Efficient Counter-factual Type Error Debugging*

Sheng Chen and Baijun Wu
The Center for Advanced Computer Studies, UL Lafayette
Lafayette, USA
{chen, bj.wu}@louisiana.edu

***Abstract*—Type inference is an important part of functional programming languages and has been increasingly adopted to imperative programming. However, providing effective error messages in response to type inference failures (due to type errors in programs) continues to be a challenge. Type error messages generated by compilers and existing error debugging approaches often point to bogus error locations or lack sufficient information for removing the type error, making error debugging ineffective. Counter-factual typing (CFT) addressed this problem by generating comprehensive error messages with each message includes a rich set of information. However, CFT has a large response time, making it too slow for interactive use. In particular, our recent study shows that programmers usually have to go through multiple iterations of updating and recompiling programs to remove a type error. Interestingly, our study also reveals that program updates are minor in each iteration during type error debugging. We exploit this fact and develop *e*CFT, an efficient version of CFT, which doesn't recompute all error fixes from scratch for each updated program but only recomputes error fixes that are changed in response to the update. Our key observation is that minor program changes lead to minor error suggestion changes. *e*CFT is based on principal typing, a typing scheme more amenable to reuse previous typing results. We have evaluated our approach and found it is about $12.4\times$ faster than CFT in updating error fixes.**

I. Introduction

Type inference allows programs to be statically typed, even without the presence of type annotations. A well-known problem in type inference is that it is very hard to locate the real error cause and generate informative feedback once type inference fails. Practical compilers pay little attention to address this problem. They usually report the place where type inference first fails as the error cause and often report errors in their internal jargon. As a result, understanding type error messages is a main challenge in learning functional programming [29].

This problem has also been intensively studied over the last three decades from different directions. One direction aims to find the most likely error causes [10], [15], [17], [21], [31]. As an example, consider the following ill-typed expression.[1]

$$\texttt{rank} = \lambda x.(x\ \texttt{'1'}, x\ \texttt{True})$$

Haskell compilers like GHC 8.0.2 and Helium [16] blame `True` as the error cause. While changing `True` may remove the type error, this is not the only possible fix. In fact, changing any of `x` (either occurrence), `'1'`, or `True` may remove the type error. We lack enough context to justify `True` is more likely the error source than other locations.

This example demonstrates the value of type error slicing [13], [26], which returns all program locations that may contribute to the type error and excludes those don't. However, a problem with this approach is that programmers still have to decide the real error cause among the returned slice, which could be comparable to the original program in size [3], [17]. Recently, Pavlinovic et al. [24] improved this by finding all possible error causes and suggesting one location at a time.

Like error slicing, counter-factual typing (CFT) [3] also finds all possible error locations in the leaves and their combinations of the program AST. However, unlike them, CFT also comes with a change suggestion for each identified location. This suggestion includes the type the identified location has in the original program, the type the identified location ought to have to remove the type error, and the result type of the changed expression if the suggestion is applied. Since some locations are more likely to be the error source than others in most common cases, CFT ranks all fixes with a list of heuristics and presents them iteratively. The evaluation result showed that CFT achieved better precision than state-of-the-art approaches when considering the first suggestions and performed even better when considering also later suggestions [3].

One problem with CFT, however, is the long response time. To find all possible error locations and change suggestions, CFT requires intensive computations. Although CFT uses variational typing [6] to reuse typing results, it still takes dozens of seconds to deliver the first error message for the programs within 100 LOC. This makes CFT slow for interactive use. In particular, our recent study of mining a program database [7], [14], [30] shows that in average students take about 29 steps to fix a type error with a maximum of 359 steps.

Fortunately, an accompanying finding of the study is that, during error debugging, the change between two consecutive versions is minor. In more than 80% cases, the change is within 10% of the old program. This result suggests to compute error fixes incrementally rather than recompute all error fixes from scratch as programs are updated.

In this work, we develop *e*CFT, an efficient version of CFT. Specifically, let P_i be the ith version of a program used in compilation, F_i be the set of all error fixes for P_i, $\Delta_{P_{ij}}$ be the difference between P_i and P_j, and $\Delta_{F_{ij}}$ be the difference between F_i and F_j, CFT recomputes all error fixes for F_j

*This work is supported by the NSF grant CCF-1750886.

[1]This paper uses notations from functional programming, which, for example, supports higher-order functions and uses spaces to denote function applications. We will also use constant values and functions, such as `True`, `succ`, `not`, and `odd`, that have self-explanatory meanings.

from P_j while *e*CFT computes $\Delta_{F_{ij}}$ based on $\Delta_{P_{ij}}$ and then merges $\Delta_{F_{ij}}$ into F_i to get F_j.

In Summary, this paper makes the following contributions after we present the background in Section II.

1) *e*CFT relies on variational typing (Section II) to find all error fixes. However, previous presentations of variational typing itself [5], [6] and its applications [3], [4] are very operational, making the type systems hard to understand and prove. In Section III, we present our first technical innovation of a declarative specification of variational typing in the presence of type errors, which simplifies the type system of *e*CFT and further applications of variational typing.
2) We present the typing rules of *e*CFT in Section IV. A subtle issue in the type system is about dealing with unbound variables, which we handle nicely with the above contribution.
3) We present three different strategies of reusing previous results to compute error fixes under program updates in Section V. Among them one relies on incremental variational unification (Section VI), our second technical innovation in this paper.
4) We have extensively evaluated the performance of *e*CFT. The result shows that in more than 80% cases *e*CFT is 12.4× faster than CFT of computing error fixes in response to program updates.

.

II. Variational Typing and Principal Typing

Variational Typing As already mentioned in Section I, both CFT and *e*CFT rely on variational typing [6] to compute informative error messages for all possible error locations. This section presents variational typing and principal typing.

Variational expressions are obtained by extending normal expressions (*plain* expressions) with named *choices* [12]. For example, the expression `e = succ` $A\langle$`1,'a'`$\rangle$ contains a choice A with two alternatives: `1` and `'a'`. We use d to range over choice names. In this paper, we use only binary choices.

An important notion in variation representations is *selector*s that have the form $d.i$, where d is a choice name and i is an alternative index. Choices can be eliminated through a process called *selection*, which takes in an expression e and a selector $d.i$ and replaces each occurrence of the choice d in e with its ith alternative. We call a set of selectors a decision and use δ to range over decisions. Selection extends naturally to decisions by iteratively selecting with all of the selectors in the decision. We write $\lfloor e \rfloor_{d.i}$ and $\lfloor e \rfloor_{\delta}$ for selections. For example, $\lfloor$`succ` $A\langle$`1,'a'`$\rangle \rfloor_{A.1}$ yields `succ 1`.

The notions and definitions of variational expressions carry over naturally to variational types. We use τ and ϕ to range over plain types and variational types, respectively.

Note that a variational program usually has thousands of independent choices [6], and the number of plain programs is exponential in the number of different choices. Therefore, it is impractical to individually type all the plain programs generated from a variational program. A more scalable way is variational typing, which types variational programs once without generating plain programs. The key idea of variational typing is reuse, and we identified three opportunities in [6] for reusing typing information.

In variational typing, one challenge is putting choice types together. For example, the expression `odd` $A\langle$`1,2`$\rangle$ seems to be ill typed at first since the argument type `Int` is not equal to the type of the argument $A\langle$`Int,Int`$\rangle$. However, we can check that both plain expressions from the variational expression are well typed. Thus, it is reasonable to require `odd` $A\langle$`1,2`$\rangle$ itself to be well typed. We satisfied this requirement by relaxing the equality relation in standard typing rules [25] to an equivalence relation [6]. Intuitively, two types are equivalent if they generate the same set of plain types. Thus, it is obvious that `Int` is equivalent to $A\langle$`Int,Int`$\rangle$. We use $\phi_1 \equiv \phi_2$ to denote that ϕ_1 and ϕ_2 are equivalent. We defer a full discussion of type equivalence to [6]. With this relation, the expression `odd` $A\langle$`1,2`$\rangle$ has the type `Int`.

Variational typing assigns types to expressions that generate only well-typed plain expressions. For example, it fails to assign a type to the expression `odd` $A\langle$`1,True`$\rangle$ since the plain expression `odd True` is ill typed. In practice, it's very useful for variational typing to assign types to the well-typed variants of a variational program even type errors exist in other variants. We addressed this problem by designing an error-tolerant type system [5], where type errors are represented explicitly by $\perp$ and variants that contain type errors receive this type [5]. For example, `odd` $A\langle$`1,True`$\rangle$ has the type $A\langle$`Bool`$,\perp\rangle$, indicating that `odd 1` has the type `Bool` and `odd True` is ill typed.

The typing rule for function applications was very complicated [5] since applications can introduce type errors in many ways. In particular, the rule has to propagate type errors from both the function and argument types and generate errors when the argument type fails to match the type of the argument exactly. Unlike standard typing rules [25] that are declarative, the rule in [6] is operational, relying on three operations that decompose type structures and introduce error types explicitly. In Section III, we propose a declarative formulation of error-tolerant type systems.

Principal Typing In the Hindley-Milner type system (HM) and its implementation, the algorithm $\mathcal{W}$ [22], type environment is an input. The type environment stores type information for free variables, and is updated accordingly as the inference algorithm traverses the program AST. As a result, type inference of the later part of the AST always depends on that of the earlier part. This bias hinders incremental type inference since even a small change in the left subtree will require almost the full type inference to be redone.

In contrast, principal typing [18] doesn't suffer from this bias. Type inference in principal typing is done bottom-up. At each variable reference, the variable is assumed to have a fresh type. The assumptions are refined as the inference gets closer to the root and are made consistent at the corresponding abstraction. Principal typing is more amenable to incremental

typing. If there is a change in a leaf, then there is no need to perform type inference for the whole tree but only for the nodes along the path from the updated leaf to the root. We will design *e*CFT based on principal typing, which has already been used in incremental type checking [2], [11], [18], [27].

III. Compute All Fixes with Variational Typing

Declarative Error-Tolerant Variational Typing Early work of variational typing [3]–[5] dealt with type errors by explicitly representing and propagating them, making the understanding and proving of relevant type systems difficult. We address this issue by using *typing patterns* to indicate which variants of the typing result are correct and which are incorrect. A typing pattern π consists of $\bot$ for ill-typed variants, $\top$ for well-typed variants, and choice patterns for variational expressions. For example, $A\langle\top,\bot\rangle$ means that the result for the alternative $A.1$ is correct while that for $A.2$ is incorrect, and $A\langle B\langle\bot,\top\rangle,\top\rangle$ means that the result in the variant $\{A.1, B.1\}$ is incorrect and those in other variants are correct.

With π, the type judgments have the form $\pi;\Gamma\vdash e:\phi$, meaning that e has the type ϕ under Γ with the validity restriction π. Intuitively, π specifies that only the variants where π contains $\top$s are valid (type correct) and those that π contains $\bot$s are invalid (type incorrect). When we later use the typing result, we should ignore the variants whose patterns are $\bot$s. For example, we have $\top;\Gamma\vdash \texttt{1}:\texttt{Int}$, saying that `1` has the type `Int`. Similarly, we have $\bot;\Gamma\vdash \texttt{1}:\texttt{Bool}$, which says that the typing result `1` has the type `Bool`. However, the pattern $\bot$ in the judgment indicates that we should ignore this typing result.

With the extended judgment form, the rule for typing function applications can be formalized as follows.

$$\pi ::= \bot \mid \top \mid d\langle\pi,\pi\rangle \qquad \frac{\forall\delta:\lfloor\pi\rfloor_\delta=\top\Rightarrow\lfloor\phi_1\rfloor_\delta\equiv\lfloor\phi_2\rfloor_\delta}{\phi_1\equiv_\pi\phi_2}$$

$$\frac{\pi;\Gamma\vdash e_1:\phi_1 \qquad \pi;\Gamma\vdash e_2:\phi_2 \qquad \phi_1\equiv_\pi\phi_2\to\phi}{\pi;\Gamma\vdash e_1\ e_2:\phi}$$

The typing rule is declarative and simple. It makes only two extensions to any standard typing rules for applications [25]: the type equivalence relation (introduced in Section II) and the typing pattern constrained judgments (introduced above). The rule can be read as: if the function e_1 has the type ϕ_1, the argument e_2 has the type ϕ_2, and ϕ_1 is equivalent to $\phi_2\to\phi$, all under the validity restriction π, then the application $e_1\ e_2$ has the type ϕ under the same π. The pattern π in $\phi_1\equiv_\pi\phi_2$ means that ϕ_1 and ϕ_2 are required to be equivalent only in the variants that π has $\top$s. Based on this new rule, we can type $A\langle\texttt{succ},\texttt{odd}\rangle\ B\langle\texttt{1},\texttt{True}\rangle$ as follows, where $\pi = B\langle\top,\bot\rangle$.

$$\pi;\varnothing\vdash A\langle\texttt{succ},\texttt{odd}\rangle:\texttt{Int}\to A\langle\texttt{Int},\texttt{Bool}\rangle$$
$$\pi;\varnothing\vdash B\langle\texttt{1},\texttt{True}\rangle:\texttt{Int}$$
$$\texttt{Int}\to A\langle\texttt{Int},\texttt{Bool}\rangle\equiv_\pi\ \texttt{Int}\to A\langle\texttt{Int},\texttt{Bool}\rangle$$
$$\pi;\varnothing\vdash A\langle\texttt{succ},\texttt{odd}\rangle\ B\langle\texttt{1},\texttt{True}\rangle:A\langle\texttt{Int},\texttt{Bool}\rangle$$

Expressions	e	::=	$c \mid x \mid \lambda x.e \mid e\ e$
Monotypes	τ	::=	$\gamma \mid \alpha \mid \tau\to\tau$
Variational types	ϕ	::=	$\tau \mid d\langle\phi,\phi\rangle \mid \phi\to\phi$
Type assumption sets	$\mathcal{A}$	::=	$\varnothing \mid \mathcal{A},(x,\phi,d)$
Substitutions	θ	::=	$\varnothing \mid \theta,\alpha\mapsto\phi$
Choice environments	Δ	::=	$\varnothing \mid \Delta,(l,d\langle\phi,\phi\rangle)$

Fig. 1: Syntax of expressions, types, and environments

Finding All Error Fixes with Variational Typing When an expression is ill typed, we generally ask two questions: Which subexpression caused the type error and how should we change the subexpression to remove the type error? Conceptually, *e*CFT addresses these problems in following steps: (1) assuming that all subexpressions may be the error causes, (2) computing the types that subexpression ought to have to remove the type error, and (3) finding real error causes by filtering out subexpressions the types they have differing from those they ought to have. However, implementing this idea directly seems to be very complex: for each of all subexpressions and their possible combinations, we have to perform type inference of the expression to find out the type the subexpression ought to have.

To combat with this high complexity, *e*CFT employs variational typing to reuse computations. Specifically, it creates at AST leaf a variational type, where the first alternative is the type of the leaf under normal type inference and the second alternative is the type the leaf ought to have to remove the type error in the whole expression. Our previous work [3] explained why CFT considered changing leaves only and how it achieved high precision. After variational typing finishes, we derive error messages from variants whose patterns are $\top$s. Note as we said earlier, we should ignore variants whose patterns are $\bot$s. We then use a list of heuristics to rank all error fixes and present most likely fixes to the user iteratively [3].

IV. Type System

In this section, we present the type system for producing a complete set of error fixes. The syntax is given in Section IV-A, basic typing rules are discussed in Section IV-B, and the top-level rule, which also handles unbound variables, is given in Section IV-C.

A. Syntax

The syntax for types, expressions, and meta environments is given in Figure 1. We use c to denote constants and x to denote variables. We stratify the type definition into two layers. First, monotypes, ranged over by τ, include constant types (γ), type variables (α), and function types. In addition to α, we use β to denote fresh type variables used for representing function application results and use κ to denote fresh type variables generated during unification. Variational types extend monotypes with choice types. We also deal with polymorphism and conditionals, but omit their presentations in this paper due to space limitations. They could be found in [7].

We use $\overline{o}$ to denote a sequence of objects $o_1,\ldots,o_n$ for any object o. We use $\theta(\phi)$ to denote the application of a

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

$$\textsc{Var}\ \frac{\exists\ d}{\pi; \{(x, \phi)\} \vdash x : d\langle\phi, \phi_1\rangle | \{(\ell(x), d\langle\phi, \phi_1\rangle)\}}$$

$$\textsc{Abs}\ \frac{\pi; \mathcal{A} \vdash e : \phi | \Delta \qquad \forall \phi_i \in \mathcal{A}^T(x) \colon \phi_1 \equiv_\pi \phi_i}{\pi; \mathcal{A} \setminus x \vdash \lambda x.e : \phi_1 \to \phi | \Delta}$$

$$\textsc{App}\ \frac{\pi; \mathcal{A}_1 \vdash e_1 : \phi_1 | \Delta_1 \quad \pi; \mathcal{A}_2 \vdash e_2 : \phi_2 | \Delta_2 \quad \phi_1 \equiv_\pi \phi_2 \to \phi}{\pi; \mathcal{A}_1 \cup \mathcal{A}_2 \vdash e_1\ e_2 : \phi | \Delta_1 \cup \Delta_2}$$

Fig. 2: Basic typing rules

type substitution. It replaces free type variables in ϕ by the corresponding images in θ. Its definition is standard except for choice types, where substitution applies to both alternatives.

An assumption set $\mathcal{A}$ maps expression variables to variational types. Moreover, $\mathcal{A}$ stores the corresponding choice name of each variable reference. We write $\mathcal{A}^T(x)$ and $\mathcal{A}^D(x)$ to get the set of types and choice names that x maps to in $\mathcal{A}$, respectively. We use l to represent program locations and 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. We assume f uniquely determines the location. The exact definition of $\ell(\cdot)$ doesn't matter. The choice environment Δ associates each leaf l with a choice type $d\langle\phi_1, \alpha_2\rangle$ during the typing process. Note that ϕ_1 is the type under normal type inference for the subexpression at l and ϕ_2 is the alternative type for the same subexpression to remove the type error.

B. Basic Typing Rules

We present the typing rules in Figure 2. The typing judgement has the form $\pi; \mathcal{A} \vdash e : \phi | \Delta$, meaning that the expression e has the type ϕ with the assumption set $\mathcal{A}$, the validity restriction π, and the change information Δ.

The conditions $\exists\ d$ in rules Con and Var are always satisfied since we have an unlimited supply of choice names. The rule Con says that if c is of the type γ, then in our type system it has the type $d\langle\gamma, \phi\rangle$ to indicate that we can change c to any type ϕ to remove the type error. No assumption is made for typing c, and thus the $\mathcal{A}$ component is empty. The choice environment records the change as $\{(\ell(c), d\langle\gamma, \phi\rangle)\}$. For this rule, we can use any π since the typing of the constant is always valid.

The rule Var for variables is similar to the rule Con. The only difference is that the variable may have any type ϕ and the assumption is recorded in $\mathcal{A}$. At variable references, we always assume variables are bound. Therefore the typing pattern component can be any value. We deal with unbound variables in Section IV-C.

Given an abstraction $\lambda x.e$, we first type the body e, which may contain multiple assumptions for the parameter x. These assumptions need to be consistent for the abstraction to be well typed. The second premise in the Abs rule ensures that all assumptions are equivalent to each other with the restriction π. The assumptions for the abstraction is the assumptions for its body minus those for the parameter x. The choice environment of the abstraction is the same as that for its body.

The App rule is very similar to the application rule discussed in Section III. The only difference is that here we have to merge assumption sets from the subexpressions and also change environments from them.

C. Handling Unbound Variables

In rule Var, we assume that all variables are bound. However, this is not always the case. We can determine if an expression contains unbound variables by checking $\mathcal{A}$. If $\mathcal{A}$ is empty, then the expression does not contain unbound variables. Otherwise, it does. For unbound variables, we want to say what are the expected types for them to remove the type error in the expressions, rather than simply reporting them as unbound.

To achieve this goal, we need to handle them in typing. Specifically, what does it mean if (x, ϕ, d) still belongs to $\mathcal{A}$ after typing the expression e with $\pi; \mathcal{A} \vdash e : \phi | \Delta$? This means that the access to x should be incorrect but when typing e we assumed it was correct under the Var rule. We can address this problem by adjusting the validity restriction π. In fact, (x, ϕ, d) still belongs to $\mathcal{A}$ means that the typing result is invalid in $d.1$. Thus, we need to worsen π by $d\langle\bot, \top\rangle$ to keep the typing result valid.

In fact, there is a simpler way to address this problem. The key observation here is that if the typing pattern π is already worse than $d\langle\bot, \top\rangle$, then we don't need to worsen it anymore. Before presenting the typing rule based on this idea, we first formalize the notion that a typing pattern (π_1) is *worse* than the other one (π_2), written as $\pi_1 \leq \pi_2$.

$$\pi \leq \top \qquad \bot \leq \pi \qquad \frac{\pi_1 \leq \pi_2 \qquad \pi_2 \leq \pi_3}{\pi_1 \leq \pi_3} \qquad \frac{\pi_1 \equiv \pi_2}{\pi_1 \leq \pi_2}$$

$$\frac{\pi_1 \leq \pi_3 \qquad \pi_2 \leq \pi_4}{d\langle\pi_1, \pi_2\rangle \leq d\langle\pi_3, \pi_4\rangle}$$

Intuitively, $\pi_1 \leq \pi_2$ expresses that, for any variant, if π_2 contains a $\bot$ then so does π_1. The first two rules say that all typing patterns are worse than $\top$ and better than $\bot$. The third rule states that the relation is transitive. In the fourth rule, we reuse the machinery of type equivalence by interpreting $\bot$ and $\top$ as two constant types. The rule then says that two equivalent patterns satisfy the $\leq$ relation. The last rule states that two choice patterns satisfy $\leq$ if both their corresponding alternatives satisfy $\leq$.

With $\leq$, we can formalize the rule for unbound variables declaratively as follows.

$$\textsc{Unbound}\ \frac{\pi; \mathcal{A} \vdash e : \phi | \Delta \qquad \forall (x, \phi, d) \in \mathcal{A} \colon \pi \leq d\langle\bot, \top\rangle}{\pi \vdash_M e : \phi | \Delta}$$

(1a) $\mathit{infer1}(x) =$
$\quad \phi \leftarrow d\langle \alpha_1, \alpha_2 \rangle$ {- d, α_1, and α_2 fresh-}
$\quad$ return $(\top, \{(x, \alpha_1, d)\}, \phi, \{(\ell(x), \phi)\})$

(1b) $\mathit{infer1}(e_1\ e_2) =$
$\quad (\pi_1, \mathcal{A}_1, \phi_1, \Delta_1) \leftarrow \mathit{infer1}(e_1)$
$\quad (\pi_2, \mathcal{A}_2, \phi_2, \Delta_2) \leftarrow \mathit{infer1}(e_2)$
$\quad (\pi, \theta) \leftarrow \mathit{vunify}(\phi_1, \phi_2 \rightarrow \beta)$ {-β fresh-}
$\quad$ return $(\pi \otimes \pi_1 \otimes \pi_2, \theta(\mathcal{A}_1 \cup \mathcal{A}_2), \theta(\beta), \theta(\Delta_1 \cup \Delta_2))$

Fig. 3: An inference algorithm that recomputes all error fixes

We can see that $\mathcal{A}$ in the premise disappears in the conclusion. Intuitively, the rule says that if π is worse enough, then the residual assumptions can simply be forgotten.

The subscript M indicates that this is the main rule on top of all other typing rules. Given any expression, we should use this rule to compute error fixes.

Although our type system is based on principal typing, it generates the same set of error fixes as CFT does, as captured in the following theorem.

Theorem 1 (CFT equivalence)**.** $\pi; \varnothing \vdash e : \phi | \Delta \Leftrightarrow \varnothing \vdash_{\mathit{CFT}} e : \phi^{\perp} | \Delta^{\perp}$ *such that* $\phi \equiv_{\pi} \phi^{\perp}$, $\Delta \equiv_{\pi} \Delta^{\perp}$, *and* $\forall \delta : \lfloor \phi^{\perp} \rfloor_{\delta} \neq \perp \Leftrightarrow \lfloor \pi \rfloor_{\delta} \neq \perp$.

In the theorem, we use $\vdash_{\mathit{CFT}}$ to denote the typing relation of CFT. The superscript $\perp$ in $\phi^{\perp}$ and $\Delta^{\perp}$ reflects that errors types are embedded in the type syntax in CFT. The relation $\Delta \equiv_{\pi} \Delta^{\perp}$ is defined as $\forall l \in \mathit{dom}(\Delta) \colon \Delta(l) \equiv_{\pi} \Delta^{\perp}(l)$. We can prove this theorem by showing that both CFT and this type system are equivalent to the type-update system [3, §4.3] through inductions over typing relations.

V. Three Different Type Inference Strategies

We present three inference algorithms in this section: an algorithm that recomputes all type error fixes as programs are updated, a coarse incremental inference algorithm that reuses results so that only nodes that are affected by the change are retyped, and a refined incremental inference algorithm on top of incremental variational unification. We use Figure 4 to illustrate the differences among them, where subfigures (b) to (d) respectively visualize the behavior of these algorithms for updating `not '1'` to `not 1`.

Recompute All Error Fixes The algorithm *infer1* in Figure 3 provides a unification-based implementation of the typing rules from Figure 2. The function *infer1* has the type $e \rightarrow \pi \times \mathcal{A} \times \phi \times \Delta$, that is, it takes in an expression and returns a typing pattern indicating which alternatives of the inference result are valid, an assumption set $\mathcal{A}$, a result type, and a choice environment storing type change information of leaves. We only present the algorithm for two cases, and a detailed version of *infer1* can be found in the longer version of this paper [7].

We now briefly go through *infer1*. The case (1a) deals with variable references. As mentioned earlier, we always assume variables are bound at variable references. Therefore, *infer1* returns $\top$ for π, indicating no errors have occurred. Note that

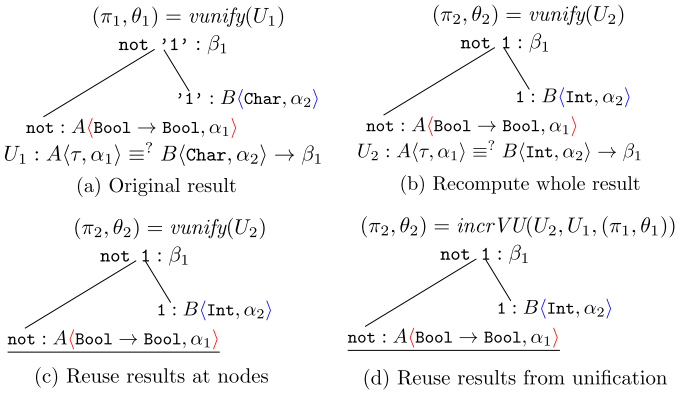

Fig. 4: A comparison of different methods for computing error fixes under program updates. The underlined typing relations are reused. To simplify our discussion, we assume `not` is a constant having the type `Bool` $\rightarrow$ `Bool`. In the figure, the value of τ is `Bool` $\rightarrow$ `Bool`.

we need two fresh type variables α_1 and α_2 in (1a), where α_1 denotes that the variable can be mapped to any type based on its uses and α_2 denotes that the variable can be changed to something of type α_2 to remove the type error if the variable is an error cause. The main difference between them is that α_1 is recorded in $\mathcal{A}$ and will be later unified with other assumptions for the same variable.

Case (1b) deals with function applications. It first computes the results for the function and the argument independently and then unifies the function type and the argument type with *vunify*, a variational unification algorithm from [6]. We now have three tying patterns, π_1 and π_2 from typing e_1 and e_2, respectively, and π from *vunify*. They will be merged together through the $\otimes$ operation, defined as follows.

$$\perp \otimes \pi = \perp \qquad \top \otimes \pi = \pi \qquad d\langle \pi_1, \pi_2 \rangle \otimes \pi = d\langle \pi_1 \otimes \pi, \pi_2 \otimes \pi \rangle$$

The result of $\otimes$ is $\top$ only if both operands are $\top$. Intuitively, $\otimes$ can be understood as the logical and operation if we interpret $\top$and $\perp$ as the truth values true and false, respectively. Based on this operation, *infer1* returns $\pi \otimes \pi_1 \otimes \pi_2$, meaning that $e_1\ e_2$ is well typed only in the alternatives that both e_1 and e_2 are correct and the argument type of e_1 unifies with the type of e_2.

infer1 is sound and complete with respect to the typing relation from Section IV. We defer a detailed discussion of the properties to [7].

Coarse Incremental Type Inference The type of *infer1*, $e \rightarrow \pi \times \mathcal{A} \times \phi \times \Delta$, indicates that the expression e decides the type and the assumption set of e. This immediately shows that if a subexpression hasn't been changed, then there is no need to perform type inference for that subexpression. In particular, when a node is changed, the type information for only the path from that node to the root needs to be recomputed. For example, if we change `True` in `rank` to `1`, we need to update the type information for the path from the right-most node to the root. This allows us to recompute type information for four nodes only, rather than eight nodes if we recompute everything

infer3 : $e \to \pi \times \mathcal{A} \times \phi \times \Delta$

(3a) *infer3*$(x) =$
  $\phi \leftarrow d^\circ\langle \alpha_1^\circ, \alpha_2^\circ \rangle$ {- reuse fresh names -}
  return $(\top, \{(x, \alpha_1^\circ, d^\circ)\}, \phi, \{(\ell(x), \phi)\})$

(3b) *infer3*$(e_1\ e_2) =$
  $(\pi_1, \mathcal{A}_1, \phi_1, \Delta_1) \leftarrow$ *infer3*(e_1)
  $(\pi_2, \mathcal{A}_2, \phi_2, \Delta_2) \leftarrow$ *infer3*(e_2)
  if $\phi_1^\circ = \phi_1$ and $\phi_2^\circ = \phi_2$
    return $(\pi^\circ \otimes \pi_1 \otimes \pi_2, \theta^\circ(\mathcal{A}_1 \cup \mathcal{A}_2), \phi_r^\circ, \theta^\circ(\Delta_1 \cup \Delta_2))$
  $(\pi, \theta) \leftarrow$ *incrVU*$(\phi_1 \equiv^? \phi_2 \to \beta^\circ, U^\circ, (\pi^\circ, \theta^\circ))$
  return $(\pi \otimes \pi_1 \otimes \pi_2, \theta(\mathcal{A}_1 \cup \mathcal{A}_2), \theta(\beta), \theta(\Delta_1 \cup \Delta_2))$

Fig. 5: A refined incremental inference algorithm.

from scratch.

We use two other tricks to save more computations for updating error fixes. The first trick is to reuse fresh variables generated for nodes. For example, in Figure 4(c), we use the same variable α_2 for the second alternative in choice B, the same as in Figure 4(a). This idea helps us to, in some situations, inferred the same type for a subexpression even it is changed, effectively quarantining the effects of changes.

The second trick is reordering the process of unifying all assumptions for the same variable, required in the type inference for abstractions, where all the assumptions for the same variable have to be unified. We could unify them in a linear ordering, but earlier unification results affect later unifications, causing a dependency. Instead, we first unify the first type with each of the rest types, yielding a list of substitutions that are combined into one substitution through the technique of substitution composition [20, §6]. We defer to [7] to a detailed exposition of these ideas.

We refer to the inference algorithm implementing these ideas *infer2*. We will omit its presentation since it is very similar to *infer3* in Figure 5. In particular, *infer2* can be obtained from Figure 5 by replacing all occurrences of *infer3* with *infer2* and replacing calls of the form *incrVU*$(\phi_1 \equiv^? \phi_2, U', (\pi', \theta'))$ with calls of the form *vunify*(ϕ_1, ϕ_2).

Refined Incremental Type Inference Coarse incremental type inference tries to maximize the reuse of typing information, but once a type is changed, the unification problems involving that type has to be resolved, even the change is minor. Observing that type unification is a main part of type inference and needs intensive computations, our main idea of refined incremental type inference is to solve variational unification problems incrementally. The algorithm *infer3* in Figure 5 implements this idea, where *incrVU* is an incremental variational unification algorithm developed in Section VI.

In the figure, we use the notation o° to denote the saved copy of o from the last run of *infer3*. If there is no saved value, then a meaningful value is returned. For example, in case (3a), we use d° to return the choice name generated last time. However, if no fresh choice name was generated and saved, then d° just generates a new fresh choice name and returns it. If U° doesn't exist, then the corresponding call of *incrVU* will call *vunify* instead.

We now briefly go through each case. Case (3a) is very similar to (1a). The only difference is that we try to reuse fresh names as much as possible. Case (3b) types applications. It first checks if both the function type and the argument type are the same as saved copies. If so, then the saved π° and θ° are used without unifying the function type and the argument type. Otherwise, it uses the incremental variational unification algorithm to solve the new unification problem.

VI. Incremental Variational Unification

The general idea of incremental variational unification is that we first compute the difference between two unification problems, yielding a delta unification problem, which is then solved and the result is merged into the original result to get a unifier for the new unification problem. Given two unification problems $U\colon \phi_l \equiv^? \phi_r$ and $U'\colon \phi_l' \equiv^? \phi_r'$ and the result (π, θ) for U, we take the following steps to solve U'.

Compute differences To compute the difference between two unification problems, we first need to compute that between two types. We use the function $\mathcal{D}(\phi_1, \phi_2)$ to compute the difference between ϕ_1 and ϕ_2. The result is a set of decisions, where each decision δ satisfies that $\lfloor \phi_1 \rfloor_\delta \not\equiv \lfloor \phi_2 \rfloor_\delta$. The function $\mathcal{D}(\phi_1, \phi_2)$ is defined as follows. The case for function types considering their respective argument types and return types and is omitted.

$$\begin{aligned}
\mathcal{D}(\tau, \tau) &= \{\} \\
\mathcal{D}(\tau_1, \tau_2) &= \{\{\}\} \\
\mathcal{D}(d\langle \phi_1, \phi_2 \rangle, d\langle \phi_3, \phi_4 \rangle) &= \{d.1\colon \delta \mid \delta \in \mathcal{D}(\phi_1, \phi_3)\} \cup \\
&\qquad \{d.2\colon \delta \mid \delta \in \mathcal{D}(\phi_2, \phi_4)\} \\
\mathcal{D}(d\langle \phi_1, \phi_2 \rangle, \phi) &= \mathcal{D}(d\langle \phi_1, \phi_2 \rangle, d\langle \lfloor \phi \rfloor_{d.1}, \lfloor \phi \rfloor_{d.2} \rangle) \\
\mathcal{D}(\phi, d\langle \phi_1, \phi_2 \rangle) &= \mathcal{D}(d\langle \phi_1, \phi_2 \rangle, \phi)
\end{aligned}$$

When two types are the same, as in the first case, the result is an empty set, meaning that there is no decision such that selecting them with the decision results in different types. In the second case, the two types are completely different. The result set in this case has one member, which is itself an empty set. Remember that selecting a type with an empty decision gives that type back. To compute the difference of two choice types with the same choice name d, $\mathcal{D}$ first computes the difference of the first alternatives of the choices and then adds $d.1$ to each decision. Similar, $\mathcal{D}$ does this for the right alternatives and extends the results with $d.2$. The cases that one argument is a choice type and the other is not are reduced to the first case.

Here are two examples of applying $\mathcal{D}$.

$$\mathcal{D}(\texttt{Int}, \texttt{Char}) = \{\{\}\} \quad \mathcal{D}(A\langle \texttt{Int}, \texttt{Char} \rangle, \texttt{Char}) = \{\{A.1\}\}$$

The result of the second example indicates that two arguments differ in only one decision, the first alternative of A. The first argument has `Int` while the second argument has `Char` in that decision.

We overload $\mathcal{D}$ to compute difference between two unification problems and define $\mathcal{D}(U, U')$ as $\mathcal{D}(\phi_l, \phi_l') \cup \mathcal{D}(\phi_r, \phi_r')$.

With this definition, we have $\mathcal{D}(U_2, U_1) = \{\{B.1\}\}$, where U_1 and U_2 are from Figure 4.

Resolve subproblems Each decision δ in $\mathcal{D}(U, U')$ represents a unification problem $\lfloor U' \rfloor_\delta$ to be resolved. We can again use *vunify* to solve these problems. However, to simplify the operation of the next step, we use *vunify*$'$, a variant of *vunify*. The main difference lies in the way they represent unification results. Given a unification problem $\phi_1 \equiv^? \phi_2$, *vunify* returns (π, θ) while *vunify*$'$ returns $\{(\delta, \pi, \theta)\}$, a set of triples. Each triple (δ, π, θ) satisfies that (1) *vunify*$(\lfloor \phi_1 \rfloor_\delta, \lfloor \phi_2 \rfloor_\delta) = (\pi, \theta)$ and (2) *vunify*$(\lfloor \phi_1 \rfloor_\delta, \lfloor \phi_2 \rfloor_\delta)$ doesn't need to decompose any choice. Consider, for example, the unification problem $A\langle \texttt{Int}, \alpha \rangle \equiv^? \texttt{Int}$. While *vunify* returns $(\top, \{\alpha \mapsto A\langle \kappa, \texttt{Int} \rangle\})$, *vunify*$'$ returns $\{(\{A.2\}, \top, \{\alpha \mapsto \texttt{Int}\})\}$. The definition of *vunify*$'$ differs from *vunify* in how it returns results only, and we will not present the definition in detail.

Based on *vunify*$'$, we define $\mathcal{R}(U, U')$ to solve all the unification subproblems and collect all the results as follows, where δ ranges over $\mathcal{D}(U, U')$.

$$\mathcal{R}(U, U') = \bigcup \{(\delta \cup \delta', \pi, \theta) \mid (\delta', \pi, \theta) \in \textit{vunify}'(\lfloor \phi_l' \rfloor_\delta, \lfloor \phi_r' \rfloor_\delta)\}$$

As an example, the result of $\mathcal{R}(U_2, U_1)$ is

$$\{(\{A.1, B.1\}, \bot, \{\}), (\{A.2, B.1\}, \top, \{\alpha_1 \mapsto \texttt{Int} \to \beta_1\})\}$$

Merge results Given each $\{(\delta', \pi', \theta')\}$ from $\mathcal{R}(U, U')$ and (π, θ) for U, we can merge them together to get the result for solving U'. First, let's try to merge π' into π with the decision δ'. Essentially, merging means that we need to change $\lfloor \pi \rfloor_{\delta'}$ to $\pi'$and don't change any other $\lfloor \pi \rfloor_{\delta''}$ if $\delta' \not\subseteq \delta'' \wedge \delta'' \not\subseteq \delta'$. We use the function *comp* from [6, §7.2] to perform this task. Given δ, ϕ', ϕ, *comp*(δ, ϕ', ϕ) replaces the type at δ in ϕ with ϕ' and leaves other parts unchanged. As we have done in Section IV-C, we can simply interpret $\bot$ and $\top$ as two constant types and use *comp*(δ, π', π) to merge π' into π. We can merge θ' into θ with δ' similarly. For each $\alpha' \mapsto \phi'$ in θ', we merge ϕ' into $\theta(\alpha')$ with δ'. If $\alpha' \notin \textit{dom}(\theta)$, we simply add $\alpha' \mapsto \textit{comp}(\delta', \phi', \kappa)$ to θ, where κ is a fresh type variable.

Now we can merge $\mathcal{R}(U_2, U_1)$ into θ_1 (from Section III), yielding $(A\langle B\langle \bot, \top \rangle, \top \rangle, \theta_2)$, where

$$\theta_2 = \{\alpha_1 \mapsto A\langle \kappa_5, B\langle \texttt{Int}, \kappa_2 \rangle \to \kappa_4 \rangle, \\ \alpha_2 \mapsto A\langle B\langle \kappa_1, \texttt{Bool} \rangle, \kappa_2 \rangle, \beta_1 \mapsto A\langle B\langle \kappa_3, \texttt{Bool} \rangle, \kappa_4 \rangle\}$$

Based on these three steps, we can define the function *incrVU* to solve U' based on the result (π, θ) for U, where $\mathcal{M}$ denotes the merging process described in the third step.

$$\textit{incrVU}(U, U', (\pi, \theta)) = \mathcal{M}(\mathcal{R}(U, U'), (\pi, \theta))$$

VII. Evaluation

To test the feasibility of *e*CFT, we have developed a prototype that implements the ideas from this paper in Haskell. The prototype supports all three inference algorithms *infer1* through *infer3* from Section V, which we will refer to as *recomputing*, *coarse*, and *refined*, respectively. In addition to the constructors presented in Figure 1, our prototype also deals with other features, including conditionals, polymorphism [7], data types, and case expressions. The prototype supports predefined (library) functions by resolving unbound variables left in $\mathcal{A}$ in the predefined type environment and introducing errors when the resolution fails.

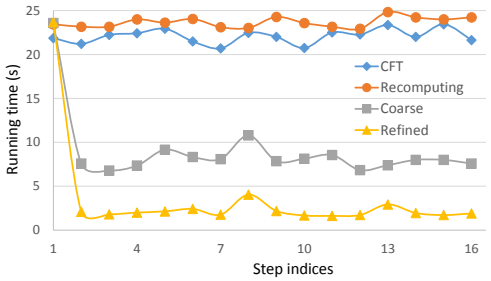

Fig. 6: Running times of different inference algorithms of *e*CFT on a student program sequence.

Section IV shows that *e*CFT and CFT produce the same set of error fixes for any given expression. We have experimentally confirmed this by running both *e*CFT and CFT prototypes over the benchmark we collected before [3]. For this reason, our evaluation focuses on performance.

Our first performance test used 60 program sequences from the student program databases [14], [30]. The initial sizes of these sequences range from 47 to 136 LOC and the numbers of steps range from 5 to 42. The results of these sequences exhibit quite similar patterns. For this reason, we present the result for a representative sequence (whose initial LOC is 125 and the number of updates is 15) in more detail only. The ratios of the differences between two consecutive program versions over the old versions range from 0.01 to 0.1 for this sequence (except for the step 8 where the ratio is 0.14). Figure 6 presents the running time of CFT and the three inference algorithms of *e*CFT for this sequence. The times are measured on a laptop with a processor having four 2.4GHz dual-cores and 8GB RAM running 64-bit Ubuntu 16.04 LTS and GHC 8.0.2.

The response time (the time delay to display the first error message) of CFT is about 22.3s in average for the presented program sequence. *e*CFT is up to 3× faster with *coarse* and 13× faster with *refined*. With *refined*, the response time ranges from 1.7s to 2.4s except for the step 8, where the response time is 4.0s. The reason is that the change ratio is 0.14, much higher those at other steps. The fact that *refined* is much faster than *coarse* reflects that unification problems are getting more and more complex as type inference moving up in the AST, although the change in each unification problem may be minor. This demonstrates the value of *refined*.

For other tested sequences, the speedup of *refined* over CFT ranges from 4.2× to 19.1× and is more than 12.4× in 80% cases. For *coarse*, the speedup over CFT ranges from 1.2× to 5.3× and is more than 2.6× in 80% cases.

In the second test, we are interested in knowing how *e*CFT behaves in general. For this reason, we have conducted another test, where we chose another 60 student programs of 71 to 142 LOC [14]. For each of the change ratios from 0.01 to 0.1 with an interval of 0.01 and from 0.1 to 0.3 with an interval of 0.05, we randomly generated changes to each original program. For

each ratio, we generated 50 unique and well-formed programs by filtering out ill-formed or repetitive generated programs.

In this test, we observed that when the change ratio is less than 0.1, relative performance of *refined*, *coarse*, and CFT exhibits the same pattern as in the previous test. We defer further details to [7] due to the space constraint. Combined with the fact that the change ratios are less than 0.1 for more than 80% of changes [7] during type error debugging, we conclude that *refined* (*e*CFT) is practical for interactive use, especially as programs are updated for fixing type errors.

VIII. Related Work

In [3], we have discussed the relation of CFT with much previous work, such as sum types [23], Seminal [19], and Chameleon [28]. These discussions also apply here and thus in this paper we mainly discuss the relation with other work.

Chitil [8] suggested that the main difficulty of understanding why a program is ill typed lies in knowing why subexpressions get certain types. Based on this observation, he developed a compositional error explanation approach using principal typing. His approach allows the user to navigate through the explanation graph and inspect the type of each node. While our approach focuses on each potential erroneous expression once a time and provides a detailed error message, his approach doesn't provide change suggestions but allow the user to have a big-picture about why errors have occurred. In this sense, these approaches are complementary to each other.

Helium [17] is designed to provide good quality type error messages for Haskell beginners. Helium uses a constraint-based type inference algorithm to generate a set of type constraints, and then uses a solver to handle all the constraints globally. When the constraints are unsatisfiable, it uses a set of heuristics to find the most suspicious constraint [15], [17], from which a few most likely error sources are identified. Haack and Wells [13] computed program slices that identify all program locations contributing to type errors. The underlying type system T [9] used in the approach to generate type constraints can be viewed as a variant of principal typing [18].

Frameworks for general incremental computing are developed by Acar et al. [1]. Unfortunately, such frameworks usually impose high overhead [11], and they don't support domain-specific optimizations, such as the tricks discussed in coarse incremental type inference. Therefore, we propose our own incremental algorithm. Erdweg et al. [11] developed a method deriving incremental type checkers with bottom-up typing flow. Their approach resolves the whole constraint once it is updated while our approach resolves only part of the constraint (Section VI).

IX. Conclusions

We have presented *e*CFT, a method to improve the performance of the highly effective type error debugging method CFT by exploiting old versions of programs. While CFT is quite effective in locating type errors and providing change suggestions, it has a long response time. To address this problem, we redesigned the type system and used principal typing to compute all error fixes. We have also developed two methods for efficiently updating error fixes as programs are changed. Our evaluation result shows that in average *e*CFT is $12.4\times$ faster than CFT in 80% cases. The response time drops from about 22.3s in CFT to about 1.7s in *e*CFT for programs with about 125 LOC in our evaluation.

References

[1] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *TOPLAS*, 2009.

[2] S. Aditya and R. S. Nikhil. Incremental polymorphism. In *FPCA*, pages 379–405, 1991.

[3] S. Chen and M. Erwig. Counter-Factual Typing for Debugging Type Errors. In *POPL*, pages 583–594, 2014.

[4] S. Chen and M. Erwig. Principal type inference for gadts. In *POPL*, pages 416–428, 2016.

[5] S. Chen, M. Erwig, and E. Walkingshaw. An Error-Tolerant Type System for Variational Lambda Calculus. In *ICFP*, 2012.

[6] S. Chen, M. Erwig, and E. Walkingshaw. Extending Type Inference to Variational Programs. *TOPLAS*, 2014.

[7] S. Chen and B. Wu. Efficient Type Error Debugging (Tech Report). 2019. Available at http://www.ucs.louisiana.edu/ sxc2311/ws/techreport/incrcft.pdf.

[8] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP*, 2001.

[9] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, 1985.

[10] H. Eo, O. Lee, and K. Yi. Proofs of a set of hybrid let-polymorphic type inference algorithms. *New Generation Computing*, 22(1):1–36, 2004.

[11] S. Erdweg, O. Bračevac, E. Kuci, M. Krebs, and M. Mezini. A co-contextual formulation of type rules and its application to incremental type checking. In *OOPSLA*.

[12] M. Erwig and E. Walkingshaw. The Choice Calculus: A Representation for Software Variation. *TOSEM*, 21(1):6:1–6:27, 2011.

[13] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *ESOP*, pages 284–301, 2003.

[14] J. Hage. Helium benchmark programs, (2002-2005). Private communication, 2013.

[15] J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In *IFL*, volume 4449 of *LNCS*, pages 199–216. 2007.

[16] B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning haskell. In *Haskell*, pages 62–71, 2003.

[17] B. J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005.

[18] T. Jim. What are principal typings and what are they good for? In *POPL*, pages 42–53, 1996.

[19] B. Lerner, M. Flower, D. Grossman, and C. Chambers. Searching for type-error messages. In *PLDI*, pages 425–434, 2007.

[20] C.-k. Lin. *Practical Type Inference for the GADT Type System*. PhD thesis, Portland State University, 2010.

[21] B. J. McAdam. *Repairing type errors in functional programs*. PhD thesis, University of Edinburgh, 2002.

[22] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[23] M. Neubauer and P. Thiemann. Discriminative sum types locate the source of type errors. In *ICFP*, pages 15–26, 2003.

[24] Z. Pavlinovic, T. King, and T. Wies. Practical smt-based type error localization. In *ICFP*, pages 412–423, 2015.

[25] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[26] T. Schilling. Constraint-free type error slicing. In *TFP*, pages 1–16. Springer, 2012.

[27] Z. Shao and A. W. Appel. Smartest recompilation. In *POPL*, pages 439–450, 1993.

[28] P. J. Stuckey, M. Sulzmann, and J. Wazny. Improving type error diagnosis. In *Haskell*, pages 80–91, 2004.

[29] V. Tirronen, S. Uusi-mäkelä, and V. Isomöttönen. Understanding beginners' mistakes with haskell. *JFP*, 25:1–31, 2015.

[30] P. van Keeken. Analyzing helium programs obtained through logging. Master's thesis, Utrecht University, October 2006.

[31] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones. Diagnosing type errors with class. In *PLDI*, pages 12–21, 2015.