Curried Thoughts 🍯 Point-Free Ramblings 🍯
Unapplied Arguments

Ziyang Liu's blog, mostly about Haskell

Un-obscuring a few GHC type error messages

09/01/2020 12 mins

Generally speaking, GHC's error messages are fairly helpful and intelligible (so long as you don't go wild with type-level programming). But there are definitely a few common but relatively less clear ones. Some of the GHC type error messages that can potentially lead to bewilderment are discussed in this post. The GHC version I used is 8.10.2.

🤔 'a' is a rigid type variable bound by...

If you've written Haskell for any amount of time, you've probably seen this error message, and it is hardly a confusing one. However, the word "rigid" may seem counter-intuitive, and one might wonder what it actually means. Consider the following function definition:

```
f :: a -> a
f x = not x
```

This definition does not compile and the error message is

```
• Couldn't match expected type ‘Bool’ with actual type ‘a’
  ‘a’ is a rigid type variable bound by
    the type signature for:
      f :: forall a. a -> a
    at Example.hs:11:1-11
• In the first argument of ‘not’, namely ‘x’
```

```
x :: a (bound at Example.hs:12:3)
f :: a -> a (bound at Example.hs:12:1)
```

It should be obvious why `f` doesn't type check, but a potential point of confusion for some, including myself when I first encountered an error message like this is: why does it say `a` is "rigid"? `a` can represent any type, so shouldn't it be "flexible"?

`a` is flexible indeed, but only from the perspective of the caller of `f`. From the perspective of the definition of `f`, it is not at all flexible. The definition of `f` must treat `a` as some unknown but fixed type, and must do the exact same thing for all possible `a`.

Haskell has several sorts of type variables, including:

- Type variables written by programmers, such as those bound by `forall`s in type signatures, and those appearing in type applications and type annotations.
- Type variables used internally by the compiler. These include
  - *Unification type variables* (or flexible type variables, meta type variables, unification variables). They are fresh variables allocated to stand for unknown types that need to be determined. One of the tasks of the type inference engine is to determine their actual types by finding a substitution for each unification variable. Unification variables are flexible in the sense that they can unify with any type or type variable that does not contain `forall`s[1] (except when they are untouchable, which will be explained later).
  - *Rigid type variables* (or skolem type variables, skolem constants, skolems). They are fresh variables allocated to stand for unknown but fixed types. Their actual types do not need to be, and cannot be determined. They are rigid in the sense that they cannot unify with anything other than themselves or unification variables. In particular, a rigid type variable cannot unify with a concrete type or type constructor, or another rigid type variable.

Unification type variables and rigid type variables are freshly allocated by the type checker. In an error message like

```
Couldn't match expected type <expected> with actual type <actual>
```

`<expected>` and `<actual>` often contain type variables not written by the programmer, because they are compiler-allocated.

When type checking the above `f` function, where `a` is a programmer-written, universally quantified type variable, GHC first allocates a rigid type variable for it (a process called skolemization), because from `f`'s point of view, `a` is unknown and fixed (or existential). The body of `f` requests that this rigid type variable be unified with `Bool`, which the type checker outright refuses, hence the above error message.

🤔 ...type variable 'a' would escape its scope. This (rigid, skolem) type variable...

A rigid/skolem type variable cannot escape, via a unification variable, the scope where it is introduced. In other words, a rigid type variable can unify with a unification variable, but not when that unification variable has a bigger scope. This can happen in two cases.

1. Type checking a polymorphic function argument:

```
fun :: (forall a. [a] -> b) -> Bool
fun _ = True

arg :: c -> c
arg c = c

x :: Bool
x = fun arg
```

This leads to the following error:

```
• Couldn't match type ‘b0’ with ‘[a]’
    because type variable ‘a’ would escape its scope
  This (rigid, skolem) type variable is bound by
    a type expected by the context:
      forall a. [a] -> b0
    at Example.hs:18:9-11
  Expected type: [a] -> b0
    Actual type: b0 -> b0
• In the first argument of ‘fun’, namely ‘arg’
  In the expression: fun arg
  In an equation for ‘x’: x = fun arg
```

In order for `x = fun arg` to be well typed, `arg`'s type must subsume (i.e., be more general than or

equal to) the type required by `fun`. This is *not* the case here. `fun` requires that its argument be able to convert a list of `a`s for any arbitrary `a`, into a *fixed* `b`. `b` can be flexibly chosen, for instance `fun (length :: [a] -> Int)` would be legal, but it is chosen before `a`, and must be fixed (i.e., does not depend on `a`) once chosen. `arg` does not satisfy this requirement, because it maps an arbitrary `a` to itself. Therefore, this program is rejected.

When type checking `x = fun arg`, the type checker allocates a rigid type variable for `a`, and allocates unification variables for `b` and `c`. It then determines that the type of both `b` and `c` is `[a]`. But sadly, `b` is not allowed to unify with `[a]`, because `b` is bound by an (implicit) top-level `forall`, while `a` is bound by an inner `forall`. If they were allowed to unify, the rigid/skolem type variable `a` would escape its scope via unification variable `b`.

By the way, this is how the ST monad works.

2. Opening existential types:

```
data A = forall a. A a
f (A x) = Just x
```

This leads to a similar skolem escape error:

```
• Couldn't match expected type ‘p’ with actual type ‘Maybe a’
    because type variable ‘a’ would escape its scope
  This (rigid, skolem) type variable is bound by
    a pattern with constructor: A :: forall a. a -> A,
    in an equation for ‘f’
    at Example.hs:12:4-6
• In the expression: Just x
  In an equation for ‘f’: f (A x) = Just x
• Relevant bindings include
    x :: a (bound at Example.hs:12:6)
    f :: A -> p (bound at Example.hs:12:1)
```

The type of the value encapsulated in an existential type (`x` in this example) is considered private. When an existential type is opened via pattern matching[2], a rigid type variable is allocated to denote the (unknown but fixed) private type, i.e., `x`'s type, and a unification variable (`p` in this example) is allocated to denote the type of the pattern matching branch. This rigid type variable must not

escape the scope of the branch via the unification variable.

In other words, Let `a` denote `x`'s type in `case A of (A x) -> ...`. The types of local bindings within `...` (declared in `let` or `where` clauses) may refer to `a` (but you can't write explicit type signatures for such bindings), but the type of the entire `...` may not. Therefore the above `f` is ill-typed because the right hand side has type `Maybe a` which of course mentions `a`.

If Haskell allowed the `exists` keyword for existential quantification, `f` would be well-typed because it would be possible to assign the following type to `f`:

```
f :: A -> exists a. Maybe a
```

But Haskell doesn't have the `exists` keyword, because `exists` can be expressed in terms of `forall` and `->`:

```
exists a. Maybe a  ≅  forall r. (forall a. Maybe a -> r) -> r
```

Intuitively, saying "I have a `Maybe a` for some `a`, but I won't tell you what `a` is" is equivalent to saying "If you give me a function that maps an arbitrary `Maybe a` to `r`, then I'll give you an `r` back, by applying your function to my secret `Maybe a`".

But note that the LHS and RHS above are only isomorphic; they are not identical. So `f` in the above example doesn't have a valid haskell type, but the following function `g`, which is isomorphic to `f`, does have a valid type:

```
g :: A -> (forall a. Maybe a -> r) -> r
g (A x) h = h (Just x)
```

Incidentally, in intuitionistic second-order propositional logic[3], not only the existential quantifier, but also negation, conjunction, disjunction and absurdity, can all be expressed in terms of the universal quantifier and `->`. For example, absurdity (corresponding to Haskell's `Void` type) is `forall a. a`, and `a \/ b` (corresponding to Haskell's sum type) is `forall r. (a -> r) -> (b -> r) -> r`. But this is off topic so I digress.

🤔 'p' is untouchable

The "untouchable" error can happen when you pattern match against a GADT without supplying a

type signature. Consider this example:

```
data A a where
  A :: A Bool

f = \case A -> True
```

This seemingly trivial and unproblematic code does not type check:

```
• Couldn't match expected type ‘p’ with actual type ‘Bool’
    ‘p’ is untouchable
      inside the constraints: a ~ Bool
      bound by a pattern with constructor: A :: A Bool,
                 in a case alternative
      at Example.hs:15:11
  ‘p’ is a rigid type variable bound by
    the inferred type of f :: A a -> p
    at Example.hs:15:1-19
  Possible fix: add a type signature for ‘f’
• In the expression: True
  In a case alternative: A -> True
  In the expression: \case A -> True
• Relevant bindings include
    f :: A a -> p (bound at Example.hs:15:1)
```

This is because `f` has more than one valid type:

```
f :: forall a. A a -> a
f :: forall a. A a -> Bool
```

Note that neither of the above two types is more general than the other. They are just two different and incomparable types. In other words, `f` lacks a *principal type* (a unique most general type)[4]. The fix is to add a type signature for `f`, which is clearly indicated in the error message.

The word “untouchable” refers to the fact that the return type of the pattern matching, although a unification variable, is considered unavailable for unification in this particular branch. Therefore it can’t actually be unified with `Bool`.

If data type `A` had other data constructors, and the pattern match in `f` had more branches, then

another branch may end up successfully unifying the result of the pattern match with Bool. For instance if `A` and `f` are defined as

```
data A a where
  A :: A Bool
  B :: A a

f = \case A -> True; B -> True
```

then it would be well typed and accepted by GHC. The reason is that `B` here is a regular, non-GADT data constructor (although it is written in GADT syntax), because it returns `A a` rather than `A <something that is not a>`. So when type checking `f`, in the `B` branch, nothing is untouchable, hence the result of the pattern match is successfully inferred to be `Bool`.

It is worth mentioning that this approach is conservative, and may fail to type check a definition with a unique, most general correct type. For instance

```
g = \case A -> 'a'
```

GHC can't infer `g`'s type, and fails with the same "untouchable" error. This can be confusing since `g` does have a unique, most general type, namely `g :: forall a. A a -> Char`.

The conservativity turns out to be unavoidable, since when GADTs are involved, the problem of determining whether a term has a principal type is undecidable. Besides, if there happens to be a type family like this:

```
type family F a where
  F Bool = Char
```

Then one can argue that `g` no longer has a principal type, because both of the following types are now valid:

```
g :: forall a. A a -> F a
g :: forall a. A a -> F Bool
```

🤔 Overloaded signature conflicts with monomorphism restriction

Consider this code in which a polymorphic constant `x` is defined:

```
x :: Num a => a
y :: ()
(x, y) = (42, ())
```

Perhaps surprisingly, this definition doesn't type check, unless monomorphism restriction is turned off by enabling `NoMonomorphismRestriction`. The error message says

```
Overloaded signature conflicts with monomorphism restriction
  x :: forall a. Num a => a
```

To understand why this happens requires some knowledge about how the monomorphism restriction works. The monomorphism restriction reduces polymorphism of a definition in certain cases. It comes in two flavors:

1. Some monomorphism restrictions can be resolved by adding a type signature to a binding.
2. Some monomorphism restrictions cannot be resolved by adding type signatures. The above example obviously belongs to this case, since both `x` and `y` already have type signatures.

The first case is applicable to *simple pattern bindings*, i.e., the pattern consists of only a single variable. For instance, all of these are simple pattern bindings:

```
x = 42
f = show
g = \a -> show a
```

Without type signatures, they either don't type check, or are assigned types that are monomorphic, e.g., the inferred type for `x` is `Integer` rather than `Num a => a`. The fix is to add type signatures to these bindings.

On the other hand, `h a = show a` is fine. It type checks and is assigned a polymorphic type. This is because `h` is a *function binding* as opposed to a pattern binding, and the monomorphism restriction only applies to pattern bindings.

The second case is applicable to non-simple pattern bindings. The above example is a non-simple pattern binding since the left-hand side is a pair. In this case, adding a polymorphic type signature does not work, and if you want `x` to have type `Num a => a`, you must define it separately.

More information about monomorphism restrictions can be found in Section 4.5.5 of the Haskell 2010

report.

Final Words

Besides type error messages, there are some other aspects of GHC that can potentially be confusing to inexperienced haskell programmers, for instance when you try to do some seemingly simple and unharmful things but GHC wouldn't let you do them. Here are some of the things off the top of my mind that I'd love to discuss in another blog post, if time permits:

- why you sometimes can't rewrite `f (g x)` into `f . g`
- why you can't partially apply type synonyms
- why sometimes you can't `deriving Show` but you can `deriving instance Show` using standalone deriving

Until then, have fun wrestling with the type checker, and stay safe and well!

1. Once impredicativity is supported, they will, in effect, be able to unify with types that do contain `forall` s. ↩

2. Existential types can only be opened by pattern matching. Use of record selectors is not allowed. ↩

3. Under the curry-howard isomorphism, intuitionistic second-order propositional logic corresponds to polymorphic lambda calculus, a.k.a. System F, which greatly overlaps with Haskell's type system. ↩

4. Another valid type of `f` is `f :: A Bool -> Bool` , but it is less general than both previous types, so this is not the cause of the problem. ↩

‹ Understanding Space Leaks From StateT

How Accursed and Unutterable is accursedUnutterablePerformIO? ›

ALSO ON **COFREE**

How Trampoline Works in Scala

8 years ago · 1 comment

Trampoline is a way to make non-tail recursive functions stack-safe. Its Scala …

Fixed Points and Non-Fixed Points of …

6 years ago · 13 comments

I've been playing with fixed points of Haskell functors lately and thought it would …

An Absolutely Elementary Introduction

8 years ago · 7 comments

When writing any non-trivial amount of code, we usually factor out the common …

D H

6

T d p

What do you think?

9 Responses

 Upvote
 Funny
 Love
 Surprised
 Angry
 Sad

4 Comments Login

G

Join the discussion...

LOG IN WITH OR SIGN UP WITH DISQUS

Name

♡ Share Best Newest Oldest

Florian Klein
5 years ago
Amazing, thanks! I would have loved seeing the correct program after each failing example, to see **how** to fix them when I encounter some of these errors.

0 0 Reply

unsafePerformIO Mod → Florian Klein
5 years ago
Thanks for the suggestion!

0 0 Reply

Providence Salumu
5 years ago
Brilliant!

0 0 Reply

unsafePerformIO Mod → Providence Salumu
5 years ago
Thanks!

0 0 Reply

Subscribe Privacy Do Not Sell My Data