Fix quiz link in types handout
hamayumm authored 2 days ago
8bb6885a

types.md 10.2 KB

Types in Programming, types in Haskell

Introduction

The topic is introduced in the video "Typing - Introduction" on Canvas.

Why are types useful?

The purpose of types in programming languages is to **disallow** certain programmes. Specifically, types ensure that we only feed those arguments to a function that the function was designed to handle.

For instance, consider a function designed to return the double of a natural number. If we were to feed it with a string instead of a natural number by mistake, the programme would not know what to do. Types allow us to discover such mistakes at compile time - and fix the mistake before shipping the software - instead of at runtime, when it is too late! In the specific instance above, the compiler could understand, and warn about, the mistake of applying a function designed to act on natural numbers to an input of type `string`.

Most programming languages have a concept of typing, where every valid expression of the language is assigned a type. However, not all programming languages pay attention to these types to the same extent.

Type inference

Every valid expression in Haskell has a type, and we can ask `ghci` to tell us that type. This is called **type inference**:

```
Prelude> :type False
False :: Bool

Prelude> :type True
True :: Bool
```

Here, the double colon `::` should be read as "has type", e.g., "False has type Bool". It corresponds to the single colon `:` in Lean, e.g., `#check ff` (you can remind yourself of Lean here).

We also have **function types** in Haskell:

```
Prelude> :type not
not :: Bool -> Bool
```

As in Lean, function application is simply written by juxtaposition in Haskell. When applying an argument to a function of suitable type, the output type is the obvious one:

```
Prelude> :type not False
not False :: Bool
```

Type checking

We can ask `ghci` to **confirm** that a given expression has a given type:

```
Prelude> :type False :: Bool
False :: Bool :: Bool

Prelude> :type not False :: Bool
not False :: Bool :: Bool
```

This is called **type checking**. (Compare this to Lean, where `#check ff : bool` checks whether `ff` has the type `bool`.)

Remark on type checking vs type inference

When writing programmes in Haskell, it is much safer to indicate to Haskell which type we expect an expression to have (and have Haskell confirm that type), than to have Haskell figure out the type of the functions we are writing. A good programmer first writes down the type of the function they are constructing, before programming the function itself.

Exercise

Use `ghci` to find out the type of the following expressions:

1. `not (not (not False))`
2. `(True,False)` (see Section 3.4 of Programming in Haskell)
3. `['a', 'b', 'x']` (see Section 3.3 of Programming in Haskell)
4. `(++)` (What is strange about this type? See "polymorphic functions" discussed later.)

Tip: You can use `:t` instead of `:type` for brevity.

Explanation: Watch this video "Type-inference in Haskell using ghci" on Canvas.

Type inference before evaluation

Importantly, the type of a Haskell expression is inferred **before** the Haskell expression is evaluated. If an expression does not have a type, then it is not a valid Haskell expression, and Haskell will not evaluate the expression. For instance, the expression `if True then 1 else "foo"` does not have a valid type, since `1` is not of the same type as `"foo"`, and hence the expression is rejected. (Reason for the rejection: in a more complicated situation where `True` is replaced by a more complicated expression of type `Bool`, Haskell would not be able to ensure what the return type is: it could be either an `Integer` or a `[Char]`. This would mean that Haskell could not ensure **type safety**: the property that type errors occur only during compilation, not during evaluation (runtime).)

Exercise

- Write five ill-typed expressions in Haskell.
- Check their types in `ghci` - what does `ghci` say?
- What happens when you try to evaluate that expression?

Explanation: Watch this video "Ill-typed expressions in Haskell" on Canvas.

Well-typed programmes can fail

Not every valid (well-typed) Haskell expression evaluates without errors:

```
Prelude> :type (!!)
(!!) :: [a] -> Int -> a
Prelude> :type ["foo", "bar"] !! 5
["foo", "bar"] !! 5 :: [Char]
Prelude> ["foo", "bar"] !! 5
"*** Exception: Prelude.!!: index too large
```

Another example is division by zero:

```
Prelude> :type 1 `div` 0
1 `div` 0 :: Integral a => a
Prelude> 1 `div` 0
*** Exception: divide by zero
```

Exercise

Use `ghci` and/or the online documentation of Hackage: The Haskell Package Repository to find out what the function `(!!)` does.

Hint: you can find the documentation for the library for lists here.

Tour of Haskell types

In this section, we look at some Haskell types, both base types and composite types. A video accompanying this section is available here.

Base types

Haskell has various base types:

1. `Bool` - the type of Booleans
2. `Char` - the type of characters
3. `Int` - fixed-precision integers (between -(2^63) and (2^63)-1); be aware of buffer overflows!

4. `Integer` - arbitrary-precision integers
5. `Float` - numbers with decimal point
6. `Double` - similar to floats, but more precision

```
Prelude> :type False
False :: Bool
Prelude> :type 'c'
'c' :: Char
Prelude> :type "foo"
"foo" :: [Char]
Prelude> :type 2 :: Int
2 :: Int :: Int
Prelude> :type 2 :: Integer
2 :: Integer :: Integer
Prelude> :type (sqrt 2)
(sqrt 2) :: Floating a => a
Prelude> :type (sqrt 2) :: Float
(sqrt 2) :: Float :: Float
Prelude> :type (sqrt 2) :: Double
(sqrt 2) :: Double :: Double
Prelude> sqrt 2 :: Float
1.4142135
Prelude> sqrt 2 :: Double
1.4142135623730951
```

Composite types

We can form complicated types from simpler ones:

1. If `a` is a type, then `[a]` is the type of lists of values in `a`.
2. If `a` and `b` are types, then `(a,b)` is the type of pairs of a value in `a` and a value in `b`
3. Generalizing the above point, we also can build triples `(a, b, c)`, quadruples `(a, b, c, d)`, etc.
4. If `a` and `b` are types, then `a -> b` is the type of functions from `a` to `b`.

```
Prelude> :type [True, False]
[True, False] :: [Bool]
Prelude> :type []     -- Polymorphic function
[] :: [a]
Prelude> :type [['a', 'b'], ['c']]
[['a', 'b'], ['c']] :: [[Char]]
Prelude> :type "How are you?"
"How are you?" :: [Char]
Prelude> :type (False, 'c')
(False, 'c') :: (Bool, Char)
Prelude> :type not
not :: Bool -> Bool
Prelude> :type (++)
(++) :: [a] -> [a] -> [a] -- Polymorphic function
```

Curried functions

(See also Chapter 3.6 of "Programming in Haskell")

This section is discussed in a video. (The video also discusses the exercise at the end of the section.)

Consider the function

```haskell

add :: (Integer, Integer) -> Integer
add (x,y) = x + y
```

Since a function can also return a function, we can instead write this function as follows:

```haskell
add' :: Integer -> (Integer -> Integer)
add' x y = x + y
```

Notes:

1. This is the same principle as the fact that proving `A → B → C` is the same as proving `A ∧ B → C` in propositional logic!
2. The parentheses in `Integer -> (Integer -> Integer)` are not necessary; we write `Integer -> Integer -> Integer` instead (as we did in Lean).
3. Similarly, `add' x y` actually means `(add' x) y`; again the parentheses are not needed. (But they are needed when we want to write something like `f (g x)`).

This principle generalizes to functions with more than two arguments:

```
add3 :: Integer -> Integer -> Integer -> Integer
add3 x y z = x + y + z
```

Exercise: Insert all the omitted parentheses into this function and its type annotation.

(The solution is given in the video for this section.)

A useful `ghci` command: `:info`

Another very useful command besides `:type` is `:info`:

```
Prelude> :info Bool
data Bool = False | True        -- Defined in ‘GHC.Types’
instance Eq Bool -- Defined in ‘GHC.Classes’
instance Ord Bool -- Defined in ‘GHC.Classes’
instance Show Bool -- Defined in ‘GHC.Show’
instance Read Bool -- Defined in ‘GHC.Read’
instance Enum Bool -- Defined in ‘GHC.Enum’
instance Bounded Bool -- Defined in ‘GHC.Enum’
```

The first line gives the definition of the datatype `Bool` and its two values `False` and `True`. The meaning of the following lines starting with `instance` will be explained in the lesson on type classes.

```
Prelude> :info not
not :: Bool -> Bool     -- Defined in ‘GHC.Classes’
```

Unfortunately, `ghci` does not currently seem to have a command to show you the **body** of a function, e.g., the implementation of the function `not :: Bool -> Bool`. For this, you need to look at the source code, e.g., on Hackage.

Exercise

Read Chapters 3.2-3.6 of "Programming in Haskell", and type the examples into `ghci`. Use `:type` and `:info` generously in your explorations.

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.

Summary

1. Haskell expressions are type-checked before they are evaluated.
2. Only well-typed expressions can be evaluated.
3. Run-time errors can still happen in Haskell, but not errors related to typing.
4. In `ghci`, `:type` (= `:t`) and `:info` (= `:i`) can be used to find out more about types and expressions.