Universiteit Utrecht

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

# Making Domain Specific Languages a Practical Reality

Jurriaan Hage

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

November 14, 2015

# The purpose of this talk

- Introduce the field of type error diagnosis
- Discuss one way of addressing the problem of type error diagnosis:
- allow the type inference process to be modified from outside the compiler
- Apply this idea to achieve domain specific type error diagnosis for EDSLs in Haskell

# The purpose of this talk

- Introduce the field of type error diagnosis
- Discuss one way of addressing the problem of type error diagnosis:
- allow the type inference process to be modified from outside the compiler
- Apply this idea to achieve domain specific type error diagnosis for EDSLs in Haskell
- But mostly: to give a glimpse of what could be yours to have, in Haskell and beyond.

# 1. Introduction

# From static guarantees to error diagnosis



- 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"

# From static guarantees to error diagnosis



- 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"
  For the right definition of wrong
- But

# From static guarantees to error diagnosis



- 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"
  For the right definition of wrong
- But if a compiler or interpreter refuses to run my program, I deserve an explanation

# From static guarantees to error diagnosis



- 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"
  For the right definition of wrong
- But if a compiler or interpreter refuses to run my program, I deserve an explanation
- Hence, the need for type error 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 include the following:
  - that a program is type incorrect (duh)
  - what kind of inconsistency was detected
  - which parts of the program contributed to the inconsistency
  - how the inconsistency may be fixed
- Functional languages have had more problems with diagnosis $\Rightarrow$ quite some research into type error diagnosis
- There is surprisingly little for O.O. languages

# Lehman's sixth law applies also to languages §1

- Java has seen the introduction of parametric polymorphism...
- ...and anonymous functions in 1.8
- What did this do to the quality of error diagnosis?

# Lehman's sixth law applies also to languages §1

- Java has seen the introduction of parametric polymorphism...
- ...and anonymous functions in 1.8
- What did this do to the quality of error diagnosis?
- Languages like Scala embrace multiple paradigms
- Odersky's "type wall": unless complicated type system features are balanced by better diagnosis, programmers will flock to dynamic languages
- But: dynamic languages grow up, and become more static
- Also here, diagnosis will rear its (time-consuming) head

# Some Haskell



$$reverse \quad xs = foldr\ (flip\ (:))\ [\,]\ xs$$
$$palindrome\ xs = reverse\ xs == xs$$

Is this program well typed?

# Some Haskell



$$reverse \quad xs = foldr\ (flip\ (:))\ [\,]\ xs$$

$$palindrome\ xs = reverse\ xs == xs$$

Is this program well typed?

```
Occurs check: cannot construct the infinite type: t ~ [[t]]
  Expected type: [t]
  Actual type: [[[t]]]
In the second argument of '(==)', namely 'xs'
In the expression: reverse xs == xs
```

# What is wrong with this message?



```
Occurs check: cannot construct the infinite type: t ~ [[t]]
  Expected type: [t]
  Actual type: [[[t]]]
In the second argument of '(==)', namely 'xs'
In the expression: reverse xs == xs
```

# What is wrong with this message?



```
Occurs check: cannot construct the infinite type: t ~ [[t]]
  Expected type: [t]
  Actual type: [[[t]]]
In the second argument of '(==)', namely 'xs'
In the expression: reverse xs == xs
```

- It does not point to the source of the error → not precise
- It's intimidating
- It shows an artifact of the implementation → mechanical
  - "Occurs check" is part of the unification algorithm
- Generally, message not very helpful
- The likely fix:

# What is wrong with this message?



```
Occurs check: cannot construct the infinite type: t ~ [[t]]
  Expected type: [t]
  Actual type: [[[t]]]
In the second argument of '(==)', namely 'xs'
In the expression: reverse xs == xs
```

- It does not point to the source of the error → not precise
- It's intimidating
- It shows an artifact of the implementation → mechanical
  - "Occurs check" is part of the unification algorithm
- Generally, message not very helpful
- The likely fix: $foldr$ should be $foldl$

# Good Error Reporting Manifesto



From Improved Type Error Reporting by Yang, Trinder and Wells

1. Correct detection and correct reporting
2. Precise: the smallest possible location
3. Succint: maximize useful and minimize non-useful info
4. Does not depend on implementation, i.e., amechanical
5. Source-based: not based on internal syntax
6. Unbiased: fixed unification order leads to bias
   $head\,[False, \texttt{'1'}, \texttt{'2'}, \texttt{'3'}, \texttt{'4'}, \texttt{'5'}, \texttt{'6'}, \texttt{'7'}]$
7. Comprehensive: enough to reason about the error

Worth striving for.

# But in the end...



- Observed quality of type error diagnosis is extremely subjective
- Depending on the programmer's intentions that a compiler cannot guess at
- And many other factors, including programmer experience, and background
- So is type error diagnosis a doomed enterprise?
- I do not think so, but there is a lot to learn, and be careful about.

# 2. Domain Specific Type Error Diagnosis in 2003

# The Helium (for Haskell) compiler



- Constraint based approach to type inferencing
- Implements many heuristics, multiple solvers
- Existing algorithms/implementations can be emulated
- ```
  cabal install helium
  cabal install lvmrun
  ```
- Haskell 98 minus type class and instance *definitions*
- Supports domain specific type error diagnosis


  ```

# What is a Domain Specific Language (DSL)? §2

- 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 Domain Specific Language (DSL)? §2

- 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:
  - domain-abstractions should not leak, e.g., in error reports

# 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.
- According to Paul Hudak "the ultimate abstraction"
- 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

# Challenges for EDSLs (in Haskell)



- 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?

# A (much simplified) EDSL for parsing



- An executable and extensible form of EBNF
  - A context free grammar production like

    `C -> DE | a | b`

    is turned into executable code:

    $$c = d \mathbin{<\!\!*\!\!>} e \mathbin{<\!|\!>} token\ \texttt{"a"} \mathbin{<\!|\!>} token\ \texttt{"b"}$$

  - Non-terminals `C`,`D`,`E` are turned into Haskell functions $c, d, e$ that do the parsing
  - We add semantics by applying a functions to the outcome(s) of the parsing:

    $$c = f \mathbin{<\!\$\!>} d \mathbin{<\!\!*\!\!>} e \ldots.$$

  - FYI, $token$, $<\!\!*\!\!>$, $<\!\$\!>$ and $<\!|\!>$ are called combinators

# A small mistake



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

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])]
```

# A closer look at the message



```
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])]
```

- Message is large and looks complicated
- You have to discover why the types don't match yourself
- No mention of "parsers" in the error message
- It happens to be a common mistake, and easy to fix

# The solution in a nutshell



1. Bring the type inference mechanism under control
   - by phrasing the type inference process as a constraint solving problem
2. Provide hooks in the compiler's type inference process to change the process for certain classes of expressions

- Changing the type system is forbidden!
  - Only the order of solving, and the provided messages can be changed

# How is this organised?



- For a given source module Abc.hs, a DSL designer may supply a file Abc.type containing directives
- The directives are automatically used when the module is imported
- The compiler will adapt the type error mechanism based on these type inference directives.

# A specialized type rule (1/3)



Consider one of the parser combinators, for instance <$>
(which today is not the generalized *Functor* field member!).

$$<\$> :: (a \to b) \to \mathit{Parser}\ s\ a \to \mathit{Parser}\ s\ b$$

We create a specialized type rule for expressions in which <$>
is given two argument expressions $x$ and $y$

$$\frac{\Gamma \vdash_{\mathrm{HM}} x : a \to b \qquad \Gamma \vdash_{\mathrm{HM}} y : \mathit{Parser}\ s\ a}{\Gamma \vdash_{\mathrm{HM}} x\ <\$>\ y : \mathit{Parser}\ s\ b}$$

# A specialized type rule (2/3) §2

- Rewrite rule to make constraints explicit

$$\frac{x : \tau_1 \qquad y : \tau_2}{x \ <\$> \ y : \tau_3}$$

$$\begin{cases} \tau_1 & \equiv & a \to b \\ \tau_2 & \equiv & \mathit{Parser}\ s\ a \\ \tau_3 & \equiv & \mathit{Parser}\ s\ b \end{cases}$$

# A specialized type rule (2/3) §2

- Rewrite rule to make constraints explicit

$$\frac{x : \tau_1 \qquad y : \tau_2}{x \ <\$> \ y : \tau_3} \qquad \begin{cases} \tau_1 & \equiv & a \to b \\ \tau_2 & \equiv & \mathit{Parser}\ s\ a \\ \tau_3 & \equiv & \mathit{Parser}\ s\ b \end{cases}$$

Split up the type constraints in "smaller" unification steps.

$$\frac{x : \tau_1 \qquad y : \tau_2}{x \ <\$> \ y : \tau_3} \qquad \begin{cases} \tau_1 & \equiv & a_1 \to b_1 & \qquad s_1 & \equiv & s_2 \\ \tau_2 & \equiv & \mathit{Parser}\ s_1\ a_2 & \qquad a_1 & \equiv & a_2 \\ \tau_3 & \equiv & \mathit{Parser}\ s_2\ b_2 & \qquad b_1 & \equiv & b_2 \end{cases}$$

# Specializing a type rule (3/3)



$$\frac{x : \tau_1 \qquad y : \tau_2}{x \ \texttt{<\$>}\ y : \tau_3}$$

$$\begin{cases} \tau_1 \equiv a_1 \to b_1 & s_1 \equiv s_2 \\ \tau_2 \equiv \mathit{Parser}\ s_1\ a_2 & a_1 \equiv a_2 \\ \tau_3 \equiv \mathit{Parser}\ s_2\ b_2 & b_1 \equiv b_2 \end{cases}$$

```
  x :: t1;   y :: t2;
--------------------
    x <$> y :: t3;

t1 == a1 -> b1
t2 == Parser s1 a2
t3 == Parser s2 b2
s1 == s2
a1 == a2
b1 == b2
```

# Special type error messages



```
  x :: t1;   y :: t2;
--------------------
    x <$> y :: t3;

t1 == a1 -> b1      : left operand is not a function
t2 == Parser s1 a2  : right operand is not a parser
t3 == Parser s2 b2  : result type is not a parser
s1 == s2 : parser has an incorrect symbol type
a1 == a2 : function cannot be applied to parser's result
b1 == b2 : parser has an incorrect result type
```

- Supply an error message for each type constraint. This message is reported if the corresponding constraint cannot be satisfied. Constraints checked top to bottom.

# Example



$$test :: Parser\ Char\ String$$
$$test = map\ toUpper\ \texttt{<\$>}\ \texttt{"hello, world!"}$$

This results in the following type error message:

```
Type error: right operand is not a parser
```

# Error message attributes



In the rule we can add important context-specific information:

```
  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
```

# Variations



```
   x :: t1;     y :: t2;
-----------------------------
  x <$> y :: Parser s b;

constraints x
t1 == a1 -> b      : left operand is not a function
constraints y
t2 == Parser s a2  : right operand is not a parser
a1 == a2 : function cannot be applied to ...
```

- Interpolate constraints into the rule (cf. $Parser\ s\ b$): no effort for default behaviour

# Other facilities



- Automatic check for soundness and completeness
- Phasing for more global control
- Siblings: suggest program fixes to replace $+$ by $++$ for incorrect Haskell expressions like $\mathbf{let}\ y = 2\ \mathbf{in}\ "" + y$
- Add your favourite siblings:

```
siblings   <$> , <$
siblings   ++ , +, .
```

# 3. DOMSTED in 2015

# The DOMSTED project



- Take the previous ideas and scale them to Haskell 2010
- Alejandro Serrano Mena started November 2013

# Isn't Haskell 98 complicated enough?



Maybe, but

| Extension | # Hackage | # Top 20 |
|---|---|---|
| FlexibleInstances | 332 | 10 |
| MultiParamTypeClasses | 321 | 9 |
| FlexibleContexts | 232 | 3 |
| ScopedTypeVariables | 192 | 3 |
| ExistentialQuantification | 149 | 6 |
| FunctionalDependencies | 139 | 4 |
| TypeFamilies | 114 | 1 |
| OverlappingInstances | 108 | 3 |
| Rank2Types | 100 | 3 |
| GADTs | 88 | 3 |
| RankNTypes | 81 | 1 |
| UnboxedTuples | 20 | 4 |
| KindSignatures | 20 | 0 |

# What have we accomplished



- Conditionals in type rules
- Regular tree expressions
- Implementation on top of `OutsideIn(X)`. Experiment at `http://cobalt.herokuapp.com/` with the latest syntax.

# Conditional matching in type rules



- Why does Haskell have $map$ and $fmap$?
  - Part tradition, part error diagnosis
- Alternative solution: provide only $fmap$, but tailor the error for $fmap$ for the case of lists
- IF we can decide before an error is discovered that we are working on lists, THEN a special rule kicks in

# Conditional matching in type rules



- Why does Haskell have $map$ and $fmap$?
  - Part tradition, part error diagnosis
- Alternative solution: provide only $fmap$, but tailor the error for $fmap$ for the case of lists
- IF we can decide before an error is discovered that we are working on lists, THEN a special rule kicks in
- Other applications:
  - Monad comprehensions vs. list comprehensions
  - Forbidding certain instances of generic constructions to novice students
  - And the list goes on...

# Regular tree expressions (RTE)



- A more general way to describe AST patterns
  - Beyond, for example, $x$ <\$> $y$ and $map\ id\ xs$
- RTEs add repetition and choice to describe our expression patterns
- Disadvantage: if you use RTEs soundness check is approximated by random testing à la QuickCheck
- Alejandro is currently making progress on fixing this.

# Example: arithmetic expressions with a GADT §3

- Consider (polymorphic) arithmetic expressions of type $Expr\ a$

  $$\begin{aligned}&\textbf{data}\ Expr\ a\ \textbf{where}\\ &\quad Plus :: Expr\ a \to Expr\ a \to Expr\ a\\ &\quad Mult :: Expr\ a \to Expr\ a \to Expr\ a\\ &\quad Lit :: Num\ a \Rightarrow a \to Expr\ a\end{aligned}$$

- Type of literals determines type of expressions: we can have $Float$ expressions and $Int$ expressions, but these may not be mixed in one expression
- $execute :: \forall\ a.Expr\ a \to a$ evaluates such an expression

# Without syntax



$$execute :: \forall\ a.Expr\ a \rightarrow a$$

We pattern match on expressions that apply $execute$ to some $Expr\ a$.

Then we have checks that

- all leafs are themselves correctly typed
- they all agree on the instance for $a$
- the instance is in the $Num$ class
- etc. in that order
- Each check has its specialized type error message.

# Conclusions



I have

- introduced the problem of type error diagnosis
- introduced specialized type rules to achieve domain-specific type error diagnosis
- informed you about some recent developments

# Ongoing and future work



- Light-weight approaches, i.e., directly in Haskell
- Integrating impredicativity and higher-ranked types into OutsideIn(X) as implemented into UHC/GHC
- Changes to syntax are ongoing
- One MSc student just started on adding this to Elm
- Another will probably shortly start on dependently typed languages (Agda, Idris, Coq).

# Ongoing and future work



- Light-weight approaches, i.e., directly in Haskell
- Integrating impredicativity and higher-ranked types into OutsideIn(X) as implemented into UHC/GHC
- Changes to syntax are ongoing
- One MSc student just started on adding this to Elm
- Another will probably shortly start on dependently typed languages (Agda, Idris, Coq).
- Interactively writing specialized type rules
- Experiment with library developers and users
- Spread the gospel (to other statically typed languages)?

# Credits



The following people have contributed to this talk:

- Alejandro Serrano Mena, current PhD student
- Bastiaan Heeren, PhD student between 2000-2004
- Patrick Bahr, visiting postdoc in 2014
- Atze Dijkstra, implementor of UHC
- Doaitse Swierstra, for initiating it in the first place
- Many people contributed to the Helium compiler (see the website)

# Questions, anyone?