Understanding Basic Haskell Error Messages

by Jan Stolarek ⟨jan.stolarek@p.lodz.pl⟩

Haskell is a language that differs greatly from the mainstream languages of today. An emphasis on pure functions, a strong typing system, and a lack of loops and other conventional features make it harder to learn for programmers familiar only with imperative programming. One particular problem I faced during my initial contact with Haskell was unclear error messages. Later, seeing some discussions on #haskell, I noticed that I wasn't the only one. Correcting a program without understanding error messages is not an easy task. In this tutorial, I aim to remedy this problem by explaining how to understand Haskell's error messages. I will present code snippets that generate errors, followed by explanations and solutions. I used GHC 7.4.1 and Haskell Platform 2012.2.0.0 [1] for demonstration. I assume reader's working knowledge of GHCi. I also assume that reader knows how data types are constructed, what type classes are and how they are used. Knowledge of monads and language extensions is not required.

Compilation errors

Simple mistakes

I'll begin by discussing some simple mistakes that you are likely to make because you're not yet used to the Haskell syntax. Some of these errors will be similar to what you know from other languages—other will be Haskell specific.

Let's motivate our exploration of Haskell errors with a short case study. Standard `Prelude` provides some basic list operating functions like `head`, `tail`, `init` and `last`. These are **partial functions**. They work correctly only for a subset of all possible inputs. These four functions will explode in your face when you apply them to an empty list:

```
ghci> head []
*** Exception: Prelude.head: empty list
```

You end up with an exception that immediately halts your program. However, it is possible to create a safe version of `head` function which will work for all possible inputs without throwing an exception. Such functions are called **total functions**.

To reach our goal we will use the `Maybe` data type. The function will return `Nothing` when given an empty list; otherwise, it will return the result of applying `head` to `xs` wrapped in the `Just` constructor. Here's our completely bugged first attempt:

```
safeHead [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Maybe head xs
```

The first line is intended to be a type annotation, while the remaining two lines are the actual code. Loading this sample into `ghci` produces a **parse error**:

```
ghci> :l tmr.hs
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:1:14: parse error on input `->'
Failed, modules loaded: none.
```

Parse errors indicate that the program violates Haskell syntax. The error message starts in the third line (not counting the blank one). It begins with name of the file and exact location of the error expressed as line and column numbers separated with colons. In this case the error is in the first line, which means our intended type annotation. The compiler complains about `->` which was not expected to appear at this location. The problem is that the name of a function should be separated from type annotation with `::`. If there's no `::`, the compiler assumes we are defining a function body, treats `[a]` as a pattern binding and expects that it is followed either by `=` or `|` (a guard). Of course there are lots of other ways to cause parse error, but they all are dealt with in the same way: use your favourite Haskell book or tutorial to check what syntax Haskell expects for a particular expression. Let's fix this particular mistake by adding the missing `::` symbol:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Maybe head xs
```

and see what happens:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:3:15: Not in scope: data constructor ‘Maybe’
Failed, modules loaded: none.
```

Now that the type annotation is correct, it turns out that there's an error in third line. **Not in scope** means that some variable, function or, as implied in this case, a data constructor, is not known to the compiler. You certainly know this kind of error from different programming languages. Let's look at the definition of `Maybe` data type to understand what is wrong:

```
ghci> :i Maybe
data Maybe a = Nothing | Just a -- Defined in ‘Data.Maybe’
```

In this definition, `Maybe` is a **type constructor**, while `Nothing` and `Just` are **value constructors**, also referred to as **data constructors**. This means that when you want to create a value of a given type you must use either `Nothing` or `Just`. In our code we have mistakenly used `Maybe`. There is no data constructor called `Maybe`: there is a type constructor called `Maybe`. Let's replace that `Maybe` with `Just`, which was our original intention:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just head xs
```

The previous error is gone, only to produce a new one, shown in Listing 1. This time, the first two lines of the error message are quite explanatory, if you know

---

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:3:15:
    The function ‘Just’ is applied to two arguments,
    but its type ‘a0 -> Maybe a0’ has only one
    In the expression: Just head xs
    In an equation for ‘safeHead’: safeHead xs = Just head xs
Failed, modules loaded: none.
```

Listing 1: Applying data constructor to too many arguments.

that every data constructor is in fact a function. The definition of `Maybe` data type shows that the `Just` value constructor takes one parameter, while in our code we have mistakenly passed two parameters: `head` and `xs`. From a formal point of view this is a type system error. These will be discussed in more detail in the next section, but we treat this one here because it is very easy to make if you forget that function application is left-associative and that functions are curried by default. This means that `Just head xs` is the same as `((Just head) xs)`. We can use either parentheses or function composition and the `$` operator to override the default associativity. I've elaborated on the second approach on my blog [2] so I will not go into explaining it here; we'll just use the parentheses:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just (head xs)
```

Surprise, surprise! The code finally compiles:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )
Ok, modules loaded: Main.
```

Our function `safeHead` now works as intended:

```
ghci> safeHead []
Nothing
ghci> safeHead [1,2,3]
Just 1
```

You could implement safe versions of other unsafe functions in the same way, but there's no need to: there already is a library called `safe` [3], which provides different safe versions of originally unsafe functions.

Let's recall the not-in-scope error that we saw earlier. It was caused by using a function which didn't exist. Another common situation in which this error arises is when you use an existing function, but fail to import it. Let's look at a simple piece of code in which a different name is given to an already existing function:

```
module Main where
  sortWrapper xs = sort xs
```

Loading this code also produces the `not in scope` error:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:22:
    Not in scope: ‘sort’
    Perhaps you meant ‘sqrt’ (imported from Prelude)
Failed, modules loaded: none.
```

GHCi doesn't know `sort` function, but it knows `sqrt` from standard `Prelude` and suggests that we might have made a typo. The `sort` function we want to use is not in the standard `Prelude` so it must be explicitly imported. A problem arises if you know that a certain function exists, but you don't know in which package it is located. For such situations use `hoogle` [4]. If you have `hoogle` installed localy you can look up package name like this:

```
[jan.stolarek@GLaDOS : ~] hoogle --count=1 sort
Data.List sort :: Ord a => [a] -> [a]
```

Module name is located before the function name. In this case it is `Data.List`. Now we can import module like this:

```
module Main where
  import Data.List (sort)
  sortWrapper xs = sort xs
```

or even more explicitly:

```
module Main where
  import qualified Data.List as Lists (sort)
  sortWrapper xs = Lists.sort xs
```

So if you get a not-in-scope error, but you know that the function really exists, the missing import is most likely the culprit.

Another common mistake made by beginners is attempting to invoke functions directly from a module. A typical example might look like this

```
module Main where
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )

print (fact 5)
```

This code gives a correct definition of factorial function and then tries to print the result of invoking this function. This, however, is incorrect and results in a following error:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:6:1: Parse error: naked expression at top level
Failed, modules loaded: none
```

In Haskell everything in the module must be enclosed within function definitions. It is forbidden to call functions directly in a way shown in the above example. There are two possible solutions: first is removing the call to `print (fact 5)`, loading module into GHCi and invoking `fact 5` from interactive prompt. Alternatively, if you want your program to run as a separate executable, enclose the call to `print (fact 5)` within the `main` function, which is an entry point to every Haskell program:

```
module Main where
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )

main = print (fact 5)
```

This code can be compiled and executed as a standalone executable:

```
[jan.stolarek@GLaDOS : ~] ghc --make tmr.hs
[1 of 1] Compiling Main             ( tmr.hs, tmr.o )
Linking tmr ...
[jan.stolarek@GLaDOS : ~] ./tmr
120
```

A naked expression error can also be easily caused by accidentally typing `Import` instead of `import`:

```
module Main where
    Import Data.List
    sortWrapper xs = sort xs
```

This results in an error, because compiler treats `Import` as an application of data constructor to the parameter `Data.List`. Remember: capitalization matters in Haskell!

Type system errors

The complexity of Haskell's strong static type system can cause problems for beginners. Seemingly identical errors can result in very different error messages, and dealing with this can be a challenge. In this section I will discuss some common type system errors.

```
ghci> True && 1

 <interactive>:23:9:
    No instance for (Num Bool)
      arising from the literal ‘1’
    Possible fix: add an instance declaration for (Num Bool)
    In the second argument of ‘(&&)’, namely ‘1’
    In the expression: True && 1
    In an equation for ‘it’: it = True && 1
```

Listing 2: Non-exhaustive pattern in a guard.

I'll begin exploring errors related to type system with a very simple example shown in Listing 2. It demonstrates what can happen when you pass parameters of the wrong type to a function. The statement `No instance for (Num Bool) arising from the literal ‘1’` is the key to understanding the error message. The last three lines of the error message provide information on exact expression that caused the error. In our case that's the second argument of `(&&)` operator[1]. Even without knowing what exactly happened, it seems clear that there's a problem with `1` literal.

To understand what is going on, you need to know that numbers in Haskell are polymorphic. This means that when you write an integer literal in Haskell—just as we've written `1` in our example—it can be interpreted as different type depending on the context it is used in. Here's an example:

```
ghci> 1 :: Int
1
ghci> 1 :: Double
1.0
```

In the first case, the literal `1` is interpreted as an `Int`; in the second one, it is interpreted as a `Double`. Both uses are correct and don't cause a type error. This

[1]The `it` value, mentioned in the last line, is equal to the value of last expression evaluated in GHCi.

```
ghci> :i Bool
data Bool = False | True -- Defined in GHC.Bool
instance Bounded Bool -- Defined in GHC.Enum
instance Enum Bool -- Defined in GHC.Enum
instance Eq Bool -- Defined in GHC.Classes
instance Ord Bool -- Defined in GHC.Classes
instance Read Bool -- Defined in GHC.Read
instance Show Bool -- Defined in GHC.Show
```

Listing 3: Information about `Bool` data type.

works as if the `fromInteger` function defined in the `Num` class was called implicitly on the numeric literal. This means that `True && 1` and `True && fromInteger 1` are equivalent expressions. Let's check the type signature of `fromInteger`:

```
ghci> :t fromInteger
fromInteger :: Num a => Integer -> a
```

This means that `fromInteger` takes an `Integer` and returns a value of **any** type `a`, with a restriction that the type `a` belongs to the `Num` type class. What type exactly should be returned? That depends on the context in which `fromInteger` was applied. For example, if the return value is required to be `Int` then the implementation of `fromInteger` defined in the `Num Int` instance declaration is called. That specific implementation returns a value of type `Int`. This mechanism allows integer literal to become an instance of a type belonging to `Num` type class.

With this knowledge, we can go back to our example in which integer literal `1` was used as a parameter to `(&&)` function. This function takes two `Bool` parameters and returns a single `Bool`:

```
ghci> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

which means that in order for literal `1` to be a valid parameter to `(&&)`, the type returned by `fromInteger` should be `Bool`. There is one problem though. The `Bool` type is not an instance of `Num` type class, as shown in Listing 3. However, it should be, since the `fromInteger` function imposes a constraint that its return value belongs to `Num` type class. This is exactly what the error message said and that is the reason why a type error occurs. The next line of the error message suggests a solution: `Possible fix: add an instance declaration for (Num Bool)`. Indeed, if we made `Bool` type an instance of `Num` type class the problem would be gone. In some cases, this may be the solution. Deriving `Bool` to be an

instance of `Num` is certainly a good exercise that you can try out. In many other cases however this error means you wanted something different than you actually wrote. Let's assume here that we wanted `1` to denote logical truth. Fixing that is easy:

```
ghci> True && True
True
```

The constraints imposed by the type classes propagate in the type system. Here's a simple example that demonstrates this: let's say we want to write a function that tells if the two parameters passed to it are equal:

```
isEq :: a -> a -> Bool
isEq x y = x == y
```

Our `isEq` function expects two parameters that are of the same type and returns a `Bool`. Our code looks perfectly reasonable, but loading that code into GHCi results with an error shown in Listing 4.

```
ghci> :l tmr.hs
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:14:
    No instance for (Eq a)
      arising from a use of ‘==’
    In the expression: x == y
    In an equation for ‘isEq’: isEq x y = x == y
Failed, modules loaded: none.
```

Listing 4: Error caused by a lack of type class constraint.

The first two lines of this message are the most important: they say that type `a` should be an instance of `Eq` type class and that this requirement is imposed by the use of `==` function. Let's inspect the type of `(==)`:

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

Indeed the `(==)` function expects that its arguments are instances of `Eq` type class. That constraint was propagated to `isEq` function, which uses `(==)`. We must therefore add a type class constraint to parameters of `isEq`:

```
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:17:
  Could not deduce (b ~ a)
  from the context (Eq a, Eq b)
    bound by the type signature for
             isEq :: (Eq a, Eq b) => a -> b -> Bool
    at tmr.hs:2:1-17
    ‘b’ is a rigid type variable bound by
      the type signature for isEq :: (Eq a, Eq b) => a -> b -> Bool
      at tmr.hs:2:1
    ‘a’ is a rigid type variable bound by
      the type signature for isEq :: (Eq a, Eq b) => a -> b -> Bool
      at tmr.hs:2:1
  In the second argument of ‘(==)’, namely ‘y’
  In the expression: x == y
  In an equation for ‘isEq’: isEq x y = x == y
Failed, modules loaded: none.
```

Listing 5: Comparing two different instances of `Eq` type class.

```
isEq :: Eq a => a -> a -> Bool
isEq x y = x == y
```

This fixes the problem. Let's now see what happens if we try to compare parameters of two different types, both belonging to `Eq` type class:

```
isEq :: (Eq a, Eq b) => a -> b -> Bool
isEq x y = x == y
```

The error message is shown in Listing 5. The `(b ~ a)` notation indicates equality of the types `a` and `b`. The compiler uses this notation to say `Could not deduce (b ~ a)`, which means that `a` and `b` should be identical, but the type definition we provided does not guarantee it. The requirement of types `a` and `b` being of the same type is a direct consequence of a type signature of `(==)` which, as we recall, requires its parameters to be of the same type. The term `rigid type variable` indicates that our types `a` and `b` have been directly specified by the type annotation [5] and the compiler is not free to unify them[2]. We have already seen a correct version of this code, but we can also make it work in a different way:

[2]Unification of two types means that they are assumed to be the same type.

```
{-# LANGUAGE TypeFamilies #-}

isEq :: (Eq a, Eq b, a ~ b) => a -> b -> Bool
isEq x y = x == y
```

Enabling `TypeFamilies` language extension and adding `a ~ b` type constraint informs the compiler that `a` and `b` can be unified to be the same type. The above code is for demonstration purposes: it makes no real sense to write the type declaration in this way, since `a` and `b` will be unified into one type. It can be written in a more straightforward way. Loading this into GHCi and checking type of `isEq` will show that it's actually `isEq :: Eq b => b -> b -> Bool`. The two distinct types introduced deliberately have disappeared because the compiler was allowed to unify them.

When you begin working with type classes, you may find it difficult what type class constraints to impose on your functions. Here's where the Haskell's type inference is useful: write down your function without the type declaration, load it into GHCi and use Haskell's type inference to determine function's type for you. This can be done using the `:t` command on a function. Suppose the `isEq` function was written without type declaration. Here's what Haskell infers about the `isEq`'s type:

```
ghci> :t isEq
isEq :: Eq a => a -> a -> Bool
```

This is a correct type signature and you can simply copy it to your code.

We've seen what will happen when type signature does not contain appropriate class restrictions. Let's now see what happens when a function's type signature is inconsistent with function's implementation. Assume we want to write a function that returns a first letter from a `String`. We could want to have a type signature like this:

```
getFirstLetter :: String -> String
```

This means that `getFirstLetter` function takes a value of type `String` and returns a value of type `String`. For example, if we pass `"Some string"` as a parameter then `getFirstLetter` will return value `"S"`. Since `String` in Haskell is a synonym for `[Char]` (list of `Char`s) we could use `head` function to take the first element of a `String`. Our `getFirstLetter` function would then look like this:

```
getFirstLetter :: String -> String
getFirstLetter = head
```

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:18:
    Couldn't match expected type 'String' with actual type 'Char'
    Expected type: String -> String
      Actual type: [Char] -> Char
    In the expression: head
    In an equation for 'getFirstLetter': getFirstLetter = head
Failed, modules loaded: none
```

Listing 6: Expected and actual type mismatch error.

The `head` function has type `[a] -> a`, which means it takes a list of some type `a` and returns a single element of type `a`, not a list. However, the type signature for `getFirstLetter` requires that the return argument be a list of `Char`s. Therefore, the provided signature is inconsistent with the actual return type of the function. Haskell will notice that when you try to load the code into GHCi and report an error as shown in Listing 6. It says that type annotation in the source code defines the return type to be `[Char]`, but the actual type inferred by the compiler is `Char`. There are two possible fixes for this problem. First, if we are really fine with getting a single `Char` instead of a `String`, we can change the type signature to `String -> Char` (or `[Char] -> Char`, since it's the same due to type synonyms). On the other hand, if we really expect to get a `String` from our function, we need to change the implementation. In this case the result of `head` should be wrapped in a list, which can be done like this:

```
getFirstLetter :: String -> String
getFirstLetter xs = [head xs]
```

or using a point-free style:

```
getFirstLetter :: String -> String
getFirstLetter = (: []) . head
```

The type mismatch error is probably the one you'll be seeing most often. There is really no way of anticipating all possible situations where it might occur. The above case was easy, because the implementation of a function was correct and the only problem was fixing the type annotation. Most often however you'll end up in situations where you've written some code and the type-checker complains that this code is type-incorrect. Here's an example of how this may look like:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:4:26:
    Couldn't match expected type 'Int' with actual type 'Char'
    In the first argument of '(*)', namely 'acc'
    In the first argument of '(+)', namely 'acc * 10'
    In the expression: acc * 10 + digitToInt x
Failed, modules loaded: none.
```

Listing 7: Expected and actual type mismatch error.

```
import Data.Char

asInt :: String -> Int
asInt = foldl (\x acc -> acc * 10 + digitToInt x) 0
```

This code takes a `String` representation of a number and turns it into an `Int`. Well... almost: if you try to compile it you'll get an error about mismatching types, as shown in Listing 7. The message says that `acc` (accumulator) is of the wrong type—`Char` instead of `Int`. In such cases it might not be immediately obvious what exactly is wrong. The only way to deal with such errors is to inspect the incorrect expressions. In this example the code isn't really complicated so this will be easy. The `asInt` function uses only `foldl` and one anonymous lambda function. Let's begin by checking the type signature of `foldl`:

```
ghci> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

This type signature says that `foldl` takes three parameters. These parameters are a function of type `a -> b -> a`, an accumulator of type `a` and a list containing elements of type `b`. By looking at the type variables in this signature—that is, `a` and `b`—you can see that the first parameter to folding function, the accumulator and the return value of `foldl` are of the same type `a`. We passed `0` as the accumulator and expect the return value of `asInt` function to be `Int`. Therefore type `a` is inferred to be `Int`. The compiler complains, however, that variable `acc` used as parameter to `(*)` is of type `Char`, not `Int`. The parameters to the lambda function are `x` and `acc`. According to type signature of `foldl`, `x` should be of type `Int`, because it is of the same type as the accumulator. On the other hand `acc` is of the same type as elements of the list passed as third parameter to `foldl`. In this

```
ghci> :m + Data.List
ghci> 'A' `isPrefixOf` "An error"

 <interactive>:8:1:
    Couldn't match expected type `[a0]' with actual type `Char'
    In the first argument of `isPrefixOf', namely 'A'
    In the expression: 'A' `isPrefixOf` "An error"
    In an equation for `it': it = 'A' `isPrefixOf` "An error"
```

Listing 8: Passing argument of wrong type to a function.

example, the list is of type `[Char]` so `acc`, being a single element, is of type `Char`. Well, this should be the other way around: `acc` should be of type `Int`; `x` should be of type `Char`. We conclude this error was caused by writing the parameters of lambda in incorrect order. The correct code therefore is:

```
import Data.Char

asInt :: String -> Int
asInt = foldl (\acc x -> acc * 10 + digitToInt x) 0
```

In some cases you will get an error message that doesn't specify concrete type. Consider example shown in Listing 8. The `isPrefixOf` function expects its both parameters to be lists of the same type:

```
ghci> :t isPrefixOf
isPrefixOf :: Eq a => [a] -> [a] -> Bool
```

As soon as compiler realizes that the first parameter is not a list, it complains. It doesn't even reach the second parameter to analyze it and infer that the first list should contain `Char`s. That is the reason why it uses a list `[a0]`, where `a0` represents any type. Listing 9 shows the result of swapping the two parameters. This time, the compiler is able to infer the exact type of the elements required in the second list. By the time compiler reaches the second parameter, it has already analyzed the first one and it knows that that parameter was a list of `Char`s.

Another common error related to types is type ambiguity. I'll demonstrate it using the `read` function, which is used to convert `String` representation of some value to that value. The `read` function is defined in the `Read` type class and has type signature `read :: Read a => String -> a`. All data types within standard Prelude, except function types and `IO` types, are instances of `Read`. Every type

```
ghci> "An error" `isPrefixOf` 'A'

 <interactive>:9:25:
    Couldn't match expected type `[Char]' with actual type `Char'
    In the second argument of `isPrefixOf', namely 'A'
    In the expression: "An error" `isPrefixOf` 'A'
    In an equation for `it': it = "An error" `isPrefixOf` 'A'
```

Listing 9: Passing argument of a wrong type to a function.

```
ghci> read "5.0"

 <interactive>:11:1:
   Ambiguous type variable `a0' in the constraint:
     (Read a0) arising from a use of `read'
   Probable fix: add a type signature that fixes these
     type variable(s)
   In the expression: read "5.0"
   In an equation for `it': it = read "5.0"
```

Listing 10: Type ambiguity.

that is an instance of `Read` type class provides its own definition of `read` function. This means that the compiler must know what resulting type to expect in order to call a correct implementation of the `read` function[3]. This is polymorphism, which was already discussed when we talked about `fromIntegral` function. Listing 10 shows that the error that occurs when the polymorphic type variable cannot be inferred to a concrete type. In this case, the compiler doesn't know which version of `read` function should be called. It suggests that we can solve the problem by adding a type signature. Let's try that:

```
ghci> read "5.0" :: Double
5.0
```

It is important to provide correct type signature:

```
ghci> read "5.0" :: Int
*** Exception: Prelude.read: no parse
```

[3]Even if there was only one instance of `Read` class, the compiler wouldn't know that and you would get type ambiguity error.

```
ghci> id -1

 <interactive>:16:4:
    No instance for (Num (a0 -> a0))
      arising from a use of '-'
    Possible fix: add an instance declaration for (Num (a0 -> a0))
    In the expression: id - 1
    In an equation for 'it': it = id - 1
```

Listing 11: Incorrect usage of unary negation.

This code produces an exception because implementation of `read` for `Int` instances of `Read` doesn't expect any decimal signs.

The type system can come into play in very unexpected moments. Let's play a little bit with `id`, the identity function, which returns any parameter passed to it:

```
ghci> id 1
1
ghci> id 0
0
ghci> id (-1)
-1
```

Notice that the negative parameter was wrapped in parentheses. Had we neglected this, it would result in a `No instance for (Num (a0 -> a0))` error, as shown in Listing 11. We've already seen this error before when we talked about polymorphism of integer literals. This time the compiler expects the `a0 -> a0` type to be an instance of `Num` type class. The `a0 -> a0` denotes a function that takes an value of type `a0` and returns a value of the same type `a0`. What seems strange is the fact that the compiler expects literal `1` to be of type `a0 -> a0`. The problem here is that the `(-)` sign is treated as an infix binary operator denoting subtraction, not as unary negation operator as we expected [4]. The strange error message blaming `1` literal is caused by the fact that `(-)` operator expects its parameters to be of the same type:

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

[4] Section 3.4 of Haskell 2010 Language Report [6] gives more detail on that matter.

```
ghci> id - "a string"

 <interactive>:11:6:
   Couldn't match expected type ‘a0 -> a0’ with actual type ‘[Char]’
   In the second argument of ‘(-)’, namely ‘"a string"’
   In the expression: id - "a string"
   In an equation for ‘it’: it = id - "a string"
```

Listing 12: Error caused by incorrect parameters that should be of the same type.

The type of first parameter was inferred to be `a0 -> a0`. Therefore the second parameter is also expected to be of type `a0 -> a0`, but this type is not an instance of `Num` type class. Just as before, this error results from the implicit use of `fromIntegral` function. In this example, however, there's one thing that you might be wondering. From the type signature of `(-)`, we can see that its parameters should belong to `Num` type class. The question is this: how can we be certain that this error is raised by the constraint in the `fromIntegral` function and not by the constraint in the `(-)` function itself? There's an easy way to verify this. Let's replace the second argument of `(-)` with a value of type `String`. We use `String`, because string literals don't implicitly call any function that would impose additional type constraints. The error that results in this case, shown in Listing 12, says that compiler expects the second argument to be the same type as the first one, which is a restriction resulting from the type signature of `(-)`. There is no complaint about `Num` type class, which allows to infer that at this point `Num` type class constraint imposed by `(-)` hasn't been checked yet. Let's verify this conclusion by supplying `(-)` with two arguments of the same type that is not an instance of `Num`. The result is shown in Listing 13. This time the compiler successfully verified that both parameters of `(-)` are of the same type `a0 -> a0`, and it could go further to check if type class constraints are satisfied. However, the `a0 -> a0` type is not an instance of `Num` type class, hence the type class constraint is violated and `No instance for` error arises.

Some runtime errors

We finally managed to get past the compilation errors. It was a lot of work, probably more than in other programming languages. That's another characteristic feature of Haskell: the strong type system moves much of the program debugging up front, into the compilation phase. You probably already heard that once a Haskell program compiles it usually does what the programmer intended. That's

```
ghci> id - id

 <interactive>:20:4:
    No instance for (Num (a0 -> a0))
      arising from a use of '-'
    Possible fix: add an instance declaration for (Num (a0 -> a0))
    In the expression: id - id
    In an equation for 'it': it = id - id
```

Listing 13: Another `No instance for (Num (Int -> Int))` error.

mostly true, but this doesn't mean that there are no runtime errors in Haskell. This section will discuss some of them.

Runtime errors in Haskell most often take the form of runtime exceptions. At the very beginning of this paper, I showed you that some functions don't work for all possible inputs and can raise an exception:

```
ghci> head []
*** Exception: Prelude.head: empty list
```

In languages like Java, exceptions are your friend. They provide a stack trace that allows to investigate the cause of an error. It is not that easy in Haskell. Runtime exceptions don't give you any stack trace, as this is not easily implemented in a language with lazy evaluation. You are usually left only with a short error message and line number.

One of the most commonly made errors resulting in runtime exceptions is non-exhaustive patterns in constructs like function definitions or guards. Let's recall `safeHead` function that we've written in the first section:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHEad xs = Just (head xs)
```

This function contains a typo: the function name in the third line is misspelled as `safeHEad` (notice the capital `E`). This code compiles perfectly, but will result in an error if we try to call `safeHead` function for non-empty list:

```
ghci> safeHead []
Nothing
ghci> safeHead [1,2,3]
*** Exception: tmr.hs:2:1-21: Non-exhaustive patterns in function
safeHead
```

The argument `[1,2,3]` passed to the function `safeHead` couldn't be matched against any pattern in the function definition. That's what **non-exhaustive pattern** error means. This is due to the typo I made in the third line. For Haskell compiler everything is perfectly fine. It treats the code as definition of two different functions: `safeHead`, matching only empty lists, and `safeHEad`, matching both empty and non-empty lists. Note that applying `safeHEad` to empty list will result in a runtime exception.

We were unexpectedly hit by the non-exhaustive pattern problem during runtime because GHCi has most warnings disabled by default. You can enable most of the warnings by passing `-Wall` command line switch when running `ghci` command[5]. Now GHCi will warn us during the compilation about non-exhaustive patterns in `safeHead` function and lack of type signature for accidentally defined `safeHEad` function. See Listing 14.

```
ghci> :l tmr.hs
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:1:
  Warning: Pattern match(es) are non-exhaustive
       In an equation for ‘safeHead’: Patterns not matched: _ : _

tmr.hs:3:1:
  Warning: Top-level binding with no type signature:
       safeHEad :: forall a. [a] -> Maybe a
Ok, modules loaded: Main.
```

Listing 14: Compilation warnings.

The non-exhaustive pattern error can also occur in an incorrect guard. To illustrate this, let's create our own signum function:

```
mySignum :: Int -> Int
mySignum x
    | x > 0  = 1
    | x == 0 = 0
```

You see the error, don't you? Listing 15 show what happens when we call `mySignum` function for negative arguments. This error is easily corrected by adding the third guard:

[5] See [7] for more details.

```
ghci> mySignum 1
1
ghci> mySignum 0
0
ghci> mySignum (-1)
*** Exception: tmr.hs:(14,1)-(16,16): Non-exhaustive patterns in
function mySignum
```

Listing 15: Non-exhaustive pattern in a guard.

```
mySignum :: Int -> Int
mySignum x
    | x > 0     = 1
    | x == 0    = 0
    | otherwise = -1
```

The `otherwise` function is defined to always return `True`, so the third guard will always evaluate if the previous guards didn't. The compiler can also give a warning about non-exhaustive guards in a same way it gives warning about non-exhaustive patterns in function definition. Remember that order of guards matters, so `otherwise` must always be the last guard.

Summary

This completes our overview of basic Haskell error messages. By now, you should know how to read error messages and how to correct the problems in your code that caused them. The above list is by no means an exhaustive one: there are a lot of different errors you will encounter when you start using some more advanced features of Haskell. Perhaps one day you'll even write a program that will cause `My brain just exploded` error. Until then, happy error solving!

Acknowledgements

I thank the great Haskell community at #haskell IRC channel. They helped me to understand error messages when I was beginning my adventure with Haskell. Many thanks go to Edward Z. Yang for his feedback. I also thank Marek Zdankiewicz for reviewing draft version of this paper.

References

[1] http://hackage.haskell.org/platform/.

[2] http://ics.p.lodz.pl/~stolarek/blog/2012/03/function-composition-and-dollar-operator-in-ha

[3] http://hackage.haskell.org/package/safe.

[4] http://www.haskell.org/hoogle/.

[5] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In **ICFP**, pages 50–61 (2006).

[6] http://www.haskell.org/onlinereport/haskell2010/.

[7] http://www.haskell.org/ghc/docs/latest/html/users_guide/options-sanity.html.