Universiteit Utrecht

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

The Usability of Static Type Systems

Jurriaan Hage

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

June 22, 2020

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
- Which properties it enforces, depends intimately on the language
  - Cf. does every function have the right number of arguments in C vs. Haskell

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

Example: one missing character

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

GPL's follow Lehmann's 6th law

- Java has seen the introduction of parametric polymorphism (and type errors suffered)
- Java has seen the introduction of anonymous functions
- 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
- The type system of Haskell is growing towards a dependently typed system, making it more powerful, but also harder to use

What's more

- Even Sun did not implement Java 1.5 faithfully
  - And neither did jikes

What's more

- Even Sun did not implement Java 1.5 faithfully
  - And neither did jikes
- Is Scala a small language?

What's more

- Even Sun did not implement Java 1.5 faithfully
  - And neither did jikes
- Is Scala a small language?
- How many Haskells can our planet support?

What's more

- Even Sun did not implement Java 1.5 faithfully
  - And neither did jikes
- Is Scala a small language?
- How many Haskells can our planet support?

Embedded Domain Specific Languages

- Embedded (internal à la Fowler) Domain Specific Languages are achieved by encoding domain-specific 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

A major challenge for EDSLs

- Achieving domain specific error diagnosis
- An implementation of the DSL should communicate with the programmer about the program in terms of the domain
  - domain-abstractions should not leak
- Error diagnosis is also necessary in an external setting, but there we have more control.
- Can we achieve this control for error diagnosis?

Philosophically...

- Aristotle is to have said “teaching is the highest form of understanding”
- To me type error diagnosis is all about explaining the type system to programmers
- If we can do that well, then we can say we really understand the type system. Just getting something to work is not enough.

Serendipity at work

- During the DOMSTED project we developed a new approach to deal with higher-rank types and impredicativity (PLDI '18/ICFP '20) that
  - has a declarative specification
  - only breaks code that can be easily fixed
  - is not broken
  - integrates well with all existing GHC type system extensions
- All from a motivation to be able to explain type inconsistencies for higher-rank types
- Although we never got round to dealing with the error diagnosis
- ICFP '20 is a variant that is also non-invasive to implement in GHC

Example #1: Siblings heuristic in Helium

- Suggest type correct replacements for functions are operators that are easily confused,
  - (:) and (++), and and *foldl* and *foldr* for novice Haskellers,
  - (.) and (++) for newcomers from PHP
  - (+) and (++) for newcomers from Java
  - $\langle *$ and $\langle * \rangle$ for people new to Applicatives
- Helium compiler takes a (editable) list of sibling pairs

Siblings in action

$$\textbf{data}\ Expr = Lambda\ [String]\ Expr$$

$$\begin{aligned} &pExpr \\ &\quad = pAndPrioExpr \\ &\;<|>\ Lambda \langle\$\ pKey\ \texttt{"\textbackslash\textbackslash"} \\ &\qquad\qquad\quad \langle*\rangle many\ pVarid \\ &\qquad\qquad\quad \langle*\ pKey\ \texttt{"->"} \\ &\qquad\qquad\quad \langle*\ pExpr \end{aligned}$$

Extremely concise:

```
(11,13): Type error in the operator <*
  probable fix: use <*> instead
```

Example #2: Specialized type rules in Helium

Control error diagnosis and solving order for programmer-definable classes of expressions.

```
  x :: t1;   y :: t2;
---------------------
    x <$> y :: t3;
...
t2 == Parser s1 a2 :
 @expr.pos@: The right operand of <$> should be a
  expression        : @expr.pp@             parser
  right operand     : @y.pp@
    type            : @t2@
    does not match  : Parser @s1@ @a2@
...
```

Example

test :: *Parser Char String*
test = *map toUpper* ⟨\$⟩ `"hello, world!"`

This results in the following type error message (including the inserted error message attributes):

```
(2,21): The right operand of <$> should be a parser
 expression          : map toUpper <$> "hello, world!"
 right operand       : "hello, world!"
   type              : String
   does not match    : Parser Char String
```

Example #3: Destructive updates (almost in Helium)

- Haskell is lazy, and therefore can have only controlled side-effects
- However, a type system can be defined that allows destructive updates (PEPM 2008)

$$\begin{aligned} &append\ [\,]\ && ys = ys \\ &append\ p@(x : xs)\ && ys = p@(x : append\ xs\ ys) \end{aligned}$$

- *p@* in rhs reuses pattern-matched cons-cell
- Only valid if the first argument to *append* is unique: nobody else can have access to the same list.
- Uniqueness type system verifies this for a given call to *append* and defaults to *append* without re-use if that is not the case.

Example #4: Type error heuristics for GADTs in Helium

- Popular type system extension in Haskell
- Includes also existential types
- Built upon OutsideIn(X), implemented in Helium alongside basic Haskell 98 solver

Example #5: diagnosis with type level programming in GHC

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

Example #5: diagnosis with type level programming in GHC

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

```
FormatEx.hs:17:9: error:
    * Dear Mr. Kernighan.
      In this programming language we distinguish
      between booleans and integers.
      Please ask your TA Bjarne for more details.

      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
```

Research challenges in the short term

Address type error diagnosis in the Helium/GHC compiler for, a.o.,

- higher-rank and impredicative types
- type class extensions (type class = ad-hoc overloading)
- type families and related issues
- And combinations of these
- Construct realistic benchmarks
  - Viscious circle in the making, ML

Research challenges in the longer term

- Type error diagnosis for subtyping based languages (Scala, Java)
- Type error diagnosis and proof assistance for dependently typed languages
  - Both harder and simpler
- Type error diagnosis for gradually typed languages
  - Both harder and simpler
- Optimisation assistance for statically typed languages

Research challenges in the longer term

- Type error diagnosis for subtyping based languages (Scala, Java)
- Type error diagnosis and proof assistance for dependently typed languages
  - Both harder and simpler
- Type error diagnosis for gradually typed languages
  - Both harder and simpler
- Optimisation assistance for statically typed languages
- Type system generator that includes error diagnosis in one unified framework