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

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

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>
```

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 x = (True, 'q' ∧ False)
<interactive>:2:16: Warning:
  Couldn't match expected type Bool with actual type Char

Prelude> fst x
True

Prelude> snd x
*** Exception: <interactive>:2:16:
  Couldn't match expected type Bool with actual type 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

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 $x = (True, \texttt{'q'} \wedge False)$; GHC compiles this into:

$$\begin{array}{l} x = \textbf{let}\ (c :: Char \sim Bool) = error\ \texttt{"Couldn't..."} \\ \quad\ \ \textbf{in}\ \ (True, (\texttt{'q'} \rhd c) \wedge False) \end{array}$$

- $\tau_1 \sim \tau_2$: type equality constraint
- *error*: runtime error
- $e \rhd c$: cast operator

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

Type inference with constraints I

Consider the expression

$$show\ xs$$

where $xs :: [Int]$, and $show :: \forall a. Show\ a \Rightarrow a \rightarrow String$.

Type inference with constraints I

Consider the expression

$show\ xs$

where $xs :: [Int]$, and $show :: \forall a. Show\ a \Rightarrow a \rightarrow String$.

GHC compiles it by first generating constraints and elaborated terms:

$$\begin{aligned} \text{Constraints:} \quad & d_6 : Show\ [Int] \\ \text{Elaborated term:} \quad & show\ [Int]\ d_6\ xs \end{aligned}$$

Type inference with constraints I

Consider the expression

$$show\ xs$$

where $xs :: [Int]$, and $show :: \forall a. Show\ a \Rightarrow a \rightarrow String$.

GHC compiles it by first generating constraints and elaborated terms:

$$\begin{array}{rl} \text{Constraints:} & d_6 : Show\ [Int] \\ \text{Elaborated term:} & show\ [Int]\ d_6\ xs \end{array}$$

Then, the constraint solver solves the generated constraints:

$$\begin{array}{l} \textbf{let}\ d_6 :: Show\ [Int] = \$dShowList\ Int\ \$dShowInt \\ \textbf{in}\ \ show\ [Int]\ d_6\ xs \end{array}$$

Type inference with constraints II

Actually, there are equality constraints too:

$$\begin{array}{rl} \text{Constraints:} & d_6 :: \mathit{Show}\ \alpha \\ & c_5 :: [\mathit{Int}] \sim \alpha \\ \text{Elaborated term:} & \mathit{show}\ \alpha\ d_6\ (\mathit{xs} \triangleright c_5) \end{array}$$

- α: fresh unification variable
- c_5: constraint arising from applying *show* to *xs*
- $xs \triangleright c_5$: *xs* with the type expected by *show*

Type inference with constraints III

Constraints: $d_6 :: Show\ \alpha$

$c_5 :: Int \sim \alpha$

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

Type inference with constraints III

$$\begin{aligned} \text{Constraints:} \quad & d_6 :: \mathit{Show}\ \alpha \\ & c_5 :: \mathit{Int} \sim \alpha \\ \text{Elaborated term:} \quad & \mathit{show}\ \alpha\ d_6\ (\mathit{xs} \triangleright c_5) \end{aligned}$$

The next step is to solve the constraints:

$$\begin{aligned} \alpha \quad &= \quad [\mathit{Int}] \\ c_5 :: [\mathit{Int}] \sim \alpha \quad &= \quad \mathit{mkRefl}\ [\mathit{Int}] \\ d_6 :: \mathit{Show}\ [\mathit{Int}] \quad &= \quad \$\mathit{dShowList}\ \mathit{Int}\ \$\mathit{dShowInt} \end{aligned}$$

- α is replaced by $[\mathit{Int}]$
- $\mathit{mkRefl}\ [\mathit{Int}]$ builds the vacuous equality $[\mathit{Int}] \sim [\mathit{Int}]$

Optimising equalities I

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:

$$\textbf{data}\ \mathit{Int} = I_{\#}\ \mathit{Int}_{\#}$$

$$\begin{aligned}
&\mathit{plusInt} :: \mathit{Int} \rightarrow \mathit{Int} \rightarrow \mathit{Int}\\
&\mathit{plusInt}\ x\ y = \textbf{case}\ x\ \textbf{of}\\
&\qquad\qquad I_{\#}\ x' \rightarrow \textbf{case}\ y\ \textbf{of}\\
&\qquad\qquad\qquad I_{\#}\ y' \rightarrow I_{\#}\ (x' +_{\#} y')
\end{aligned}$$

Optimising equalities II

At the call site, the inliner and optimiser work together to simplify the expression:

$$x \,\text{`}\mathit{plusInt}\text{`}\, x$$

$$\equiv$$

$$\begin{array}{l}\textbf{case}\ x\ \textbf{of} \\ \quad I_{\#}\ a \rightarrow \textbf{case}\ x\ \textbf{of} \\ \qquad\qquad\quad I_{\#}\ b \rightarrow I_{\#}\ (a +_{\#} b)\end{array}$$

$$\equiv$$

$$\begin{array}{l}\textbf{case}\ x\ \textbf{of} \\ \quad I_{\#}\ a \rightarrow I_{\#}\ (a +_{\#} a)\end{array}$$

Optimising equalities III

The same goes for type equalities; $(\sim)$ is just boxed type equality:

$$\textbf{data}\ a \sim b\ \textbf{where}$$
$$\quad Eq_{\#} :: (a \sim_{\#} b) \rightarrow a \sim b$$

And $(\triangleright)$ is a wrapper for the internal cast $(\triangleright_{\#})$:

$$(\triangleright) :: \forall (a\ b :: \star).a \rightarrow (a \sim b) \rightarrow b$$
$$(\triangleright) = \Lambda (a\ b :: \star) .\ \lambda (x :: a) .\ \lambda (eq :: a \sim b) .$$
$$\quad \textbf{case}\ eq\ \textbf{of}$$
$$\quad\quad Eq_{\#}\ (c : a \sim_{\#} b) \rightarrow x \triangleright_{\#} c$$

Optimising equalities IV

Here we see the optimisation of a trivial equality constraint:

$$
\begin{aligned}
&\textbf{let}\ (c :: \mathit{Char} \sim \mathit{Char}) = \mathit{mkRefl}\ \mathit{Char} \\
&\textbf{in}\ \ \ldots(e \triangleright c)\ldots \\
\equiv\ & \\
&\textbf{let}\ (c :: \mathit{Char} \sim \mathit{Char}) = \mathit{Eq}_{\#}\ \mathit{Char}\ \mathit{Char}\ \langle \mathit{Char} \rangle \\
&\textbf{in}\ \ \ldots(\textbf{case}\ c\ \textbf{of}\ \mathit{Eq}_{\#}\ c' \rightarrow e \triangleright_{\#} c')\ldots \\
\equiv\ & \\
&\ldots(e \triangleright_{\#} \langle \mathit{Char} \rangle)\ldots
\end{aligned}
$$

Optimising equalities V

Deferred type errors, on the other hand, optimise to calls to *error*:

$$\textbf{let}\ (c :: \mathit{Char} \sim \mathit{Bool}) = \mathit{error}\ \texttt{"Couldn't match..."}$$
$$\textbf{in}\ \ \mathit{snd}\ (\mathit{True}, (\texttt{'a'} \triangleright c) \wedge \mathit{False})$$

$$\equiv$$

$$\mathit{snd}\ (\mathit{True}, (\textbf{case}\ \mathit{error}\ \texttt{"..."}\ \textbf{of}\ \mathit{Eq}_{\#}\ c \rightarrow \texttt{'a'} \triangleright_{\#} c) \wedge \mathit{False})$$

$$\equiv$$

$$\mathit{snd}\ (\mathit{True}, \mathit{error}\ \texttt{"..."})$$

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

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!