Works in GHCi> but not when loaded?

Asked 12 years, 2 months ago Modified 12 years, 2 months ago Viewed 351 times

0

I cant figure out why I get two different results but I'm sure it has to do with `IO`, which I am beginning to hate!

For example:

```
ghci> x <- readFile "foo.txt"
ghci> let y = read x :: [Int]
ghci> :t y
y :: [Int]
```

Now when I create that file and do the same thing it comes out as `IO [Int]` ?

`foo.txt` is a txt file containing only this: `12345`

Someone that can explain this to me? As I'm about to snap it!

Thanks for any insight!

haskell

Share
Improve this question
Follow

edited Oct 1, 2012 at 6:42

ДМИТРИЙ МАЛИКОВ
21.9k ● 11 ● 80 ● 133

asked Sep 30, 2012 at 18:56

user1501127
865 ● 1 ● 19 ● 32

2 probably you should write the whole program and add that here too ,,, – Satvik Sep 30, 2012 at 18:59

4 Answers

Sorted by: Highest score (default)

6

Read about ghci. To quote

> The syntax of a statement accepted at the GHCi prompt is exactly the same as the syntax of a statement in a Haskell do expression. However, there's no monad overloading here: statements typed at the prompt must be in the IO monad.

Basically you are inside the `IO` Monad when you are writing anything in ghci.

Share Improve this answer Follow

answered Sep 30, 2012 at 19:01

Satvik

11.2k ● 1 ● 39 ● 46

5

Main idea

Be clear on the distinction in Haskell between an `IO` operation that produces a value, and the value itself.

Recommended Reading

A good reference to IO in Haskell that doesn't expect you to want to know the theoretical basis for monads is sigfpe's [The IO Monad for People who Simply Don't Care](#).

Note it's the monad-theoretical bit that he's assuming you don't care about. He's assuming you do care about doing IO. It's not very long, and I think it's very much worth a read for you, because it makes explicit some 'rules' that you're not aware of so are causing you irritation.

Your code

Anyway, in your code

```
x <- readFile "foo.txt"
```

the `readFile "foo.txt"` bit has type `IO String`, which means it's an operation that produces a String. When you do `x <- readFile "foo.txt"`, you use `x` to refer to the String it produces. Notice the distinction between the output, `x` and the operation that produced it, `readFile "foo.txt"`.

Next let's look at `y`. You define `let y = read x :: [Int]`, so `y` is a list of Ints, as you specified. However, `y` isn't the same as the whole chunk that defines it.

```
example = do
  x <- readFile "foo.txt"
  let y = read x :: [Int]
  return y
```

Here `example :: IO [Int]`, whereas `y` itself has type `[Int]`.

The cause of your frustration

If you come from an imperative language, this is frustrating at first - you're used to being able to use functions that produce values wherever you'd use values, but you're used to those functions also being allowed to execute arbitrary IO operations.

In Haskell, you can do whatever you like with 'pure' functions (that don't use IO) but not IO operations. Haskell programmers see a whole world of difference between an IO operation that returns a value, which can only be reused in other IO operations, and a pure function which can be used anywhere.

This means that you can end up trapped in the awkward IO monad all the time, and all your functions are full of IO datatypes. This is inconvenient and you write messy code.

How to avoid IO mess

First solve the problem you have in its entirety *without* using external (file or user) data:

- Write sample data that you normally read from a file or user as values in your source code.
- Write your functions with any data you need in the definition as parameters to the function. This is the only way you can get data when you're writing pure code. Write pure code first.
- Test your functions on the specimen data. (If you like, you can reload in ghci every time you write a new function, making sure it does what you expect.)
- Once your program is complete without the IO, you can introduce it as a wrapper around the pure code at the end.

This means that in your program, I don't think you should be writing any `readFile` or other IO code until you're nearly finished.

It's a completely different workflow - in an imperative language you'd write code to read your data, then do stuff, then write your data. In Haskell it's better to write the code that does stuff first, then the code to read and write the data at the end, once you know the functionality is right.

answered Sep 30, 2012 at 22:13

AndrewC

32.4k ● 7 ● 80 ● 115

4

You can't actually do exactly the same thing in a Haskell source file, so I suspect what you did actually looks like this:

```
readFoo = do
    x <- readFile "foo.txt"
    let y = read x :: [Int]
    return y
```

And are surprised that the type of `readFoo` comes out as `IO [Int]`, even though it's returning `y` which is of type `[Int]`.

If this is the case, the source of your confusion is that `return` in Haskell isn't a return statement from imperative languages.

`return` in Haskell is a function. A perfectly ordinary function. Like any other function it takes a value of some type as input and gives you a value of some other type as output. Specialised to the case of `IO` (`return` can be used with any monad, but we'll keep it simple here), it has this type:

```
a -> IO a
```

So it takes a value of any type and gives you a value in the same type wrapped up in the `IO` monad. So if `y` has type `[Int]`, then `return y` has type `IO [Int]`, and that's what you get as the result of `readFoo`.

There's no way to get the `[Int]` "out" of the `IO [Int]`. This is *deliberate*. The whole point of `IO` is that any value which is dependent on anything "outside" the program can only appear in an `IO x` type. So a function which internally reads "foo.txt" and returns a list of the integers in it must be impossible to write in Haskell, or the whole house of cards falls down. If we saw a function with a type like `readFoo :: [Int]`, as you were trying to write, then we know that `readFoo` can only be one particular list of integers; it can't be a list that depends on the contents of a file on disk.

Share Improve this answer Follow

answered Oct 1, 2012 at 2:50

Ben

71.3k • 21 • 132 • 183

2

GHCi does a little bit of magic to make it easier to quickly test stuff at the command prompt. If this were part of a program, then `IO [Int]` would indeed be the correct type. GHCi is letting you treat it as just `Int` to save you a bit of typing.

So that's the "why".

Now, if you have a specific question like "how do I do X given that the type signature is this?"...

Share Improve this answer Follow

answered Sep 30, 2012 at 19:50

MathematicalOrchid
62.8k ● 21 ● 129 ● 227