Improving Type Error Reporting for Type Classes

Sheng Chen(✉) and Md Rabib Noor

UL Lafayette, Lafayette, LA 70503, USA
{chen,md-rabib.noor1}@louisiana.edu

Abstract. Debugging type errors when type inference fails is a challenging problem since there are many different ways to remove the type error, and it's unclear which way is intended. While numerous approaches have been proposed to more precisely locate the real error causes, most of them do not deal with popular type system extensions, such as type classes. A second problem is that most approaches do not provide enough information for removing the type error or do so for a few error causes only.

In this work, we develop an approach called TEC to address both problems. Given an ill-typed expression that may involve type classes, TEC finds comprehensive error causes and generates for each cause an error fix with detailed information for removing the type error. TEC computes all error fixes, ranks them, and iteratively presents the most likely error fix to the user until a fix is accepted. TEC reduces the problem of finding all error fixes to variational typing, which systematically reuses typing results. Our main technical innovation is a *variational context reduction* algorithm that simplifies type class constraints containing variations. We have evaluated the precision of TEC and found that it outperforms existing approaches in locating type errors arising from type class uses.

1 Introduction

Type inference allows programs to be statically typed, even without the presence of type annotations. However, it is particularly difficult to locate the real error causes and generate informative error messages when type inference fails. In the last thirty years, numerous approaches have been proposed to address this problem [4,5,10,13,17,18,23–26,30,31,35,38,39,41–43].

However, while plentiful type system extensions have been proposed and integrated into languages like Haskell, most of the error debugging methods focused on the Hindley-Milner type system (HM) plus some basic extensions, such as algebraic data types. On one hand, as type classes in Haskell are so popular nowadays, it is hard to write a program without involving them, particularly as types for numbers (such as `Int`, `Integer`, and `Double`) are members of different classes (such as `Num`, `Real`, and `Integral`) and lists are a member of `Traversable`. On the other hand, type error debugging for type classes is rudimentary. All Haskell compilers and error debuggers SHErrLoc [43] and Chameleon [33,34] diagnose type errors only and do not provide change

This work is supported by the National Science Foundation under the grant CCF-1750886.

M. Hanus and A. Igarashi (Eds.): FLOPS 2022, LNCS 13215, pp. 19–38, 2022.
https://doi.org/10.1007/978-3-030-99461-7_2

suggestions. Moreover, they usually report only several likely error causes and in many cases miss the real cause.

Consider, for example, the erroneous expression `rank1 x = (x 1, x True)` adapted from [2]. This expression is ill-typed because it applies the argument to values of different types. For this expression the most widely used Haskell compiler GHC version 8.10.6 displays the following message.

```
* No instance for (Num Bool) arising from the literal ‘1’
* In the first argument of ‘x’, namely ‘1’
  In the expression: x 1
  In the expression: (x 1, x True)
```

This error message is not helpful for removing the type error for several reasons. First, the type information and the location information is inconsistent. The message seems to say that `1` requires the type system to prove `Bool` be an instance of the class `Num`. While `1` gives rise to the type class `Num`, it doesn't have anything to do with `Bool`. Second, it doesn't say how to remove the type error, for example, adding an instance definition `Num Bool`, changing `1` to something of `Bool`, and so on. In this small example, it is not hard to infer that `Num Bool` is caused by the conflict of `1` and `True`. However, when the program is large, the conflicting subexpressions may be far away, and figuring out the exact problem can be hard. Third, GHC only reports the problem at `1` while, in fact, changing any of `True` or either of `x` will make `rank1` well-typed. Moreover, there is no evidence that `1` is more likely to cause the type error than `True`. Thus, reporting the problem at `1` is biased for users. For this example, SHErrLoc produces a similar message, except that it mentions both `1` and `True`.

We will use the terms *error cause*s and *error fix*es throughout the paper. Given an expression, an error cause is a set of subexpressions such that changing them appropriately can make the expression well-typed. We omit the set delimiters if the cardinality is 1. For example, all of `x` (either occurrence), `1`, `True`, and $\{\texttt{1}, \texttt{True}\}$ are possible error causes for `rank1`. An error fix is an error cause plus change information, for each member of the error cause, to make the expression well-typed. For example, the error cause `1` plus the change of `1` to something of type `Bool` is an error fix for `rank1`. Given an oracle (for example, specified by the paper where the example was introduced), we say an error cause or error fix is correct if it is consistent with the oracle and incorrect otherwise. In these terms, both GHC and SHErrLoc only identify error causes and do not generate error fixes.

When multi-parameter classes and functional dependencies [22] are involved, the messages become worse. Consider, for example, the expression `insert2` adapted from [34]. The functional dependency `ce -> e` specifies that the container type `ce` uniquely determines the element type `e`. Many interesting `Collects` instances may be defined, such as lists, characteristic functions, bit sets for representing collections of characters, etc.

```
class Collects ce e | ce -> e where
    empty :: ce
    insert :: e -> ce -> ce

insert2 c = insert 1 (insert True c)
```

The expression insert2 contains a type inconsistency between 1 and True because they violate the functional dependency [34]. For insert2, GHC generates a similar message as before while SHErrLoc generates the following message, where the text with a grey background is the error cause identified by SHErrLoc. The corresponding constraint for each error given by SHErrLoc is attached to the end of each message in italics. As a result, according to [34], these messages are incorrect.

```
 2 errors found
 A value with type cons_2 (ce_aL3) (e_aL4) is being used at type Collects
 insert2 c = insert 1 (insert True c)  cons_2 (ce_aL3) (e_aL4) <= Collects


A value with type cons_2 (ce_aHQ) (e_aHR) is being used at type Collects
insert2 c = insert 1 (insert True c)   cons_2 (ce_aHQ) (e_aHR) <= Collects
```

Based on previous examples, we observe that the tool support for debugging type errors for type systems with type classes is inadequate. To address this problem, we develop TEC, which (1) finds comprehensive error causes, (2) generates an error fix for each cause, and (3) ranks all error fixes and presents them iteratively. For insert2, TEC generates the following error fix.

```
The expression contains type errors. Possible fix:
Change: "True", of type: "Bool", to something of type: "Num f => f"
the resulting type will be: (Collects c f, Num f) => c -> c

Show more one-change fixes? (y/n)
```

We can see that each fix includes the error cause (True), the type it has under normal type inference (Bool), the type it ought to have to remove the type error (Num f => f), and the type of the resulting expression after applying this fix. We will refer to these three type parts as *source type*, *target type*, and *consequent type*, respectively.

This message provides abundant information to remove the type error: the consequent type allows the user to quickly decide if this message is useful. For example, if the user's expected type of insert2 is Collects c Bool => c -> c, then the user can simply skip this message and ask for the next one. Otherwise if the consequent type is intended, the user can turn to the target type to further decide how she can fix the type error. In this case, the user can figure out that she should change True to something that is an instance of the Num type class.

If this message is not useful, the user can hit the letter y to ask for the next message, which suggests to change 1, with the source type Num a => a, the target type Bool, and the consequent type Collects c Bool => c -> c. This process continues until all error fixes are displayed. Later error fixes may involve multiple subexpressions. For example, after the first four fixes, TEC starts to generate fixes involving two subexpressions.

We have evaluated the precision of TEC in more depth for two benchmarks (Sect. 5) and the result shows that TEC is precise in locating type errors. Also, TEC is fast enough for practical use. For example, for programs of about 100 LOC and 300 LOC, TEC delivers the first error message within 1.6s and 5.7s, respectively. While the response time is still slower than compilers, this cost pays off as effective and informative error messages generated by TEC can save beginners dozens of minutes for fixing type errors, a view shared by [25].

Overall, our contributions in this paper is developing an error debugger, TEC, that considers type classes and functional dependencies, finds complete error fixes in leaves and their combinations under moderate conditions, and is fast enough for practical use. Each error fix provides abundant information to remove the type error. Along the way, we formally develop a type system for finding comprehensive error fixes and a variational context reduction for simplifying type class constraints.

We give an overview of TEC in Sect. 2, present a type system in Sect. 3, develop constraint generation and variational context reduction in Sect. 4, present evaluation in Sect. 5, discuss related work in Sect. 6, and conclude in Sect. 7.

2 TEC, Informally

TEC relies on the machineries developed in variational typing [6,7] to efficiently find comprehensive error fixes. In this section, we first present background on variational typing and then use an example to illustrate the idea of TEC.

Background. Variational typing introduces variations to the type syntax. For example, the type $A\langle \texttt{Int}, \texttt{Bool}\rangle$ contains a variation named A, which has two *alternatives*: `Int` and `Bool`. An expression having the type $A\langle \texttt{Int}, \texttt{Bool}\rangle$ means that the expression has either the type `Int` or `Bool`, depending on which alternative is taken. Variations can be considered as a binary type constructor, so $A\langle \texttt{Int}, B\langle \texttt{Bool}, \texttt{Int}\rangle\rangle$, $A\langle \texttt{Int}, \texttt{Bool}\rangle \rightarrow \texttt{Bool}$, and $A\langle \texttt{Int}, \texttt{Bool}\rangle \rightarrow B\langle \texttt{Bool}, \texttt{Int}\rangle$ are all valid types.

Variations can be eliminated through a process called *selection*, which takes a type ϕ and a selector s of the form $d.i$ and replaces all occurrences of variations named d with their ith alternatives in ϕ. We write $\lfloor \phi \rfloor_{d.i}$ for selecting ϕ with $d.i$. For instance, $\lfloor A\langle \texttt{Int}, \texttt{Bool}\rangle \rfloor_{A.2}$ yields `Bool`. A decision is a set of selectors. We use ω to range over decisions. Selection extends naturally to decisions as $\lfloor \phi \rfloor_{s\omega} = \lfloor \lfloor \phi \rfloor_{s} \rfloor_{\omega}$. Note that the ordering of selection doesn't matter. We say ω is complete with respect to ϕ if $\lfloor \phi \rfloor_{\omega}$ yields a *plain* type, which doesn't contain any choice. Selecting a plain type with any selector yields the type itself. For instance, $\lfloor \texttt{Int} \rfloor_{B.2} = \texttt{Int}$. Based on the definition of selection, choices with the same name are synchronized in the sense that we have to select the same alternatives from them and those with different names are independent. Thus, while $A\langle \texttt{Int}, \texttt{Bool}\rangle \rightarrow A\langle \texttt{Bool}, \texttt{Int}\rangle$ can generate two plain types, $A\langle \texttt{Int}, \texttt{Bool}\rangle \rightarrow B\langle \texttt{Bool}, \texttt{Int}\rangle$ can generate four with two different parameter types and two different return types.

An Example of Debugging with TEC. During type inference, compilers infer the most general types for all the subexpressions visited, which defers the detection of type errors. Thus, compiler error messages are often biased and incomplete. To find comprehensive error fixes, TEC systematically assume that each leaf may cause the type error and find the target type of each leaf to remove the type error in the expression. TEC also computes the source type and consequent type of changing each leaf (see Page 3 below our error message for the meanings of the terms source type and consequent type). In fact, TEC also finds error fixes in arbitrary combinations of leaves, and in this case computes related type information for each leaf in the error fix. After the computation is finished, TEC determines that a leaf indeed causes the type error if the source type differs from the target type for that leaf.

TEC reduces this computation process to variational typing by traversing the expression just once. For each leaf, TEC assigns a variational type whose left and right alternatives represent the source and target types, respectively. The source type is the type a leaf has under normal type inference and the target type is the type that makes the context well-typed. When a leaf is first visited, the target type is a fresh type variable, which will be refined to the correct type after the typing constraints are solved.

We illustrate this process with type error debugging for `abs True`, where `abs` has the type $\forall\alpha_1.\texttt{Num}\ \alpha_1 \Rightarrow \alpha_1 \rightarrow \alpha_1$. The following table lists types for subexpressions and the generated constraints, which are numbered. Each leaf receives a choice type that includes the source type and the target type for it. In the constraint (1), $\texttt{Num}\ \alpha_2$, the constraint in the left alternative, is come from the constraint of $\texttt{Num}\ \alpha_2 \Rightarrow \alpha_2 \rightarrow \alpha_2$, which is the type of `abs`. In that constraint, ε in the right alternative means that the type α_3 has no constraint. The constraint (2) expresses the equivalence constraint between two types, and we use $\equiv^?$ to denote such constraints.

Subexpr.	Types	Constraints	Idx
`abs`	$A\langle\alpha_2 \rightarrow \alpha_2, \alpha_3\rangle$	$A\langle\texttt{Num}\ \alpha_2, \varepsilon\rangle$	(1)
`True`	$B\langle\texttt{Bool}, \alpha_4\rangle$		
`abs True`	β_1	$A\langle\alpha_2 \rightarrow \alpha_2, \alpha_3\rangle \equiv^? B\langle\texttt{Bool}, \alpha_4\rangle \rightarrow \beta_1$	(2)

In general, traversing an expression generates both type class constraints and type equivalence constraints. Type equivalence constraints are solved with the variational unification algorithm from [6]. In addition to a unifier, constraint solving also returns a *pattern* to indicate at which variants constraint solving is successful and at which it is not. Specifically, a pattern, written as π, can be $\top$ (denoting constraint solving success), $\bot$ (denoting solving failure), or a variation between two patterns (such patterns can be useful when constraint solving in one variant fails while in the other variant succeeds). For example, the pattern for solving the constraint $A\langle\texttt{Int}, \texttt{Bool}\rangle \equiv^? \texttt{Int}$ is $A\langle\top, \bot\rangle$ since the constraint solving in the second alternative fails. Type class constraints are solved using variational context reduction, to be developed in Sect. 4.1. Similarly, a pattern is returned to indicate where reduction is successful and where is not.

For the constraints (2) above, the solution is as follows. The returned pattern is $\top$, since constraint solving is successful for all variants. κs in the solution represent fresh type variables introduced during unification.

$$\theta = \{\alpha_2 \mapsto A\langle B\langle\texttt{Bool}, \alpha_4\rangle, \kappa_1\rangle, \alpha_3 \mapsto A\langle\kappa_3, B\langle\texttt{Bool}, \alpha_4\rangle \rightarrow \kappa_2\rangle, \beta_1 \mapsto A\langle B\langle\texttt{Bool}, \alpha_4\rangle, \kappa_2\rangle\}$$

After that, we apply θ to the constraint (1) and remove the dead alternative (κ_1), which yield the new constraint $C_1 = A\langle\texttt{Num}\ B\langle\texttt{Bool}, \alpha_4\rangle, \varepsilon\rangle$. With variational context reduction, C_1 is reduced to (π, C) where $\pi = A\langle B\langle\bot, \top\rangle, \top\rangle$ and $C = A\langle B\langle\texttt{Num Bool}, \texttt{Num}\ \alpha_4\rangle, \varepsilon\rangle$. The π indicates that C_1 can not be reduced to the normal form [21] in variant $\{A.1, B.1\}$. Overall, the result type is $\phi = C \Rightarrow \theta(\beta_1)$ with the pattern π.

With ϕ, θ, and π, we can generate error fixes by considering different decisions. For any ω, if $\lfloor\pi\rfloor_\omega$ is $\top$, then ω corresponds to an error fix that will remove the type error in the expression. The reason is that, as discussed earlier, only variants where both variational unification and context reduction are successful will receive $\top$. If either fails, then the variant contains a $\bot$. Given a ω, if $d.1 \in \omega$, it means that we do not change the subexpression where d is created. Otherwise, we change the corresponding subexpression to something of type $\lfloor \theta(\alpha) \rfloor_\omega$, where α is the target type for the subexpression. For example, let's consider generating the error fix for the decision $\omega = \{A.1, B.2\}$. Since $\lfloor \pi \rfloor_{\{A.1,B.2\}} = \top$, this fix will remove the type error. The decision ω corresponds to changing `True` only. The target type is α_4 and the constraint for the target type is $\lfloor C \rfloor_\omega$ = `Num` α_4, meaning that the overall target type for `True` is `Num` $\alpha_4 \Rightarrow \alpha_4$. The consequent type type is $\lfloor \phi \rfloor_\omega$, which is `Num` $\alpha_4 \Rightarrow \alpha_4$. This provides all the information needed for generating our error message in Sect. 1.

3 Type System

This section presents a type system for computing comprehensive error fixes and studies its properties. While supporting type class constraints, our formalization is made general so that it can be instantiated to support other type constraints.

3.1 Syntax

Figure 1 collects the syntax for types, expressions, and related environments. We consider a formalization for HM plus multi-parameter type classes. We use c to range over constants. Our formalization omit functional dependencies for simplicity though our implementation supports them. Types are stratified into three layers. Monotypes include constant types (γ), type variables (α), and function types. Variational types extend monotypes with choice types. We use τ and ϕ to range over monotypes and variational types, respectively. Type schemas, ranged over by σ, has the form $\forall \overline{\alpha}.C \Rightarrow \phi$, where C specifies the requirements of types substituting $\overline{\alpha}$ in ϕ. We use $FV(\sigma)$ to return the set of free type variables in σ.

$$\text{Expressions } e, f ::= c \mid x \mid \lambda x.e \mid e\ e \mid \textbf{let}\ x = e\ \textbf{in}\ e$$

$$\begin{array}{llcl}
\text{Monotypes} & \tau & ::= & \gamma \mid \alpha \mid \tau \to \tau \\
\text{Variational types} & \phi & ::= & \tau \mid d\langle \phi, \phi \rangle \mid \phi \to \phi \\
\text{Type schemas} & \sigma & ::= & \phi \mid \forall \overline{\alpha}.C \Rightarrow \phi \\
\text{Constraints} & C & ::= & \varepsilon \mid \phi \equiv \phi \mid C \wedge C \mid \mathtt{G}\ \overline{\phi} \mid d\langle C, C \rangle \\
\text{Axiom schemes} & Q & ::= & C \mid Q \wedge Q \mid \forall \overline{\alpha}.C \Rightarrow \mathtt{G}\ \overline{\tau} \\
 & & \mid & \forall \overline{\alpha}.C \Leftarrow \mathtt{G}\ \overline{\alpha} \\
\text{Typing patterns} & \pi & ::= & \top \mid \bot \mid d\langle \pi, \pi \rangle \\
\text{Type environments} & \Gamma & ::= & \varnothing \mid \Gamma, x \mapsto \sigma \\
\text{Substitutions} & \theta & ::= & \varnothing \mid \theta, \alpha \mapsto \phi \\
\text{Choice environments} & \Delta & ::= & \varnothing \mid \Delta, (l, d\langle \phi, \phi \rangle)
\end{array}$$

Fig. 1. Syntax

There are two main forms of primitive constraints. The first form is the type equivalence requirement $\phi_1 \equiv \phi_2$, which specifies that ϕ_1 and ϕ_2 must be equivalent. Two types are equivalent if selecting them with the same decision always yields the same type. For example, $d\langle$`Int`, `Int`$\rangle \equiv$ `Int`. The second is the type class constraint $\mathtt{G}\ \overline{\tau}$. Compound constraints include $C_1 \wedge C_2$, where $\wedge$ is commutative, and $d\langle C_1, C_2 \rangle$, a variation between C_1 and C_2 under the name d.

$$\text{E1}\quad Q \wedge C \Vdash C \qquad \text{E2}\ \frac{Q \wedge C_1 \Vdash C_2 \qquad Q \wedge C_2 \Vdash C_3}{Q \wedge C_1 \Vdash C_3} \qquad \text{E3}\ \frac{Q \Vdash C}{\theta(Q) \Vdash \theta(C)}$$

$$\text{E4}\ \frac{Q \Vdash C_1 \qquad Q \Vdash C_2}{Q \Vdash C_1 \wedge C_2} \qquad \text{E5}\ \frac{Q \wedge C_1 \Vdash C_2 \qquad Q \wedge C_3 \Vdash C_4}{Q \wedge d\langle C_1, C_3\rangle \Vdash d\langle C_2, C_4\rangle}$$

$$\text{E6}\ \frac{\forall \omega. \lfloor \pi \rfloor_\omega = \top \Rightarrow \lfloor Q \rfloor_\omega \Vdash \lfloor C \rfloor_\omega}{Q \Vdash_\pi C}$$

$$\text{T1}\ \frac{Q \Vdash \phi_1 \equiv \phi_2}{Q \Vdash \phi_2 \equiv \phi_1} \qquad \text{T2}\ \frac{Q \Vdash \phi_1 \equiv \phi_2 \qquad Q \Vdash \phi_2 \equiv \phi_3}{Q \Vdash \phi_1 \equiv \phi_3} \qquad \text{T3}\quad Q \Vdash \phi \equiv \phi$$

$$\text{T4}\ \frac{Q \Vdash \phi_1 \equiv \phi_2}{Q \Vdash C[\phi_1] \leftrightarrow C[\phi_2]} \qquad \text{T5}\quad Q \Vdash d\langle \phi, \phi \rangle \equiv \phi$$

$$\text{T6}\quad Q \Vdash d\langle \phi_1, \phi_2 \rangle \equiv d\langle \lfloor \phi_1 \rfloor_{d.1}, \lfloor \phi_2 \rfloor_{d.2} \rangle \qquad \text{T7}\ \frac{\forall \overline{\alpha}.C \Rightarrow \mathtt{G}\ \overline{\tau} \in Q \qquad Q \Vdash [\overline{\alpha \mapsto \phi}](C)}{Q \Vdash [\overline{\alpha \mapsto \phi}](\mathtt{G}\ \overline{\tau})}$$

$$\text{T8}\ \frac{\forall \overline{\alpha}.C \Leftarrow \mathtt{G}\ \overline{\alpha} \in Q \qquad Q \Vdash [\overline{\alpha \mapsto \phi}](\mathtt{G}\ \overline{\alpha}) \qquad C_G \in C}{Q \Vdash [\overline{\alpha \mapsto \phi}](C_G)}$$

Fig. 2. Entailment relation of constraints

Axiom schemes include constraints (C), abstractions of class declarations ($\forall \overline{\alpha}.C \Leftarrow \mathtt{G}\ \overline{\alpha}$), and those of instance declarations ($\forall \overline{\alpha}.C \Rightarrow \mathtt{G}\ \overline{\tau}$). For example, the declaration `class Eq a => Ord a where...` gives rise to $\forall \alpha.\mathtt{Eq}\ \alpha \Leftarrow \mathtt{Ord}\ \alpha$. We use a left arrow in the scheme to reflect that any type that is an instance of a subclass (`Ord`) is also an instance of the parent class (`Eq`). Similarly, the instance declaration `instance Eq a => Eq [a] where...` gives rise to $\forall \alpha.\mathtt{Eq}\ \alpha \Rightarrow \mathtt{Eq}\ [\alpha]$.

We use l to range over program locations. Each program element has a unique location in the program. We use the function $\ell_e(f)$ to return the location of f in e. We may omit the subscript e when the context is clear. For simplicity, we assume that f uniquely determines the location. The exact definition of $\ell(\cdot)$ doesn't matter.

The definitions of the type environments Γ and the substitutions θ are conventional, mapping expression variables to type schemas and type variables to variational types, respectively. The choice environment Δ associates each program location l to a choice type $d\langle \phi_1, \phi_2 \rangle$, where d, ϕ_1, and ϕ_2 are the choice, the source type, and the target type for the subexpression at l, respectively.

3.2 Type System

Constraint Entailment. Constraints are related together through the entailment relation defined in Fig. 2. The relation $Q \Vdash C$ specifies that under axiom Q, the constraint C is satisfied. The first three rules are standard in constrained type systems [19,28]. The rules E1 and E2 specify that the relation is reflexive and transitive. The rule E3 states

$$\boxed{C;\pi;\Gamma \vdash e : \phi | \Delta} \qquad \textsc{Con}\ \frac{c \text{ is of type } \gamma \qquad \exists\, d}{C;\pi;\Gamma \vdash c : d\langle \gamma,\phi \rangle | \{(\ell(c), d\langle \gamma,\phi \rangle)\}}$$

$$\textsc{Var}\ \frac{\Gamma(x) = \forall\overline{\alpha}.C \Rightarrow \phi \qquad \exists\, d \qquad C_1 \Vdash_{d\langle \lfloor\pi\rfloor_{d.1},\bot \rangle} [\overline{\alpha \mapsto \phi'}](C)}{C_1;\pi;\Gamma \vdash x : d\langle [\overline{\alpha \mapsto \phi'}](\phi),\phi_1 \rangle | \{(\ell(x), d\langle [\overline{\alpha \mapsto \phi'}](\phi),\phi_1 \rangle)\}}$$

$$\textsc{Unbound}\ \frac{x \notin dom(\Gamma) \qquad \exists\, d \qquad \pi \leq d\langle \bot,\top \rangle}{C;\pi;\Gamma \vdash x : \phi | \{(\ell(x),\phi)\}} \qquad \textsc{Abs}\ \frac{C;\pi;\Gamma, x \mapsto \phi \vdash e : \phi' | \Delta}{C;\pi;\Gamma \vdash \lambda x.e : \phi \to \phi' | \Delta}$$

$$\textsc{Let}\ \frac{C;\pi;\Gamma, x \mapsto \phi \vdash e : \phi | \Delta \qquad \overline{\alpha} = FV(\phi) - FV(\Gamma) \qquad C_1;\pi;\Gamma, x \mapsto \forall\overline{\alpha}.C \Rightarrow \phi \vdash e_1 : \phi_1 | \Delta_1}{C \wedge C_1;\pi;\Gamma \vdash \textbf{let } x \texttt{ = } e \textbf{ in } e_1 : \phi_1 | \Delta \cup \Delta_1}$$

$$\textsc{App}\ \frac{C;\pi;\Gamma \vdash e_1 : \phi_1 | \Delta_1 \qquad C;\pi;\Gamma \vdash e_2 : \phi_2 | \Delta_2 \qquad C \Vdash_\pi \phi_1 \equiv \phi_2 \to \phi}{C;\pi;\Gamma \vdash e_1\ e_2 : \phi | \Delta_1 \cup \Delta_2}$$

$$\boxed{Q;\pi;\Gamma \vdash_M e : \phi | \Delta} \qquad \textsc{Main}\ \frac{C;\pi;\Gamma \vdash e : \phi | \Delta \qquad \pi_1 \leq \pi \qquad Q \Vdash_{\pi_1} C}{Q;\pi_1;\Gamma \vdash_M e : \phi | \Delta}$$

Fig. 3. Typing rules

that the entailment relation is stable under type substitution. Type substitution, written as $\theta(\sigma)$, substitutes free type variables in σ with their corresponding mappings in θ. For instance and class declaration constraints, substitution has no effect. For other constraints, substitution applies to their type components. The rules E4 and E5 show how to satisfy the compound constraints $C_1 \wedge C_2$ and $d\langle C_3, C_4 \rangle$. The rule E6 introduces the partial entailment relation $\Vdash_\pi$, which specifies that the validity of $\Vdash$ is only limited to the variants where π has $\top$s.

The rest of the rules in Fig. 2 specify the relations between type equivalence constraints. The rules T1 through T3 express that this relation is reflexive, symmetric, and transitive. In T4, we use $C[\,]$ to denote a constraint context into which we can plug a type. The rule says that the satisfiability of a constraint is preserved if a type of it is replaced by an equivalent type. The notation $Q \Vdash C_1 \leftrightarrow C_2$ is a shorthand for $Q \wedge C_1 \Vdash C_2$ and $Q \wedge C_2 \Vdash C_1$. The rule T5 states that a choice with same alternatives is equivalent to its alternatives. The rule T6 says that dead alternatives can always be removed without affecting type equivalence. Here $\lfloor \phi_1 \rfloor_{d.1}$ removes all the second alternatives of d choices inside ϕ_1, which are unreachable because ϕ_1 is in the first alternative of d. For example, $Q \Vdash A\langle \texttt{Int}, A\langle \texttt{Char},\texttt{Bool} \rangle\rangle \equiv A\langle \texttt{Int},\texttt{Bool} \rangle$. The rules T7 and T8 deal with entailments brought in by instance and class declarations, respectively. We use C_G to denote type class constraints having type variables as their arguments.

Typing Rules. Typing rules are presented in Fig. 3. Type judgment has the form $C;\pi;\Gamma \vdash e : \phi | \Delta$, which is read as: under the type environment Γ and constraint C the expression e has the type ϕ with the validity restriction π, with type change information collected in Δ. Here Γ, C, π, and e are inputs, and ϕ and Δ are outputs. The π indicates that the typing result is required to be correct only in the alternatives that π has $\top$s and is not required to be correct in alternatives that π has $\bot$s. For example, disregarding Δ, $\varepsilon;\top;\Gamma \vdash 1 : \texttt{Int}|\Delta$ is valid. Interestingly, while 1 does not have the type `Bool`, $\varepsilon;\bot;\Gamma \vdash 1 : \texttt{Bool}|\Delta$ is also a valid type judgment since $\bot$ in the judgment says the result does not need to be correct. However, the judgment $\varepsilon;\top;\Gamma \vdash 1 : \texttt{Bool}|\Delta$ is invalid because $\top$ requires the result to be correct but 1 does not have the type `Bool`. Intuitively,

In many rules, we use the condition $\exists\, d$, where d can be a fresh or existing choice name. This condition allows us to maintain the flexibility of assigning variations to leaves. If we assign unique variations to leaves, we can change leaves independently. Otherwise, if some leaves receive the same variation, then either all or none of them will be changed. This condition is always satisfied.

The rule Con, dealing with unconstrained constants, says that for a constant c of the type γ, the source type is γ and the target type is ϕ, which is unconstrained, meaning that we can change c to anything to remove the type error. The π component can be any value since changing a constant will not introduce any error. Here Δ is assigned $\{(\ell(c), d\langle\gamma,\phi\rangle)\}$ to record the change information. The rule for constrained constants is very similar to Var and we will not present it here. The rule Var for variables has a similar pattern as Con has. The source type for a variable x is any valid instantiation of the polymorphic type $\Gamma(x)$ and the target type ϕ_1 is again unconstrained. Since typing a variable is always correct, π can be any value. The rule records a change for x in Δ.

We also need a rule for unbound variables since we do not want the typing process to be terminated. As always, the target type can be an unconstrained type. The question is, what should be the source type for the variable? Since the variable is unbound, we couldn't find out the correct source type. Fortunately, we can avoid this problem by choosing an appropriate π. Specifically, the first alternative of the typing pattern must be $\bot$, denoting that the typing result of the first alternative that accesses the unbound variable is invalid. As always, the second alternative can be any value.

To formally express this idea, we first define the *more-defined* relation between typing patterns as follows. We write $\pi_1 \leq \pi_2$ to express that π_2 is more-defined than π_1. Intuitively, $\pi_1 \leq \pi_2$ if for any alternative π_2 has an $\bot$ then so does π_1.

$$\pi \leq \top \qquad \bot \leq \pi \qquad \frac{\pi_1 \leq \pi_2 \qquad \pi_2 \leq \pi_3}{\pi_1 \leq \pi_3} \qquad \frac{\pi_1 \leq \pi_3 \qquad \pi_2 \leq \pi_4}{d\langle\pi_1,\pi_2\rangle \leq d\langle\pi_3,\pi_4\rangle} \qquad \frac{\Vdash \pi_1 \equiv \pi_2}{\pi_1 \leq \pi_2}$$

The first two rules indicate that all typing patterns are more-defined than $\bot$ and less-defined than $\top$. The third rule states that $\leq$ is transitive. The fourth rule says that two choice typing patterns satisfy $\leq$ if both corresponding alternatives satisfy this relation. Finally, the last rule reuses the entailment relation defined for variational types by interpreting $\top$ and $\bot$ as two distinct constant types. The rule says that two typing patterns that satisfy the equivalence relation also satisfy the $\leq$ relation. This allows us to derive $\top \leq d\langle\top,\top\rangle$ since the two sides are equivalent according to the rule T2 in Fig. 2.

We formalize the idea for typing unbound variables in the rule Unbound with the help of $\leq$. The rules Abs for abstractions and Let for **let** expressions are similar to those in the constraint-based formalization of HM, except for the threading information for π and Δ here. The rule App deals with applications. The rules for deriving type equivalence ($\equiv$) are given in the upper part of Fig. 2.

$$\text{E6}\ \frac{\lfloor C_4 \rfloor_{\{A.1,B.1\}} \Vdash \texttt{Num Bool} \qquad \lfloor C_4 \rfloor_{\{A.1,B.2\}} \Vdash \texttt{Num}\ \alpha}{C_4 \Vdash_{\pi_5} [\alpha_1 \mapsto B\langle \texttt{Bool}, \alpha \rangle](\texttt{Num}\ \alpha_1)}$$

$$\frac{\Gamma(\texttt{abs}) = \forall \alpha_1.\texttt{Num}\ \alpha_1 \Rightarrow \alpha_1 \to \alpha_1 \qquad \exists\ A \qquad \vdots}{C_4; \pi_4; \Gamma \vdash \texttt{abs} : \phi_5 | \Delta_5}\ \text{Var}$$

$$\text{E6}\ \frac{\begin{array}{c} C_4 \Vdash \lfloor \phi_5 \rfloor_{\{A.1\}} \equiv B\langle \texttt{Bool}, \alpha \rangle \to B\langle \texttt{Bool}, \alpha \rangle \\ C_4 \Vdash \lfloor \phi_5 \rfloor_{\{A.2,B.1\}} \equiv \texttt{Bool} \to \texttt{Bool} \end{array}}{C_4 \Vdash_{\pi_4} \phi_5 \equiv B\langle \texttt{Bool}, \alpha \rangle \to A\langle B\langle \texttt{Bool}, \alpha \rangle, \texttt{Bool} \rangle}$$

$$\text{Con}\ \frac{\texttt{True}\ \text{is of type}\ \texttt{Bool} \qquad \exists\ B}{C_4; \pi_4; \Gamma \vdash \texttt{True} : B\langle \texttt{Bool}, \alpha \rangle | \Delta_4}$$

$$\text{App}\ \frac{\vdots \qquad C_4; \pi_4; \Gamma \vdash \texttt{True} : B\langle \texttt{Bool}, \alpha \rangle | \Delta_4 \qquad \vdots}{C_4; \pi_4; \Gamma \vdash \texttt{abs True} : A\langle B\langle \texttt{Bool}, \alpha \rangle, \texttt{Bool} \rangle | \Delta_6}$$

$$C_4 = A\langle B\langle \texttt{Num Bool}, \texttt{Num}\ \alpha \rangle, \varepsilon \rangle \qquad \Delta_4 = \{(\ell(\texttt{True}), B\langle \texttt{Bool}, \alpha \rangle)\}$$

$$\pi_4 = A\langle \top, B\langle \top, \bot \rangle \rangle \qquad \Delta_5 = \{(\ell(\texttt{abs}), \phi_5)\}$$

$$\pi_5 = A\langle \lfloor \pi_4 \rfloor_{A.1}, \bot \rangle = A\langle \top, \bot \rangle \qquad \Delta_6 = \Delta_4 \cup \Delta_5$$

$$\phi_5 = A\langle B\langle \texttt{Bool} \to \texttt{Bool}, \alpha \to \alpha \rangle, \texttt{Bool} \to \texttt{Bool} \rangle$$

Fig. 4. The derivation tree for the expression `abs True`.

In general, we are interested in generating error fixes for an expression given user defined axioms Q. We formalize this idea in the rule Main in Fig. 3. The type judgment for this rule is very similar to that for other rules and can be read similarly.

Example. Let's consider typing `abs True` under the constraint $C_4 = A\langle B\langle \texttt{Num Bool}, \texttt{Num}\ \alpha \rangle, \varepsilon \rangle$, where $\Gamma(\texttt{abs}) = \forall \alpha_1.\texttt{Num}\ \alpha_1 \Rightarrow \alpha_1 \to \alpha_1$ and `True` is of the type `Bool`. We present the derivation tree, together with values for symbols (such as C_4, π_4, etc.), for typing this expression in Fig. 4. Most of the derivations are simple instantiations of the typing rules in Fig. 3, except the two using E6. We defer the detailed explanation of deriving them to the long version of this paper [8]. Overall, we derive the typing result $C_4; \pi_4; \Gamma \vdash \texttt{abs True} : A\langle B\langle \texttt{Bool}, \alpha \rangle, \texttt{Bool} \rangle | \Delta_6$ for the expression `abs True`.

We observe that C_4 includes the constraints `Num Bool` and `Num` α. What is the result of typing the expression under the axiom $Q_6 = \texttt{Num}\ \alpha$? Let $\pi_6 = A\langle B\langle \bot, \top \rangle, B\langle \top, \bot \rangle \rangle$, we have $\pi_6 \leq \pi_4$ and $Q_6 \Vdash_{\pi_6} C_4$. According to the rule Main, we have $Q_6; \pi_6; \Gamma \vdash_M \texttt{abs True} : A\langle B\langle \texttt{Bool}, \alpha \rangle, \texttt{Bool} \rangle | \Delta_6$. From π_6, Δ_6, and the result type $A\langle B\langle \texttt{Bool}, \alpha \rangle, \texttt{Bool} \rangle$, we can extract error fixes as we did in Sect. 2. For example, changing `abs` to something of type `Bool` → `Bool` the resulting expression will have the type `Bool`.

Properties. We now investigate the properties of finding error fixes by our type system. We first present a few definitions. Given an ω and a Δ, we can generate an *alteration* for fix type errors as follows, where $locToChc(l)$ returns the variation name generated at l during the typing process.

$$alt(\omega, \Delta) = \{(l, \lfloor \phi \rfloor_\omega) | (l, \phi) \in \Delta \wedge locToChc(l).2 \in \omega\}$$

An alteration specifies the target type of each location in it. We use δ to range over alterations. For example, $alt(\{A.2, B.1\}, \Delta_6)$ yields $\{(\ell(\texttt{abs}), \texttt{Bool} \to \texttt{Bool})\}$, where Δ_6 was introduced in Sect. 3.2 for typing `abs True`.

To reason about the correctness of our type system, we need a notion of applying alterations to the Hindley-Milner typing process extended with type classes for ill-typed expressions. For this purpose, we introduce the typing judgment $Q;\Gamma;\delta \vdash_{ALT} e : \tau$, which says that under Q, Γ, and the alteration δ, the expression e has the type τ. The type system is mostly standard but for each subexpression e' satisfying $\ell(e') \mapsto \tau' \in \delta$, e' has the type τ'. Since the specification of $\vdash_{ALT}$ is simple, we defer it to [8].

The correctness of our type system consists of the soundness and completeness of error fixes, shown in the following theorems. The proofs for both theorems are constructed through induction on the corresponding typing process.

Theorem 1 (Error fix soundness). *If* $Q;\pi;\Gamma \vdash_M e : \phi|\Delta$, *then for any* ω $\lfloor \pi \rfloor_\omega = \top$ *implies* $Q;\Gamma;alt(\omega, \Delta) \vdash_{ALT} e : \lfloor \phi \rfloor_\omega$.

Theorem 2 (Error fix completeness). *Given* Q, *e, and* Γ, *if* $Q;\Gamma;\delta \vdash_{ALT} e : \tau$, *then there exists some* ϕ *and* Δ *such that* $Q;\pi;\Gamma \vdash_M e : \phi|\Delta$, $\lfloor \phi \rfloor_\omega = \tau$, *and* $alt(\omega, \Delta) = \delta$ *for some* ω.

4 Constraint Generation and Solving

When an expression is visited, both type constraints and class constraints are generated. Type constraints are solved using variational unification from [6], yielding substitutions, which will be applied to class constraints. After that, class constraints are simplified.

Constraint Generation. Since constraint generation rules can be systematically derived from the typing rules in Fig. 3, we present the rules only for variables and applications. Other constraint generation rules can be derived similarly.

$$\textsc{I-Var}\ \frac{\Gamma(x) = \forall\overline{\alpha_1}.C \Rightarrow \phi \qquad d, \overline{\alpha_2}, \alpha_3 \text{ fresh} \qquad \phi_1 = [\overline{\alpha_1 \mapsto \alpha_2}](\phi)}{d\langle[\overline{\alpha_1 \mapsto \alpha_2}](C), \varepsilon\rangle;\Gamma \vdash_I x : d\langle\phi_1, \alpha_3\rangle|\{(\ell(x), d\langle\phi_1, \alpha_3\rangle)\}}$$

$$\textsc{I-App}\ \frac{C_1;\Gamma \vdash_I e_1 : \phi_1|\Delta_1 \qquad C_2;\Gamma \vdash_I e_2 : \phi_2|\Delta_2 \qquad \beta \text{ fresh}}{C_1 \wedge C_2 \wedge \phi_1 \equiv^? \phi_2 \to \beta;\Gamma \vdash_I e_1\ e_2 : \beta|\Delta_1 \cup \Delta_2}$$

The judgment for the inference rules has the form $C;\Gamma \vdash_I e : \phi|\Delta$, read as: given Γ, e has the type ϕ with the generated constraint C and the change information Δ. All components are output except e and Γ. The I-Var rule is simple: the variable receives a choice type where the source type is an instantiation of the type schema and the target type is a fresh type variable. We use ε for the second alternative of the output constraint since that alternative is unconstrained. With these rules, we can generate the constraints shown in Sect. 2 for the example `abs True`.

4.1 Variational Context Reduction

Our algorithm in this subsection follows the idea of Jones [21] but has to deal with variations. Given a constraint, context reduction first transforms it into a head-normal form and then simplifies it with an entailment relation. We discuss them below in detail.

Constraint Transformation. We define the function $\mathit{toHnf}(C,Q)$ to transform C into the head-normal form with the given Q. A type class constraint is in head-normal form if at least one argument is a type variable. The result is (C_2,π), meaning that C_2 is the normal form for C but transformation was successful only in variants where π have $\top$s. When Q is ε, it means that no axiom is given. For example, $\mathit{toHnf}(A\langle\texttt{Num Bool},\texttt{Num Int}\rangle,\varepsilon)$ yields $(\varepsilon,A\langle\bot,\top\rangle)$, meaning that the transformation is failed in $A.1$ but is successful in $A.2$. The operation $(C_1,\pi_1)\ (op_1,op_2)\ (C_2,\pi_2)$ below is defined as $(C_1\ op_1\ C_2,\pi_1\ op_2\ \pi_2)$ and $o_1\ d\langle,\rangle\ o_2$ yields $d\langle o_1,o_2\rangle$ where o denotes any object.

$$\begin{aligned}
\mathit{toHnf}(C_1\wedge C_2,Q) &= \mathit{toHnf}(C_1,Q)\ (\wedge,\otimes)\ \mathit{toHnf}(C_2,Q)\\
\mathit{toHnf}(d\langle C_1,C_2\rangle,Q) &= \mathit{toHnf}(C_1,Q)(d\langle,\rangle,d\langle,\rangle)\mathit{toHnf}(C_2,Q)\\
\mathit{toHnf}(C,Q) &= \mathit{toHnf}'(\mathit{inHnf}(C),C,Q)
\end{aligned}$$

The operation $\otimes$ in the rule puts two patterns together. It is defined with three rules: (1) $\top\otimes\pi=\pi$, (2) $\bot\otimes\pi{=}\bot$, and (3) $d\langle\pi_1,\pi_2\rangle\otimes\pi=d\langle\pi_1\otimes\pi,\pi_2\otimes\pi\rangle$. $\otimes$ can be understood as logical **and** if we view $\top$ and $\bot$ as logical **true** and **false**, respectively.

When the constraint is a class constraint, *toHnf* delegates the real task to toHnf', which, in addition to C and Q, takes a ϖ, a *variational boolean value*, as its first argument. This argument indicates whether C is already normalized or not. It could be `True`, `False`, or a variation over ϖs.

$$\begin{aligned}
\mathit{toHnf}'(\texttt{True},C,Q) &= (C,\top)\\
\mathit{toHnf}'(d\langle\varpi_1,\varpi_2\rangle,C,Q) &= \mathit{toHnf}'(\varpi_1,\lfloor C\rfloor_{d.1},Q)(d\langle,\rangle,d\langle,\rangle)\ \mathit{toHnf}'(\varpi_2,\lfloor C\rfloor_{d.2},Q)\\
\mathit{toHnf}'(\texttt{False},C,Q) &= \begin{cases}(\varepsilon,\top) & \mathit{byInst}(C,Q)=(\varepsilon,\top)\\ \mathit{toHnf}(C_2,Q) & \mathit{byInst}(C,Q)=(C_2,\top)\\ (\varepsilon,\bot) & \text{otherwise}\end{cases}
\end{aligned}$$

toHnf' pattern matches against ϖ. When ϖ is `True`, it means that C is already normalized and C itself, together with a $\top$, is returned. Note that ϖ equals `True` doesn't mean that C is plain, but rather, all possible variants in C are normalized. When ϖ is a variational value, it means that at least some variant in C is not normalized. In this case, the constraint C is broken down into two constraints $\lfloor C\rfloor_{d.1}$ and $\lfloor C\rfloor_{d.2}$, which are normalized and the results are packed back using the choice d. When ϖ is `False`, then, due to how ϖ is computed, C must be plain. We call *byInst* to possibly reduce C using instance declarations.

There are three possible outcomes of *byInst*. First, *byInst* returns $(\varepsilon,\top)$, which means that C is successfully reduced with no new constraint generated. For example, $\mathit{byInst}(\texttt{Ord Bool},Q)$ belongs to this case if Q includes the default instance declarations in Haskell. Second, *byInst* returns $(C_2,\top)$, which means that C is successfully reduced to C_2, which is in turn normalized using *toHnf*. For example, $\mathit{byInst}(\texttt{Ord}\ [\alpha], Q)$ belongs to this case. Third, *byInst* returns $(\varepsilon, \bot)$, which means that no rule in Q can be used to reduce C and the reduction fails. For example, if Q includes Haskell default class instances, then $\mathit{byInst}(\texttt{Num Bool}, Q)$ belongs to this case. For each case, *toHnf′* returns corresponding values. Since the argument C to *byInst* is always plain, the definition of *byInst* is very similar to that in [21]. We omit its definition here.

The value ϖ for C is computed with the function *inHnf*. Thus, the results from different arguments are combined together through the $\oplus$ operation, which is defined below. In this section, we use the notation $d\langle\overline{o}\rangle$ to denote $d\langle o_1, o_2\rangle$ for any object o.

$$\mathit{inHnf}(\texttt{G}\ \overline{\phi}) = \bigoplus \overline{\mathit{hnf}(\phi)}$$

$$\texttt{True} \oplus \varpi = \texttt{True} \quad \texttt{False} \oplus \varpi = \varpi \quad d\langle\overline{\varpi}\rangle \oplus \varpi_3 = d\langle\overline{\varpi \oplus \varpi_3}\rangle$$

The following function $\mathit{hnf}(\phi)$ decides if ϕ is normalized. The notation $\texttt{True} \parallel d\langle\overline{\mathit{hnf}(\phi)}\rangle$ means that $d\langle\overline{\mathit{hnf}(\phi)}\rangle$ is simplified to True if both alternatives of d are True and is unchanged otherwise.

$$\mathit{hnf}(\alpha) = \texttt{True} \qquad \mathit{hnf}(\gamma) = \texttt{False} \qquad \mathit{hnf}(\phi_1 \to \phi_2) = \texttt{False} \qquad \mathit{hnf}(d\langle\overline{\phi}\rangle) = \texttt{True} \parallel d\langle\overline{\mathit{hnf}(\phi)}\rangle$$

Algorithmic Constraint Entailment. We define the function $\mathit{entail}(C_1, C_2, Q)$ to decide whether C_2 is entailed by C_1. In [21], *entail* returns a boolean value. However, here we need a variational boolean value, reflecting that entailment may only hold for certain variants. For example, $\mathit{entail}(A\langle\texttt{Num}\ \alpha, \texttt{Ord}\ \alpha\rangle, \texttt{Ord}\ \alpha, Q)$ yields $A\langle\texttt{False}, \texttt{True}\rangle$, indicating that the entailment relation holds only in $A.2$.

Since a normalized constraint can not be simplified by instance declarations anymore, our definition of *entail* is quite simple: just checking if C_2 belongs to the super classes of C_1, as expressed below.

$$\begin{aligned}
\mathit{entail}(C_1, C_2, Q) &= \mathit{belong}(C_2, \mathit{bySuper}(C_1, Q)) \\
\mathit{bySuper}(d\langle\overline{C}\rangle, Q) &= d\langle\overline{\mathit{bySuper}(C, Q)}\rangle \\
\mathit{bySuper}(C_1 \wedge C_2, Q) &= \mathit{bySuper}(C_1, Q) \wedge \mathit{bySuper}(C_2, Q) \\
\mathit{bySuper}(\texttt{G}\ \overline{\phi}, Q) &= \texttt{G}\ \overline{\alpha} \wedge\ \mathit{bySuper}(C_G, Q) \quad \text{where } \forall\overline{\alpha}.C_G \Leftarrow \texttt{G}\ \overline{\alpha} \in Q
\end{aligned}$$

The definition of *bySuper* is quite simple. In the third case we find super classes for constraints like $\texttt{G}\ \overline{\phi}$, which can just be simplified to a sequence of type variables. Given C_1 and C_2, *belong* returns True if they are the same and False if they are primitive and have different class names. We omit the definition of *belong* here because it is quite simple. For plain class constraints, a simple equality testing is enough, and for variational class constraints, the definition recurse into their alternatives. When the second argument of *belong* is a $\wedge$ over two constraint operands, *belong* returns True if the first argument of *belong* belongs to any operand.

With *entail*, we can now define the simplification operation as follows. $\mathit{simp}(C_1, C_3, Q)$ simplifies the constraint C_1 with the already simplified constraint C_3

and the axiom $\mathcal{Q}$.

$$simp(\varepsilon, C_3, \mathcal{Q}) = C_3 \qquad simp(d\langle\overline{C}\rangle, C_3, \mathcal{Q}) = d\langle\overline{simp(C, C_3, \mathcal{Q})}\rangle$$
$$simp(C_1 \wedge C_2, C_3, \mathcal{Q}) = simp(C_2, entail(C_2 \wedge C_3, C_1, \mathcal{Q}) \lhd C_1, \mathcal{Q})$$

When C_1 becomes ε, *simp* terminates and returns C_3. For a variational constraint, *simp* simplifies its individual alternatives. To simplify the compound constraint $C_1 \wedge C_2$, we check to see if we can simplify each of C_1 and C_2 by the other. Formally, we first decide the result of $\varpi = entail(C_2 \wedge C_3, C_1, \mathcal{Q})$. If the result is `True`, then C_1 can simply be dropped. However, the entailment may hold in certain variants only, and we can not drop the whole C_1 in this case. We handle this situation by replacing each variants in C_1 whose ϖ is `True` with a ε and leaving other variants unchanged. This process is defined through the operation $\varpi \lhd C_1$. Its definition is quite simple and is omitted here.

Overall, with *toHnf* and *simp*, context reduction for C under the axiom $\mathcal{Q}$ is defined as $(simp(C_1, \varepsilon, \mathcal{Q}), \pi)$, where $(C_1, \pi) = toHnf(C, \mathcal{Q})$.

5 Evaluation

Following the ideas in this paper, we have implemented a prototype of TEC. The prototype supports functional dependencies using the idea from [20]. Our prototype is implemented into Helium [17] rather than GHC due to the overhead of implementing it into the latter. Our implementation generates constraints and solves them. During constraint generation, a Δ is generated to record location information and choices created. Constraint solving will update Δ with concrete type information. Type error messages are generated from Δ, the typing pattern from constraint solving, and the result type of the expression, following the method described in Sect. 2.

TEC reuses the heuristics in counter-factual typing [4] for ranking all the error fixes calculated and introduces two more heuristics. First, we favor fixes whose consequent types have lower ratios of unification variables. Given a type, the ratio is defined as the number of unification variables over the number of primitive types. The rationale of this rule is that less internal information should be leaked to the outside of the ill-typed expression. Second, we prefer fixes whose source type and target type have closer class constraints. Two class constraints are closer if they have a similar number of primitive constraints or share a class hierarchy. The rationale of this rule is to avoid exotic changes related to type classes.

Error Locating Precision. To evaluate the precision of TEC, we created two benchmarks. The first is created by taking all examples involving type classes from all the papers [33,34,39] in the literature (We do not include class directive examples from [17] since none of TEC, GHC, and SHErrLoc supports them). This benchmark contains 17 programs. The second benchmark is extracted from student programs [36], which were logged as undergraduate students in an introductory Haskell course were attempting their homework assignments. All intermediate program versions were logged. We obtained about 5000 programs in total, filtered out all programs that have type errors related to type classes, and then randomly chose 60 from them.

We next investigate how different tools perform in the presence of type classes. We consider TEC, GHC 8.10.6, and SHErrLoc. The precisions of these tools for Benchmarks 1 and 2 are shown in the following table. In both benchmarks, the first several messages are already correct and later messages (after 2 messages in Benchmark 1 and 3 in Benchmark 2) do not help.

Tool	Precision (%) after # of msgs (Bench. 1)			Precision (%) (Bench. 2)			
	1	2	> 2	1	2	3	> 3
TEC	65	88	88	55	72	78	78
SHErrLoc	47	59	59	42	58	63	63
GHC	41	53	53	33	40	40	40

Figure 5 presents the result for Benchmark 1 in more detail. A tool receives a filled circle if the first two messages from the tool help to locate the real error cause and an unfilled circle otherwise. Some examples from the paper contain an accompanying "oracle" stating how to remove the type error. For these examples, we compare reported error message against the oracles. For other messages, no oracles are given. Since there are usually many different causes for a type error, a message is regarded as correct if it points to at least one error cause. If the message is not helpful at all, then it is classified as incorrect. One such instance is that SHErrLoc says that Example 3 [34] is well-typed while in fact it is ill-typed, and the message of SHErrLoc is regarded as incorrect.

Both GHC and SHErrLoc are very good at locating type errors when the type annotations do not subsume the real inferred types. Both of them are successful for all the 6 examples of this kind. TEC also performs quite well for this kind. It can suggest a correct type annotation as well as find fixes in expressions so that the signature becomes valid. GHC also performs well in locating type errors violating functional dependencies when involved literals (for example `'1'` and `True`) have constant types (for example `Char` and `Bool`). However, when the involved literals have constrained types, the messages are always not helpful (for example the message for `insert2`). TEC always works well since it always finds an alternative type for the erroneous expression. SHErrLoc doesn't work well for functional dependencies. For other examples, TEC and SHErrLoc work better than GHC. The Example 6 [33] requires one to report that `Integral` and `Fractional` has no overlapping instances, and none of the evaluated tools are able to report this.

The following function demonstrates the shortcoming of TEC.

```
class Collects ce e where insert :: e -> ce -> ce

insert3 :: Collects ce Bool => ce -> ce
insert3 c = insert 1 (insert False c)
```

The type error is due to the mismatch between the type `1` and the `Bool` in the type signature. TEC doesn't work well because the target type α_4 of `1` gives rise to the wanted constraint `Collects ce` α_4, which can not be deduced from the given constraint `Collects`

Tool	Example # [33]		Example # [34]							Example # [39]					This paper		
	5	6	2	3	6	10	12	13	14	3	6	38	70	73	rank1	insert2	insert3
TEC	●	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○
SHErrLoc	●	○	●	○	●	○	●	●	●	○	●	○	●	●	●	○	○
GHC	●	○	●	○	●	●	●	●	○	○	●	○	○	●	○	○	●

Fig. 5. A comparison of TEC, SHErrLoc and GHC for examples from the literature. Filled and unfilled circles denote that the first two messages from the corresponding tool are useful and not useful, respectively.

`ce Bool`. Moreover, suggesting `(Collects ce Bool, Collects ce` α_4`) => ce -> ce` as the type signature is incorrect since this type is ambiguous [39] as α_4 doesn't appear in `ce -> ce`. As a result, we can not identify `1` as an error cause. One way to fix this problem is to unify the wanted constraint with the given constraint if the deduction fails. However, it's unclear how to systematically apply this idea and we leave it for future work. SHErrLoc again reports that `insert3` is well-typed. This example demonstrates that the folklore knowledge [3,25] about the completeness of error debugging by searching does not hold when the type system extends to type classes. Note, this is not inconsistent with Theorem 2 because this happens when the condition $Q;\Gamma;\delta \vdash_{ALT} e : \tau$ of the Theorem does not hold.

Benchmark 2, among 60 programs, contains 13 programs where type errors are due to missing or extra parentheses. No existing tools are known to handle such errors well. The programs for which SHErrLoc doesn't perform well are those that are ill-typed but SHErrLoc reports that they are well-typed. GHC doesn't work well for programs where no type annotations are present: the reported locations are quite far away from the real error causes. TEC works well for both situations, and achieves a better precision. In this benchmark, we did not observe any program like `insert3` that TEC could not find an error fix. In the future, we will investigate how often programs like `insert3` happen in practice.

From the evaluation results of these benchmarks, we conclude that TEC is quite effective in locating the real error causes. We have also investigated the running time of TEC in detail. However, due to space limitation, we do not elaborate further. Briefly, the response time of TEC for programs around 100 LOC is about 1.6 s and that for about 300 LOC is 5.7 s. The reason that TEC has a long response time is that it first calculates all possible fixes (including changing 1 location, 2 locations, and up to all locations) at once before they are ranked. In the future, we will work on improving the response time by computing typing results in phases such that fixes that change only one or two locations will be returned and ranked before computing other fixes. This strategy works because in fixing type errors users prefer to change fewer locations.

6 Related Work

Approaches Supporting Type Classes. Besides GHC, SHErrLoc [42,43], Chameleon [33,34,39], and the Top framework [17] also debug type errors involving type class constraints. SHErrLoc is a general method for locating static inconsistencies, including type errors. SHErrLoc treats many Haskell features, such as type classes, type families, and type signatures. Given a program, it works by calling GHC to generate typing constraints, builds a graph representation from the constraints, and then locates the most likely error cause using a Bayesian model. Chameleon is an extensible framework that supports HM and type classes. Given an ill-typed program, it first expresses typing constraints in constraint handling rules (CHRs) and then finds the minimal unsolvable constraints and reports the corresponding program source. Chameleon can also explain why a type error happens while our approach gives concrete messages for removing the type error. In this sense, they are complementary to each other. Another difference is that our approach finds all error causes while Chameleon doesn't.

Top aims to generate high quality error messages. It supports type classes through overloading of identifiers. A main feature of Top is that it uses type inference directives [16] and type class directives [15] to customize error messages. Directives allow certain type errors to be detected earlier. TEC doesn't support directives but focuses on how to change expressions to, for example, avoid class constraints, satisfy class constraints, and make context reduction successful. While TEC generates informative messages at all error causes, Top does so for several, usually one or two, locations only. Therefore, these two approaches are again complementary.

Other Approaches. Due to the large body of previous work in error debugging, we are able to only give a brief discussion. This work is similar to CFT [4] but we deal with type errors in type class uses while that does not. As type classes are prevalent in Haskell, we believe that this difference is significant. This work can be viewed as searching for type error fixes at the type level, and [37] searches error fixes at the expression level. Thus, while this work can reuse typing processes to scale to large programs, the scalability of that work is unclear. Also, the error messages from this work contains more information than that work. [4] discussed the relation between CFT and discriminative sum types for locating type errors [27]. As the difference between our work and [27] is similar, we will not discuss that work further in this paper.

It has been a long history of locating the most likely error cause in the research community. The first approach, developed by [18], is based on maximal flow of the graph representation of typing constraints. Essentially, it could be understood as majority voting. Numerous approaches based on reordering the unification process have been developed, including algorithms $\mathcal{M}$ [24], $\mathcal{G}$ [12], and the symmetrical versions of $\mathcal{W}$ and $\mathcal{M}$ [26]. Recent developments include dedicated heuristics [14] and a Bayesian model [42] for locating the most likely error cause.

Instead of just finding the mostly likely error causes, many error slicing approaches have been developed [13,31,35], which highlight all program text that contributes to the type error. The shortcoming of error slicing approaches, as noted in [17], is that they usually cover too many locations, and give no hint about which location more likely causes the type error. The approach by [29,30] takes the advantages of approaches locating most likely error causes and those slicing type errors in that it can find all error causes and iteratively presents the most likely one. TEC takes this approach one step further of providing an informative message for each error cause. A very different line of research explains type errors from various perspectives [1,9,11,32,38,40].

7 Conclusions

We have presented TEC, an efficient approach for debugging type errors in type systems with type classes and its extensions of multi-parameter classes and functional dependencies. For most expressions, TEC finds error causes in all possible combinations of leaves of the program AST and generates an informative error message for each error cause. We have evaluated TEC and the result shows that it can locate type errors quite precisely.

In some rare cases, TEC fails to find complete error fixes when the generalization of an error fix causes both context reduction failure and ambiguous types due to class usage. In the future, we will investigate how we can systematically fix this problem and integrate our solution into TEC. In the future, we also plan to perform a user study to find out how well TEC helps students in fixing type errors, such as how many type errors students investigate for fix type errors and whether TEC helps shorten students' type error debugging time. Finally, we plan to collect users' feedback for fine tuning our heuristics for ranking error fixes.

References

1. Beaven, M., Stansifer, R.: Explaining type errors in polymorphic languages. ACM Lett. Program. Lang. Syst. **2**, 17–30 (1994)
2. Bernstein, K.L., Stark, E.W.: Debugging Type Errors. State University of New York at Stony Brook, Technical report (1995)
3. Braßel, B.: Typehope: there is hope for your type errors. In: International Workshop on Implementation of Functional Languages (2004)
4. Chen, S., Erwig, M.: Counter-factual typing for debugging type errors. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 583–594 (2014)
5. Chen, S., Erwig, M.: Guided type debugging. In: Codish, M., Sumii, E. (eds.) FLOPS 2014. LNCS, vol. 8475, pp. 35–51. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-07151-0_3
6. Chen, S., Erwig, M., Walkingshaw, E.: An error-tolerant type system for variational lambda calculus. In: ACM International Conference on Functional Programming, pp. 29–40 (2012)
7. Chen, S., Erwig, M., Walkingshaw, E.: Extending type inference to variational programs. ACM Trans. Program. Lang. Syst. **36**(1), 1:1–1:54 (2014)
8. Chen, S., Noor, R.: Improving Type Error Reporting for Type Classes. Technical report, UL Lafayette (2022). https://people.cmix.louisiana.edu/schen/ws/techreport/longtec.pdf
9. Chitil, O.: Compositional explanation of types and algorithmic debugging of type errors. In: ACM International Conference on Functional Programming, pp. 193–204, September 2001
10. Choppella, V.: Unification source-tracking with application to diagnosis of type inference. Ph.D. thesis, Indiana University (2002)
11. Duggan, D., Bent, F.: Explaining type inference. Sci. Comput. Program. **27**, 37–83 (1995)
12. Eo, H., Lee, O., Yi, K.: Proofs of a set of hybrid let-polymorphic type inference algorithms. New Gener. Comput. **22**(1), 1–36 (2004). https://doi.org/10.1007/BF03037279
13. Haack, C., Wells, J.B.: Type error slicing in implicitly typed higher-order languages. In: Degano, P. (ed.) ESOP 2003. LNCS, vol. 2618, pp. 284–301. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36575-3_20
14. Hage, J., Heeren, B.: Heuristics for type error discovery and recovery. In: Horváth, Z., Zsók, V., Butterfield, A. (eds.) IFL 2006. LNCS, vol. 4449, pp. 199–216. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74130-5_12

15. Heeren, B., Hage, J.: Type class directives. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2005. LNCS, vol. 3350, pp. 253–267. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30557-6_19
16. Heeren, B., Hage, J., Swierstra, S.D.: Scripting the type inference process. In: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, pp. 3–13. ACM, New York (2003). https://doi.org/10.1145/944705.944707
17. Heeren, B.J.: Top quality type error messages. Ph.D. thesis, Universiteit Utrecht, The Netherlands (2005)
18. Johnson, G.F., Walz, J.A.: A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In: ACM Symposium on Principles of Programming Languages, pp. 44–57 (1986)
19. Jones, M.P.: A theory of qualified types. In: Krieg-Brückner, B. (ed.) ESOP 1992. LNCS, vol. 582, pp. 287–306. Springer, Heidelberg (1992). https://doi.org/10.1007/3-540-55253-7_17
20. Jones, M.P.: Simplifying and improving qualified types. In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, FPCA 1995, pp. 160–169. ACM, New York (1995). https://doi.org/10.1145/224164.224198
21. Jones, M.P.: Typing Haskell in Haskell. In: Haskell Workshop (1999)
22. Jones, M.P.: Type classes with functional dependencies. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 230–244. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-46425-5_15
23. Lee, O., Yi, K.: Proofs about a folklore let-polymorphic type inference algorithm. ACM Trans. Program. Lang. Syst. **20**(4), 707–723 (1998)
24. Lee, O., Yi, K.: A generalized let-polymorphic type inference algorithm. Technical report, Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology (2000)
25. Lerner, B., Flower, M., Grossman, D., Chambers, C.: Searching for type-error messages. In: ACM International Conference on Programming Language Design and Implementation, pp. 425–434 (2007)
26. McAdam, B.J.: Repairing type errors in functional programs. Ph.D. thesis, University of Edinburgh. College of Science and Engineering. School of Informatics (2002)
27. Neubauer, M., Thiemann, P.: Discriminative sum types locate the source of type errors. In: ACM International Conference on Functional Programming, pp. 15–26 (2003)
28. Odersky, M., Sulzmann, M., Wehr, M.: Type inference with constrained types. Theor. Pract. Object Syst. **5**(1), 35–55 (1999)
29. Pavlinovic, Z., King, T., Wies, T.: Finding minimum type error sources. In: ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2014, pp. 525–542. ACM, New York (2014). https://doi.org/10.1145/2660193.2660230
30. Pavlinovic, Z., King, T., Wies, T.: Practical SMT-based type error localization. In: ACM SIGPLAN International Conference on Functional Programming, pp. 412–423 (2015)
31. Schilling, Thomas: Constraint-free type error slicing. In: Peña, R., Page, R. (eds.) TFP 2011. LNCS, vol. 7193, pp. 1–16. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32037-8_1
32. Seidel, E.L., Jhala, R., Weimer, W.: Dynamic witnesses for static type errors (or, ill-typed programs usually go wrong). In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, pp. 228–242. ACM, New York (2016). https://doi.org/10.1145/2951913.2951915
33. Stuckey, P.J., Sulzmann, M., Wazny, J.: Interactive type debugging in Haskell. In: ACM SIGPLAN Workshop on Haskell, pp. 72–83 (2003)

34. Stuckey, P.J., Sulzmann, M., Wazny, J.: Improving type error diagnosis. In: ACM SIGPLAN Workshop on Haskell, pp. 80–91 (2004)
35. Tip, F., Dinesh, T.B.: A slicing-based approach for locating type errors. ACM Trans. Softw. Eng. Methodol. **10**(1), 5–55 (2001)
36. Tirronen, V., Uusi-Mäkelä, S., Isomöttönen, V.: Understanding beginners' mistakes with Haskell. J. Funct. Program. **25**, e11 (2015). https://doi.org/10.1017/S0956796815000179
37. Tsushima, K., Chitil, O., Sharrad, J.: Type debugging with counter-factual type error messages using an existing type checker. In: IFL 2019: Proceedings of the 31st Symposium on Implementation and Application of Functional Languages, ICPS. ACM, New York (2020). https://kar.kent.ac.uk/81976/
38. Wand, M.: Finding the source of type errors. In: ACM Symposium on Principles of Programming Languages, pp. 38–43 (1986)
39. Wazny, J.R.: Type inference and type error diagnosis for Hindley/Milner with extensions. Ph.D. thesis, The University of Melbourne (2006)
40. Yang, J.: Explaining type errors by finding the source of a type conflict. In: Trends in Functional Programming, pp. 58–66. Intellect Books (2000)
41. Yang, J.: Improving Polymorphic Type Explanations. Ph.D. thesis, Heriot-Watt University (2001)
42. Zhang, D., Myers, A.C.: Toward general diagnosis of static errors. In: ACM Symposium on Principles of Programming Languages, pp. 569–581 (2014)
43. Zhang, D., Myers, A.C., Vytiniotis, D., Peyton-Jones, S.: Diagnosing type errors with class. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 12–21 (2015)