Fix broken link to quiz
hamayumm authored 2 days ago

463c2d2a

polymorphism.md 5.25 KB

Polymorphism

Introduction

Some functions act on elements of different types, independently of the precise shape of the elements. This is known as **polymorphism**. The topic is introduced in the video "Polymorphism - Introduction" on Canvas.

Example: the type family of lists

Let's look at the datatype of lists. We have seen in Section 3.3 of our textbook that lists can be filled with elements of different types:

```
Prelude> :t ['a', 'b', 'c']   -- a list of characters
['a', 'b', 'c'] :: [Char]
Prelude> :t [False, True, True, True]   -- a list of Booleans
[False, True, True, True] :: [Bool]
Prelude> :t ["foo", "bar"]   -- a list of lists of characters, aka, a list of strings
["foo", "bar"] :: [[Char]]
```

For any type `a`, we can form the type `[a]`. Its elements are lists of elements of type `a`.

How to build lists

There are two ways of creating a list:

1. The empty list is a list, `[]`.
2. To a given list `xs` we can prepend an element `x`, written `x:xs`.

The empty list

Firstly, for any type `a`, there is the empty list of elements of `a`:

```
Prelude> :t []
[] :: [a]
```

Here, `a` in `[] :: [a]` is a **type variable**, which can be instantiated with any type. For instance, `a` can be set to be `Integer`, the type of integer numbers in GHC:

```
Prelude> :t [] :: [Integer]
[] :: [Integer] :: [Integer]
```

In this example, we did not use type *inference*, but instead type *checking*: we suggest to `ghci` that `[]` should have type `[Integer]`, and ask `ghci` to confirm, which it does. Similarly, `ghci` is happy to confirm that `[]` has type `[Char]`:

```
Prelude> :t [] :: [Char]
[] :: [Char] :: [Char]
```

Or:

```
Prelude> :t [] :: [[Char]]
[] :: [[Char]] :: [[Char]]
```

Or, for that matter,

```
Prelude> :t [] :: [[[[[[[Char]]]]]]]
[] :: [[[[[[[Char]]]]]]] :: [[[[[[[Char]]]]]]]
```

Exercise

For each of these examples, what is the type that the variable `a` is instantiated with?

Explanation: Watch the video "Polymorphism: empty list []" on Canvas.

Adding an element to a list

There is also a way to build a new list from a smaller one, by adding a new element at the beginning:

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

The operator `:` is an **infix** operator; that means that it is used **between** its two arguments (similar to the symbol "+" for addition):

```
Prelude> :t False:[True, False]
False:[True, False] :: [Bool]
```

We can evaluate the expression `False:[True, False]`:

```
Prelude> False:[True, False]
[False,True,False]
```

It is **crucial** that in the expression `x:xs`, where `x :: a`, `xs :: [a]`, **for the same** `a`. The following fails:

```
Prelude> 'a':[True, False]

<interactive>:11:6: error:
    • Couldn't match expected type ‘Char’ with actual type ‘Bool’
    • In the expression: True
      In the second argument of ‘(:)’, namely ‘[True, False]’
      In the expression: 'a' : [True, False]

<interactive>:11:12: error:
    • Couldn't match expected type ‘Char’ with actual type ‘Bool’
    • In the expression: False
      In the second argument of ‘(:)’, namely ‘[True, False]’
      In the expression: 'a' : [True, False]
```

Exercise

Read the error message in the example above, and explain in your own words what it means.

Explanation: Watch the video "Polymorphism: consing (:)" on Canvas.

Polymorphism

A function whose type contains one (or several) variables is called **polymorphic**. The functions `[]` and `(:)` are examples of polymorphic functions.

Here is an example of a function whose type contains **two** type variables:

```
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Quiz time

Test your understanding by taking this quiz. Don't worry, it is not marked, and you can take it as many times as you want.

Exercises:

1. Explain, in your own words, what the function `zip` does. In the expression `zip ['x', 'y'] [False]`, what are the type variables `a` and `b` of `zip :: [a] -> [b] -> [(a, b)]` instantiated by?
2. Find out the types of the following functions. Decide if they are polymorphic.
    1. `fst`
    2. `(++)`
    3. `not`
    4. `head`

    5. `tail`
    6. `id`
3. Find a polymorphic function in the GHC standard library whose type contains 3 type variables or more.
4. Read Section 3.7 of Programming in Haskell. Compare the types of the examples given there with the types `ghci` indicates. (Note: some of the types that `ghci` shows use "type classes" - you will learn about these in the next lesson.)

Summary:

1. A **polymorphic** function is a function whose type contains **type variables** (which are typically called `a`, `b`, etc.).
2. When applying a polymorphic function to an input, the type variables are suitably **instantiated**, e.g., for `(++) : [a] -> [a] -> [a]` applied to `[True, False]` and `[False, False]`, Haskell instantiates `a` to `Bool`.