Let's Hear Both Sides: On Combining Type-Error Reporting Tools

Sheng Chen
Oregon State University
chensh@eecs.oregonstate.edu

Martin Erwig
Oregon State University
erwig@eecs.oregonstate.edu

Karl Smeltzer
Oregon State University
smeltzek@eecs.oregonstate.edu

***Abstract*—Producing precise and helpful type error messages has been a challenge for the implementations of functional programming languages for over 3 decades now. Many different approaches and methods have been tried to solve this thorny problem, but current type-error reporting tools still suffer from a lack of precision in many cases. Based on the rather obvious observation that different approaches work well in different situations, we have studied the question of whether a combination of tools that exploits their diversity can lead to improved accuracy. Specifically, we have studied Helium, a Haskell implementation particularly aimed at producing good type error messages, and Lazy Typing, an approach developed previously by us to address the premature-error-commitment problem in type checkers. By analyzing the respective strengths and weaknesses of the two approaches we were able to identify a strategy to combine both tools that could markedly improve the accuracy of reported errors. Specifically, we report an evaluation of 1069 unique ill-typed programs out of a total of 11256 Haskell programs that reveals that this combination strategy enjoys a correctness rate of 79%, which is an improvement of 22%/17% compared to using Lazy Typing/Helium alone. In addition to describing this particular case study, we will also report insights we gained into the combination of error-reporting tools in general.**

I. Introduction

One of the major challenges faced by current implementations of type inference algorithms is the production of error messages that help programmers fix mistakes in the code. Cryptic, complex, and misleading compiler error messages have been understood to be severe barriers to programmers, especially novices, going back several decades (to the original Hindley-Milner type system, even) [1], [2], [3], [4], and this problem has continued to be acknowledged much more recently [5], [6].

Expert programmers familiar with the way their compiler performs type inference may develop some intuition for recognizing error message patterns. Novice programmers, however, might find some type errors—especially those using type system jargon such as infinite types, or containing many polymorphic type variables—to be more confusing than helpful. Sometime these errors are not only complex, but even incorrect or misleading. In particular, type errors sometimes include line numbers which do not point to the actual error. In some cases, this location is quite far away from the actual error source, or even in the wrong file.

Quite a few solutions have been proposed to more accurately locate type errors. One approach is to eliminate the left-to-right bias of type inference [7], [8]. Another is to report several program sites that most likely contribute to the type inconsistency [9], [10] rather than committing to only one error location. A related technique uses program slicing to determine all the positions that are related to the type errors [11]. Finally, constraint solving has been used to minimize the number of possible error locations [12].

However, despite the considerable research efforts devoted to this problem, and the improvements made, each of the proposed solutions has its own shortcomings and performs poorly for certain programs. Consider, for example, the following function [10] that splits a list into two lists by placing elements at odd positions in the first list and those at even positions into the second.

```
split xs = case xs of
             []       -> ([], [])
             [x]      -> ([], x)
             (x:y:zs) -> let (xs, ys) = split zs
                         in (x:xs, y:ys)
```

Even though the type error in this definition is not hard to spot—it occurs because the variable `x` in the second case alternative is used as if it were itself a list rather than a single value [10]—existing type inference systems have a hard time locating it precisely.

When type checking this example, the Glasgow Haskell Compiler (GHC) 7.6,[1] which is the most widely used Haskell compiler, produces a jargon-filled error message about being unable to construct infinite types and points the programmer to the very last line in the function.

The Chameleon Type Debugger [10], which represents the more advanced error slicing tools, notes type mismatches in three different lines, which still requires the programmer to distill the information into an actual fix.

Helium [12] was designed to produce well-rounded results that are more helpful than those from other tools. It suggests changing the cons function `(:)` to the list append function `(++)`. This change would indeed fix the type error, but would cause the types of the other two alternatives to change as well. In general, it seems preferable to change the definition of a function rather than the use of one and, if possible, to minimize the effect of the change.

We have recently developed a new approach, termed lazy typing (LT) [13], which improves type error messages for many of these situations. For this particular example, LT identifies the error as occurring in the expression `x` on the right-hand side of the second case expression—exactly where we would hope.

[1] www.haskell.org/ghc/

The error message produced suggests that `x` is of polymorphic type `a`, but should be of type `[a]`, or "list of `a`".

However, having seen so many different approaches one might wonder whether any single approach will ever be able to handle every conceivable type error properly. Consequently, we ask the question of whether it is possible instead to combine existing techniques together in complementary ways. Strategically combined, this could allow a given tool to be used in situations where it excels, and another tool where it does not. To this end, we examine the specific case of combining LT and Helium, and from that we extract principles and recommendations for the general case.

In Section II, we introduce and explain the principles behind lazy typing and expand upon Helium as a representative for comparison. Section III evaluates Helium and LT separately in order to specifically identify conditions under which each is strong. Section IV discusses how Helium and LT can be combined and evaluates the success of doing so. In Section V we extract general principles for combining other type error reporting methods and tools. Finally, we discuss related work in Section VI and provide conclusions and general advice in Section VII.

II. LAZY TYPING AND HELIUM

Here we examine LT and Helium in slightly greater depth. This will help to illustrate the complementary nature of these two techniques, and justify why we have selected these two tools in particular to combine and discuss.

The motivating idea behind lazy typing is to better exploit the context of expressions containing type errors. This context, in principle, can support finding more accurate type error locations and also improve potential change suggestions provided to the programmer [13].

We are able to exploit context information by delaying the decision about the type of an expression until we can better leverage the type information gathered from its context. In cases where the expression turns out to exhibit a type error, the availability of this contextual type information can help in deciding what is wrong with the expression and therefore point more precisely to the source of the type error.

The basic idea of this delaying strategy is to turn an equality constraint between types, such as $\tau = \tau'$, into a *choice* between the two types, which we write as $A\langle\tau,\tau'\rangle$ [14]. Instead of enforcing the constraint, which potentially causes an immediate failure of type checking, we continue the type inference process with the two possibilities τ and τ'. If $\tau \neq \tau'$, the inference will eventually fail too, but at a later point when additional context information is available. We call this strategy *lazy typing*.

By contrast, Helium is based on a constraint solving approach. Helium can be roughly broken down into three phases. In the first, constraints describing the program or expression are gathered. The second stage reorders the type constraints in a tree, which largely determines where Helium finds and identifies the type error. Finally, the collection of constraints is passed to a solver to type the code.

The differences between LT and Helium are indicative of the diversity present among type checking and type error reporting techniques. As might be expected, this variety present between LT and Helium causes them to excel in quite different situations. Consider the following type-incorrect function definition.

```
insertRowInTable :: [String] -> [[String]] -> [[String]]
insertRowInTable r t = r ++ t
```

This function takes two parameters, namely a data row represented by a list of strings and a table in which to insert that row represented as a list of list of strings. The implementation then uses the `(++)` function, which appends two lists. This causes a type error, however, because `(++)` has the type `[a] -> [a] -> [a]` rather than `[a] -> [[a]] -> [[a]]`.

LT types the expression before it considers the type annotation. Because of this, the error is determined to be a mismatch between the expression and the annotation, and no suggestion is generated. Helium, on the other hand, assumes the correctness of the type annotation and suggests changing the implementation to use `(:)`, which inserts a single value at the front of the list, rather than `(++)`. This will, indeed, fix the type error, although it may not reflect what the programmer had in mind.

However, introducing additional code to this example can change the results dramatically and thereby illustrate some of the practical differences between LT and Helium. Suppose we add the following definition somewhere else in the file, which tries to make use of our `insertRowInTable` function.

```
v = insertRowInTable ["Bread"] [["Beer"]]
```

LT is able to make use of this additional context to see that this definition agrees with the original type annotation, and so determines the error is most likely in the body of the implementation, and it suggests to change `(++)` to something of type `[String] -> [[String]] -> [[String]]`. Helium does not account for the additional context and suggests the same change as before. In this case, the additional context makes LT more accurate, but not more so than Helium. However, consider the case where, instead of the previous definition of `v`, we have the following.

```
v = insertRowInTable [["Bread"]] [["Beer"]]
```

Here, LT sees that the function implementation and the definition of `v` agree, while the type annotation does not. Because of this, it suggests changing the type annotation to `[[a]] -> [[a]] -> [[a]]`.

By contrast, Helium continues to trust the type annotation and produces two distinct type errors. The first is unchanged from the previous examples while the second is to use a character literal rather than a string, effectively changing the type of the first parameter from `[[String]]` to `[String]`.

Because Helium trusts type annotations in all cases, it will typically produce the most useful error messages in cases where that type annotation is indeed correct. When that type annotation is the cause of the type error, however, LT tends to produce more accurate error messages.

While Helium and LT both suggest expression changes in some circumstances, they do so in different ways. Helium frequently makes use of what it calls sibling functions. Siblings are pairs of functions which are in some way similar or offer related functionality. Example of this include `(:)` and `(++)` as

already witnessed, as well as `max` and `maximum` which find the maximum of two values and the maximum in a list of values, respectively. Literals can also be considered siblings, such as the string and character versions of a single letter (`"c"` and `'c'`) or the floating point and integer versions of a number.

In the following section we present a more detailed analysis of the particular situations for which Helium and LT are particularly strong and examine their overall success rates.

III. Evaluation of Lazy Typing and Helium

We are interested in leveraging the diversity of techniques among type checking tools, but combining them most effectively requires understanding the strengths and weaknesses of each. For this reason, we begin by evaluating both Helium and LT independently. This will then help to inform more general conclusions about combining type checking tools in later sections.

For evaluation purposes, we obtained a database of programs collected at Utrecht University in 2005 [15]. The full collection contains 11256 real programs, written by first-year undergraduate students learning Haskell. While each program is unique, some are sequences of programs which the students fix or improve over time. These are particularly useful, as some of these sequences involve fixing type errors. We can therefore use these fixed programs as oracles for correcting the earlier programs. This provides a practical, realistic, and objective way to evaluate and compare type checking techniques

We filtered the set of programs down to those which contained type errors in earlier iterations of the same program. To achieve this, we produced a script to run GHC on every program. We kept those which contained type errors, but omitted those which also contained other issues such as parsing errors as those are outside the scope of this work. Additionally, Helium allows for some extended, non-standard notation such as the `(*.)` operation for multiplying floating point values. We also excluded these programs, as LT does not support such non-standard syntax. After this process, we were left with 1069 unique, ill-typed programs. Some programs contained more than one type error, meaning we actually investigated 1133 separate type errors.

With this filtered database, we were able to run both our LT prototype and the Helium compiler on each program in order to compare the type error messages. The output of each was compared against the changes made in the oracle programs. The output of Helium/LT was deemed to be helpful and correct when the output agreed with the fix actually applied by the student programmer. This process was performed manually, and took approximately 200 hours of work.

To more clearly demonstrate how we determined whether an error message was deemed correct and helpful or not, let us return the example from Section I, which is, in fact, a snippet from one of the actual student programs [15]. The example is reproduced below, and the remaining lines show five different error messages of different kinds and for different locations. These messages were produced by LT, Helium, and some other tools; the purpose is to show some of the kinds of errors that can be found.

```
insertRowInTable :: [String] -> [[String]] -> [[String]]
insertRowInTable r t = r ++ t

(1) Change (++) to (:).
(2) Change (++) of type [a] -> [a] -> [a] to
      something of type [String] -> [[String]] -> [[String]].
(3) Change type annotation to
      [[String]] -> [[String]] -> [[String]].
(4) The type of insertRowInTable is incompatible
      with its signature.
(5) In the first argument of (++),
      couldn't match type Char with [Char].
```

For each type error message, we record the following information: (a) Whether the suggestion or error actually helps to repair the type error. Because some changes may technically fix the type error but completely change the behavior of the program, we used the oracle programs to ensure that the suggestions agreed with the actual intent of the programmer. (b) Whether or not it is an expression change suggestion. Expression change suggestions are those which are specific, code-based changes rather than more general messages (which frequently only refer to types). For example, messages `(1)` and `(3)` from above are deemed to be expression change suggestions because they recommend specific code changes, while the others are less specific and refer only to types. (c) When appropriate, why a message does not help to remove the type error. For example, if the true cause of the type error in the above example is the type annotation, then the messages `(1)`, `(2)`, and `(5)` are considered to be not helpful because they report errors at the wrong locations. Additionally, message `(4)` would also be considered unhelpful in this case. While technically correct, it is vague and fails to suggest a way to fix the error. Alternatively, if the true cause is the use of the `(++)` function, then the messages `(3)` and `(5)` are considered unhelpful because they point to wrong locations. Message `(4)` is still too vague, suggesting no specific changes, and is therefore still considered unhelpful. (d) Whether the type annotation is correct or not.

This information allows us to separate all messages into one of three error categories, which will help with the analysis. The categories are based on level of concreteness, which is also roughly analogous to usefulness. The least concrete category of error messages are those we call *fault location* messages. Typical type error messages have two components, namely the location in the source code (such as a line and column number) at which the error was determined to occur and the message itself, which explains why code at that location caused a type error. This category is only concerned with the former. In particular, this category is concerned with those error messages which provide a misleading fault location for the type error. Regardless what kind of type error has occurred, or what the suggestion is, if the oracle programs determine that the message produced an incorrect fault location, it belongs to this error category. Therefore, if an error message meets the criteria for both this and another category, then this category takes priority.

At the second level, slightly more concrete than fault location messages, we have *type change* suggestions. Type change suggestions are those which produce vague recommendations for changing the types of definitions in the code. A good example of this is message `(2)` from above. It follows the form "change X of type T_1 to something of type T_2", which is common for type change suggestions. Type change messages can be either correct or incorrect as well.

Finally, the most concrete category of error message is that

	Overall		Expression change (EC)				Type change (TC)				Fault location (FL)	
	C_o	C_o/N	N_e	N_e/N	C_e	C_e/N_e	N_t	N_t/N	C_t	C_t/N_t	N_l	N_l/N
Lazy typing	645	56.9%	252	22.2%	249	98.8%	579	51.1%	396	68.4%	302	26.7%
Helium	703	62.0%	309	27.3%	298	96.3%	673	59.4%	405	60.2%	151	13.3%
LT/H Oracle	1010	89.1%	506	44.7%	505	99.8%	605	53.4%	505	83.5%	22	1.95%

Fig. 1: Evaluation results for different type-checking approaches applied to ill-typed programs. The left-most columns, labeled "Overall", show the number of cases in which type errors are fixed (C_o) and the percentage of programs this represents (C_o/N). The N_e, N_t, and N_l columns show the total number of programs determined to be of kind expression change, type change, or fault location, respectively. The C_e and C_t columns report the number of expression change and type change suggestions, respectively, that actually fix the error. The remaining columns show derived ratio information. The third row (LT/H Oracle) denotes results obtained by always using the better output from either LT or Helium, serving as a theoretical maximum.

which we call an *expression change* suggestion, also described above. It suggests a specific change in the source code of the program, such as applying a different function or changing a type signature in a particular way. Like type change suggestions, expression change suggestions can be either correct or incorrect.

These three error categories together partition all of the error messages. This will be useful in analyzing which kinds of errors Helium and LT handle well and which they do not.

Figure 1 presents the results of running LT and Helium separately on each of the 1133 type errors in our database, broken down by the aforementioned classification scheme. The third row (LT/H Oracle) presents a theoretical maximum that could be achieved by always using the better output from either LT or Helium. That is, if LT produces a message which points to the wrong location for a given type error, but Helium has a correct type-change suggestion for that same error, then we say that the LT/H Oracle has the correct type-change suggestion and ignore the LT failure.

From these data we can observe a number of things. LT produced useful and correct error messages in 57% of all cases and Helium did so in 62% of all cases. Also, although neither tool produces a high ratio of expression change suggestions, those that are produced tend to be accurate. Type change suggestions account for the biggest partition of all error messages. Helium produces type change suggestions for more cases than LT, but suffers from slightly lower precision. Finally, LT produces more error messages than Helium that fall into the fault location category. Speculatively, this could be because LT, unlike Helium, does not always trust type annotations. Since the example programs in which the type annotations are correct outnumber those in which the annotations are incorrect, Helium gains an advantage by trusting them.

Both tools are substantially more effective in locating type errors and suggesting changes than common Haskell compilers. For the sake of comparison, a recent study showed that GHC produces useful error messages in approximately 20% of cases [16]. Helium provides useful feedback in more cases than LT, though not by a wide margin. More importantly, both Helium and LT still offer substantial room for improvement, as neither produces strong error messages in all situations.

What is not obvious from the data, however, is that LT and Helium do not completely overlap in the programs for which they are successful. The different approaches are strong in different situations. This means that, with a hypothetical oracle, a programmer that simply runs both LT and Helium and automatically selects the better of the two suggestions would receive helpful suggestions in 89% of the cases. This LT/H Oracle information is shown in the third row of Figure 1.

Of particular interest are the LT/H Oracle expression change suggestions. Helium and LT combine for a total of 561 such suggestions, of which only 56 occur for the same program. This means that, while Helium and LT can only make expression change suggestions in 27% and 22% of the examples, respectively, the LT/H Oracle can improve this to an impressive 45%. Since the accuracy of expression change suggestions is so high, this is a promising statistic. This increase in expression change suggestions comes with a cost, however, in that the maximum number of possible type change suggestions is actually reduced from those that Helium itself is capable of producing. This occurs because many of the type change suggestions that Helium and LT are able to produce separately become expression change suggestions when using the LT/H Oracle. Finally, we can observe that the number of unhelpful error messages pointing to the wrong source code location can be reduced remarkably using the LT/H Oracle. This should not be a surprise given the increases elsewhere.

Unfortunately, however, we cannot rely on the LT/H Oracle. Simply applying both techniques and producing two error messages is not always helpful since the programmer will not know which tool to trust when they disagree. Instead, we need a way to determine which of the two approaches is most likely to be correct in a given situation.

IV. Integrating Lazy Typing and Helium

We have seen that LT and Helium could potentially be combined to produce useful error messages in up to 89.1% of our test cases. Of course, this figure represents the theoretical best case, which could be achieved by someone who knows in advance what the correct answer is. The interesting question is how we could possibly integrate the two tools to *automatically* produce error messages that improve on both tools.

As mentioned previously, the high success rate of the theoretical combination is largely due to Helium and LT being strong in different situations. That this is true is witnessed by the fact that, of the 703 type errors correctly identified by Helium and the 645 identified by LT, only 338 are correctly found by both.

Following the results in Figure 1 and the corresponding analysis in Section III, we have derived a strategy to combine LT and Helium that consists of the following three rules.

			Lazy Typing			
			EC		¬EC	
			✓	✗	✓	✗
Helium	EC	✓	51	0	67	180
		✗	3	5	7	9
	¬EC	✓	76	2	-	-
		✗	119	15	-	-

Fig. 2: A breakdown of the cases in which LT or Helium (or both) made an expression change suggestion (EC). Note: ¬EC= TC∨FL.

		Lazy typing	
		✓	✗
Helium	✓	38	2
	✗	81	19

Fig. 3: A comparison of correctness for LT and Helium type error messages for programs that LT determines to have wrong type annotations.

- If either LT or Helium provides an expression change suggestion, accept it. If both suggest expression changes, prefer the suggestion from LT.
- Otherwise, if LT suggests that the type annotation is wrong, use the suggestion made by LT.
- Otherwise, use the suggestion made by Helium.

The motivation for the first rule can be gained by examining Figure 2. This table shows two rows for expression (EC) and non-expression (¬EC) changes by Helium, and two such columns for LT. Each row/column is further split into correct (✓) and incorrect (✗) cases. This leads to 12 possible combinations. The four empty cells are coincidental and show situations with no expression change suggestions, which are not relevant. The remaining cells show the number of programs in which that combination of suggestions occurred. For example, the cell with the value 7 tells us that there are 7 cases in which Helium made an incorrect expression change suggestion while LT made a correct non-expression change suggestion.

Please note that there may seem to be a discrepancy between the values in Figures 2 and 1. This occurs because in Section III the fault location category took precedence over the others, i.e. some expression change suggestions were categorized in fault location. In the current Section we discuss *all* expression change suggestions, even those that were previously categorized as fault location.

From the table, we can extract several specific reasons to justify always trusting expression change suggestions. (1) There is only minimal overlap in these cases. Out of the 534 cases in which either Helium or LT makes an expression change suggestion, only 59 of them are shared. (2) Expression changes are typically accurate, and a tool that trusts them will often deliver correct error messages. We can see that trusting Helium's expression change suggestions rather than LT's non-expression change suggestions only leads to 7 cases in which we could have done better. Similarly, trusting LT's expression change suggestions over Helium's non-expression change suggestions only leads to 2 situations in which we could do better. (3) In only very few cases do both tools have different expression change suggestions. There are only 3 such cases, all of which occur where LT is correct and Helium is incorrect. This directly supports preferring LT in cases where both produce an expression change suggestion. (4) There is little overlap between correct expression changes and correct non-expression changes when compared to correct expression changes and incorrect non-expression changes. We can see that Helium expression change suggestions provide useful error messages in 180 cases where LT would provide an incorrect message. In the other direction, LT provides correct expression change suggestions in 119 cases where Helium provides an incorrect non-expression suggestion.

Finally, we can see that out of the 534 expression change cases, there are only 38 cases which would be improved by the oracle solution, namely those in which either both expression change suggestions are incorrect (5) or only one is made and it is incorrect (that is, $7+9+2+15=33$).

To justify the second rule of our strategy, we need to examine information about the correctness of the type annotations in our programs. Unfortunately, we cannot simply rely on type annotations being correct. This rules out the use of Helium for these cases, which automatically trusts them. Fortunately, LT is reasonably accurate at finding incorrect type annotations. The program database contained 264 incorrect annotations and LT identified 243 of them. Out of the 1133 total type errors, LT produced 21 false negatives and 40 false positives, producing an acceptable margin of error in deciding the correctness of type annotations.

We can extract yet more information to justify our second rule from the table in Figure 3. Of the messages produced for the 264 incorrect type annotations, 140 do not contain an expression change suggestion from either tool. The other 124 are therefore handled by application of the first rule and so are not relevant here. Figure 3 presents a breakdown of these examples by whether or not LT and Helium have correct type change suggestions. We can observe that there are 38 cases in which both approaches are correct, 81 cases in which LT is correct and Helium is incorrect, and only 2 in which LT is incorrect and Helium is correct. This argues strongly in favor of preferring LT in cases where programs are reported to have incorrect type annotations.

To summarize the justification for the second rule, it correctly handles 119 out of 140 possible cases. There are only 2 programs in our database which the oracle would improve upon, where Helium is correct and LT is not. In the remaining 19 cases, neither Helium nor LT produces a correct message, and so no strategy will succeed.

The third rule in our strategy proves the most difficult. We have no syntactic way of distinguishing these cases, and neither tool produces special error messages for particular cases.

Figure 4 presents a breakdown of how LT and Helium perform on the remaining 459 cases. We observe that in 106 cases both tools produce useful error messages and in 75 cases the choice is irrelevant as both are incorrect. Of the remaining cases, 97 favor LT while 181 favor Helium. This suggests that our combined approach should default to favoring Helium in all

		Lazy typing	
		✓	✗
Helium	✓	106	181
	✗	97	75

Fig. 4: A breakdown of type error messages from LT and Helium for programs with correct type annotations and no expression change suggestions.

	Overall	Correct	Fault location	Other error
Rule 1	534	496	31	7
Rule 2	140	119	14	7
Rule 3	459	287	58	114
Sum	1133	902	103	128

Fig. 5: Effectiveness of the rules in the LT/H strategy.

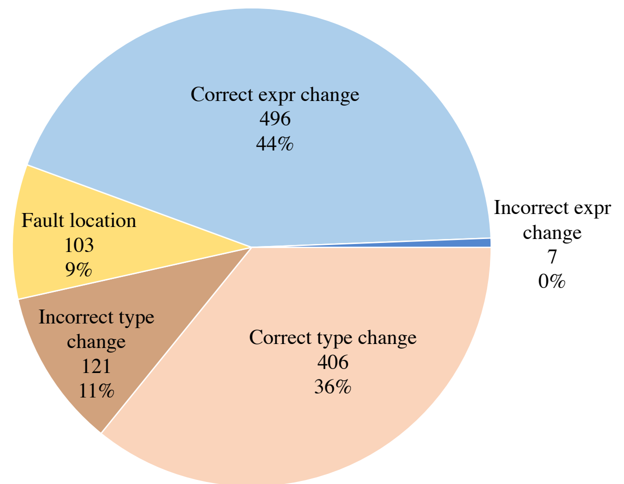

Fig. 6: Performance of the LT/H approach

remaining cases, simply because it is more likely to be correct for these situations. As a consequence of this, our strategy fails to produce useful error message in 172 cases.

By using these three rules, we have produced a strategy for integrating LT and Helium that always selects only one error message, eliminating the problems faced by a naive combination. From the user's perspective, this combination works just as Helium or LT would as a standalone tool except that it produces useful error messages in more cases than either.

Figure 5 summarizes the effectiveness of the LT/H strategy and the individual rules. For each, we present the number of cases that satisfy the condition of that rule, the number of programs for which the tool that the rule selects is able to suggest a useful type error fix, the number of cases in which the corresponding tool produces an unhelpful fault location, and all remaining possible mistakes. Overall, LT/H produces correct error change suggestions in 902 cases, representing 79.4% of our program database. It produces 103 total type error messages that provide an unhelpful fault location. In the remaining 128 cases, LT/H produces some other kind of unhelpful error message, for example, a type change suggestion at the correct fault location but with the wrong target type.

Finally, Figure 6 summarizes the overall performance of LT/H. In it, we categorize each type error message according to the classification scheme described and used in Section III. LT/H is able to achieve a correctness rate of 79.4%, improving substantially on Helium and LT as separate tools, which achieved 62.0% and 56.9%, respectively. From this we can conclude that the LT/H strategy is indeed effective.

V. General Strategies for Combining Tools

By reflecting on the process that we used to identify the LT/H strategy in Section IV, we can extract guidelines that apply to the general case of combining type error reporting (or other static analysis) tools.

In the following, we use A and B to refer to two arbitrary tools to be combined, and we use AB_s to denote the combination with respect to strategy s. In cases where the strategy is irrelevant to the discussion, we may simply drop s.

For a given combination of tools AB_s, there are two primary factors that affect its performance. First, both A and B have inherent limitations which will naturally restrict the performance of AB_s, regardless of the strategy used. This occurs in cases when neither A nor B is able to produce a helpful error message. Even a perfect oracle cannot improve this situation, as was the case in our running example. See Figure 1, which shows the inherent limitations of both Helium and LT, as well as the LT/H Oracle.

The second factor affecting the performance of AB_s is simply the number of cases in which our strategy is able to select the strongest tool for the situation. Unlike the LT/H Oracle, it is possible that the strategy's selected approach is the weaker of the two, resulting in cases we refer to as being misclassified.

In our running example, LT/H, there are 9 misclassified cases out of those handled by rule 1, which always trusts expression change suggestions. Rules 2 and 3 produce 2 and 97 misclassified suggestions, respectively. In total, then, LT/H produces 108 misclassified suggestions. Together with the 123 cases for which Helium and LT both produce incorrect suggestions, this sums to 231 cases in which LT/H produces an unhelpful message. This result can also be calculated from Figure 5.

From these data, we can conclude that, while the specific limitations of A and B are important, the strategy s is the most important aspect. In order to derive an effective general strategy, the first task must be to label the problem cases and classify them into categories, treating each separately. This enables the analysis of the problem in parts, maximizing the performance of s for each of the problem cases. Without understanding these problem cases, it is difficult to do better than by random chance.

In the case of LT/H, we were able to classify all cases into three different error categories: cases for which at least one tool produced an expression change suggestion, cases for which LT reported an incorrect type annotation, and all other cases. If, like this, problem cases can be divided into categories, then we can apply two principles that we call *category analysis* and *category-wise method analysis* (hereafter just *method analysis*) to help decide how to handle them.

Category analysis considers the manner in which the

individual problem cases are classified into categories most effectively. Individually, A and B might classify a single case into different categories, and so care needs to be taken when AB_s choosing between them. For example, for some of the programs we evaluated, Helium reported an expression change suggestion while LT reported an incorrect type annotation and made a type change suggestion. In this case, according to the category analysis principle, we should choose the category that has the highest probability of producing a correct result. Returning again to Figure 1, we can see that expression changes have a substantially higher accuracy rate than any other category, and so we chose to give preference to the expression change whenever one is produced.

In this example we have the good fortune to see that both tools are accurate when producing an expression change suggestion. This might not always be the case, however. In a different situation, we would derive a different strategy to take advantage of the strengths of the tools. In general, however, the category analysis principle offers guidance for making this determination.

Method analysis is the principle which guides the selection of a tool or technique for a given category. We assume that category analysis has already been applied, and thus that each problem case has been assigned to a category. As an example, we decided that, in the case where LT reports an incorrect type annotation (and no expression change suggestion is made by Helium) to always trust LT, as the data suggests this is more likely to be correct. Intuitively, the principle is to maximize the correctness rate for each of the categories.

This principle works particularly well when there are many small categories, as well as when two tools have high accuracy and correctness for disparate sets of categories. By the same logic, method analysis is less effective when two tools have similar performance for a single category, or when categories are large. This is also demonstrated by the LT/H example. Our strategy does well for the second category, which contains the programs that correspond to LT reports about incorrect type annotations, because LT handles this case much better than Helium with relatively few false positives. However, the category of all remaining programs is large and Helium outperforms LT by only a small margin, which means our strategy is not as effective. Of the programs that fall into the third category, LT/H misclassifies 21% of them, accounting for nearly all of the misclassifications.

One way of addressing this situation is to further refine the analysis and break down the large categories into several smaller categories. Unfortunately, this finer-grained analysis requires additional information, which may not always be available for the given tools. One possibility for the LT/H example would be to identify cases in the third category for which Helium only reports a unification failure.[2] We leave this for future work.

In summary, developing a strategy for combining tools comes down to two principles: category analysis and category-wise method analysis. Category analysis assigns a category for each problem case while method analysis decides which tool to use for each category. Increasing the number of categories will complicate the former while improving the precision of the latter.

We have employed a common machine learning cross-validation technique [17] to verify the effectiveness of our method-combining strategy and rules. First, we randomly divided the programs into two groups, using one half to derive rules based on our principles, which were then evaluated on the other half. We repeated this technique using one-third of the programs to derive rules, tested on the remaining two-thirds. In both cases, we derived the same set of rules discussed previously (Section IV). Moreover, the correctness ratios were both within $79.4 \pm 0.32\%$, a difference of no more than 0.32% compared to the original result.

A. Threats to Validity

There are three main threats to the validity of this work. First, all sample programs share a single data source. This could lead to a homogeneous programming style with a proclivity for avoiding or making certain types of errors. Until additional data sets are available, this is difficult to work around. Second, despite efforts to make a neutral and representative choice, it is possible that the combination of LT and Helium represents an anomalous case. Since we have not yet had the opportunity to perform an evaluation of other tool combinations, we cannot generalize the results. Finally, the coding of error messages was performed by one researcher, which could potentially lead to errors.

VI. Related Work

The challenge of accurately reporting type errors and producing helpful error messages has received considerable attention in the research community. Improvements for type inference algorithms have been proposed that are based on changing the order of unification [8], [7], suggesting program fixes [7], and program slicing techniques [11] to find all program locations involved in type errors. The discussion of technical and behavioral differences among disparate approaches is widely available in [12], [10] and in our technical report of LT [13]. We will therefore instead focus our discussion on the work most related to our own, as well as that which was not covered by these summaries.

The most recent approaches to debugging type errors are [16], [18], and [19]. The idea of [16] is to find all possible changes that would fix a given type error, and then to use heuristics to order those changes according to likelihood of being correct. Although that idea is potentially more powerful than LT, the approach does not offer the same diversity as that between Helium and LT, making it a poor choice for studying strategies for combining diverse tools. The idea of [18] is to generate a constraint graph for type inference problems. Rather than reporting all constraint satisfaction errors, however, Bayesian reasoning is applied to detect the most suspicious constraint, which is then reported to the user. One downside of this approach is that errors are only reported in terms of constraint satisfaction issues, which is rather low level and makes error messages quite difficult to understand. The approach described in [19] extends the work in citeCE14popl by taking the programmer's intentions into account when producing change suggestions. These intentions are elicited in the form of type annotations.

[2] An example of a unification failure is `Occurs check: cannot construct the infinite type: a0 = [a0]`.

Seminal takes the unique approach searching for a well-typed program that is similar to the ill-typed one by creating mutations of the original program and applying heuristics [20]. This search-based approach is both an advantage and a disadvantage. In some cases it is able to make correct change suggestions where other tools fail, but is prone to exotic suggestions in others.

Like LT, Johnson and Walz's unification-based approach also uses contextual information to help locate faults more accurately [2]. While LT resolves conflicts under the directive of ensuring the overall program is well-typed, their approach uses "usage voting" to resolve conflicts, in which the most common successful unification result is used.

Muşlu et al. investigated providing programmers with information about the consequences of suggested error fixes [21]. In order to speculatively apply them, this work relies on error suggestions having already been generated. Such an approach could, however, be used to supplement the suggestions LT provides users.

A number of previous works have shown success in combining multiple, complementary techniques or tools in order to utilize the strengths of both or to mitigate weaknesses [22], [23], [24], [25]. Using these examples as justification, we adopt a similar method and apply it to the domain of generating type error messages. In addition to identifying an effective combination (LT and Helium), we make the additional contribution of extracting general strategy-creation techniques that can be applied to any combination of type-error reporting tools.

VII. Conclusions

We have successfully improved the accuracy of two type-checking approaches by combining them into one tool. We did so by a careful analysis of the situations in which the individual tools succeed or fail. Reflecting on this approach we have also identified a general strategy for exploiting the diversity in tools to craft tools that are more powerful than the sum of their parts.

In future work we plan to investigate the combination of other type debugging tools, and also to refine the evaluation to more error categories. Additionally, we plan to investigate other general tool-combining strategies.

Acknowledgments

We thank Jurriaan Hage for sharing his collection of student Haskell programs with us. This work is supported by the National Science Foundation under the grants CCF-1219165 and IIS-1314384.

References

[1] P. J. Brown, "Error messages: The neglected area of the man/machine interface," *Commun. ACM*, vol. 26, no. 4, pp. 246–249, Apr. 1983.

[2] G. F. Johnson and J. A. Walz, "A maximum-flow approach to anomaly isolation in unification-based incremental type inference," in *ACM Symp. on Principles of Programming Languages*, 1986, pp. 44–57.

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

[4] M. Wand, "Finding the source of type errors," in *ACM Symp. on Principles of Programming Languages*, 1986, pp. 38–43.

[5] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *ACM SIGCHI Conf. on Human Factors in Computing Systems*, 2010, pp. 1019–1028.

[6] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: What can help novices?" in *ACM SIGCSE Symp. on Computer Science Education*, 2008, pp. 168–172.

[7] B. J. McAdam, "Repairing type errors in functional programs," Ph.D. dissertation, University of Edinburgh. College of Science and Engineering. School of Informatics., 2002.

[8] O. Lee and K. Yi, "Proofs about a folklore let-polymorphic type inference algorithm," *ACM Trans. on Programming Languages and Systems*, vol. 20, no. 4, pp. 707–723, Jul. 1998.

[9] J. Yang, "Explaining type errors by finding the source of a type conflict," in *Trends in Functional Programming*. Intellect Books, 2000, pp. 58–66.

[10] J. R. Wazny, "Type inference and type error diagnosis for hindley/milner with extensions," Ph.D. dissertation, The University of Melbourne, January 2006.

[11] T. Schilling, "Constraint-free type error slicing," in *Trends in Functional Programming*. Springer, 2012, pp. 1–16.

[12] B. J. Heeren, "Top quality type error messages," Ph.D. dissertation, Universiteit Utrecht, The Netherlands, 2005.

[13] S. Chen and M. Erwig, "Better Type-Error Messages Through Lazy Typing," Oregon State University, Technical Report, 2014, http://web.engr.oregonstate.edu/~chensh/Docs/tr-lt.pdf.

[14] M. Erwig and E. Walkingshaw, "The Choice Calculus: A Representation for Software Variation," *ACM Trans. on Software Engineering and Methodology*, vol. 21, no. 1, pp. 6:1–6:27, 2011.

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

[16] S. Chen and M. Erwig, "Counter-Factual Typing for Debugging Type Errors," in *ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 2014, pp. 583–594.

[17] R. Kohavi, "A study of cross-validation and bootstrap for accuracy estimation and model selection," in *Int. Joint Conf. on Artificial Intelligence (Vol. 2)*, 1995, pp. 1137–1143.

[18] D. Zhang and A. C. Myers, "Toward General Diagnosis of Static Errors," in *ACM Symp. on Principles of Programming Languages*, 2014, pp. 569–581.

[19] S. Chen and M. Erwig, "Guided Type Debugging," in *Int. Symp. on Functional and Logic Programming*, ser. LNCS 8475, 2014, pp. 35–51.

[20] B. Lerner, M. Flower, D. Grossman, and C. Chambers, "Searching for type-error messages," in *ACM Int. Conf. on Programming Language Design and Implementation*, 2007, pp. 425–434.

[21] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," in *Proceedings of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications*, 2012, pp. 669–682.

[22] J. Lawrence, R. Abraham, M. M. Burnett, and M. Erwig, "Sharing Reasoning about Faults in Spreadsheets: An Empirical Study," in *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, 2006, pp. 35–42.

[23] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourw, "Applying and combining three different aspect mining techniques," *Software Quality Journal*, vol. 14, no. 3, pp. 209–231, 2006.

[24] R. Pelánek, V. Rosecký, and P. Moravec, "Complementarity of error detection techniques," *Electronic Notes in Theoretical Computer Science*, vol. 220, no. 2, pp. 51–65, 2008.

[25] J. Tschannen, C. Furia, M. Nordio, and B. Meyer, "Usable verification of object-oriented programs by combining static and dynamic techniques," in *Software Engineering and Formal Methods*, 2011, pp. 382–398.