

The *rough* guide to GHC Type Checker Messages

Joanna Sharrad
jks31@kent.ac.uk
University of Kent

1 Introduction

This unpolished paper presents a set of programs that trigger error messages from the Glasgow Haskell Compiler version 8.4.3. All of the following messages are from TcErrors (ghc/compiler/typecheck/TcErrors.hs), however we shall also briefly mention other modules when the error messages we discuss employ them.

According to TcErrors the messages are separated into four categories:

- Irreducible predicate errors.
- Equality Errors.
- Type-Class Errors.
- Error from the canonicaliser.

Each error message in these categories is split into three parts to ease the presentation to the programmer. TcErrors state the parts are:

- Main Message.
- Context Box.
- Relevant Bindings Block.

To gently approach each message we will start with discussing the 'Main Message' segment of the error, however as we move further into TcErrors we will start to present the entire error message for dissection. The paper layout is formed by giving an example program that will trigger the error, showing the relevant error message we receive, and describing where the error message is called from within TcErrors. The programs for our examples have come from a mixture of online resources representing real programs that cause these messages, however, where an online resource was not found, we took an example program from the GHC test suites.

2 TcErrors.hs

2.1 Irreducible predicate errors

Irreducible predicate error messages start from line *1056* in TcErrors.

2.1.1 IPred Error 1

mkHoleError (*line 1074*) is the name of three functions, each dealing with a different type of error. The first applies to 'not in scope' errors for Data Constructors and Variables. Below we give an example of each.

The following program triggers the first mkHoleError function concerned with Data Constructors:

Listing 1: Example Program [ONi15]

```
data Days = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
    deriving(Eq, Ord, Show, Read, Bounded, Enum)

main =
    putStrLn $ show $ Days !! 1
```

Listing 2: Error

```
Data constructor not in scope: Days :: [a0]
```

2.1.2 IPred Error 2

This second example runs the same function as above but delivers a different message when it is a variable not in scope:

Listing 3: Example Program [Kad16]

```
main = interact $ show . maxsubseq . map read . words

maxsubseq :: (Ord a, Num a) => [a] -> (a, [a])
maxsubseq = snd . foldl f ((0, []), (0, [])) where
f ((h1, h2),sofar) x = (a, b) where
a = max (0, []) (h1 + x , h2 ++ [x])
b = max sofar a
```

Listing 4: Error

```
Variable not in scope: h1
Variable not in scope: x
Variable not in scope: h2 :: [a1]
Variable not in scope: sofar :: (a, [a1])
```

The code for these error messages can be found on lines *1100* and *1101* respectively. Variable not in scope can also give more details when the error concerns Template Haskell, and using splice. Unfortunately, we could not find a real world example for this error, so show one from the GHC test suite.

2.1.3 IPred Error 3

Listing 5: Example Program [Eis16]

```
{-# LANGUAGE TemplateHaskell #-}
{-# LANGUAGE DuplicateRecordFields #-}
module Foo where

import qualified Data.List           as List
import      Language.Haskell.TH.Syntax (addTopDecls)

ex9 :: ()
ex9 = cat

$(do
    ds <- [d| f = cab
              cat = ()
            ]
    addTopDecls ds
    [d| g = cab
        cap = True
      ])
```

Listing 6: Error

```
Variable not in scope: cat :: ()
'cat' (splice on lines 13-20) is not in scope before line 13
```

The function `mk_bind_scope_msg` starting on line *1118* is what provides us with the above message.

2.1.4 IPred Error 4

The following program triggers the **second** `mkHoleError` function (starting on *1137*), first for type holes in expressions (*1164*):

Listing 7: Example Program

```
insert x [] = _
```

Listing 8: Error

```
Found hole: _ :: [a]
  Where: 'a' is a rigid type variable bound by
        the inferred type of insert :: Ord a => a -> [a] -> [a]
        at Insert.hs:(1,1)-(3,40)
```

2.1.5 IPred Error 5

The second error that comes from the second `mkHoleError` function covers typed holes in signatures (*1167*):

Listing 9: Example Program [lef17]

```
module Foo where
myFunction :: _ -> String
myFunction x = "The value is " ++ show x
```

Listing 10: Error

```
Found type wildcard '_' standing for 'a0'
  Where: 'a0' is an ambiguous type variable
  To use the inferred type, enable PartialTypeSignatures
```

The last error message in this section deals with implicit parameters.

2.1.6 IPred Error 6

The message is on line *1259* from within the `mkIPerr` function(*1251*)and we can prompt it with the following:

Listing 11: Example Program [Cli17]

```
{-# LANGUAGE ImplicitParams #-}

main = print (f z)

f g =
  let
    ?x = 42
    ?y = 5
  in
    g

z :: (?x :: Int) => Int
z = ?x
```

Which gives us the following error:

Listing 12: Error

```
Unbound implicit parameter (?x::Int) arising from a use of 'z'
```

Next we shall look at the next set of error messages that form the Equality Errors.

2.2 Equality Errors

Equality Errors messages start from line 1394. From this point on in the paper we shall show the entire error message for each faulty program.

2.2.1 Eq Error 1

To start; 'Couldn't match...' comes from `mkEqErr1(1440)` which calls the function `mkEqErr_help(1435)`. `mkEqErr_help` employs the function `mkTyVarEqErr(1597)` which in turn calls `mkTyVarEqErr'(1601)` which uses `misMatchMsg(1866)`. `misMatchMsg` provides all the 'Couldn't match...' messages as well as messages concerning lifting.

Concerning the 'When matching...' message; `mkEqErr1(1440)` calls function `mk_wanted_extra(1463)`, this provides the second part of the error. The example below states 'types' but the function also has the ability to change this to 'kind' if and when needed:

Listing 13: Example Program [Hes16]

```
{-# LANGUAGE KindSignatures #-}
{-# LANGUAGE OverloadedStrings #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE DataKinds #-}

import GHC.TypeLits (Symbol, Nat, KnownNat, natVal, KnownSymbol, symbolVal)
import Data.Text (Text)
import qualified Data.Text as Text
import Data.Proxy (Proxy(..))

data TextMax (n :: Nat) = TextMax Text
  deriving (Show)

textMax :: KnownNat n => Text -> Maybe (TextMax n)
textMax t
  | Text.length t <= (fromIntegral $ natVal (Proxy :: Proxy n)) = Just (TextMax t)
  | otherwise = Nothing
```

Listing 14: Error

```
Couldn't match kind '*' with 'Nat'
  When matching types
    proxy0 :: Nat -> *
    Proxy :: * -> *
  Expected type: proxy0 n1
  Actual type: Proxy n0
```

Lastly, this function `mk_wanted_extra(1463)` also calls another function `mkExpectedActualMsg(1921)`. `mkExpectedActualMsg` gives us the 'Expected type' and 'Actual type' part of the message, as seen on the last two lines of the example. These functions also deal with other messages but we will come to those later in the paper.

2.2.2 Eq Error 2

The next error message we meet is for coercion. `mkCoercibleExplanation(1497)` provides the following message when triggered:

Listing 15: Example Program [iva17]

```

{-# LANGUAGE GeneralizedNewtypeDeriving #-}

class NameOf a where
  nameOf :: proxy a -> String

instance NameOf Int where
  nameOf _ = "Int"

newtype MyInt = MyInt Int
  deriving (NameOf)

```

Listing 16: Error

```

Couldn't match representation of type 'proxy Int'
      with that of 'proxy MyInt'
    arising from the coercion of the method 'nameOf'
      from type 'forall (proxy :: * -> *). proxy Int -> String'
      to type 'forall (proxy :: * -> *). proxy MyInt -> String'
NB: We cannot know what roles the parameters to 'proxy' have;
    we must assume that the role is nominal

```

'Couldn't match....' is again provided by `misMatchMsg(1866)`, it is worth noting that this error message can also use the word 'expected' instead of 'representation' depending on the error received. The second part 'arising from...' is supplied by the `TcRnTypes` module (`ghc/compiler/typecheck/TcRnTypes.hs`) on 3529 and 3656 respectively. Lastly the 'NB' message is managed by `mkCoercibleExplanation(1497)` back in `TcErrors`.

`mkTyVarEqErr'(1601)` is the next function in `TcErrors` we shall look at. As well as being called as part of other messages as seen above the function also provides several messages of its own, the first, one that gets seen quite often is the Occurs Check.

2.2.3 Eq Error 3

Listing 17: Example Program [Flo09]

```

module Foo where

intersperse :: a -> [[a]] -> [a]

intersperse _ [] = []
intersperse _ [x] = x
intersperse s (x:y:xs) = x:s:y:intersperse s xs

```

This time our error message is very large, it is the first to show the three different error sections placed together to provide enough information to the programmer, 'Message', 'Context Box' and 'Relevant Bindings'.

The 'message' part of the error relating to 'Occurs check...' is found on line 1621. `mkExpectedActualMsg(1921)` gives us the 'Expected type' and 'Actual type' part of the message as before. The 'In the second...' segment of the error message is produced by `TcHsType.hs` (`ghc/compiler/typecheck/TcHsType.hs`) in function `funAppCtxt(2772)` starting on line 2774. Back into `TcErrors.hs` and our Relevant Bindings are generated from the function `relevantBindings(2865)` with the error message starting on line 2898.

Listing 18: Error

```
Occurs check: cannot construct the infinite type: a ~ [a]
  Expected type: [[a]]
  Actual type: [a]
In the second argument of '(:)', namely 'intersperse s xs'
In the second argument of '(:)', namely 'y : intersperse s xs'
In the second argument of '(:)', namely 's : y : intersperse s xs'
Relevant bindings include
  xs :: [[a]]
    (bound at /path/test/error_tests/test16.hs:7:20)
  y :: [a]
    (bound at /path/test/error_tests/test16.hs:7:18)
  x :: [a]
    (bound at /path/test/error_tests/test16.hs:7:16)
  s :: a
    (bound at /path/test/error_tests/test16.hs:7:13)
  intersperse :: a -> [[a]] -> [a]
    (bound at /path/test/error_tests/test16.hs:5:1)
```

With that error digested we move onto the next. The occurs check is made up of two cases, OC_Occurs which we have seen above and OC_Bad which we shall show below.

2.2.4 Eq Error 4

Listing 19: Example Program [0xd17]

```
{-# LANGUAGE RankNTypes #-}

data Foo a

type A a = forall m. Monad m => Foo a -> m ()
type PA a = forall m. Monad m => Foo a -> m ()
type PPFA a = forall m. Monad m => Foo a -> m ()

_pfa :: PPFA a -> PA a
_pfa = undefined

_pa :: PA a -> A a
_pa = undefined

_pp :: PPFA a -> A a
_pp = undefined

main :: IO ()
main = putStrLn "yay"
```

As before mkTyVarEqErr' function starts on line 1601 but our OC_Bad case starts on 1640 with the message on 1641.

Listing 20: Error

```
Cannot instantiate unification variable 'a0'
  with a type involving forall: PPFA a -> Foo a -> m ()
  GHC doesn't yet support impredicative polymorphism
In the expression: undefined
In an equation for '_pp': _pp = undefined
Relevant bindings include
  _pp :: PPFA a -> A a
    (bound at /path/test/error_tests/test18.hs:16:1)
```

Following each segment in this message we get 'In the expression..' which is taken from Tc-Expr.hs(ghc/compiler/typecheck/TcExpr.hs) in function exprCtxt on 2514, and like previously, the

relevant bindings are produced by the the function `relevantBindings(2865)` with the error message starting on line `2898`.

Using the same program as before we can enact the next error message that `mkTyVarEqErr` provides.

2.2.5 Eq Error 5

Listing 21: Example Program [0xd17]

```
{-# LANGUAGE RankNTypes #-}

data Foo a

type A a = forall m. Monad m => Foo a -> m ()
type PA a = forall m. Monad m => Foo a -> m ()
type PPFA a = forall m. Monad m => Foo a -> m ()

_pfa :: PPFA a -> PA a
_pfa = _pfa

_pa :: PA a -> A a
_pa = _pa

_pp :: PPFA a -> A a
_pp = _pa . _pfa

main :: IO ()
main = putStrLn "yay"
```

Listing 22: Error

```
Couldn't match type 'm0' with 'm2'
  because type variable 'm2' would escape its scope
This (rigid, skolem) type variable is bound by
  a type expected by the context:
    PA a
  at /path/test/error_tests/test17_skolem.hs:16:7-9
Expected type: (Foo a -> m0 ()) -> Foo a -> m ()
Actual type: PA a -> A a
In the first argument of '(.)', namely '_pa'
In the expression: _pa . _pfa
In an equation for '_pp': _pp = _pa . _pfa
```

As we have covered several times where 'Couldn't match...', 'Expected types...', 'In the first....' come from, we shall concentrate on the section starting 'This (rigid...)'. TcErrors state this is to capture "skolem escape". The check for this starts on `1680` with the error message starting on `1685`.

The last error of the function concerns what the TcErrors file states as: "Nastiest case: attempt to unify an untouchable variable".

2.2.6 Eq Error 6

Our program to cause this message is:

Listing 23: Example Program [nh215]

```
{-# LANGUAGE GADTs #-}

data My a where
  A :: Int -> My Int
  B :: Char -> My Char

main :: IO ()
main = do
  let x = undefined :: My a

  case x of
    A v -> print v

  print x
```

Giving the error:

Listing 24: Error

```
Couldn't match type 'a1' with '()'
  'a1' is untouchable
    inside the constraints: a2 ~ Int
    bound by a pattern with constructor: A :: Int -> My Int,
        in a case alternative
    at /path/test/error_tests/test19.hs:13:5-7
Expected type: IO a1
Actual type: IO ()
In the expression: print v
In a case alternative: A v -> print v
In a stmt of a 'do' block: case x of { A v -> print v }
```

This is handled with TcErrors at 1708 onwards but the message starts at 1713.

That was the last of the messages from mkTyVarEqErr' so moving away from the function we get to mkEqInfoMsg(1749).

2.2.7 Eq Error 7

Listing 25: Example Program [Rya18]

```
{-# LANGUAGE AllowAmbiguousTypes #-}
{-# LANGUAGE DefaultSignatures #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE TypeInType #-}
{-# LANGUAGE TypeOperators #-}
module SGenerics where

import Data.Kind (Type)
import Data.Type.Equality ((~:)(..), sym, trans)
import Data.Void

data family Sing (z :: k)

class Generic (a :: Type) where
  type Rep a :: Type
  from :: a -> Rep a
```

```

to :: Rep a -> a

class PGeneric (a :: Type) where
  type PFrom (x :: a)    :: Rep a
  type PTo  (x :: Rep a) :: a

class SGeneric k where
  sFrom :: forall (a :: k). Sing a -> Sing (PFrom a)
  sTo   :: forall (a :: Rep k). Sing a -> Sing (PTo a :: k)

class (PGeneric k, SGeneric k) => VGeneric k where
  sTof :: forall (a :: k). Sing a -> PTo (PFrom a) :~: a
  sFot :: forall (a :: Rep k). Sing a -> PFrom (PTo a :: k) :~: a

data Decision a = Proved a
                | Disproved (a -> Void)

class SDecide k where
  (%~) :: forall (a :: k) (b :: k). Sing a -> Sing b -> Decision (a :~: b)
  default (%~) :: forall (a :: k) (b :: k). (VGeneric k, SDecide (Rep k))
    => Sing a -> Sing b -> Decision (a :~: b)
  s1 %~ s2 = case sFrom s1 %~ sFrom s2 of
    Proved (Refl :: PFrom a :~: PFrom b) ->
      case (sTof s1, sTof s2) of
        (Refl, Refl) -> Proved Refl
    Disproved contra -> Disproved (\Refl -> contra Refl)

```

This large example gives us the following message:

Listing 26: Error

```

Could not deduce: PFrom a ~ PFrom a
from the context: b ~ a
  bound by a pattern with constructor:
    Refl :: forall k (a :: k). a :~: a,
  in a lambda abstraction
  at /path/test/error_tests/test20.hs:44:37-40
Expected type: PFrom a :~: PFrom b
Actual type: PFrom a :~: PFrom a
NB: 'PFrom' is a non-injective type family
In the first argument of 'contra', namely 'Refl'
In the expression: contra Refl
In the first argument of 'Disproved', namely
  '(\ Refl -> contra Refl)'
Relevant bindings include
  contra :: (PFrom a :~: PFrom b) -> Void
    (bound at /path/test/error_tests/test20.hs:44:15)
  s2 :: Sing b
    (bound at /path/test/error_tests/test20.hs:40:9)
  s1 :: Sing a
    (bound at /path/test/error_tests/test20.hs:40:3)
  (%~) :: Sing a -> Sing b -> Decision (a :~: b)
    (bound at /path/test/error_tests/test20.hs:40:3)

```

'Could not deduce...' comes the function `couldNotDeduce(1810)` which in turn also calls `pp_givens(1815)` which gives us the 'from context...' and 'bound by...' messages. The 'bound by...' message also calls for skol information which gives us the 'pattern with constructor..' part of the message, this comes from `TcRnTypes.hs` in the function `pprPatSkolInfo(3333)`.

'in a lambda abstraction' is actually called from `HsExpr(2804)` which is outside the typechecker folder in `/compiler/hsSyn/HsExpr.hs`. This file also contains some nice features as pointed out to us by Lindsey Kuper [Kup18] showing the way GHC makes the choice of using 'a' or 'an' on line 2784 using the `pprMatchContext` function, with the actual message itself provided by the function `pprMatchContextNoun(2792)`. Now, back to our message the last piece of the message we have

not covered is 'NB:....' which takes us back to our original function `mkEqInfoMsg` in `TcErrors.hs` on line 1777.

The next function we have already spoken about quite a bit in its usage with other error messages previously in this paper, but one error we did not cover are about lifted and unlifted types.

2.2.8 Eq Error 8

Listing 27: Example Program [nom17]

```
{-# LANGUAGE MagicHash #-}
import GHC.Prim
import GHC.Types

main = do
  let primDouble = 0.42## :: Double#
  let double = 0.42 :: Double
  IO (\s -> mkWeakNoFinalizer# double () s)
```

This program gives the error:

Listing 28: Error

```
Couldn't match a lifted type with an unlifted type
When matching types
  a :: *
  Weak# () :: TYPE 'UnliftedRep
Expected type: (# State# RealWorld, a #)
Actual type: (# State# RealWorld, Weak# () #)
In the expression: mkWeakNoFinalizer# double () s
In the first argument of 'IO', namely
  '(\ s -> mkWeakNoFinalizer# double () s)'
In a stmt of a 'do' block:
  IO (\ s -> mkWeakNoFinalizer# double () s)
Relevant bindings include
  main :: IO a
    (bound at /path/test/error_tests/test22.hs:5:1)
```

This is dealt with by `misMatchMsg(1869)`. 'Couldn't match...' is dealt with on line 1917, with the rest of the error coming from the same places as previously explained. One section of this error message that we haven't seen before is 'In a stmt of a...' this error is retrieved from the function `pprStmtInCtxt` from within `HsExpr.hs(2892)`.

We will now leave Equality Errors and move on to the next category of errors in `TcErrors`, Type-Class Errors.

2.3 Type-Class Errors

Type-Class Error messages start from line 2287.

2.3.1 TC Error 1

Our first error message in this section comes from the `mk_dict_err` function(2334). We can bring it about with the following code.

Listing 29: Example Program [Adr18]

```
module Tclass where
  import System.Environment

  class Console a where
    writeLine::a->IO()
    readLine::IO a
```

```

instance Console Int where
  writeLine= putStrLn . show

  readLine = do
    a <- getLine
    let b= (read a)::Int
    return b

useInt::IO()
useInt =putStrLn . show $ (2+readLine)

```

This program returns three error messages but we have already covered the first two earlier so we only present the third and concentrate on its message:

Listing 30: Error

```

Ambiguous type variable 'a0' arising from a use of 'readLine'
prevents the constraint '(Console a0)' from being solved.
Probable fix: use a type annotation to specify what 'a0' should be.
These potential instance exist:
  instance Console Int
    -- Defined at /path/test/error_tests/test23.hs:8:14
In the second argument of '(+)', namely 'readLine'
In the second argument of '($)', namely '(2 + readLine)'
In the expression: putStrLn . show $ (2 + readLine)

```

The first sentence of the error comes from the function `mkAmbigMsg(2779)`, it is called on 2799. The 'arising from...' part of this sentence gets its information from the following: First we head back into the `mk_dict_err` function to call `pprArising(2404)`. The `pprArising(927)` function employs `pprCtOrigin` from `TcRnTypes.hs`. `pprCtOrigin(3615)` uses the `ctoHerald(3528)` function which is contained in the same file and finally gives us the 'arising from' string.

Back to `TcErrors.hs` and `mk_dict_err` for the 'prevents constraints...' which is found on line 2402 along with 'Probable fix...' and 'These potential...' just a little lower starting on 2422. The last three sentences we have already cover above in other examples.

Again in `mk_dict_err` we see another error message at line 2433.

2.3.2 TC Error 2

Listing 31: Example Program [mpi15]

```

{-# LANGUAGE PatternSynonyms #-}
module Foo where
pattern Pat :: () => Show a => a -> Maybe a
pattern Pat a = Just a

```

Listing 32: Error

```

No instance for (Show a)
  arising from the "provided" constraints claimed by
  the signature of 'Pat'
In other words, a successful match on the pattern
  Just a
does not provide the constraint (Show a)
In the declaration for pattern synonym 'Pat'

```

'No instance for...' is see on line 2409 of `mk_dict_err` again using the 'arising from...' the same as the last error, and 'In other words...' is provided to us from `mb_patsyn_prov(2429)` a function within `mk_dict_err`. A different ending for this error can be seen in the next example where we receive a suggestion of '(maybe you....' given to us by the `extra_note(2443)` section of `mk_dict_err`.

2.3.3 TC Error 3

Listing 33: Example Program [Cau17]

```
module Foo where
foo4 x    = (x ==) . (&&)
```

Listing 34: Error

```
No instance for (Eq (Bool -> Bool)) arising from a use of '=='
  (maybe you haven't applied a function to enough arguments?)
In the first argument of '(.)', namely '(x ==)'
In the expression: (x ==) . (&&)
In an equation for 'foo4': foo4 x = (x ==) . (&&)
```

Three more functions provide error messages in `mk_dict_err`. The first `drv_fix(2462)` we could not find an example for its main case ‘fill in the wildcard constraint yourself...’, however, we did find an example for its ‘otherwise’ case ‘use a standalone....’ as shown below.

2.3.4 TC Error 4

Listing 35: Example Program [Rya17]

```
newtype Foo f a = Foo (f (f a)) deriving Eq
```

Listing 36: Error

```
No instance for (Eq (f (f a)))
  arising from the 'deriving' clause of a data type declaration
Possible fix:
  use a standalone 'deriving instance' declaration,
  so you can specify the instance context yourself
When deriving the instance for (Eq (Foo f a))
```

‘Possible fix’ is taken from the function `show_fixes(2664)` with the actual message about ‘use a standalone...’ on 2466. ‘When deriving...’ then comes from the function `derivInstCtxt(605)` that is in the `TcDerivInfer.hs` (`ghc/compiler/typecheck/TcDerivInfer.hs`) file.

Now back to the `TcErrors` file. Still within the `mk_dict_err` we get to the function `overlap_msg(2470)`. The following program triggers it:

2.3.5 TC Error 5

Listing 37: Example Program [Vis17]

```
{-# LANGUAGE PolyKinds, RankNTypes, ConstraintKinds, FlexibleInstances,
      UndecidableInstances, MultiParamTypeClasses, FunctionalDependencies #-}
module Foo where
class Class1 b h | h -> b
instance Class1 Functor Applicative
instance Class1 Applicative Monad

class SuperClass1 b h
instance {-# OVERLAPPING #-} SuperClass1 b b
instance {-# OVERLAPPABLE #-} (SuperClass1 b c, Class1 c h) => SuperClass1 b h

newtype HFree c f a = HFree { runHFree :: forall g. c g => (forall b. f b -> g b) -> g
  a }

instance SuperClass1 Functor c => Functor (HFree c f)
instance SuperClass1 Applicative c => Applicative (HFree c f)
instance SuperClass1 Monad c => Monad (HFree c f)
```

```
test :: (a -> b) -> HFree Monad f a -> HFree Monad f b
test = fmap
```

With the relevant error messages being found on lines 2472 and 2495 respectively.

Listing 38: Error

```
Overlapping instances for SuperClass1 Functor c1
  arising from the superclasses of an instance declaration
Matching instances:
  instance [overlappable] forall k1 k2 k3 (b :: k3) (c :: k2) (h :: k1).
    (SuperClass1 b c, Class1 c h) =>
      SuperClass1 b h
    -- Defined at /path/test/error_tests/test29.hs:9:31
  instance [overlapping] forall k (b :: k). SuperClass1 b b
    -- Defined at /path/test/error_tests/test29.hs:8:30
(The choice depends on the instantiation of 'c1, k1'
To pick the first instance above, use IncoherentInstances
when compiling the other instance declarations)
In the instance declaration for 'Applicative (HFree c f)'
```

Lastly we come to the final message that `mk_dict_err` provides. This message is found within `safe_haskell_msg(2522)`. We could not find a real world example to get this message so took the one from the GHC test suite.

2.3.6 TC Error 6

Listing 39: Example Program [Jon11]

```
{-# LANGUAGE Trustworthy #-}
module Main where

import safe SafeLang10_A -- trusted lib
import safe SafeLang10_B -- untrusted plugin

main = do
  let r = res [(1::Int)]
  putStrLn $ "Result: " ++ show r
  putStrLn $ "Result: " ++ show function
```

Listing 40: Error

```
Unsafe overlapping instances for Pos [Int]
  arising from a use of 'res'
The matching instance is:
  instance [overlapping] [safe] Pos [Int]
    -- Defined at SafeLang10_B.hs:13:30
It is compiled in a Safe module and as such can only
overlap instances from the same module, however it
overlaps the following instances from different modules:
  instance Pos [a] -- Defined at SafeLang10_A.hs:13:10
In the expression: res [(1 :: Int)]
In an equation for 'r': r = res [(1 :: Int)]
In the expression:
  do let r = res ...
     putStrLn $ "Result: " ++ show r
     putStrLn $ "Result: " ++ show function
```

We now move out of `mk_dict_err` and into the `pprPotentials(2669)` function for our last error message in the Type-Class section.

2.3.7 TC Error 7

Listing 41: Example Program [wer16]

```
{-# LANGUAGE FlexibleInstances #-}
module Style where

import Control.Monad.Writer.Lazy

type StyleM = Writer [(String, String)]
newtype Style = Style { runStyle :: StyleM () }

class Term a where
  term :: String -> a

instance Term String where
  term = id

instance Term (String -> StyleM ()) where
  term property value = tell [(property, value)]

display :: String -> StyleM ()
display = term "display"

flex :: Term a => a
flex = term "flex"

someStyle :: Style
someStyle = Style $ do
  flex "1"      -- [1] :: StyleM ()
  display flex -- [2]
```

Most of this error message we have covered already apart from one covered by pprPotentials beginning with 'one instance involving.....' and that can be found starting on line 2718.

Listing 42: Error

```
Ambiguous type variable 'a0' arising from a use of 'flex'
prevents the constraint '(Term
  ([Char]
    -> WriterT
      [(String, String)]
      Data.Functor.Identity.Identity
      a0))' from being solved.
(maybe you haven't applied a function to enough arguments?)
Probable fix: use a type annotation to specify what 'a0' should be.
These potential instance exist:
  one instance involving out-of-scope types
  (use -fprint-potential-instances to see them all)
In a stmt of a 'do' block: flex "1"
In the second argument of '($)', namely
  'do flex "1"
    display flex'
In the expression:
  Style
    $ do flex "1"
      display flex
```

This section provides only one Error message which starts from line *3006* in TcErrors.hs.

Listing 43: Example Program [luq17]

And its error message:

Reduction stack overflow; size = 201

Use `-freduction-depth=0` to disable this check

TcValidity.hs can be found in `ghc/compiler/typecheck/`.

Listing 45: Example Program [CE14]

And its error message:

Listing 46: Error

```
Non type-variable argument in the constraint: Num [a]
  (Use FlexibleContexts to permit this)
When checking the inferred type
  sum2 :: forall a. Num [a] => [[a]] -> [a]
```

'Non type-variable argument....' is provided by function `predTyVarErr` starting line 1042 and the 'When checking the inferred type...' is provided by another module `TcBinds(ghc/compiler/typecheck/TcBinds.hs)` with function `mk_inf_msg(1054)`.

4 TcRnDriver.hs

`TcRnDriver.hs` can be found in `ghc/compiler/typecheck/`.

4.0.1 Dri Error 1

Listing 47: Example Program [CE14]

```
import Data.Char

-- Problem: type of (++) doesn't match String -> Char -> String
test12 :: Parser Char String
test12 = (++) <$> token "hello world"
          <*> symbol '!'
```

Listing 48: Error

```
Not in scope: type constructor or class 'Parser'
```

The 'Not in scope: part of this message is found within `tcRnLookupRdrName(2512)`. However, the 'type constructor...' is held within a competently different module, `OccName.hs(ghc/compiler/basicTypes/OccName.hs)` with the message being generated by `pprNameSpace` starting on line 192.

5 Conclusion and Future Work

This paper covered the error messages found in the GHC compiler within the `TcError.hs` file. We have provided an example of how to trigger each error along with describing where each place of the error occurs in the compiler.

So far we have only covered the errors mentioned in `TcErrors.hs`, briefly mentioning other modules if they are called. For the future we would like to cover all the modules within the type checker folder.

References

- [Oxd17] 0xd34df00d. Function composition and forall'ed types, 2017. <https://stackoverflow.com/questions/43943793/function-composition-and-forallded-types> Accessed July 2018.
- [Adr18] Bercovici Adrian. Ambiguous type variable that prevents the constraint, 2018. <https://stackoverflow.com/questions/51042183/ambiguous-type-variable-that-prevents-the-constraint> Accessed July 2018.
- [Cau17] Daniel Causebrook. Function composition errors, 2017. <https://stackoverflow.com/questions/47298872/function-composition-errors> Accessed July 2018.
- [CE14] Sheng Chen and Martin Erwig. Counter-factual typing for debugging type errors. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 583–594, 2014.

- [Cli17] Clinton. A function that sets an implicit parameter context, 2017. <https://stackoverflow.com/questions/41661714/a-function-that-sets-an-implicit-parameter-context> Accessed July 2018.
- [Eis16] Jason Eisenberg. Testsuite test t11680.hs, 2016. <https://github.com/ghc/ghc/blob/master/testsuite/tests/th/T11680.hs> Accessed July 2018.
- [Flo09] Charlie Flowers. Why does this haskell code produce the “infinite type” error?, 2009. <https://stackoverflow.com/questions/795317/why-does-this-haskell-code-produce-the-infinite-type-error> Accessed July 2018.
- [Hes16] Sean Clark Hess. Couldn’t match kind ‘*’ with ‘nat’, 2016. <https://stackoverflow.com/questions/40731220/couldnt-match-kind-with-nat> Accessed July 2018.
- [iva17] ivanm. Generalizednewtyperderiving and polymorphic arguments don’t play nicely together, 2017. <http://haskell.1045720.n5.nabble.com/GHC-14220-GeneralizedNewtypeDeriving-and-polymorphic-arguments-don-t-play-nicely-together-td5863971.html> Accessed July 2018.
- [Jon11] Simon Peyton Jones. Safelang10, 2011. <https://git.science.uu.nl/f100183/ghc-invariant/blob/1ae5fa451f4f554e0d652d55f9312a585188ce13/testsuite/tests/safeHaskell/safeLanguage/SafeLang10.hs> Accessed July 2018.
- [Kad16] Zubin Kadva. Haskell: Variable not in scope, 2016. <https://stackoverflow.com/questions/39403798/haskell-variable-not-in-scope> Accessed July 2018.
- [Kup18] Lindsey Kuper. Reply to : Help with finding the term lambda abstraction in ghc, 2018. <https://twitter.com/JoannaSharrad/status/1016662521757958145> Accessed July 2018.
- [lef17] leftaroundabout. Passing any type in function signature in haskell, 2017. <https://stackoverflow.com/questions/45983399/passing-any-type-in-function-signature-in-haskell> Accessed July 2018.
- [luq17] luqui. How can undecidable instances actually hang the compiler?, 2017. <https://stackoverflow.com/questions/42356242/how-can-undecidable-instances-actually-hang-the-compiler> Accessed July 2018.
- [mpi15] mpickering. Bad error message for incorrect pattern synonym signature, 2015. <https://ghc.haskell.org/trac/ghc/ticket/10873> Accessed July 2018.
- [nh215] nh2. Type inference with gadt - a0 is untouchable, 2015. <https://stackoverflow.com/questions/28582210/type-inference-with-gadts-a0-is-untouchable> Accessed July 2018.
- [nom17] nomeata. Unhelpful error messages about lifted and unlifted types, 2017. <https://ghc.haskell.org/trac/ghc/ticket/13610> Accessed July 2018.
- [ONi15] ONiel. Data constructor not in scope, 2015. <https://www.dreamincode.net/forums/topic/396776-data-constructor-not-in-scope/> Accessed July 2018.
- [Rya17] RyanG1Scott. Allow partialtypesignatures in the instance context of a standalone deriving declaration, 2017. <https://ghc.haskell.org/trac/ghc/ticket/13324> Accessed July 2018.
- [Rya18] RyanG1Scott. Ghc 8.4.1-alpha regression with typeintype, 2018. <https://ghc.haskell.org/trac/ghc/ticket/14720> Accessed July 2018.
- [Vis17] Sjoerd Visscher. Unexpected overlapping instances error, 2017. <https://stackoverflow.com/questions/42053915/unexpected-overlapping-instances-error> Accessed July 2018.
- [wer16] wereHamster. Ghc doesn’t pick the only available instance, 2016. <https://stackoverflow.com/questions/41086039/ghc-doesnt-pick-the-only-available-instance> Accessed July 2018.