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

September 3, 2018

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

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

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])]
```

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
- 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

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
- 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

A challenge for EDSLs

- How to achieve: 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?

A challenge for EDSLs

- How to achieve: 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?
- Quite a bit says work with Bastiaan Heeren and Alejandro Serrano' Mena
- But do these scale to an industrial strength compiler like GHC?

Work with Bastiaan Heeren ($<=$ 2006)

- Constraint-based type inferencing
- Programmable diagnosis for classes of expressions
- $<$ Haskell 98
- Implemented in Helium

Work with Alejandro Serrano Mena (2013-2017)

- Based on OutsideIn (GHC's constraint solver)
- More control over diagnosis (ESOP 2016)
- Making OutsideIn higher-ranked (PLDI 2018)
  - Why is that important?
- Building some of ESOP 2016 into an industrial strength compiler (IFL 2017)

Future work (I)

- Type error diagnosis for advanced Haskell type system features (GADTs, Type Families, type classes).
  - Driven by constraints, based on heuristics
- Type error diagnosis for Scala (Java,...)
  - Challenge: the integration of subtyping with type inferencing and parametric polymorphism is tricky. Languages are large.
- Type error diagnosis and proof assistance for the dependently typed languages Agda and Idris

But what about optimising analyses?

- Functional language compilers have to be aggressive optimizers
- In the FP world, optimising analyses are phrased as a non-standard type system
- Reuse of vocabulary, concepts and implementation suggests that we may be able to reuse facilities for diagnosis
- How would that work?

Strictness analysis

$fs\ x = x + 2$

$fl\ y = 3$

- Both functions can have type $Int \rightarrow Int$, but there is a difference too:
  - *fs* uses its single argument (it is strict): *fs forever* will not terminate
  - *fl* does not (it is lazy): *fl forever* returns 3
- In Haskell, lazy is the default
- For performance, strict evaluation is to be preferred, but only if we are sure that an argument will in fact be used
  - *fl forever* will not terminate if we evaluate *fl*s argument *forever* before the call
- Strictness analysis decides for every function which of its arguments can be evaluated eagerly before the call.

Some problems that arise in this setting

- Strictness information is stored as part of the generated object code, but is not human-readable
  - Solution: provide strictness signatures (cf. Clean's uniqueness types)
- Sometimes programmers know better (every analysis has limited precision), and want to override the analysis information
  - Propagating the override information can be semantically tricky (PEPM 10)
- Analyses can "help" each other: sharing analysis and absence analysis can be combined to yield better results
- Note: strictness is just one example

Future Work (II)

- In the setting of Haskell
  - Transparency of optimisation information
  - Exploiting programmer knowledge, and providing feedback about programmer suggestions
  - Exploit interdependencies between analyses to achieve better results
  - Can we then implement domain-specific optimisations on top of a range of such built-in analyses?
- Note: this research can take place in other technological settings as well

Thank you for your attention