Universiteit Utrecht

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

Customizing type error diagnosis in the Glorious Glasgow Haskell Compilation System (as of version 8.1.x)

Jurriaan Hage (collaboration with Alejandro Serrano Mena)

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

August 28, 2017

Introduction and Motivation

About me

- Assistant professor in Utrecht, teaching compiler construction and formal languages (all use Haskell)
- Programming FP (first Scheme, then Haskell) since 1991
  - and some non-FP languages of course
- Ran a large FP bachelor course in 2013-2016
- Maintainer of the Haskell Helium compiler
- The Utrecht compiler technology master track uses FP/Haskell throughout

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

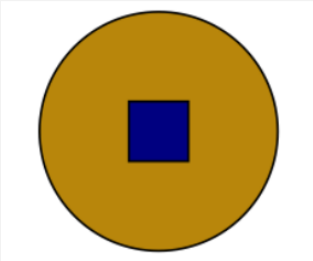
- Languages likes Java and Haskell grow and grow
- Languages like Scala embrace multiple paradigms
- Martin Odersky's "type wall": unless complicated type system features are balanced by better diagnosis, programmers will flock to dynamic languages
- From the other direction: dynamic languages are becoming more static (Hack, Typescript)

Example 1: domain-specific terms in Diagrams

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

```
atop :: (OrderedField n, Metric v, Semigroup m)
     => QDiagram b v n m ->
        QDiagram b v n m ->
        QDiagram b v n m
```

```
square 0.5 # fc navy
  `atop`
circle 1 # fc darkgoldenrod
```



Example 1: domain-specific terms in Diagrams

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

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

But writing *atop True* gives

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

Example 1: domain-specific terms in Diagrams

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

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

Writing *atop cube3d plane2d* might give

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

Instead, we would like to see domain terms, telling us about non-matching 'vector spaces' in the messages.

Example 2: missing instances 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: big consequences

$$
\begin{aligned}
pExpr = {} & pAndPrioExpr \\
<|> {} & sem\_Expr\_Lam \quad \text{-- Semantics for lambda expressions} \\
& \langle\$\ pKey\ \texttt{"\textbackslash\textbackslash"} \\
& \langle * \rangle pFoldr1\ (sem\_LamIds\_Cons, sem\_LamIds\_Nil)\ pVarid \\
& \langle * \rangle pKey\ \texttt{"->"} \\
& \langle * \rangle pExpr
\end{aligned}
$$

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 4: big consequences

$$
\begin{aligned}
pExpr = {} & pAndPrioExpr \\
<|> {} & sem\_Expr\_Lam \quad \text{-- Semantics for lambda expressions} \\
& \langle\$\ pKey\ \texttt{"\textbackslash\textbackslash"} \\
& \langle * \rangle pFoldr1\ (sem\_LamIds\_Cons, sem\_LamIds\_Nil)\ pVarid \\
& \langle *\ pKey\ \texttt{"->"} \\
& \langle * \rangle pExpr
\end{aligned}
$$

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) 'a'

notokay :: Maybe Char
notokay = (return >=> (\ x -> Nothing)) '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
- As we say: "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 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
- But how can we teach the Haskell compiler to communicate with the programmer in domain terms?

III. Customizing type error diagnosis in GHC

A great mistake

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

A great mistake

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

```
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
```

- What is code and what is type?
- :◇: and :$$: simply put error messages (type level texts) together (next to or on top of eachother)
- a :⊀: b means: if we know that input and output of id' can never become equal, give the message after the :⇛:

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*, and it's just Haskell

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
```

- E qualifier to employ type level *Text*

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
```

- $id' = id$ 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
```

- *VS* is a reusable type level function

A bit more on apartness

Examples of types that are not apart:

- *Int* and *Int*
- *a* and *Int*, since *a* can still become an *Int*
- *a* -> *b* and *Bool* -> [*a*]

These are:

- *Int* and *Bool*
- [*a*] and *b* -> *b*

Apartness implies we have a type error, non-apartness by itself does not give information.

Apartness is simply a check, and does not imply a unification/inference step

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

Apartness ($=$ can never become equal again) is represented by the operator $:\not\sim:$

We can deal with two kinds of failure:

data *ConstraintFailure* $=$
$\forall\ t\ .\ t :\not\sim: t \mid$ *Undischarged Constraint*

A *CustomError* is then a failure and a message

data *CustomError* $=$
ConstraintFailure $:\Rrightarrow:$ *ErrorMessage* $\mid$ *Check Constraint*

The latter if we want the default message.

Running back to our example

$$
\begin{aligned}
&\mathit{atop} :: \mathit{CustomErrors}\ [\\
&\quad d_1 :\not\sim: \mathit{QDiagram}\ b_1\ v_1\ n_1\ m_1\\
&\qquad :\Rrightarrow: \mathit{Text}\ \texttt{"Arg. \#1 to 'atop' must be a diagram"},\\
&\quad d_2 :\not\sim: \mathit{QDiagram}\ b_2\ v_2\ n_2\ m_2\\
&\qquad :\Rrightarrow: \mathit{Text}\ \texttt{"Arg. \#2 to 'atop' must be a diagram"},\\
&\quad b_1 :\not\sim: b_2\\
&\qquad :\Rrightarrow: \mathit{Text}\ \texttt{"Back-ends do not coincide"},\\
&\quad \ldots\\
&\quad \mathit{Check}\ (\mathit{OrderedField}\ n_1), \mathit{Check}\ (\mathit{Metric}\ v_1),\\
&\quad \mathit{Check}\ (\mathit{Semigroup}\ m_1)\\
&\quad] \Rightarrow d_1 \rightarrow d_2 \rightarrow d_1
\end{aligned}
$$

Abstraction and reuse

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

type *DoNotCoincide what a b* =
  a :$\not\sim$: *b* :$\Rrightarrow$: *Text what* :$\diamond$: *Text* `" do not coincide: "`
    :$\diamond$: *ShowType a* :$\diamond$: *Text* `" vs. "` :$\diamond$: *ShowType b*

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

The EDSL-developer facing API (version 2) §III

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?

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

It may not be what the programmer intends, but the change will resolve the type error.

The EDSL-developer facing API (version 3) §III

```
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.

A Very Short Time to Format

Go there, now!

How did we achieve this?

- Piggybacking on the *TypeError* class and the *Constraint* kind in GHC
- Constraint resolution needs some changes to track messages and deal with priorities
- A few changes 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 GHC 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 *AllowAmbiguousTypes*)

What else?

§III

- We have worked out some rules for
  - *path*, *diagrams*, *persistent*, *formatting*
  - *map*, *Eq*, *foldr* and *foldl*
  - Students have worked on a few more
- I omitted
  - siblings: suggest a similar function that has a type that fits
  - the monad license example
- Alejandro is presenting this at IFL in Bristol this week

I'll leave you with our slogan

Expression level type error messages
by
type level programming

Wanna play?
Contact us at J.Hage@uu.nl or A.SerranoMena@uu.nl

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. >=>)
```

Questions you could have asked

§III

- 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