Diagnosing Type Errors with Class

Danfeng Zhang
Andrew C. Myers
Cornell University

Dimitrios Vytiniotis
Simon Peyton-Jones
MSR Cambridge

PLDI 2015
Distinguished Paper Award
Error localization is difficult for ML type systems

“It is a truism that most bugs are detected only at a great distance from their source.”

Mitchell Wand
Finding the source of type errors, POPL’86

Even worse in sophisticated type systems

The Glasgow Haskell Compiler (GHC)
- Type classes
- Type families
- GADTs
- Type signatures
Error messages are sometimes confusing.
SHEErrLoc: Static Holistic Error Locator

A general, expressive and accurate error localization method, which handles the highly expressive type system of GHC
General Error Localization [Zhang&Myers’14]

Cannot diagnose Haskell errors

Based on Bayesian interpretation

General Diagnosis Heuristics
The error cause is likely to be
• Simple
• Able to explain all errors
• Not used often on correct paths

Programs

| OCaml | Jif | Others |

Constraints

\[
\text{unit} = acc_5 \\
acc_5 = acc_3 \\
acc_3 = (\text{float*float}) \text{ list}
\]

Constraints Analysis

Cause
Key Contributions

Haskell Program

```haskell
fact n = if n == 0 then 1
else n * fac (n == 1)
```

Constraints

- `Bool = α_n`
- `α_n = α_1`
- `α_1 ≤ Num`
- `α_1 = α_*`
- `α_* ≤ Num`
- `α_n = α_0`
- `α_0 ≤ Num`

Constraints Analysis

A Bayesian model that accounts for the richer graph representation

A highly expressive constraint language

A decidable and efficient constraint analysis algorithm
Roadmap

Haskell Program

```haskell
fact n = if n == 0 then 1
        else n * fac (n == 1)
```

Constraints

\[
\begin{align*}
\text{Bool} &= \alpha_n \\
\alpha_n &= \alpha_1 \\
\alpha_n &= \alpha_* \\
\alpha_n &= \alpha_0
\end{align*}
\]

\[
\begin{align*}
\alpha_1 &\leq \text{Num} \\
\alpha_* &\leq \text{Num} \\
\alpha_0 &\leq \text{Num}
\end{align*}
\]

A highly expressive constraint language
Type Checking as Constraint Solving

• ML type system
  – Constraint elements: types
  – Constraints: type equalities

Constructors: Int, Bool, List
Variables: $\alpha, \beta, \gamma$

Syntax of Constraints

\[
\begin{align*}
E & ::= \alpha \mid \text{con}(E_1, \ldots, E_n) \\
I & ::= E_1 = E_2 \\
C & ::= \bigwedge_i I_i
\end{align*}
\]
Type Classes

Instances of a type class, called **Num**

Intuitively, a set of types
Modeling Type Class Constraints

<table>
<thead>
<tr>
<th>Syntax of Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E ::= \alpha \mid \text{con}(E_1, \ldots, E_n)$</td>
</tr>
<tr>
<td>$I ::= E_1 \equiv E_2 \mid \text{cla}(E_1, \ldots, E_n)$</td>
</tr>
<tr>
<td>$C ::= \land_i I_i$</td>
</tr>
</tbody>
</table>

Our constraint language

$$E ::= \alpha \mid \text{con}(E_1, \ldots, E_n)$$

$$I ::= E_1 \leq E_2$$

$$C ::= \land_i I_i$$

$$\llbracket \text{Num } \alpha \rrbracket ::= \alpha \leq \text{Num}$$

$$\llbracket E_1 = E_2 \rrbracket ::= E_1 \leq E_2 \land E_2 \leq E_1$$

A type class is a set of its instances
## Modeling Type Class Constraints

### Syntax of Constraints

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E ::= \alpha \mid \text{con}(E_1, \ldots, E_n)$</td>
<td>Type expression $E$ consists of a type class name $\alpha$ followed by a constraint $\text{con}$ over the types $E_1, \ldots, E_n$.</td>
</tr>
<tr>
<td>$I ::= E_1 = E_2 \mid \text{cla}(E_1, \ldots, E_n)$</td>
<td>Constraint $I$ is an equality $E_1 = E_2$ or a conjunction $\text{cla}$ of constraints $E_1, \ldots, E_n$.</td>
</tr>
<tr>
<td>$C ::= \bigwedge_i I_i$</td>
<td>Constraint $C$ is a conjunction of individual constraints $I_i$.</td>
</tr>
</tbody>
</table>

### Our constraint language

- **Type expression** $E$ consists of a type class name $\alpha$ followed by a constraint $\text{con}$ over the types $E_1, \ldots, E_n$.
- **Constraints** $I$ are either an equality $E_1 = E_2$ or a conjunction $\text{cla}$ of constraints $E_1, \ldots, E_n$.
- **Constraint** $C$ is the conjunction of individual constraints $I_i$.

### Concise; inequalities directly map to edges in a graph

A type class is a set of its instances.

Concise; inequalities directly map to edges in a graph.
Types are Checked Under Hypotheses

- Type signatures and GADTs introduce *hypotheses*

Haskell Program

```haskell
double :: Num a => a -> a
double n = n * 2
```

Constraints

```plaintext
a ≤ Num ⊢ a ≤ Num
```

- Hypothesis: a is an instance of Num
- Constraint hypothesis

Constraints are checked under hypotheses
Types are Checked Under Axioms

• Instance declaration may introduce (global) axioms

For all a, a is an instance of \texttt{Eq} implies list of a is an instance of \texttt{Eq}

\[
\text{instance Eq a} \Rightarrow \text{Eq [a]}
\]

Haskell Program

Constraint example with axioms:

\[
(\text{Int} \leq \text{Eq}) \land (\text{Int} \leq \text{Eq} \Rightarrow [\text{Int}] \leq \text{Eq}) \vdash [\text{Int}] \leq \text{Eq}
\]

Hypothesis \hspace{1cm} Axiom
Modeling Hypothesis and Axioms

<table>
<thead>
<tr>
<th>Syntax of SHErrLoc Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E ::= \alpha \mid \text{con}(E_1, ..., E_n)$</td>
</tr>
<tr>
<td>$I ::= E_1 \leq E_2$</td>
</tr>
</tbody>
</table>

- Constraints ($C$): inequalities under quantified axioms
- Quantified axioms ($Q$): implication rules
- Hypotheses (e.g., $\text{Int} \leq \text{Num}$): degenerate axioms
The Full Constraint Language

• Also supports
  – Functions on constraint elements
  – Nested universally and existentially quantified variables

• Is expressive enough to model
  – type classes, type families, GADTs, type signatures

(refer to the paper for more details)
Roadmap

Haskell Program

```haskell
fact n = if n == 0 then 1 else n * fact (n == 1)
```

Constraints

- `Bool = α_n`
- `α_n = α_1`
- `α_n = α_0`
- `α_1 ≤ Num`
- `α_0 ≤ Num`
- `α_1 ≠ Num`

Constraints Analysis

A highly expressive constraint language

A decidable and efficient constraint analysis algorithm
Constraint Graph in a Nutshell

- Graph construction (simple case)
  - Node: constraint element
  - Directed edge: partial ordering

\[
\begin{align*}
\text{Bool} &= \alpha_n \\
\alpha_n &= \alpha_1 \\
\alpha_n &= \alpha_* \\
\alpha_n &= \alpha_0 \\
\alpha_1 &\leq \text{Num} \\
\alpha_* &\leq \text{Num} \\
\alpha_0 &\leq \text{Num}
\end{align*}
\]
Constraint Analysis in a Nutshell

\[
\text{fact } n = \begin{cases} 
  0 & \text{if } n == 0 \\
  1 & \text{if } n == 1 \\
  n \times \text{fac} (n == 1) & \text{else}
\end{cases}
\]

Bool is not an instance of Num
Limitations of Previous Algorithms

[Barrett et al.’00, Melski&Reps’00, Zhang&Myers’10]

\[ \text{Int} \leq C \vdash \alpha = \text{Bool} \land [\alpha] \leq C \]

Previous algorithms *under-saturates* the graph

Satisfiable: \( \alpha = \text{Int} \)

\[ [\alpha] \]

\( \alpha \) \( \leftarrow \) \( \text{Bool} \)

Satisfiable: \( \alpha = \text{Bool} \)

Previous algs only add edges (\([\text{Bool}]\) is not in the graph!)
New Algorithm

$[\text{Int}] \leq C \vdash \alpha = \text{Bool} \land [\alpha] \leq C$

Key idea: add new edges and nodes during saturation

Key challenge: naive algorithms either fail to terminate, or under-saturate the graph
New Algorithm in a Nutshell

Black node: node before saturation
White node: added during saturation

Nodes added based on patterns
1. one edge with two black nodes
2. a black/white node

Recursion check: if white node, not added based on the edge in pattern

Lemma: the algorithm always terminates
Constraint Analysis

• The analysis also handles
  – Functions on constraint elements
  – Hypotheses
  – Quantified axioms
  (refer to the paper for more details)

• Performance
  – Empirically: quadratic in graph size
Roadmap

Haskell Program

```haskell
fact n = if n == 0 then 1 else n * fact (n == 1)
```

A highly expressive constraint language

A decidable and efficient constraint analysis algorithm

A Bayesian model that accounts for the richer graph representation

Bayesian reasoning

Constraints

\[
\begin{align*}
\text{Bool} &= \alpha_n \\
\alpha_n &= \alpha_1 \\
\alpha_n &= \alpha_+ \\
\alpha_n &= \alpha_0 \\
\alpha_1 &\leq \text{Num} \\
\alpha_+ &\leq \text{Num} \\
\alpha_0 &\leq \text{Num}
\end{align*}
\]

Constraints Analysis

A decidable and efficient constraint analysis algorithm
Likelihood Estimation [Zhang & Myers’14]

A ranking metric based on Bayesian reasoning

\[ P_1^{\lvert E \rvert} \left( \frac{P_2}{1 - P_2} \right)^{k_E} \]

(\( P_1, P_2 \) are tunable parameters)

- Simplifying assumption
  - Satisfiability of paths are independent

White nodes break this assumption
Observation: some paths using white nodes provide neither positive nor negative evidence.

Lemma: the satisfiability of any redundant path depends on non-redundant paths.

Redundant Paths (definition in paper)

- Satisfiability depends on edges between α and Bool.
New Ranking Metric

\[ P_1^{\mid E \mid} \left( \frac{P_2}{1 - P_2} \right)^{k_E} \]

• Intuitively,

General Diagnosis Heuristics

The error cause is likely to be
• Simple
• Able to explain all errors
• Not used often on correct non-redundant paths

• Top candidates returned by an efficient A* algorithm
  [Zhang&Myers’14]
Evaluation

• Implementation
  – From Haskell programs to constraints
  – SHErrLoc

Modified GHC

50 atop 20K+ LOC

GHC Constraints

Constraint Translator

~400 LOC

SHErrLoc

~7500 LOC

SHErrLoc Constraints

Constraint Graph

Error Diagnosis

Reports

little effort

Modified GHC

GHC Constraints

Constraint Translator

SHErrLoc

Constraint Graph

Error Diagnosis

Reports
Evaluation Setup

• Benchmarks
  – CE Benchmark: analyzed 77 Haskell programs collected from papers about type-error diagnosis, used in [Chen&Erwig’14]
  – Helium benchmark: analyzed 228 programs with type-checking errors, logged by the Helium tool [Hage’14]

• Ground truth
  – CE Benchmark: already well-marked
  – Helium benchmark: user’s actual fix

• Correctness
  – only when the programmer mistake is returned by tools
Accuracy on the CE Benchmark

Comparison with GHC

Comparison with the Helium tool

SHErrLoc uses no Haskell-specific heuristics!
Accuracy on the Helium Benchmark

Comparison with GHC

- SHErrLoc finds the correct error
- Other tool misses the error

Comparison with the Helium tool

- Both find the correct error
- Both miss the correct error
- Other tool finds the correct error
- SHErrLoc misses the error
Related Work

• General error localization [Zhang & Myers’14]
  – Cannot handle the type system of GHC
  – Simpler constraints and constraint analysis algorithm

• Program analyses as constraint solving [e.g., Aiken’99, Foster et al.’06]
  – No support for hypotheses and axioms

• Diagnosing Haskell error [e.g., Heeren et al.’03, Hage & Heeren’07, Chen & Erwig’14]
  – Haskell-specific heuristics
  – Unable to handle all of the sophisticated features of GHC
SHErrLoc

General, expressive and accurate error localization

– Applies to the highly expressive type system of GHC

– A novel graph-based constraint analysis algorithm

– Bayesian reasoning => more accurate error locations than with existing tools