Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Customizing type error diagnosis in GHC

Jurriaan Hage (collaboration with Alejandro Serrano Mena)

Department of Information and Computing Sciences, Universiteit Utrecht
J.Hage@uu.nl

December 5, 2017

Introduction and Motivation

Static type systems

- Statically typed languages come equiped with an intrinsic type system, preventing some structurally correct programs from being compiled
- Well-worn slogan: "well-typed programs can't go wrong"
- type incorrect programs $\Rightarrow$ the need for diagnosis

What is type error diagnosis?

- Type error diagnosis is the problem of communicating to the programmer that and/or why a program is not type correct
- This may involve information
  - that a program is type incorrect
  - which inconsistency was detected
  - which parts of the program contributed to the inconsistency
  - how the inconsistency may be fixed
- Traditionally, functional languages have more room for inconsistencies $\Rightarrow$ at least some attention was paid to type error diagnosis

Languages follow Lehmann's sixth law

- Java has seen the introduction of parametric polymorphism (and type errors suffered)
- Java has seen the introduction of anonymous functions (I have not dared look)
- Languages like Scala embrace multiple paradigms
- Martin Odersky's "type wall": unless complicated type system features are balanced by better diagnosis, programmers will flock to dynamic languages
- And what do implicits do to type error diagnosis?
- New trends: dynamic languages becoming more static
- Again, diagnosis rears its head

Example 1: domain-specific terms in Diagrams

From the *diagrams* library (Yorgey, 2012/2016)

$$\begin{aligned} atop \; :: \; & (OrderedField\ n, Metric\ v, Semigroup\ m) \\ \Rightarrow\; & QDiagram\ b\ v\ n\ m \rightarrow \\ & QDiagram\ b\ v\ n\ m \rightarrow \\ & QDiagram\ b\ v\ n\ m \end{aligned}$$

writing *atop True* gives

```
Couldn't match type 'QDiagram b v n m' with type 'Bool'
```

or for *atop cube3d plane2d* might give

```
Couldn't match type 'V2' with type 'V3'
```

We would like to see domain terms, like 'vector spaces' in the messages.

Example 2: Left undischarged in Persistent

From the *persistent* library (Snoyman, 2012)

$$\begin{aligned} \mathit{insertUnique} \;::\; & (\mathit{MonadIO}\ m, \mathit{PersistUniqueWrite}\ \mathit{backend}, \\ & \mathit{PersistEntity}\ \mathit{record}) \\ \Rightarrow\; & \mathit{record} \rightarrow \\ & \mathit{ReaderT}\ \mathit{backend}\ m\ (\mathit{Maybe}\ (\mathit{Key}\ \mathit{record})) \end{aligned}$$

use of *insertUnique* gives rise to type class predicates that may be left undischarged, because the programmer forgot to write a *PersistEntity* instance.

We'd like to get something like:

```
Data type 'Person' is not declared as a Persistent
entity. Hint: entity definition can be automatically
derived. Read more at http://www.yesodweb.com/...
```

Example 3: Formatting (type safe printf)

$hello = format\ (now$ `"Hello, World!"`$)$

```
FormatEx-orig.hs:26:21:
    Couldn't match expected type 'T.Builder'
       with actual type '[Char]'
    In the first argument of 'now', namely
       '"Hello, World!"'
    In the first argument of 'format', namely
       '(now "Hello, World!")'
    In the expression: format (now "Hello, World!")
```

It would be helpful to have a hint on how to fix the problem.

Example 4: can we have a sibling, please?

```
pExpr = pAndPrioExpr
    <|> sem_Expr_Lam    -- Semantics for lambda expressions
      <$ pKey "\\"
      <*> pFoldr1 (sem_LamIds_Cons, sem_LamIds_Nil) pVarid
      <*> pKey "->"
      <*> pExpr
```

The error message that results:

```
ERROR "BigTypeError.hs":1 - Type error in application
*** Expression     : sem_Expr_Lam <$ pKey "\\" <*> pFoldr1 (sem_LamIds_Cons,sem_
LamIds_Nil) pVarid <*> pKey "->"
*** Term           : sem_Expr_Lam <$ pKey "\\" <*> pFoldr1 (sem_LamIds_Cons,sem_
LamIds_Nil) pVarid
*** Type           : [Token] -> [((Type -> Int -> [([Char],(Type,Int,Int))] -> I
nt -> Int -> [(Int,(Bool,Int))] -> (PP_Doc,Type,a,b,[c] -> [Level],[S] -> [S]))
-> Type -> d -> [([Char],(Type,Int,Int))] -> Int -> Int -> e -> (PP_Doc,Type,a,b
,f -> f,[S] -> [S]),[Token])]
*** Does not match : [Token] -> [([Char] -> Type -> d -> [([Char],(Type,Int,Int)
)] -> Int -> Int -> e -> (PP_Doc,Type,a,b,f -> f,[S] -> [S]),[Token])]
```

Example 5: simplifying monads

$okay :: IO\ ()$
$okay = (return >=> putChar)\ \text{'a'}$

$notokay :: Maybe\ Char$
$notokay = (return >=> (\backslash x \to Nothing))\ \text{'a'}$

Forbid to use monads unless with IO:

```
ClassExperiment.hs:28:12: error:
    * Illegal use of monads: you are allowed to use IO,
      but not the Maybe monad
      According to your teacher, you have yet to pass
      your monad-license
    * In the expression: return >=> (\ x -> Nothing)
      In the expression: (return >=> (\ x -> Nothing)) 'a'
      In an equation for 'notokay':
          notokay = (return >=> (\ x -> Nothing)) 'a'
```

Domain Specific Type Error Diagnosis

What is a DSL?

Examples 1 - 4 dealt with embedded domain-specific languages.

- Walid Taha:
  - the domain is well-defined and central
  - the notation is clear,
  - the informal meaning is clear,
  - the formal meaning is clear and implemented.

What is a DSL?

Examples 1 - 4 dealt with embedded domain-specific languages.

- Walid Taha:
  - the domain is well-defined and central
  - the notation is clear,
  - the informal meaning is clear,
  - the formal meaning is clear and implemented.
- Missing is:
  - and an implementation of the DSL can communicate with the programmer about the program in terms of the domain
- "domain-abstractions should not leak"

Embedded Domain Specific Languages

- Embedded (internal à la Fowler) Domain Specific Languages are achieved by encoding the DSL syntax inside that of a host language.
- Some (arguable) advantages:
  - familiarity host language syntax
  - escape hatch to the host language
  - existing libraries, compilers, IDE's, etc.
  - combining EDSLs
- At the very least, useful for prototyping DSLs
- According to Hudak "the ultimate abstraction"

What host language?

- Some languages provide extensibility as part of their design, e.g., Ruby, Python, Scheme
- Others are rich enough to encode a DSL with relative ease, e.g., Haskell, C++
- In most languages we just have to make do
- In Haskell, EDSLs are simply libraries that provide some form of "fluency"
  - Consisting of domain terms and types, and special operators with particular priority and fixity

Challenges for EDSLs

- How to achieve:
  - domain specific optimisations
  - domain specific error diagnosis
- Optimisation and error diagnosis are also costly in a non-embedded setting, but there we have more control.
- Can we achieve this control for error diagnosis?

Challenges for EDSLs

- How to achieve:
  - domain specific optimisations
  - domain specific error diagnosis
- Optimisation and error diagnosis are also costly in a non-embedded setting, but there we have more control.
- Can we achieve this control for error diagnosis?
- Yes, says work with Bastiaan Heeren and Alejandro Serrano Mena
- But which of these ideas can we easily build into GHC?

III. Customizing type error diagnosis in GHC

Over to GHC...


```
instance TypeError (Text "Cannot 'Show' functions." :$$:
                    Text "Perhaps a missing argument?")
      => Show (a -> b) where ...
```

- Leverages type-level programming techniques in GHC (Diatchki, 2015)
- Very restricted:
  - Only available for type class and family resolution
  - May not influence the ordering of constraints
  - Messages cannot depend on who generated the constraint

How far can we take this?

We provide

- control over the content of the type error message
  - the same constraint may result in different messages
- (some) control over the order in which constraints are checked
- GHC's abstraction facilities allow for reuse and uniformity
  - A type level embedded DSL for diagnosing embedded DSLs
- integrated as a patch in GHC version 8.1.20161202 (and 8.3.some)
- soundness and completeness for free!

How far can we take this?

We provide

- control over the content of the type error message
  - the same constraint may result in different messages
- (some) control over the order in which constraints are checked
- GHC's abstraction facilities allow for reuse and uniformity
  - A type level embedded DSL for diagnosing embedded DSLs
- integrated as a patch in GHC version 8.1.20161202 (and 8.3.some)
- soundness and completeness for free!
- Expression level error messages by type level programming

How much effort is involved (on our part)?

- We get a lot for a few non-invasive changes to GHC, with *TypeError* and the *Constraint* kind as enablers
- Constraint resolution needs some changes to track messages, and deal with priorities
- A few additions to *TypeLits.hs* in the base library and a new module *TypeErrors.hs* (62 lines) that exposes the API
- One additional compiler pragma `CHECK_ARGS_BEFORE_FN`.
- We employ many language extensions:

  `DataKinds`, `TypeOperators`, `TypeFamilies`, `ConstraintKinds`, `FlexibleContexts`, `PolyKinds`, `UndecidableInstances`, `UndecidableSuperclasses`

  but the EDSL programmer only the first four, the EDSL user none. (Since 8.3 sometimes also *AllowAmbiguousTypes*)

A great mistake

$$intid :: Int$$
$$intid = id' \; True$$

A great mistake

$$\mathit{intid} :: \mathit{Int}$$
$$\mathit{intid} = \mathit{id}' \; \mathit{True}$$

```
FormatEx.hs:17:9: error:
    * Hi! Please read this error message. It's a
      great error message.
      The argument and result types of 'id' do not
      coincide: Bool vs. Int
    * In the expression: id' True
      In an equation for 'intid': intid = id' True
```

Our wrapped, highly tailored identity function §III

```
id′ :: CustomErrors
   '[ '[a :≁: b
          :⇛: E.Text "Hi! Please read this error message."
          :◇: E.Text " It's a great error message."
          :$$:
          E.Text "The argument and result types of 'id'"
          :◇: E.Text " do not coincide: " :◇: VS a b]
    ] => a -> b
id′ = id
```

- *id′* is a type error aware wrapper for *id*

Our wrapped, highly tailored identity function

```
id' :: CustomErrors
   '[ '[a :≁: b
         :⇛: E.Text "Hi! Please read this error message."
         :◇: E.Text " It's a great error message."
         :$$:
        E.Text "The argument and result types of 'id'"
         :◇: E.Text " do not coincide: " :◇: VS a b]
    ] => a -> b
id' = id
```

- *id'* is a type error aware wrapper for *id*
- *E* qualifier to employ type level *Text*

Our wrapped, highly tailored identity function


```
id′ :: CustomErrors
  '[ '[a :≁: b
        :⇛: E.Text "Hi! Please read this error message."
        :◇: E.Text " It's a great error message."
        :$$:
        E.Text "The argument and result types of 'id'"
        :◇: E.Text " do not coincide: " :◇: VS a b]
   ] => a -> b
id′ = id
```

- id' is a type error aware wrapper for id
- E qualifier to employ type level *Text*
- $id' = id$ ensures id' is sound; can do completeness

Our wrapped, highly tailored identity function §III

id' :: $CustomErrors$
  '['[a :≁: b
      :⇛: $E.Text$ "Hi! Please read this error message."
      :◇: $E.Text$ " It's a great error message."
      :$$:
      $E.Text$ "The argument and result types of 'id'"
      :◇: $E.Text$ " do not coincide: " :◇: $VS\ a\ b$]
  ] => a -> b
$id' = id$

- id' is a type error aware wrapper for id
- E qualifier to employ type level $Text$
- $id' = id$ ensures id' is sound; can do completeness
- VS is a reusable type level function

Our wrapped, highly tailored identity function §III

```
id' :: CustomErrors
  '[ '[a :≁: b
        :⇒: E.Text "Hi! Please read this error message."
        :◇: E.Text " It's a great error message."
        :$$:
       E.Text "The argument and result types of 'id'"
        :◇: E.Text " do not coincide: " :◇: VS a b]
   ] => a -> b
id' = id
```

- *id′* is a type error aware wrapper for *id*
- *E* qualifier to employ type level *Text*
- *id′* = *id* ensures *id′* is sound; can do completeness
- *VS* is a reusable type level function
- With `{#- INLINE id' -#}` no run-time overhead

Before we go on: *Constraints*

GHC supports a special kind *Constraint* so that type level programming can be applied to constraints

type *JSONSerializable a* = (*FromJSON a*, *ToJSON a*)

and use type families as type-level functions:

type *family All* (*c* :: *k* -> *Constraint*) (*xs* :: [*k*]) **where**
All c [] = ()
All c (*x* : *xs*) = (*c x*, *All c xs*)

so we can write *All Show* [*Int*, *Bool*] instead of (*Show Int*, *Show Bool*)

This is what opens the door to manipulating constraints and type error messages in a reusable fashion.

The running example

$$
\begin{aligned}
atop \;::\; & (OrderedField\ n, Metric\ v, Semigroup\ m) \\
\Rightarrow\; & QDiagram\ b\ v\ n\ m \rightarrow \\
& QDiagram\ b\ v\ n\ m \rightarrow \\
& QDiagram\ b\ v\ n\ m
\end{aligned}
$$

can also be written as

$$
\begin{aligned}
atop \;::\; & (d_1 \sim QDiagram\ b_1\ v_1\ n_1\ m_1, \\
& \ d_2 \sim QDiagram\ b_2\ v_2\ n_2\ m_2, \\
& \ b_1 \sim b_2, v_1 \sim v_2, n_1 \sim n_2, m_1 \sim m_2, \\
& \ OrderedField\ n_1, Metric\ v_1, Semigroup\ m_1) \\
\Rightarrow\; & d_1 \rightarrow d_2 \rightarrow d_1
\end{aligned}
$$

Failure to satisfy either $b_1 \sim b_2$ or $v_1 \sim v_2$ should lead to different messages.

The EDSL-developer facing API (version 1) §III

Apartness (= can never become equal again) is represented by the operator

infixl 5 $:\not\sim:$

We deal with two kinds of failure:

$$\textbf{data}\ \mathit{ConstraintFailure} = \forall\ t\ .\ t :\not\sim: t \mid \mathit{Undischarged}\ \mathit{Constraint}$$

A *CustomError* is then a failure and a message

$$\textbf{infixl}\ 4 :\Rrightarrow:$$
$$\textbf{data}\ \mathit{CustomError} = \mathit{ConstraintFailure} :\Rrightarrow: \mathit{ErrorMessage} \mid \mathit{Check}\ \mathit{Constraint}$$

The latter if we do not want a message.

Running back to our example

$atop :: CustomErrors$ [

$\quad d_1 :\not\sim: QDiagram\ b_1\ v_1\ n_1\ m_1$

$\qquad :\Rrightarrow: Text$ `"Arg. #1 to 'atop' must be a diagram"`,

$\quad d_2 :\not\sim: QDiagram\ b_2\ v_2\ n_2\ m_2$

$\qquad :\Rrightarrow: Text$ `"Arg. #2 to 'atop' must be a diagram"`,

$\quad b_1 :\not\sim: b_2$

$\qquad :\Rrightarrow: Text$ `"Back-ends do not coincide"`,

$\quad \ldots$

$\quad Check\ (OrderedField\ n_1), Check\ (Metric\ v_1),$

$\quad Check\ (Semigroup\ m_1)$

$\quad] \Rightarrow d_1 \rightarrow d_2 \rightarrow d_1$

CustomErrors is a type family that builds the constraint structure. To the programmer, a syntactic wrapper around his/her diagnosis.

Abstraction and reuse

For consistency and conciseness we can define a type level implementation for the checks of back-ends, vector spaces, etc.

$$\textbf{type}\ \mathit{DoNotCoincide}\ \mathit{what}\ a\ b = \\ \quad a :\not\sim: b :\Rrightarrow: \mathit{Text}\ \mathit{what} :\diamond: \mathit{Text}\ \texttt{" do not coincide: "} \\ \qquad :\diamond: \mathit{ShowType}\ a :\diamond: \mathit{Text}\ \texttt{" vs. "} :\diamond: \mathit{ShowType}\ b$$

Note that *ShowType* and type level *Text*s are provided by GHC.

The EDSL-developer facing API (version 2)

Some constraints can be checked independently: partition constraints into a list of lists.

```
atop :: CustomErrors [
    [d1 :≁: QDiagram b1 v1 n1 m1
          :⇛: Text "Arg. #1 to 'atop' must be a diagram",
     d2 :≁: QDiagram b2 v2 n2 m2
          :⇛: Text "Arg. #2 to 'atop' must be a diagram"],
    [DoNotCoincide "Back-ends"           b1 b2,
     DoNotCoincide "Vector spaces"       v1 v2,
     DoNotCoincide "Numerical fields"    n1 n2,
     DoNotCoincide "Query annotations"   m1 m2],
    [Check (OrderedField n1), Check (Metric v1),
     Check (Semigroup m1)]
  ] => d1 -> d2 -> d1
```

Alternatives and conversions

- *diagrams* distinguishes vectors from points
- You can compute the perpendicular of a vector (but not a point (pair)) with *perp*
- Can we provide a hint on how to convert a pair to a vector if the argument happens to be a pair, like

```
* Expecting a 2D vector but got a tuple.
  Use 'r2' to turn the tuple into a vector.
```

Although the fix may not be what the programmer intends, it will resolve the type error.

```
perp :: CustomErrors [
  [v :≁: V2 a :⇛?:
    ([v ∼ (a, a) :⇛!:
      Text "Expecting a 2D vector but got a tuple."
      :$$: Text "Use r2 to turn a tuple into a vector."
      ],
      Text "Expected a 2D vector, but got "
      :◇: ShowType v)],
  [Check (Num a)]] => v -> v
```

With every apartness check we can associate a list of further checks on what in this case *v* might actually be.

Restricting the Monad


```
(>=>) :: CustomErrors
  [
    [m :≁: IO :⇛: E.Text "Illegal use of monads: ... "
                  :◇: ShowType m
                  :◇: E.Text " monad"
                  :$$:
                  E.Text "...to pass your monad-license"
    ]
  ] => (a -> m b) -> (b -> m c) -> a -> m c
(>=>) = (M. >=>)
```

Time to Format

Sorry, but I have to skip the demo.

Time to Format

Sorry, but I have to skip the demo.
But we can look at some code.

What we did on our holidays

- We have worked out some rules for
  - *path*
  - *diagrams*
  - *persistent*
  - *map*, *Eq*, and making *foldr* and *foldl* siblings
  - *formatting*
  - Students are working on *uulib*, *copilot* and a few more

Recapping our slogan

Expression level type error messages
by
type level programming

Thank you for your attention

Questions you could have asked

- Does this work with type classes?

Questions you could have asked

- Does this work with type classes?
- Can we specialize per instance?

Questions you could have asked

- Does this work with type classes?
- Can we specialize per instance?

Questions you could have asked

- Does this work with type classes?
- Can we specialize per instance?
- Can you apply your work to your error diagnosis EDSL?

Questions you could have asked

- Does this work with type classes?
- Can we specialize per instance?
- Can you apply your work to your error diagnosis EDSL?
- What do I see in ghci when I ask for the type of *now*?

Questions you could have asked

- Does this work with type classes?
- Can we specialize per instance?
- Can you apply your work to your error diagnosis EDSL?
- What do I see in ghci when I ask for the type of *now*?
- And what about Haddock?

Questions you could have asked

- Does this work with type classes?
- Can we specialize per instance?
- Can you apply your work to your error diagnosis EDSL?
- What do I see in ghci when I ask for the type of *now*?
- And what about Haddock?
- Can I help?

Questions you could have asked

- Does this work with type classes?
- Can we specialize per instance?
- Can you apply your work to your error diagnosis EDSL?
- What do I see in ghci when I ask for the type of *now*?
- And what about Haddock?
- Can I help? Mail me J.Hage@uu.nl