisque effectué au bon moment
- Inconvénient : non-différentiation des « erreurs » possibles

  (puisque toute erreur donne `Nothing`) ⟹ `Either`

Either

Either (documentation complète)

```haskell
data Either a b = Left a | Right b
```

« Vrai » type union ⟹ filtrage par motifs uniquement

Fonctions de test prédéfinies

```haskell
isLeft :: Either a b -> Bool
isLeft (Left  _) = True
isLeft (Right _) = False
```

```haskell
isRight :: Either a b -> Bool
isRight (Left  _) = False
isRight (Right _) = True
```

Fonctions de transformation prédéfinies

```haskell
fromLeft :: a -> Either a b -> a
fromLeft _ (Left  x) -> x
fromLeft z (Right _) -> z
```

```haskell
fromRight :: b -> Either a b -> b
fromRight z (Left  _) -> z
fromRight _ (Right x) -> x
```

```haskell
either :: (a -> c) -> (b -> c) -> Either a b -> c
either fLeft _      (Left  x) -> fLeft  x
either _     fRight (Right y) -> fRight y
```

Fonctions à resultat de type `Either a b`

Motivation

- Renvoie un résultat au lieu de provoquer une erreur

  ⇒ Fonctionnement normal non interrompu

  ⇒ Pas de `try` … `catch`
- **Par convention, `Right _` est un résultat correct (« *right* »).**
- `Left _` ≡ absence (documentée) de résultat

L'absence de résultat devient un résultat en soi.

- Possibilité de transmettre l'« erreur » à la fonction appelante

  ⇒ traitement optimisé puisque effectué au bon moment
- Avantage : différentiation des « erreurs » possibles

  (la valeur contenue dans `Left` renseigne l'erreur)

Types récursifs

Types récursifs

Type récursif : type exprimé en fonction de lui-même

La définition d'un type récursif nécessite :

- au moins un cas terminal (pour que l'inférence termine) ;
- au moins une définition en fonction du type lui-même.

⟹ Type récursif : « vrai » type union

Exemple : liste chaînée

Une liste est :

- soit vide *(cas terminal)* ;
- soit la donnée d'un élément (tête) et d'une liste (queue).

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

Types récursifs : exemple détaillé

Une expression arithmétique est :

- soit un nombre *(cas terminal)* ;
- soit un opérateur (+, -, ×) appliqué à deux expressions.

```
data ArithExpr a = Value a
                 | Plus  (ArithExpr a) (ArithExpr a)
                 | Minus (ArithExpr a) (ArithExpr a)
                 | Times (ArithExpr a) (ArithExpr a)
```

La définition du type fournit ses règles de validation.

- Impossible de construire une expression incorrecte

```
fortyTwo = Times (Value 6) (Plus (Value 4) (Value 3))
Times (Value 6)  -- second paramètre de Times manquant
Plus (Value 4) 3 -- second paramètre de Plus incorrect
```

Types récursifs : exemple détaillé

Traitement sur un type récursif

- Filtrage par reconnaissance de motifs dans la majorité des cas
- Traitement récursif dans la très grande majorité des cas

```
eval :: Num a => ArithExpr a -> a
eval (Value x)       = x
eval (Plus  e1 e2) = (eval e1) + (eval e2)
eval (Minus e1 e2) = (eval e1) - (eval e2)
eval (Times e1 e2) = (eval e1) * (eval e2)
```

NB: Le motif _ est inutile ici. *(Pourquoi ?)*

NB: La mention `Num a =>` est obligatoire ici. *(Pourquoi ?)*

Expression	**Résultat**
`eval fortyTwo`	`42`

Modules

Module

Regroupement de déclarations/implémentations

- types/classes de types
- instanciation de classes de types
- constantes
- fonctions

Syntaxe

```
module <nomModule>(<listePublique>) where
  <corpsModule>
```

- `<listePublique>` contient la liste des éléments publics :
  - si elle n'est pas spécifiée, tout élément déclaré est public ;
  - les éléments issus d'une instanciation sont publics (ne pas les spécifier)
- `<corpsModule>` contient l'ensemble des déclarations et implémentations

Utilisation d'un module : `import`

```
import [qualified] <NomModule> [as <AliasModule>]
       [(listeElements) | hiding (<listeElements>)]
```

- **`qualified`** impose l'utilisation *explicite* des éléments (i.e. prefixés avec le nom du module).
- **`as`** permet d'utiliser un alias pour le nom du module dans le fichier courant
- **`(...)`** contient la liste des constantes importées
- **`hiding (...)`** empêche l'import des constantes spécifiées *(souvent utilisé pour éviter des conflits de noms avec d'autres modules)*

Module : exemple

Fichier `ArithExpr.hs` (1/2)

```
module ArithExpr(ArithExpr, val, (+!), (-!), (*!), eval) where

data ArithExpr a = Value a
                 | Plus  (ArithExpr a) (ArithExpr a)
                 | Minus (ArithExpr a) (ArithExpr a)
                 | Times (ArithExpr a) (ArithExpr a)

val :: a -> ArithExpr a
val = Value

(+!) :: ArithExpr a -> ArithExpr a -> ArithExpr a
(+!) = Plus
(-!) :: ArithExpr a -> ArithExpr a -> ArithExpr a
(-!) = Minus
(*!) :: ArithExpr a -> ArithExpr a -> ArithExpr a
(*!) = Times
```

Module : exemple

Ficher ArithExpr.hs (2/2)

```
eval :: Num a => ArithExpr a -> a
eval (Value x)      = x
eval (Plus  e1 e2) = (eval e1) + (eval e2)
eval (Minus e1 e2) = (eval e1) - (eval e2)
eval (Times e1 e2) = (eval e1) * (eval e2)

fortyTwo :: ArithExpr Integer
fortyTwo = Times (Value 6) (Plus (Value 4) (Value 3))
```

Utilisation

```
import ArithExpr
e = val 6 *! (val 4 +! val 3)
```

Expression	Résultat
`eval ArithExpr.fortyTwo`	`Not in scope`
`(Plus (Value 4) (Value 3))`	`Data constructor not in scope`
`eval e`	`42`

Documentation : *Haddock*

Commentaires en Haskell

Commentaire sur une ligne -- (cf // en C ou Java)

- -- commente le contenu le suivant et ce jusqu'à la fin de la ligne.
- -- ne commente pas le contenu le précédant.
- -- ne commente pas le contenu de la ligne suivante.

```
Partie non commentée -- Partie commentée
Partie non commentée
```

Bloc de commentaire {- ... -} (cf /* ... */ en C ou Java)

- Tout le contenu entre {- et -} est commenté

```
Partie non commentée {-
Partie commentée
Partie commentée
-} Partie non commentée
```

Documentation d'un code Haskell

Haddock **(Documentation complète)**

- Créé en 2002
- Licence BSD
- Dernière version : **2.29.1** (22 septembre 2023)
- Inspiré par d'autres systèmes comme *Doxygen*

Principes d'utilisation

- Tout élément de documentation est un commentaire Haskell.
- `-- |` et `{- | ... -}` documentent l'élément **qui suit**.
- `-- ^` et `{- ^ ... -}` documentent l'élément **qui précède**.

Documentation d'un module

```haskell
{- |
  Module      : ArithExpr
  Description : Expressions arithmétiques
  Copyright   : (c) Romain Dujol, 2023
  Maintainer  : romain.dujol@cyu.fr
-}
module ArithExpr
  ( -- * Data type
    ArithExpr,
    -- * Opérations
    val, (+!), (-!), (*!),
    -- * Évaluation
    eval
  ) where
  ...
```

Documentation d'un type

```haskell
-- | Type d'une expression arithmétique
data ArithExpr a =
   Value -- ^ Constante
         a -- ^ Type numérique utilisé
 | Plus  (ArithExpr a) (ArithExpr a) -- ^ Addition
 | Minus -- ^ Soustraction
         (ArithExpr a) -- ^ Opérande gauche
         (ArithExpr a) -- ^ Opérande droit
 | Times -- ^ Multiplication
         (ArithExpr a) -- ^ Opérande gauche
         (ArithExpr a) -- ^ Opérande droit
```

Documentation d'une fonction

```haskell
-- | Addition
(+!) :: ArithExpr a -- ^ Opérande gauche
     -> ArithExpr a -- ^ Opérande droit
     -> ArithExpr a -- ^ Somme des deux opérandes
(+!) = Plus
```

Mise en forme du texte

- Italique : `/Texte/`
- Gras : `__Texte__`
- Largeur fixe : `@Texte@`
- Lien vers une valeur Haskell : `'ArithExpr'`
  (*quote* ou *backquote*)