Universiteit Utrecht

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

Domain Specific Type Error Diagnosis for Embedded Domain Specific Languages

Jurriaan Hage

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

April 19, 2012

About me

- PhD at University Leiden somewhere between 1995 and 2000 under Grzegorz Rozenberg on algorithms and combinatorics of graphs and groups (switching classes)
- Commercial educator at Leiden during 1999-2000
- Assistant professor with Doaitse Swierstra at Utrecht University since Nov 2000
- Topics of interest:
  - static analysis and software analysis
  - mostly functional languages
  - plagiarism detection
  - testing
  - SOA
  - type error feedback
- I try to frequent conferences like POPL, ICFP, PEPM

Embedded Domain Specific Languages

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

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 this presentation I work with the pure, lazy, higher-order, polymorphic functional language Haskell (www.haskell.org)
- 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 smattering of Haskell

- Fortunately, we do not need much of Haskell for this talk
- Lambda's, higher-order, operator slices(, infinite lists)

$$
\begin{aligned}
&work\ n = \mathbf{let} \\
&\qquad pl = map\ (\lambda z \to z + n)\ [1\,..] \\
&\quad \mathbf{in} \\
&\qquad \mathbf{if}\ (n \geqslant 0)\ \mathbf{then} \\
&\qquad\quad take\ n\ (map\ (*2)\ pl) \\
&\qquad \mathbf{else} \\
&\qquad\quad take\ 10\ pl \\
&main = \mathbf{do} \\
&\qquad putStrLn\ (show\ (work\ 40))
\end{aligned}
$$

Challenges for EDSLs

- How to achieve:
  - domain specific optimisations
  - domain specific error diagnosis
- Optimisations and error diagnosis also take up time in a non-embedded setting, but there we have more control.
- Can we attain this control for error diagnosis?

Our case study

- Parser combinators: an EDSL for describing parsers
- An executable and extensible form of EBNF
  - Concatenation/juxtaposition: p `<*>` q, and p `<*` q
  - Choice: p `<|>` q
  - Semantics: f `<$>` p and f `<$` p
  - Repetition: $many$, $many1$, ...
  - Optional: $option\ p\ \mathbf{default}$
  - Literals: $token$ `"text"`, $pKey$ `"->"`
  - Others introduced as needed, and defined at will

```
pExpr = pAndPrioExpr
  <|> sem_Expr_Lam   -- a function of two arguments
    <$  pKey "\\"
    <*> pFoldr1 (sem_LamIds_Cons, sem_LamIds_Nil) pVarid
    <*  pKey "->"
    <*> pExpr
```

My first mistake

$$
\begin{aligned}
pExpr = {} & pAndPrioExpr \\
<|> {} & sem\_Expr\_Lam \\
& <\$ \ pKey \ \texttt{"\textbackslash\textbackslash"} \\
& <*> \ pFoldr1 \ (sem\_LamIds\_Cons, sem\_LamIds\_Nil) \ pVarid \\
& <*> \ pKey \ \texttt{"->"} \\
& <*> \ pExpr
\end{aligned}
$$

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 problems

Type error messages typically suffer from the following problems.

1. **A fixed type inference process.** The order in which types are inferred strongly influences the reported error site, and there is no way to depart from it.
2. **The size of the mentioned types.** Irrelevant parts are shown, and type synonyms are not always preserved.
3. **The standard format of type error messages.** Domain specific terms are not used.
4. **No anticipation for common mistakes.** Error messages focus on the problem, and not on how to fix it.

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
   - specialize type error messages for a particular domain
   - control the order in which constraints are solved
   - drive heuristics that suggest fixes for often-made mistakes

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
   - specialize type error messages for a particular domain
   - control the order in which constraints are solved
   - drive heuristics that suggest fixes for often-made mistakes

- 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 Abs.type containing the 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.
- The directives themselves are also a(n external) DSL!

The type inference process

- We piggy-back ride on Haskell's underlying type system
- Type rules for functional languages are often phrased as a set of logical deduction rules
- Inference is then implemented by means of an AST traversal
  - Ad-hoc or using attribute grammars
- Type inference is beyond the scope of a short talk like this, but we can't escape it completely

The rule for type checking applications

$$\frac{\Gamma \vdash_{\mathrm{HM}} f : \tau_a \to \tau_r \qquad \Gamma \vdash_{\mathrm{HM}} e : \tau_a}{\Gamma \vdash_{\mathrm{HM}} f\ e : \tau_r}$$

- Γ is an environment, containing the types of identifiers defined elsewhere
- Rules for variables, anonymous functions and local definitions omitted
- Algorithm $\mathcal{W}$ is a (deterministic) implementation of these typing rules.

Specializing a type rule (1/3)

Applying the type rule for function application twice in succession results in the following:

$$\frac{\Gamma \vdash_{\mathrm{HM}} op : \tau_1 \to \tau_2 \to \tau_3 \qquad \Gamma \vdash_{\mathrm{HM}} x : \tau_1 \qquad \Gamma \vdash_{\mathrm{HM}} y : \tau_2}{\Gamma \vdash_{\mathrm{HM}} x \text{ `}op\text{` } y : \tau_3}$$

Specializing a type rule (1/3)

Applying the type rule for function application twice in succession results in the following:

$$\frac{\Gamma \vdash_{\mathrm{HM}} op : \tau_1 \to \tau_2 \to \tau_3 \qquad \Gamma \vdash_{\mathrm{HM}} x : \tau_1 \qquad \Gamma \vdash_{\mathrm{HM}} y : \tau_2}{\Gamma \vdash_{\mathrm{HM}} x \text{ `}op\text{` } y : \tau_3}$$

Consider one of the parser combinators, for instance $<\$>$.

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

We can now create a specialized type rule by filling in this type in the type rule

Specializing a type rule (1/3)

Applying the type rule for function application twice in succession results in the following:

$$\frac{\Gamma \vdash_{\text{HM}} op : \tau_1 \to \tau_2 \to \tau_3 \qquad \Gamma \vdash_{\text{HM}} x : \tau_1 \qquad \Gamma \vdash_{\text{HM}} y : \tau_2}{\Gamma \vdash_{\text{HM}} x \text{ `}op\text{` } y : \tau_3}$$

Consider one of the parser combinators, for instance $<\$>$.

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

We can now create a specialized type rule by filling in this type in the type rule (x and y stand for arbitrary expressions of the given type)

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

Specializing a type rule (2/3)

- Use equality constraints to make the restrictions that are imposed by the type rule explicit.
- Γ is unchanged, and therefore omitted from the rule
- Type rules are invalidated by shadowing, here, <$>.

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

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

Specializing a type rule (2/3)

- Use equality constraints to make the restrictions that are imposed by the type rule explicit.
- Γ is unchanged, and therefore omitted from the rule
- Type rules are invalidated by shadowing, here, <$>.

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

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.

Example

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

This results in the following type error message:

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

Example

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

This results in the following type error message:

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

Important context specific information is missing, for instance:

- Inferred types for (sub-)expressions, and intermediate type variables
- Pretty printed expressions from the program
- Position and range information

Error message attributes

The error message attached to a type constraint might now look like:

```
  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
```

Implicit constraints

A type constraint can be "moved" from the constraint set to the deduction rule.

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

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

An implicit constraint with a default error message is inserted for the type in the conclusion.

Order of the type constraints

Each meta-variable represents a subtree for which also type constraints are collected. This constraint set can be explicitly mentioned in the type rule.

```
   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 parser's
                                              result
```

Soundness and completeness

The soundness of a specialized type rule with respect to the default type rules is examined at compile time.

- Because a mistake is easily made
- Invalid type rules are rejected when a Haskell file is compiled
- Type safety can still be guaranteed at run-time
- The type rule may not be too restrictive, so we are also complete
  - This restriction may be dropped

Example

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

t1 == a1 -> b      : left operand is not a function
t2 == Parser s a2  : right operand is not a parser
```

This specialized type rule is not restrictive enough:

```
The type rule for "x <$> y" is not correct
  the type according to the type rule is
    (a -> b, Parser c d, Parser c b)
  whereas the standard type rules infer the type
    (a -> b, Parser c a, Parser c b)
```

Another example

```
   x :: a -> b;    y :: Parser Char a;
-------------------------------------
      x <$> y :: Parser Char b;
```

This specialized type rule is too restrictive: there is no reason to demand that we parse streams of characters.

```
The type rule for "x <$> y" is not correct
  the type according to the type rule is
    (a -> b, Parser Char a, Parser Char b)
  whereas the standard type rules infer the type
    (a -> b, Parser c a, Parser c b)
```

Another directive: siblings

- Certain combinators are known to be easily confused:
  - cons $(:)$ and append $(+\!\!+)$
  - $<\$>$ and $<\$$
  - $(.)$ and $(+\!\!+)$ (PHP programmers)
  - $(+)$ and $(+\!\!+)$ (Java programmers)
- These combinations can be listed among the specialized type rules.

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

- The siblings heuristic will try a sibling if an expression with such an operator fails to type check.

Example

$$\begin{aligned}
&\textbf{data}\ Expr = Lambda\ [String]\ Expr\\
&pExpr\\
&\quad = pAndPrioExpr\\
&\ <|>\ Lambda\ <\$\ pKey\ \texttt{"\textbackslash\textbackslash"}\\
&\qquad\qquad <\!*\!>\ many\ pVarid\\
&\qquad\qquad <\!*\ pKey\ \texttt{"->"}\\
&\qquad\qquad <\!*\ pExpr
\end{aligned}$$

Extremely concise:

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

Concluding remarks

- I have shown what can be achieved in the context of Haskell 98 when it comes to domain specific error diagnosis.
- Implemented in the Helium compiler (www.cs.uu.nl/wiki/bin/view/Helium/WebHome)
- More details in an ICFP paper from 2003: Heeren, Hage, Swierstra, Scripting The Type Inference Process. Eighth International Conference on Functional Programming. ACM.
- See the paper and a follow-up paper on type classes at PADL '05 for many more details.

Future Work

- Scaling up to Haskell 2010 (or later)
- Because most libraries/EDSLs use extensions that we do not yet support
  - existentials
  - GADTs
  - type families