УДК 519.6

Галкін О.В., к. ф.-м. н.,доц.

Поліморфізм та класи в мові Haskell

У даній статі проаналізовано типи поліморфізму, реалізовані в мові програмування Haskell

Ключові слова: поліморфізм, параметричний поліморфізм, мова Haskel

Київський національний університет імені Тараса Шевченка,03680, м. Київ, пр-т Глушкова 4д, e-mail:galkin@unicyb.kiev.ua

O.V.Galkin, Ph.D

Polymorphism and classes in Haskell.

In this paper we consider the kinds of polymorphism implemented,in Haskell.

Key Words:polymorphism,parametric polymorphism, language Haskell.

Taras Shevchenko National University of Kyiv, 03680, Kyiv, Glushkova av., 4d,
e-mail: galkin@unicyb.kiev.ua

Статтю представив д. т. н., проф. Заславський В.А.

Вступ.

Haskell є чисто функціональною мовою програмування загального призначення, яка включає багато останніх інновацій у розробці мов програмування. Haskell забезпечує функції високого порядку, нестрогу семантику, статичну поліморфну типізацію, визначені користувачем алгебраїчні типи даних, зіставлення зі зразком, опис списків, модульну систему, монадичну систему введення - виведення і багатий набір примітивних типів даних, включаючи списки, масиви, цілі числа довільної і фіксованої точності і числа з плаваючою крапкою [1].

Однією з головних особливостей мови Haskell є те, що вона є мовою з відкладеними обчисленнями. Це означає, що нічого не буде обчислене до початку самого процесу обчислення. Наприклад, можна визначити в нескінченній рекурсії нескінченний список чисел. Тільки ті елементи списку, які насправді будуть використані, будуть обчислені. Це дає можливість для розв'язання багатьох проблем використовувати дуже елегантні методи. Типовим зразком вирішення проблеми могло б бути задання списку всіх можливих методів вирішення з наступним відсіюванням неправильних. Такий список згодом буде містити тільки допустимі розв'язки. Відкладені обчислення роблять цю операцію дуже прозорою [3-4].

Варто також зазначити, що мова Haskell є суворо типізованою мовою. Неможливо необережно привести Double до Int або викликати метод по порожньому покажчику. Це приводить до меншої кількості помилок. У деяких випадках може бути потрібно явно приводити Int до Double тоді, коли це потрібно перед виконанням якоїсь операції, але на практиці таке трапляється не так часто, щоб викликати незручності. В дійсності, примусове явне приведення часто допомагає виділити проблемну частину коду.

На відміну від інших мов, типи в мові Haskell визначаються автоматично. Це означає, що дуже рідко необхідно оголошувати типи функцій, крім тих випадків, які обумовлені документацією коду.

Наприклад, якщо написати функцію сортування без опису типу, мова Haskell упевниться, що вона буде працювати для всіх значень, які можуть бути відсортовані. Для порівняння, в мові програмування Java для досягнення поліморфізму необхідно було б використовувати базові класи, а після цього оголосити змінні як екземпляри підкласу цього

базового класу. Все це призводить до необхідності писати додатковий код. До того ж для явного приведення типів доведеться провести безліч перетворень типів, що є не найелегантнішим рішенням. Написання поліморфної функції на мові Java призведе до необхідності використання шаблонів, що еквівалентно оголошенню параметрів функції як об'єктів типу Object, а це по суті дозволяє програмісту посилати до функції що завгодно, навіть об'єкти, які за логікою не можуть пройти в функцію. В результаті виходить, що більшість функцій, які пишуться на мові Java, не є загальними [2]. Нижче розглянемо реалізацію поліморфізму в мові Haskell і проаналізуємо різні типи поліморфізму, присутні в цій мові програмування.

Поліморфізм функцій

В мові Haskell існує два види поліморфізму - параметричний і спеціальний (названий в англійських джерелах взятим з латини терміном «ad hoc»).

Параметричний поліморфізм

Параметричний поліморфізм - це можливість визначення узагальнених структур даних і функцій, поведінка яких не залежить від типів значень, якими вони оперують. У випадку типів даних (конкретніше, алгебраїчних типів даних, які, як показано в [5], можна інтерпретувати як контейнерні типи) значення довільних типів можуть тим чи іншим чином використовуватися всередині контейнерів (безпосередньо міститися в контейнерах, або вміст контейнерів залежатиме від таких довільних типів). У випадку функцій їх поведінка не залежить від типів переданих значень в якості вхідних параметрів.

Класифікація параметричного поліморфізму заснована на обмеженні рангу поліморфізму і на обмеженні використання типових змінних (за аналогією з термінами «рядкова змінна», «булева змінна» та ін.). Виділяють наступні типи параметричного поліморфізму:

а) непредикативний поліморфізм - дозволяє інстанціювати типові змінні при конкретизації довільними типами, в тому числі і поліморфними;

б) предикативний поліморфізм - на відміну від непредикативного поліморфізму інстанціювання типових змінних при конкретизації типу може проводитися тільки неполіморфними (мономорфними) типами, які іноді називаються «монотипія»;

в) поліморфізм рангу * (rank * polymorphism) - замість символу підстановки * можуть використовуватися значення «1», «k» і «N». В поліморфізмі першого рангу (цей тип поліморфізму ще називають «випереджальним поліморфізмом» або «let-поліморфізмом») типові змінні можуть отримувати конкретні значення мономорфних типів. Поліморфізм рангу k припускає, що в формулах, які описують λ-терми, квантор загальності (∀) може стояти перед не більше ніж k стрілками. Слід зауважити, що при k = 2 проблема виведення типів розв'язана, в той час як при k > 2 ця проблема нерозв'язна. Нарешті, поліморфізм вищого рангу (або поліморфізм рангу N) визначається тим, що квантори загальності можуть стояти перед довільною кількістю стрілок.

В мові Haskell досить багато прикладів параметричного поліморфізму, наприклад, у функціях length, head, teal, curry, uncurry, map, filter та ін. Наведемо реалізацію деяких з них:

```
length :: [a] -> Int
length [] = []
length (_: xs) = 1 + length xs

head :: [a] -> a
head (x: _) = x

tail :: [a] -> [a]
tail (_: xs) = xs

curry :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)

uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry g (x, y) = g x y
```

У перших трьох функцій вхідний параметр типу a, а в curry та uncurry ще b і c. Замість конкретного типу даних (Int, Bool, Char, ...) використовується типізація. Приклад на java:

```
public class Test <A> {...}
```

Тут теж використовується A - невідомий тип даних, який буде визначений під час компіляції. Поліморфізм використовується тоді, коли невідомо, який тип матиме параметр, але відомі операції, що над ним (або з ним) будуть проводитися.

Всі ідентифікатори типових змінних в мові Haskell повинні починатися з маленької літери (наприклад, a, abc, aA101), а всі конкретні типи (конструктори) – з великої (наприклад String, Int, Node). Параметр a може приймати будь-які типи: Int, String чи інші або навіть функції (наприклад, length [f, g, h], де f, g, h - функції, які мають однакові типи). Варто зауважити, що тип b може також приймати будь-які типи, в тому числі і тип параметра a.

Тип функції в інтерпретаторі GHCi (і в Hugs) завжди можна дізнатися за допомогою команди : t, наприклад:

```
Main>: t length
length :: [a] -> Int
```

Спеціальний (ad hoc) поліморфізм

Спеціальний (ad-hoc) поліморфізм, який ще називається поліморфізмом спеціального виду або «перевантаженням імен», дозволяє давати однакові імена програмним сутностям з різним поведінкою. Такий поліморфізм широко використовується в математиці, коли подібні математичні операції отримують одні й ті ж знаки (наприклад, арифметичні знаки (+), (-), (×) і (/) використовуються для позначення операцій додавання, віднімання, множення і ділення відповідно для довільних чисел - цілих, дійсних, комплексних та ін.). Це більш слабкий тип поліморфізму, ніж параметричний. Візьмемо для прикладу оператор (функцію) додавання (+). Вирази такого вигляду

```
(+) 2 3 - >> 5
(+) 2.5 3.85 - >> 6.35
```

відрізняються від виразів

```
(+) True False
(+) [1,2,3] [3,2,1]
```

тим, що в першому випадку були використані чисельні значення, а в другому значення типу Bool і [Int]. Оператор додавання не визначений для нечисельних типів. Все тому, що ця функція має тип не

```
(+) :: A -> a -> a
```

а такий

```
(+) :: Num a => a -> a -> a.
```

Тобто тут вводиться обмеження на типи даних, що вводяться (і виводяться). Обмеження, яке накладено на тип a: Num a говорить, що тип a повинен бути елементом класу Num. Такі класи типів дуже важливі в Haskell, так як вони додають додатковий захист від помилок при програмуванні, а також можуть зменшити кількість написаного коду в рази. Це ми побачимо в наступному прикладі.

Є наступні типи двійкових дерев:

Двійкове дерево, яке може зберігати будь-який тип даних

```
data Tree1 a = Nil | Node1 a (Tree1 a) (Tree1 a)
```

Двійкове дерево, яке може зберігати два елементи, і кожне відгалуження закінчується «листом» с одним елементом (a не Nil):

```
data Tree2 ab = Leaf2 b | Node2 ab (Tree2 ab) (Tree2 ab)
```

Двійкове дерево, яке може зберігати дані типу String і теж закінчується листом:

```
data Tree3 = Leaf3 String | Node3 String Tree3 Tree3
```

І, скажімо, є ще два види списків:

звичайний

```
type Lst a = [a]\
```

і тип "мотузка"

```
data Rope a b = Nil | Twisted b (Rope b a)
```

Маючи всі ці структури даних, можна написати функцію size, яка незалежно від структури даних, виводила б або її глибину (для дерев), або довжину (для списків), або одним словом - розмір. Наївним рішенням було б написати для кожного свою функцію:

```
sizeT1 :: Tree1 a -> Int – підраховуємо кількість
```

```
вузлів
sizeT1 Nil = 0
sizeT1 (Node1 _ l r) = 1 + sizeT1 l + sizeT1 r

sizeT2 :: (Tree2 ab) -> Int - підраховуємо к-ть елементів
sizeT2 (Leaf 2 _) = 1
sizeT2 (Node2 __ l r) = 2 + sizeT2 l + sizeT2 r

sizeT3 :: Tree3 -> Int - піраховуємо довжини елементів типу String
sizeT3 (Leaf3 m) = length m
sizeT3 (Node mlr) = length m + sizeT3 l + sizeT3
I для списків:
sizeLst :: [a] -> Int
sizeLst = length

sizeRope :: (Rope a b) -> Int
sizeRope Nil = 0
sizeRope (Twisted _ ls) = 1 + sizeRope ls
```

Тепер якщо з'явиться ще якась структура даних, то доведеться робити нову функцію, яка буде підходити тільки для цієї структури. А нам хотілося б, щоб однією функцією можна було отримати розмір будь-якої структури даних. Як у функції (+) було обмеження класом Num, так і в даному прикладі треба зробити обмеження класом Size, а для цього цій клас треба спочатку створити:

```
class Size a where
   size :: a -> Int
```

Таким чином, залишилося зробити екземпляри цього класу під конкретні значення a (тобто під конкретні типи даних):

```
instance Size (Tree1 a) where
   size Nil = 0
   size (Node1 _ l r) = 1 + size l + size r

instance Size (Tree2 a b) where
   size (Leaf2 _) = 1
   size (Node2 __ l r) = 2 + size l + size r

instance Size Tree3 where
   size (Leaf3 m) = length m
   size (Node3 m l r) = length m + size l + size r

instance Size [a] where
   size = length

instance Size (Rope a b) where
   size Nil = 0
   size (Twisted _ ls) = 1 + size ls
```

Тепер якщо в GHCi написати: t size, то побачимо size :: Size a => a -> Int. Тобто отримаємо:

```
size Nil - >> 0
size (Node1 "foo" (Node1 "bar" Nil Nil) Nil) - >>
2
size (Leaf 2 "foo") - >> 1
size (Node3 "foo" (Node3 "bar" (Leaf3 "abc")
(Leaf "cba")) (Leaf "tst")) - >> 15 - 3 * 5
size [1 .. 5] - >> 5
size (Rope 2 (Rope 'a' (Rope 5 Nil))) - >> 3
```

Кожен екземпляр класу Size реалізував функцію size, яка застосовна лише до значень певного типу. В даному випадку ця функція, як і інші предвизначені оператори (+), (*), (-), перевантажена. Але є і свої мінуси у такого рішення. Наприклад, хотілося дізнатися кількість елементів у парному списку, тобто

```
size [(1,2), (3,4)] - >> 4
size [('a', 'b'), ('c', 'd'), ('e', 'f')] - >> 6
```

Очевидно можна було б зробити наступне:

```
instance Size [(a, b)] where
   size = (* 2). Length
```

Але тут виникне проблема, через те, що вже є екземпляр для звичайного списку (instance Size [a] where) та більше не можна використовувати інше визначення, тому що, як вже говорилося раніше, тип a може бути будь-яким типом, в тому числі і (b, c), тобто. [A] == [(b, c)] Для вирішення даної проблеми можна використовувати Overlapping Instances (англ. перекривають екзепляри класу). У цього рішення є і свої мінуси (імпорт одного з модулів може поміняти значення програми, може викликати помилки, і так далі).
Розглянемо класи в Haskell більш докладно.

Класи

У попередньому прикладі був створений клас Size. Неповна декларація класу в Haskell має наступний вигляд: (tv = type variable):

```
class Name tv where
   сигнатура включає змінну tv
```

Неповна тому, що в декларацію класу також можуть бути введені обмеження на tv (наприклад, tv повинна бути елементом класу Ord). Сигнатур може бути скільки завгодно,

але кожна з них повинна включати (як вхідний так / або вихідний параметр) змінну tv.
Типові класи - це колекції типів, для яких визначені спеціальні функції. Ось деякі (одні з найважливийших) класи в Haskell:

Eq - клас для тесту на рівність (нерівність) двох типів даних (операції ==, / =)
Ord - клас для визначення порядку типів, тобто який елемент більше, який менше (операції>,> =, <, <=, min, max ...)
Enum - клас для типів, чиї значення можна «порахувати» (наприклад, [1 .. 10])
Bounded - клас теж використовується для типів класу Enum. Використовується для найменування
нізжшей і вищої межі типу.
Show - клас для типів, значення яких можна перетворити в рядок, (= можна представити як символи)
Read - клас для типів, значення яких можна перетворити з рядка

Тільки ці класи можна автоматично успадковувати будь-якого типу даних (= помістити в секцію deriving). Між класами існує теж залежність. Наприклад, якщо відомо як порівнювати два елементи (клас Ord), то тоді треба заздалегідь вміти визначати чи дорівнює один елемент іншому (клас Eq). Адже для реалізації оператора (> =) треба мати реалізований оператор (==). Тому, можна сказати, що клас Ord залежить (успадковує) від класу Eq. Більш детальна схема залежності класів представлена на рис.1. Розглянемо клас еквівалентності Eq більш детально. Як вже раніше було згадано, цей клас повинен мати дві функції (==) та (/ =):

```
class Eq a where
   (==), (/ =) :: A -> a -> Bool - так як сигнатури (==) та (/ =) однакові, їх можна записати в 1 рядок

   x / = y = not (x == y)
   x == y = not (x / = y)
```

З коду видно, що для будь-якого екземпляру класа Eq, достатньо реалізувати одну з двох функцій. Наприклад, для типу Bool (два типи дорівнюють лише тоді, коли вони обидва True або False) це зроблено таким чином:

```
instance Eq Bool where
   (==) True True = True
   (==) False False = True
   (==) __ = False
```

Як видно, випадок True False і False True (так само як і останній рядок) не обов'язково писати, тому нерівність є зворотною операцією рівності. Якщо до цього у нас було так, що тип є кокретізацією іншого типу (наприклад, [Int] є інстанцією типу [a]), то тепер тип може бути екземпляр класу (наприклад, Bool є інстанцією класу Eq). З цієї ж аналогією, можна написати функції рівності тих двійкових дерев, що ми використовували вище. Наприклад, для Tree2:

```
instance (Eq a) => Eq (Tree2 a) where - накладаємо умову на a, що цей тип може порівнюватися
   (==) (Leaf2 s) (Leaf2 t) = (s == t)
   (==) (Node2 s t1 t2) (Node2 t u1 u2) = (s == t) && (t1 == u1) && (t2 == u2)
   (==) __ = False
```

Для (Tree3 ab) ми вже повинні будемо накладати умову як на a, так і на b, тобто
instance (Eq a, Eq b) => Eq (Tree3 a b)

Все що лівіше символу => називається контекстом, все що правіше цього символу повинне бути або базовим типом (Int, Bool, ...), або конструктори типів (Tree a, [...], ...). Оператор (==) є перевантаженою функцією (ad-hoc поліморфа).

Рівність (==) є тіпозавісимим властивістю, яке вимагає реалізації під кожен тип (як наприклад реалізація рівності двійкових дерев). Хотілося б, щоб можна було порівнювати дві функції таким чином:
(==) Fac fib - >> False - де fac - факторіал числа, а fib - число Фібоначчі

```
(==) (\ X -> x + x) (\ x -> 2 * x) - >> True
(==) (+2) (2 +) - >> True
```

Це вимагало б в Haskell написання такого коду:

```
instace Eq (Int -> Int) where
(==) F g = ... - Де f і g є якимись функціями
```

Чи можливо написати таку функию, яка б видавала результат рівності двох будь-яких

функцій (True або False)? На жаль, з тези Черча-Тьюринга випливає, що рівність двох будь-яких функцій можна визначити. Не існує алгоритму, який би для двох будь-яких даних функцій, завжди і через кінцеве кількість кроків вирішував, чи є дві функції рівними чи ні. Такий алгоритм можна запрограмувати ні на одній мові програмування. Замість реалізації своєї функції для рівності, наприклад двійкових дерев, можна завжди помістити ті класи, які успадковуються даним типом автоматично після ключового слова deriving. Тобто можна написати:

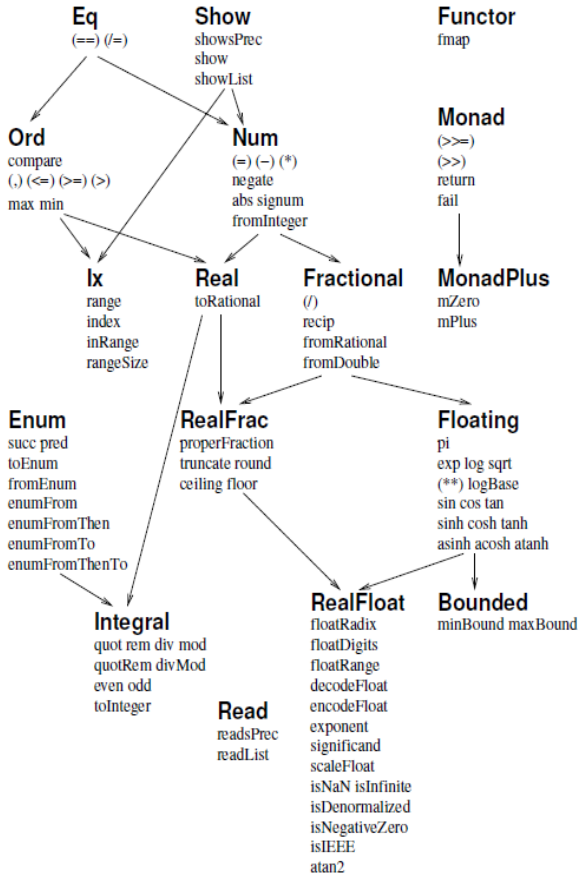

Рис.1 Залежність стандартних класів в мові Haskell

data Tree a = Nil | Node a (Tree a) (Tree a) deriving Eq

Це дозволяє не писати власноручно інстанцію класу Eq (instance Eq a => Eq (Tree a) where ...). Але для будь-якої функції, тоді доведеться накладати умову на змінну a, що ця змінна є елементом класу Eq. Іноді, все-таки доводиться писати ці функції, так як «автоматичне» порівняння не завжди працює так як ми б того хотіли. Наприклад для двійкового дерева
data Tree3 ab = Leaf3 bl | Node3 ab (Tree3 ab) (Tree3 ab) deriving Eq
буде реалізована (автоматично) наступна функція:

```
instance (Eq a, Eq b) => Eq (Tree3 a b) where
(==) (Leaf3 q) (Leaf3 s) = (q == s)
(==) (Node3 _ q t1 t2) (Node3 _ s u1 u2) = (q == s) && (t1 == u1) && (t2 == u2)
(==) __ = False
```

Як ми бачимо, перший аргумент Node3 був опущений, чого би не хотілося в деяких випадках.

Висновки.

Таким чином, в даній статті було розглянуто реалізацію поліморфізму в функціональній мові програмування Haskell. Проаналізовано відмінність між двома формами поліморфізму: спеціальним та параметричним. Неформальне відмінність двох типів поліморфізму:

параметричний - один і той же код незалежно від типу

ad-hoc - різний код, хоча однакові імена функцій

Список використаних джерел

1. www.haskell.org
2. N.A.Roganova. Functional programming. M.:MGIU 2007
3. J.Harrison. Introduction in functional programming. University of Cambridge 1998
4. A.Field, P.Harrison. Functional programming. M.:Mir, 1993.
5. R.Dushkin. Algebraic data types and their use in programming. Practice of functional programming, №2, p.86–105, 2009.

Надійшла до редколегії: 05.03.2013