Вивчить собі Хаскела на велике щастя!

Автор: Міран Ліповача

Переклад здійснили: Ганна Лелів, Семен Тригубенко, Богдан Пеньковський, Марина Стрельчук і Тетяна Богдан

Мовні редактори: Тетяна Богдан і Ганна Лелів
Науковий редактор: Семен Тригубенко

Переклад виконано за підтримки
Словненя

2017-05-21T00:06:15Z
Версія v4.7-54-gda41cf2

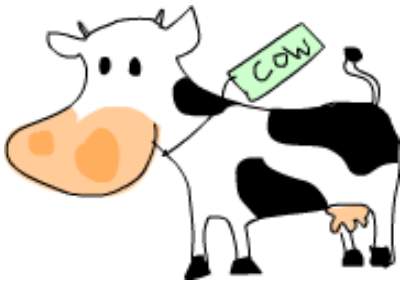
Зміст

Розділ 3

Типи і типокласи

Переклад українською Ганни Лелів

3.1 Повірте типові



Я вже згадував, що Хаскел має статичну систему типів. Тип кожного виразу є відомим під час компіляції, і тому код виходить надійніший. Якщо ви напишете програму, де ви спробуєте розділити булів тип із якимось числом, то вона навіть не скомпілюється. І це добре, адже краще вловити такі помилки під час компіляції, аніж мати аварійне завершення програми при виконанні. У Хаскелі все має тип, тому компілятор зробить чимало висновків про вашу програму ще до того, як її скомпілює.

На відміну від Java чи Pascal, Хаскел має виведення типів. Коли ми пишемо число, то не мусимо вказувати Хаскелу, що це є число. Він може самостійно це *вивести*, тому нам не треба явно розписувати типи функцій і виразів під час написання коду. В двох попередніх розділах ми зробили огляд головних ідей Хаскела, буквально у двох словах зачепивши типи. Проте, розуміння системи типів є дуже важливою частиною вивчення Хаскел.

Тип — це щось на кшталт ярлика, що його має кожен вираз. Він каже нам, якій категорії речей «підходить» цей вираз. Вираз `True` — це булів вираз, `”hello”` — це рядок, тощо.

А зараз скористаймося GHCi, щоб розглянути типи деяких виразів. У цьому нам допоможе команда `:t`, яка, якщо після неї подати коректний вираз, вкаже нам його тип. Ну що — поїхали!

```
ghci> :t 'a'
'a' :: Char
ghci> :t True
True :: Bool
ghci> :t "HELLO!"
"HELLO!" :: [Char]
ghci> :t (True, 'a')
(True, 'a') :: (Bool, Char)
ghci> :t 4 == 5
4 == 5 :: Bool
```

Застосування `:t` до виразу виводить той самий вираз, а після нього `::` і його тип. `::` читаємо як «має тип». Типи завжди пишуться з великої літери. Як бачимо, схоже на те, що `'a'` має тип `Char`. Не важко зрозуміти (якщо знати, як перекладається *character* з англійської!), що Char тут означає *символ*. `True` має тип `Bool`. Наразі все зрозуміло. Але що *це* таке?! Перевірка типу `"HELLO!"` вертає `[Char]`. Квадратні дужки позначають список. Отож, це читається *список символів*. На відміну від списків, кортежі різних довжин — різні типи. Тобто вираз `(True, 'a')` має тип `(Bool, Char)` тоді як вираз `('a','b','c')` матиме тип `(Char, Char, Char)`. Вираз `4 == 5` завжди вертатиме `False`, тому його тип є `Bool`.

Кожна функція теж має тип. Коли ми пишемо свої власні функції, то можемо, якщо захочемо, оголошувати їхні типи. Добрі програмісти завжди так і роблять, за винятком, можливо, випадків коли пишуть дуже короткі функції. Відтепер ми оголошуватимемо тип усіх функцій які ми писатимемо. Пригадуєте, як ми створили списковий характер, який фільтрує рядок так, щоб залишалися тільки великі літери? Ось як він виглядає разом із оголошенням типу.

```
removeNonUppercase :: [Char] -> [Char]
removeNonUppercase st = [ c | c <- st, c `elem` ['A'..'Z']]
```

`removeNonUppercase` має тип `[Char] -> [Char]`, тобто, ця функція перетворює рядок на рядок. Це тому що вона бере один рядок як параметр і повертає інший рядок як результат. Тип `[Char]` — ідентичний до `String`, тому зрозуміліше буде написати: `removeNonUppercase :: String -> String`. Ми не мусили оголосити тип цієї функції, тому що компілятор може самотужки вивести, що це функція, яка перетворює рядки на рядки. Але ми все-таки це зробили. А як написати тип функції, яка приймає кілька параметрів? Ось простенька

функція, яка бере три цілі числа та додає їх:

```
addThree :: Int -> Int -> Int -> Int
addThree x y z = x + y + z
```

Параметри відокремлюються `->`, а між параметрами і результатом обрахунку функції немає особливої різниці. Тип результату — це останній елемент в оголошенні, а перші три — це типи параметрів. Згодом я поясню, чому ми відокремлюємо і параметри і результат `->`, а не робимо якоїсь більш очевидної різниці між типом результату функції та типами параметрів на кшталт `Int, Int, Int -> Int` або ще якось.

Якщо ви хочете оголосити тип своєї функції, але не впевнені, яким саме він повинен бути, то просто напишіть функцію без оголошення, а тоді дізнайтеся її тип за допомогою `:t`. Функції — це вирази також, тож `:t` працюватиме з ними без жодних проблем.

Ось короткий огляд найпоширеніших типів.

Тип `Int` використовується для цілих чисел. Назва — скорочення англійського слова *integer*. `7` може бути `Int`, а `7.2` — ні. `Int` обмежений, тобто, він має мінімальне та максимальне значення. На 32-бітних машинах максимальне можливе значення `Int` зазвичай є 2147483647, а мінімальне — -2147483648.

Тип `Integer` — це теж цілі числа… [От лише назвою він ніяк не зв'язаний із *integer* з англійської… (жартую!)]. Різниця в тому, що цей тип необмежений, тобто він може представляти дійсно величезні числа. Просто гігантські. Водночас, `Int` — ефективніший.

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

```
ghci> factorial 50
30414093201713378043612608166064768844377641568960512000000000000
```

Тип `Float` — це дійсні числа, представлені як числа із плаваючою комою, одинарної точності.

```
circumference :: Float -> Float
circumference r = 2 * pi * r
```

```
ghci> circumference 4.0
25.132742
```

Тип `Double` — це дійсні числа також, представлені як числа із плаваючою комою, але подвійної точності!

```
circumference' :: Double -> Double
circumference' r = 2 * pi * r
```

```
ghci> circumference' 4.0
25.132741228718345
```

Тип `Bool` — це булів тип. Він може мати лише два значення: `True` і `False`.

Тип `Char` представляє символ. Символи беруться в ординарні лапки. Список символів — це рядок.

Кортежі — це типи, але вони залежать не тільки від типів своїх складових, а й від своєї довжини також. Тому, теоретично, існує нескінченна кількість типів кортежів, і її неможливо описати в цьому посібнику. Зверніть увагу, що порожній кортеж `()` — це тип також, і він може мати лише одне значення: `()`.

3.2 Змінні типу

Як ви гадаєте — який тип має функція `head`? `head` бере список з елементів будь-якого типу та повертає перший елемент. То якого вона типу? Перевірмо!

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

Гммм! Що таке `a`? Це тип? Пригадуєте, я вже казав, що типи пишуться з великої літери, отож це не може бути типом. Оскільки воно пишеться з малої літери — це **змінна типу**. А це значить, що `a` може мати будь-який тип. Схоже на узагальнені засоби в інших мовах програмування, але в Хаскелі така штука називається параметричним поліморфізмом, і в Хаскелі вона набагато могутніша, адже дає змогу легко написати дуже загальні функції (якщо вони не використовують ніякі спеціалізовані поведінки типів із якими оперують). Функції, в сиґнатурі яких є змінні типу, називаються **поліморфними функціями**. Оголошення типу функції `head` каже, що вона бере список з елементів будь-якого типу та повертає один елемент такого типу.

Імена змінних типу можуть містити понад один символ, але зазвичай їх називають a, b, c, d...

Пригадуєте функцію `fst`? Вона повертає перший компонент пари. Перевіримо її тип.

```
ghci> :t fst
fst :: (a, b) -> a
```

Як бачимо, `fst` бере кортеж, який містить два типи, та повертає елемент, який має такий самий тип, як перший компонент пари. Ось чому `fst` можна

застосувати для пари, що містить будь-які два типи. Зверніть увагу — `a` і `b` не мусять бути різного типу тільки тому, що це дві різні змінні типу. Тип функції всього лиш каже, що тип першого компонента і тип результату — одинакові.

3.3 Вступ до вступу до типокласів

Типоклас — це щось на кшталт інтерфейсу, який означує якусь поведінку. Якщо тип належить до певного типокласу, це означає, що він підтримує та реалізує поведінку, що її описує цей типоклас. Це збиває з пантелику чимало людей з шкіл об'єктно-орієнтованого програмування, тому що вони гадають, що це те саме, що класи у об'єктно-орієнтованих мовах. Але це не так. Типокласи нагадують Java інтерфейси. Тільки вони кращі.

Яка сиґнатура функції `==`?

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Примітка: Оператор рівності `==` — це функція. Як і `+`, `*`, `-`, `/` та майже всі інші оператори. Якщо назва функції складається тільки зі спеціальних символів, функція за замовчуванням вважається інфіксною. Якщо ми хочемо перевірити її тип, передати її іншій функції чи викликати її як префіксну функцію, ми мусимо взяти її в дужки.

Цікаво. Ми побачили щось нове — символ `=>`. Усе, що передує символові `=>` називається **умовою типокласу**. Попереднє оголошення можна прочитати ось так: функція перевірки на рівність бере будь-які два значення однакового типу та повертає `Bool`. Тип обидвох значень мусить належати до типокласу `Eq` (це і була умова типокласу).

Типоклас `Eq` надає інтерфейс для перевірки на рівність. Будь-який тип, для двох значень якого має сенс перевірка на рівність, має належати до типокласу `Eq`. У Хаскелі усі стандартні типи — окрім IO (тип, що має справу з вводом і виводом) і функцій — належать до типокласу `Eq`.

Функція `elem` має тип `Eq a => a -> [a] -> Bool`, тому що вона використовує `==` для перевірки, чи є в списку потрібне нам значення.

Ось кілька основних типокласів:

Типоклас `Eq` використовують для типів, що підтримують перевірку на рівність. Його члени реалізують функції `==` і `/=`. Отож, якщо на змінну типу у функції накладено умову `Eq`, десь в означенні цієї функції використовується `==` або `/=`. Усі раніше згадані типи, окрім функцій, належать до `Eq`, тому значення таких типів можна перевіряти на рівність.

```
ghci> 5 == 5
True
ghci> 5 /= 5
False
ghci> 'a' == 'a'
True
ghci> "Ho Ho" == "Ho Ho"
True
ghci> 3.432 == 3.432
True
```

Типоклас `Ord` — для типів, що мають впорядковування.

```
ghci> :t (>)
(>) :: Ord a => a -> a -> Bool
```

Усі вищезгадані типи, окрім функцій, належать до `Ord`. `Ord` охоплює усі типові функції порівняння, як-от `>`, `<`, `>=` і `<=`. Функція `compare` бере два члени `Ord` однакового типу та повертає упорядкування. `Ordering` — це тип, що може мати три значення — `GT`, `LT` або `EQ`, що означає *більший ніж*, *менший ніж* і *рівний*, відповідно.

Щоб належати до `Ord`, тип мусить спершу бути членом престижного та ексклюзивного клубу `Eq`.

```
ghci> "Abrakadabra" < "Zebra"
True
ghci> "Abrakadabra" `compare` "Zebra"
LT
ghci> 5 >= 2
True
ghci> 5 `compare` 3
GT
```

Членів типокласу `Show` можна серіалізувати в рядок. Усі вищезгадані типи, окрім функцій, належать до `Show`. Найпоширеніша функція, що працює з типокласом `Show` — це `show`. Вона бере значення, тип якого належить до `Show`, переводить його в рядок і показує його.

```
ghci> show 3
"3"
ghci> show 5.334
"5.334"
ghci> show True
"True"
```

Типоклас `Read` — це по суті протилежний типоклас до `Show`. Функція `read` бере рядок та повертає тип, який належить до `Read`.

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

Наразі все ОК. Знову ж таки — усі вищезгадані типи належать до цього типокласу. А що трапиться, якщо виконати просто `read "4"`?

```
ghci> read "4"
<interactive>:1:0:
    Ambiguous type variable `a' in the constraint:
      `Read a' arising from a use of `read' at <interactive>:1:0-7
    Probable fix: add a type signature that fixes these type variable(s)
```

GHCi каже нам, що не знає, *що* ми хочемо отримати. Зверніть увагу: коли ми задіювали `read` раніше, ми потім якось використовували її результат. Завдяки цьому GHCi міг вивести, якого типу результат ми хотіли отримати від `read`. Якщо ми використовували його як булеве значення, то він повертав `Bool`. Але тепер, він знає, що ми хочемо якийсь тип, що належить до класу `Read`, але не знає, який саме. Погляньмо на типосигнатуру `read`.

```
ghci> :t read
read :: Read a => String -> a
```

Бачите? Він повернув тип, що належить до `Read`, але якщо ми не будемо його якось використовувати пізніше, немає можливості **визначити** [to determine] який саме тип треба. Ось для чого існують **анотації типу**. За допомогою анотацій типу можна явно вказати якого типу вираз має бути. Щоб створити таку анотацію треба додати `::` наприкінці виразу, а тоді вказати його тип. Дивіться:

```
ghci> read ”5” :: Int
5
ghci> read ”5” :: Float
5.0
ghci> (read ”5” :: Float) * 4
20.0
ghci> read ”[1,2,3,4]” :: [Int]
[1,2,3,4]
ghci> read ”(3, ’a’)” :: (Int, Char)
(3, ’a’)
```

Компілятор зазвичай може самостійно **вивести** [to infer] тип більшості виразів. Але часом компілятор не знає, чи повертати значення типу `Int` чи `Float` для виразу на кшталт `read ”5”`. Щоб **визначити** [to determine] тип, Хаскел мусить вирахувати `read ”5”`. Але ж Хаскел — це статично типізована мова, тому він мусить знати всі типи наперед, до компіляції (або, у випадку GHCi — до обрахунку). Тому мусимо сказати Хаскелові: «Гей, на випадок, якщо ти цього не знаєш, цей вираз повинен мати *оцей* тип!».

Члени типокласу `Enum` — це типи, втілення яких можна перелічити і впорядкувати. Основна перевага типокласу `Enum` полягає в тому, що його типи можна використовувати в діапазонах списків. Такі типи також мають означені наступні та попередні елементи, що їх можна отримати за допомогою функцій `succ` і `pred`, відповідно. Типи в цьому типокласі: `()`, `Bool`, `Char`, `Ordering`, `Int`, `Integer`, `Float` і `Double`.

```
ghci> [’a’..’e’]
”abcde”
ghci> [LT .. GT]
[LT,EQ,GT]
ghci> [3 .. 5]
[3,4,5]
ghci> succ ’B’
’C’
```

Члени типокласу `Bounded` мають верхню і нижню межу.

```
ghci> minBound :: Int
-2147483648
ghci> maxBound :: Char
’\1114111’
ghci> maxBound :: Bool
True
ghci> minBound :: Bool
```

```
False
```

`minBound` і `maxBound` варті уваги, тому що вони мають тип `Bounded a => a`. До певної міри, вони — поліморфні сталі.

Кортеж належить до `Bounded`, якщо до `Bounded` належить кожна з його компонент.

```
ghci> maxBound :: (Bool, Int, Char)
(True,2147483647,'\1114111')
```

Клас `Num` — це чисельний типоклас. Його елементи здатні діяти як числа. Розгляньмо тип числа:

```
ghci> :t 20
20 :: Num t => t
```

Здається, що цілі числа — це теж поліморфні сталі. Вони можуть поводитися як будь-який тип, що належить до типокласу `Num`.

```
ghci> 20 :: Int
20
ghci> 20 :: Integer
20
ghci> 20 :: Float
20.0
ghci> 20 :: Double
20.0
```

Це типи, що входять до типокласу `Num`. Коли ми перевіримо тип `*`, то побачимо, що він працює із усіма членами чисельного типокласу.

```
ghci> :t (*)
(*) :: Num a => a -> a -> a
```

Він бере два числа однакового типу та повертає число такого ж типу. Ось чому `(5 :: Int) * (6 :: Integer)` видасть помилку типу, тоді як `5 * (6 :: Integer)` спрацює і поверне `Integer`, адже `5` здатний поводитися як `Integer`, так і як `Int`.

Щоб приєднатися до `Num`, тип уже має приятелювати з `Show` і `Eq`.

Типоклас `Integral` — це також чисельний типоклас. `Num` включає всі числа — дійсні та цілі, а `Integral` — тільки цілі. До цього типокласу належать `Int` і `Integer`.

До типокласу `Floating` належать тільки числа з плаваючою комою, тобто `Float` and `Double`.

Функція `fromIntegral` — це вкрай корисна функція для роботи з числами. Її сиґнатура — `fromIntegral :: (Num b, Integral a) => a -> b`, і з неї випливає, що ця функція бере ціле число з `Integral` та перетворює його у загальне число з `Num`. Вона стане вам у пригоді, коли ви захочете, щоб цілі числа і числа з плаваючою комою гарненько співпрацювали. До прикладу, сиґнатура функції `length` є `length :: [a] -> Int`, а не більш загальна `length :: Num b => [a] -> b`. Напевно так склалося історично, хоча на мою думку — це тупо. Хай там як, якщо ми спробуємо отримати довжину списку, а тоді додати її до `3.2`, то отримаємо помилку, бо ми намагаємося додати докупи `Int` і число з плаваючою комою. Спробуймо це обійти — виконаймо `fromIntegral (length [1,2,3,4]) + 3.2` — і все супер.

Зверніть увагу, що `fromIntegral` має кілька обмежень класу у сиґнатурі. Це цілком правильно. Як бачите, обмеження класу відокремлюються комами всередині дужок.

Покажчик