Equality Proofs and Deferred Type Errors
A Compiler Pearl

Dimitrios Vytiniotis  Simon Peyton Jones  José Pedro Magalhães

September 11, 2012
An interactive GHC session in 7.6

```haskell
> ghci -fdefer-type-errors
GHCi, version 7.6.1: http://www.haskell.org/ghc/

Prelude>
```

José Pedro Magalhães, University of Oxford
Equality Proofs and Deferred Type Errors: a Compiler Pearl, ICFP 2012
An interactive GHC session in 7.6

> ghci -fdefer-type-errors
GHCi, version 7.6.1: http://www.haskell.org/ghc/

Prelude> let \(x = (\text{True}, 'q' \land \text{False})\)
<interactive>:2:16: Warning:
   Couldn’t match expected type \text{Bool} with actual type \text{Char}

Prelude>
An interactive GHC session in 7.6

> ghci -fdefer-type-errors
GHCi, version 7.6.1: http://www.haskell.org/ghc/

Prelude> let x = (True, 'q' \& False)
<interactive>:2:16: Warning:
   Couldn’t match expected type `Bool' with actual type `Char'

Prelude> fst x
True

Prelude>
An interactive GHC session in 7.6

> ghci -fdefer-type-errors
GHCi, version 7.6.1: http://www.haskell.org/ghc/

Prelude> let \textit{x} = (\textit{True}, 'q' \&\& False)
<interactive>:2:16: Warning:
   Couldn’t match expected type \textit{Bool} with actual type \textit{Char}

Prelude> \textit{fst} \textit{x}
\textit{True}

Prelude> \textit{snd} \textit{x}
*** Exception: <interactive>:2:16:
   Couldn’t match expected type \textit{Bool} with actual type \textit{Char}
   (deferred type error)
Why would one want to defer type errors?

- During prototyping, commenting out parts of code
  - But if you comment out $f$, you must comment out everything that calls $f$...
  - Deferring type errors is like lazily commenting out code

- During refactoring, making large changes that affect a whole project
  - But you want to be able to test each part of the project as you complete refactoring
  - With deferred type errors, only the errors that “matter” show up

Useful for an IDE

It’s a simple, instructive, and elegant application of multiple GHC features:

- Equality proofs (coercions)
- Laziness
- Kind polymorphism
- Coercion optimisation
Why would one want to defer type errors?

- During prototyping, commenting out parts of code
  - But if you comment out \( f \), you must comment out everything that calls \( f \) ...
  - Deferring type errors is like lazily commenting out code
- During refactoring, making large changes that affect a whole project
  - But you want to be able to test each part of the project as you complete refactoring
  - With deferred type errors, only the errors that “matter” show up
Why would one want to defer type errors?

- During prototyping, commenting out parts of code
  - But if you comment out \( f \), you must comment out everything that calls \( f \)...
  - Deferring type errors is like lazily commenting out code
- During refactoring, making large changes that affect a whole project
  - But you want to be able to test each part of the project as you complete refactoring
  - With deferred type errors, only the errors that “matter” show up
- Useful for an IDE
Why would one want to defer type errors?

- During prototyping, commenting out parts of code
  - But if you comment out \( f \), you must comment out everything that calls \( f \)...
  - Deferring type errors is like lazily commenting out code
- During refactoring, making large changes that affect a whole project
  - But you want to be able to test each part of the project as you complete refactoring
  - With deferred type errors, only the errors that “matter” show up
- Useful for an IDE

It’s a simple, instructive, and elegant application of multiple GHC features:

- Equality proofs (coercions)
- Laziness
- Kind polymorphism
- Coercion optimisation
Equality proofs

Back to our example \( x = (True, \ 'q' \land False) \); GHC compiles this into:

\[
x = \text{let } (c :: \text{Char} \sim \text{Bool}) = \text{error } "\text{Couldn’t…}"
\text{in } (True, (\ 'q' \triangleright c) \land False)
\]

- \( \tau_1 \sim \tau_2 \): type equality constraint
- \text{error}: runtime error
- \( e \triangleright c \): cast operator

Due to laziness, we only get an error if the second component of the pair is evaluated.
Consider the expression

\[ \text{show } \mathit{xs} \]

where \( \mathit{xs} :: [\text{Int}] \), and \( \text{show} :: \forall a. \text{Show } a \Rightarrow a \rightarrow \text{String} \).
Consider the expression

\[ \text{show } \mathbf{x s} \]

where \( \mathbf{x s} :: [\text{Int}] \), and \( \text{show} :: \forall a.\text{Show } a \Rightarrow a \rightarrow \text{String} \).

GHC compiles it by first generating constraints and elaborated terms:

\[
\begin{align*}
\text{Constraints:} & \quad d_6 : \text{Show } [\text{Int}] \\
\text{Elaborated term:} & \quad \text{show } [\text{Int}] d_6 \ \mathbf{x s}
\end{align*}
\]
Consider the expression

\[ \text{show } \mathit{xs} \]

where \( \mathit{xs} :: [\text{Int}] \), and \( \text{show} :: \forall a. \text{Show } a \Rightarrow a \rightarrow \text{String} \).

GHC compiles it by first generating constraints and elaborated terms:

Constraints: \( d_6 : \text{Show } [\text{Int}] \)

Elaborated term: \( \text{show } [\text{Int}] d_6 \mathit{xs} \)

Then, the constraint solver solves the generated constraints:

\[
\begin{align*}
\text{let } & d_6 :: \text{Show } [\text{Int}] = \$d\text{ShowList Int } \$d\text{ShowInt} \\
\text{in } & \text{show } [\text{Int}] d_6 \mathit{xs}
\end{align*}
\]
Actually, there are equality constraints too:

Constraints: 
\[ d_6 :: Show \alpha \]
\[ c_5 :: [Int] \sim \alpha \]

Elaborated term: 
\[ show \alpha d_6 (xs \triangleright c_5) \]

- $\alpha$: fresh unification variable
- $c_5$: constraint arising from applying $show$ to $xs$
- $xs \triangleright c_5$: $xs$ with the type expected by $show$
Constraints:  
\[ d_6 :: \text{Show } \alpha \]
\[ c_5 :: \text{Int } \sim \alpha \]
Elaborated term:  
\[ \text{show } \alpha \ d_6 \ (xs \triangleright c_5) \]
Type inference with constraints III

Constraints:  
\[ d_6 :: \text{Show } \alpha \]
\[ c_5 :: \text{Int } \sim \alpha \]

Elaborated term:  
\[ \text{show } \alpha \ d_6 \ (xs \triangleright c_5) \]

The next step is to solve the constraints:

\[ \alpha = [\text{Int}] \]
\[ c_5 :: [\text{Int}] \sim \alpha = \text{mkRefl } [\text{Int}] \]
\[ d_6 :: \text{Show } [\text{Int}] = \$dShowList \text{ Int } \$dShowInt \]

- \( \alpha \) is replaced by \([\text{Int}]\)
- \( \text{mkRefl } [\text{Int}] \) builds the vacuous equality \([\text{Int}] \sim [\text{Int}]\)
The optimisation of type equalities uses machinery that has been in place for over twenty years. This is how integer arithmetic is implemented in GHC:

```haskell
data Int = I# Int#

plusInt :: Int → Int → Int
plusInt x y = case x of
    I# x' → case y of
        I# y' → I# (x' +# y')
```

José Pedro Magalhães, University of Oxford
At the call site, the inliner and optimiser work together to simplify the expression:

\[ x \text{` plusInt` } x \]

\[ \equiv \]

\[ \text{case } x \text{ of} \]

\[ l_\# a \rightarrow \text{case } x \text{ of} \]

\[ l_\# b \rightarrow l_\# (a + \# b) \]

\[ \equiv \]

\[ \text{case } x \text{ of} \]

\[ l_\# a \rightarrow l_\# (a + \# a) \]
The same goes for type equalities; (∼) is just boxed type equality:

```haskell
data a ∼ b where
    Eq# :: (a ∼# b) → a ∼ b
```

And (▷) is a wrapper for the internal cast (▷#):

```haskell
(▷) :: ∀(a b :: ⋆). a → (a ∼ b) → b
(▷) = Λ(a b :: ⋆). λ(x :: a). λ(eq :: a ∼ b).
    case eq of
        Eq# (c :: a ∼# b) → x ▷# c
```
Here we see the optimisation of a trivial equality constraint:

\[
\begin{align*}
&\text{let } (c :: \text{Char }\sim \text{Char}) = \text{mkRefl Char} \\
&\text{in } \ldots (e \triangleright c) \ldots \\
\equiv \\
&\text{let } (c :: \text{Char }\sim \text{Char}) = \text{Eq}\# \text{Char Char }\langle\text{Char}\rangle \\
&\text{in } \ldots (\text{case } c \text{ of } \text{Eq}\# c' \rightarrow e \triangleright\# c') \ldots \\
\equiv \\
&\ldots (e \triangleright\# \langle\text{Char}\rangle) \ldots
\end{align*}
\]
Deferred type errors, on the other hand, optimise to calls to `error`:

```haskell
let (c :: Char ~ Bool) = error "Couldn’t match..."
in  snd (True, ('a' ▶ c) ∧ False)
≡
snd (True, (case error "..." of Eq# c → 'a' ▶# c) ∧ False)
≡
snd (True, error "...")
```

Again, no changes to the optimiser were required!
Conclusion

Deferring type errors:

- Is a valuable feature, and simple to implement in GHC
- Makes good use of coercions, coercion optimisation, laziness, and kind polymorphism

What it doesn’t do:

- Defer errors other than type errors:
  - Parse errors?
  - Kind errors?
- Deferring type errors $\not\equiv$ dynamic typing!
Conclusion

Deferring type errors:

- Is a valuable feature, and simple to implement in GHC
- Makes good use of coercions, coercion optimisation, laziness, and kind polymorphism

What it doesn’t do:

- Defer errors other than type errors:
  - Parse errors?
  - Kind errors?
- Deferring type errors $\not\equiv$ dynamic typing!