Exploiting Diversity in Type Checkers for Better Error Messages

Sheng Chen

CACS, UL Lafayette

Martin Erwig, Karl Smeltzer

School of EECS, Oregon State University

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 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 82%, which is an improvement of 25%/20% 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.

Email addresses: chen@louisiana.edu (Sheng Chen), {erwig,smeltzek}@oregonstate.edu (Martin Erwig, Karl Smeltzer)

Keywords: Type-error diagnosing, tool combining

1. 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 (Brown, 1983; Johnson and Walz, 1986; Milner, 1978; Wand, 1986). This problem goes back several decades (even to the original Hindley-Milner type system), and is still considered to be unsolved (Hartmann et al., 2010; Nienaltowski et al., 2008).

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. Sometimes error messages are not only complex, but even incorrect or misleading. In particular, type error messages sometimes include line numbers which do not point to the actual error. In some cases, this location is far away from the actual error source, or even in the wrong file.

Quite a few solutions have been proposed to locate type errors more accurately. One idea is to eliminate the left-to-right bias of type inference (McAdam, 2002; Lee and Yi, 1998). Another is, instead of committing to just one error location, to report several program sites that most likely contribute to the type inconsistency (Yang, 2000; Wazny, 2006). A related technique uses program slicing to determine all the positions that are related to the type errors (Schilling, 2012). Finally, constraint solving has been proposed as a means to minimize the number of possible error locations (Heeren, 2005).

However, despite considerable research effort 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 introduced in Wazny (2006) 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)
```

This definition contains a type error. In the body of the second case alternative, the variable `x` should be replaced with `[x]` (Wazny, 2006). Existing type inference systems have a hard time locating it correctly.

When type checking this example, the Glasgow Haskell Compiler (GHC) 7.6,[1] which is the most widely used Haskell compiler, produces the following message.

```
Occurs check: cannot construct the infinite type: a0 = [a0]
In the first argument of ‘(:)’, namely ‘y’
In the expression: y : ys
In the expression: (x : xs, y : ys)
```

This message about being unable to construct infinite types employs compiler jargon and incorrectly points the programmer to the very last line in the function.

The Chameleon Type Debugger (Wazny, 2006), a representative of the more advanced error slicing tools, displays the following information.

```
ERROR: Type error
Problem : Case alternatives incompatible
Types   : [a] -> (b,a)
          [a] -> (c,[a])
Conflict: split xs = case xs of
            [] -> ([],[])
            [x] -> ([],x)
            (x:y:zs) -> let (xs, ys) = split zs in (x:xs, y:ys)
```

The advantage of slicing approaches is that they include all locations that may cause the type error and exclude all locations that don't contribute to the error. A drawback of this approach is, however, that it often includes too many locations (Chen and Erwig, 2014b). In this example, Chameleon notes type mismatches across multiple lines, which still requires the programmer to distill the information into an actual fix for the problem.

[1] `www.haskell.org/ghc/`

Helium (Heeren, 2005) was designed to produce well-rounded results that are more helpful than those from other tools. For our example it suggests changing the cons function `(:)` to the list append function `(++)`. This change would indeed fix the type error, but it would also cause the types of the other two alternatives to change as well. Helium's change suggestion also doesn't agree with the type error fix described in (Wazny, 2006), where the original example was introduced.

We have recently developed a new approach, called *Lazy Typing* (LT) (Chen and Erwig, 2014a), which improves type error messages for many of these situations. For this particular example, LT presents the following message.

```
(3,31): Type error in expression:
  x
Of type:  a
Should have type:  [a]
```

It identifies the error as occurring in the expression `x` on the right-hand side of the second case expression—exactly where we would hope. The error message produced suggests that `x` is of polymorphic type `a`, but should be of type `[a]`, or "list of `a`".

With 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 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.

Previous Work and Structure of the Paper

This paper is an extended version of Chen et al. (2014a). It contains the following changes and additions compared to the original paper.

1. We give a more through discussion of the examples in Section 1. Specifically, we present example error messages from several tools to explain the different approaches to type error debugging.

2. We present a more detailed discussion of different kinds of type error messages in Section 3.

3. We have refined our combining strategy by adding a fourth rule to the original set of three rules. This refinement gains about 3% of accuracy over the original combining strategy. Although 3% may look like only a small improvement, it is actually 30% of all the improvement that is possible, because our original strategy achieved 79%, which is only 10% below the theoretically best strategy which has an accuracy of 89%. Section 4.2 presents this strategy in detail.

4. We present an improved version of general strategies for combining tools in Section 5. Specifically, the refined analysis discussed in Section 4.2 revealed that, in general, an iterative process is needed to produce a good combining strategy.

5. We have expanded the discussion of related work in Section 6.

6. This paper is not about visual languages per se. However, the results and insights of this paper can be translated to visual languages in several ways. For example, many visual languages (Burnett, 1993; Djang et al., 2000; Clerici et al., 2014) also use type inference, and the work in this paper provides a method to improve type error debugging in these languages. Moreover, type error debugging can be integrated into visual approaches to type inference. For example, the visual transformation steps described in (Erwig, 2006) can serve as explanations for how to arrive at types and type errors. That method could be adapted to the different approaches for producing type error messages studied in this paper, and thus, in addition to producing correct type error messages in more cases, we could also obtain explanations for how the conclusion about the errors were arrived at.

In Section 2, we introduce and explain the principles behind lazy typing and expand upon Helium as a representative for comparison. Section 3 evaluates Helium and LT separately in order to specifically identify conditions under which each is strong. Section 4 discusses how Helium and LT can be combined and evaluates the success of doing so. In Section 5 we extract general principles for combining other type error reporting

methods and tools. Finally, we discuss related work in Section 6 and provide conclusions in Section 7.

2. Lazy Typing and Helium

Next we will examine LT and Helium in greater detail. This will help us 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 can, at least in principle, support finding more accurate type error locations and also improve potential change suggestions provided to the programmer (Chen and Erwig, 2014a).

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 $A\langle\tau, \tau'\rangle$ (Erwig and Walkingshaw, 2011). 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* (*LT*). This strategy is based on the concept of variational types and a variational unification algorithm presented in Chen et al. (2014b).

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.

As might be expected, the difference between LT and Helium causes them to excel in

quite different situations. Consider, for example, 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, represented as a list of list of strings, into which the row is to be inserted. 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 change suggestion is generated since there is not enough evidence about where type error occurs. 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 correct. When that type annotation is the cause for 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 expressions. One form of siblings is 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 summary, we choose to combine LT and Helium in this paper, because they are likely to produce correct error messages in different situations. For example, LT works better when the type annotation is incorrect and Helium works better otherwise. As another example, they can provide expression change suggestions for very different cases. More reasons for choosing these two tools are provided in Section 3.2 when more detailed examples and terminologies have been introduced.

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

3. 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.

3.1. Evaluation Programs

For evaluation purposes, we obtained a database of programs collected at Utrecht University from 2002 to 2005 (Hage, 2013). The full collection contains 11256 real programs, written by first-year undergraduate students learning Haskell using Helium. More detailed information concerning this collection can be found in (Hage and van Keeken, 2009). 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 references for correcting the earlier programs. To provide a practical, realistic, and objective way to evaluate and compare type checking techniques, we define the notion of a reference program as follows.

Definition 1 (Reference program). *Given an ill-typed program P_i, if P_i appears in the program editing sequence $\ldots, P_i, \ldots, P_j, \ldots$, then P_j is the reference program for P_i if P_j is first well typed program after P_i in the sequence.*

Note that this definition doesn't ensure that the reference program actually fixes the type error. It could very well happen that an expression causing the type error is simply removed (together with all expressions that directly or indirectly depend on it). This would be bad, because it would avoid the problem of fixing the error, and in that case the reference program would not provide a good oracle for the evaluation. Fortunately, this did not occur in any of the ill-typed programs we investigated.

Thus, intuitively, P_j reflects how the student fixed the type error in P_i. In the definition, P_i can appear anywhere but at the end of a sequence. We found reference programs for all of the ill-typed programs we investigated.

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 we cannot run type checkers on programs that fail to parse. 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.

3.2. Error Message Categories

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. A manifesto for measuring the quality of type error messages was proposed in Yang et al. (2000). Unfortunately this manifesto is not helpful for our scenario since it judges all of the error messages generated by LT and Helium as good messages. Instead, we compare the output of both LT and Helium against the changes made in the reference programs to determine their correctness. The following definition specifies more formally when error messages are considered to be correct or incorrect.

Definition 2 (Error message correctness). *An error message em is said to be* correct *for the ill-typed program P_i if following the suggestions given by em will transform P_i to its reference program P_j. Otherwise, the message is* incorrect*.*

Based on this definition, we classified all of the Helium/LT messages as either correct or incorrect. This process was performed manually and took approximately 200 hours of work. To explain how we determined whether an error message was deemed correct or not, let us return to the example from Section 1, which is, in fact, a snippet from one of the actual student programs (Hage, 2013). The example is reproduced below; the numbered lines show different error messages of different kinds and for different locations. These messages were produced by LT, Helium, and GHC; 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].
(6) Type error in variable
    expression: (++)
    type:          [a] -> [a] -> [a]
    expected type: b   -> [b] -> [b]
    because: unification would give infinite type
```

For each type error message, we record the following information:

(a) Whether the suggestion is correct. Because some changes may technically fix the type error but completely change the behavior of the program, we determine if a message is correct based on Definitions 1 and 2.

(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 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 reference program shows the type error in the above example is the type annotation, then the messages `(1)`, `(2)`, and `(5)` are considered to be incorrect, because they report errors at the wrong locations. Additionally, message `(4)` would also be considered incorrect in this case. While technically correct, it is vague and fails to suggest a way to fix the error. Alternatively, if the reference program shows the error is the use of the `(++)` function, then messages `(3)` and

`(5)` are considered incorrect because they point to wrong locations. Message `(4)` is still too vague, suggesting no specific changes, and is therefore still considered incorrect.

(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 an indication of how directly they can be employed to fix an error. 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 includes all the error messages whose locations differ from the fixes in the corresponding reference programs for the ill-typed programs. Regardless what kind of type error has occurred, or what the suggestion is, if the reference program determines that the message produced an incorrect 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. The reason for this decision is that any correct error message must get the location right. If an error mesage cannot even point to the proper location, then any further information it provides is meaningless and only confusing and misleading.

At the second level, slightly more concrete than fault location messages, we have *type change* suggestions. Type change suggestions are those which produce recommendations for changing the types of definitions in the code. A good example is message `(2)` from above. It follows the form "change X of type τ_1 to something of type τ_2", which is common for type change suggestions. Type change messages can be either correct or incorrect as well. In many cases, Helium produces another form of type changes in terms of unification failures. One such case is the message `(6)` in the `insertRowInTable` example. The message `(6)` is produced by Helium if we change the type annotation of `insertRowInTable` to `a -> [a] -> [a]`.

Finally, the most concrete category of error message is that 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. An expression change suggestion is more concrete than a type change suggestion because a type change suggestion still requires the programmer to determine a specific expression to use, while an expression change suggestion doesn't. Consider the type change suggestion `(2)` and the expression change suggestion `(1)` in the `insertRowInTable` example. Given the message `(1)`, the programmer simply changes `(++)` to `(:)`. For the message `(2)`, the programmer has to decide what function to replace `(++)` with. While `(:)` is a candidate, some other possibilities exist, for example, the `intersperse` function from the Haskell library. Like type change suggestions, expression change suggestions can be either correct or incorrect.

Now that we have seen a detailed example of different categories of error messages, we can give concrete reasons for choosing to combine LT and Helium.

1. *Internal diversity.* Each tool can generate different types of error messages. We have seen that Helium can report unification failures, type change suggestions, and expression change suggestions. Likewise, LT can report problems related to type annotations, type change suggestions, and expression change suggestions. The existence of different categories allows us to take different actions for different categories and creates fine-grained combining strategies. This is the reason why we did not choose, for example, the Chameleon Type Debugger (Wazny, 2006), which reports minimum error sources, Skalpel (Haack and Wells, 2003), which slices type errors, nor GHC, which mainly reports unification failures as combining candidates.

2. *External diversity.* Different tools produce different kinds of information or produce the same kind of information based on different methods. In our case, LT can report problems in type annotations while Helium can report unification failures. Although they both produce expression change suggestions, they do so based on different methods. This is the reason why we did not choose, for example, to combine Chameleon and Skalpel— they can both be considered

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

Figure 1: Evaluation results for different type-checking approaches applied to ill-typed programs in number (upper part) and in percentage (lower part). Note that N denotes the number of all type errors inspected and thus is 1133. The left-most column, labeled "Overall", shows 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 LT/H Oracle rows denote results obtained by always using the better output from either LT or Helium, serving as a theoretical maximum.

slicing tools—different versions of Helium, nor LT and counter-factual typing since they share many commonalities.

3.3. Results of Individual Tools

The three error categories described in Section 3.2 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 LT/H Oracle rows present a theoretical maximum that could be achieved by always using the better output from either LT or Helium. That is, if LT produces a fault location message 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 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 correct error messages in approximately 20% of cases (Chen and Erwig, 2014b). Helium provides correct 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 correct 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 correct suggestions in 89% of the cases. This LT/H Oracle information is shown in the Lt/H Oracle rows in 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

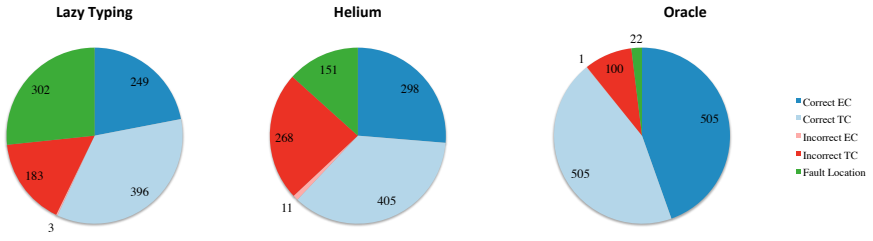

Figure 2: Performance of Lazy Typing, Helium, and the LT/H Oracle.

this to 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 fault location messages can be reduced remarkably using the LT/H Oracle. This should not be a surprise given the increases elsewhere. Figure 2 presents a summary of the performances of LT, Helium, and the theoretical Oracle.

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. Our goal here is to develop a strategy for combining the tools so that for each ill-typed program only one error message is displayed, which is more likely to be correct when any of LT or Helium is correct. The cost for programmers when using the combining approach is that they may have to get used to three or four kinds of error messages produced by the tool they don't already use. For example, if programmers previously used Helium, then they need to get used to three types of error messages produced by LT when the combining approach is used.

4. 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 correct 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 3, we have derived two strategies to combine LT and Helium. The first strategy is the one presented in the conference version of this paper (Chen et al., 2014a), and the second strategy is a refinement of the first strategy. Both strategies share the following first two rules. The remaining rules for both strategies are presented in Subsections 4.1 and 4.2, respectively.

Rule 1. *If either LT or Helium provides an expression change suggestion, accept it. If both suggest expression changes, pick the suggestion from LT.*

Rule 2. *Otherwise, if LT suggests that the type annotation is wrong, use the suggestion made by LT.*

The motivation for the first rule can be understood by examining Figure 3. 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.

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

Figure 3: A breakdown of the cases in which LT or Helium (or both) made an expression change suggestion (EC). Note: ¬EC= TC∨FL.

Please note that there may seem to be a discrepancy between the values in Figures 3 and 1. This occurs because in Section 3 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.

The table provides several specific reasons for 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 different

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 correct 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 4. 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 4 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 is a strong point 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

		Lazy typing	
		✓	✗
Helium	✓	38	2
	✗	81	19

Figure 4: A comparison of correctness for LT and Helium type error messages for programs that LT determines to have wrong type annotations.

		Lazy typing	
		✓	✗
Helium	✓	106	181
	✗	97	75

Figure 5: A breakdown of type error messages from LT and Helium for programs with correct type annotations and no expression change suggestions using the original strategy.

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.

4.1. Completing the first strategy

In our original strategy (Chen et al., 2014a), the following rule 3 is used for all the cases that are not handled by rules 1 and 2.

Rule 3. *Use the suggestion made by Helium.*

This rule in our strategy proves the most difficult as we have no obvious syntactic way of distinguishing these cases.

Figure 5 presents a breakdown of how LT and Helium perform on the remaining 459 cases. We observe that in 106 cases both tools produce correct 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

	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

Figure 6: Effectiveness of the rules in the LT/H strategy.

favoring Helium in all 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 rules 1, 2, and 3, 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 6 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 produce a correct message, the number of cases in which the corresponding tool produces fault location messages, 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 are fault location. In the remaining 128 cases, LT/H produces some other kind of incorrect error message, for example, a type change suggestion with a wrong target type.

Finally, Figure 7 summarizes the overall performance of the original strategy with rules 1, 2, and 3. In it, we categorize each type error message according to the classification scheme described and used in Section 3. This strategy 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 this combining strategy is quite effective.

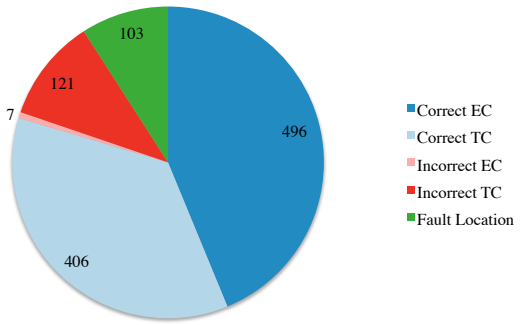

Figure 7: Performance of the LT/H approach using rules 1, 2, and 3

4.2. Refined rules for an improved strategy

We observe that rule 3 presented in the previous subsection handles 459 programs, which is about 41% of all the 1133 ill-typed programs. Moreover, Helium outperforms LT by only a small margin, which means that the combining strategy is not that effective in this case. Of the programs that fall into this category, this rule misclassifies 97 programs, which is about 21% of them. As rules 1 and 2 only have 11 misclassifications, rule 3 accounts for almost all misclassifications.

We address this issue by employing the additional information about unification failures in error messages reported by Helium. Using this information, we can refine rule 3 as follows by using two rules.

Rule 3a. *For all the cases that are not handled by rules 1 and 2, if Helium complains about unification failures, use the suggestion made by LT.*

Rule 3b. *Otherwise, use the suggestion made by Helium.*

The rule 3a says that for the cases that are not handled by the first two rules, we use LT's messages if Helium reports a unification failure. This rule takes the advantage of Helium's goal to produce high-quality error messages. Since messages about unification failure usually involve compiler jargon and are hard to understand, Helium tries to avoid such messages and falls back to using them only when no better suggestions can be made. Thus, the messages that involve unification failure reports are usually incorrect.

		Lazy typing	
		✓	✗
Helium	✓	7	5
	✗	35	20

Figure 8: A breakdown of type error messages from LT and Helium for programs for which Helium complains about unification failure.

		Lazy typing	
		✓	✗
Helium	✓	99	176
	✗	62	55

Figure 9: A breakdown of type error messages from LT and Helium for programs with correct type annotations, no expression change suggestions, and no unification failures reported by Helium.

In these cases, LT performs much better. Figure 8 presents the details about using this rule. We observe that, in total, 67 cases are handled by this rule. Among these cases, Helium outperforms LT in 5 cases while LT outperforms Helium in 35 cases. Thus, we choose to use LT's messages in these cases. Our strategy fails to produce correct error messages in 25 cases by applying this rule.

All the remaining cases are handled by rule 3b. Figure 9 presents a breakdown of how LT and Helium perform on the remaining 392 cases. We observe that in 99 cases both tools produce correct error messages and in 55 cases the choice is irrelevant as both are incorrect. Of the remaining cases, 62 favor LT while 176 favor Helium. Thus we use Helium's messages simply because they are more likely to be correct in this situation. As a result, our strategy fails to produce correct error message in 117 cases.

Like the original strategy, the refined strategy using rules 1, 2, 3a, and 3b also selects one message from either Helium or LT. Figure 10 summarizes the effectiveness of the refined strategy and the individual rules. To increase readability, we repeat the results

	Overall	Correct	Fault location	Other error
Rule 1	534	496	31	7
Rule 2	140	119	14	7
Rule 3a	67	42	11	14
Rule 3b	392	275	38	79
Sum	1133	932	94	107

Figure 10: Effectiveness of the rules in the refined strategy.

for rules 1 and 2. Similar to Figure 6, Figure 10 shows for each rule the number of cases handled by that rule, the number of cases that rule picks correct messages, the number of cases that rule picks messages have fault locations, and the number of all other cases.

Overall, the refined strategy produces correct error change suggestions in 932 cases, representing 82.3% of our program database. It produces 94 total type error messages that have fault locations and 107 messages that are incorrect for removing type errors for other reasons.

By comparing Figures 8 and 9 with Figure 5, we can see that the refined strategy helps to improve the combining performance. In particular, while rule 3 misclassifies 97 of all the 459 cases handled by that rule, the rules 3a and 3b misclassify 67 cases in those 459 cases. While rule 3 can produce correct error messages in 287 cases, the number for rules 3a and 3b is 317. Overall, this adds 30 more cases to the original 902 cases yielding 932 cases for which this combining strategy can help remove the type error. Thus, the refined combining strategy achieves 82.3% of accuracy, compared to the 79.4% of the original strategy. Considering that the oracle achieves the maximum possible 89.1% of accuracy, using rules 3a and 3b over rule 3 mobilizes 30% of the remaining potential for further improving the accuracy of the combining approach.

Finally, Figure 11 summarizes the overall performance of the refined strategy. We categorize each type error message as we did in Figure 2. The refined strategy is able to achieve a correctness rate of 82.3%. Compared to the accuracy rate of Helium and LT as separate tools, which are 62.0% and 56.9%, respectively, the refined strategy performs quite well.

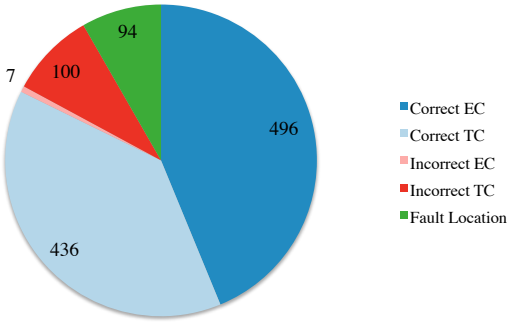

Figure 11: Performance of the refined strategy using rules 1, 2, 3a, and 3b

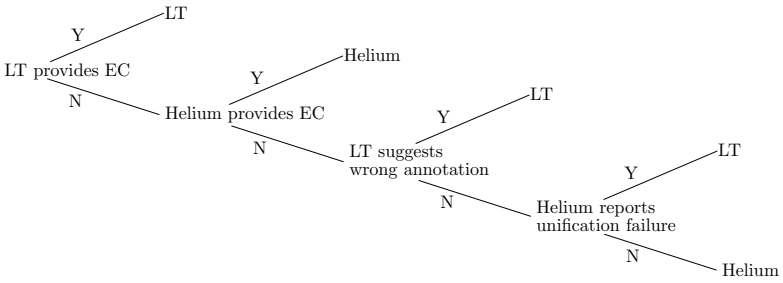

Figure 12: Decision tree representation of the refined combining strategy

Relationship to machine learning. We can represent combining strategies using decision trees. For example, Figure 12 shows the decision tree for the refined strategy. Given an ill-typed program, the internal nodes ask questions about the properties of the messages produced by LT and Helium, and the leaf nodes decide which tool to use to produce the error message. For example, if neither Helium nor LT produces expression change suggestions and LT suggests that the type annotation is wrong, then the message produced by LT should be used. This structure suggests a close relation between the work in this paper and supervised machine learning techniques, such as decision-tree-based learning.

In fact, we can use an existing algorithm, such as CART (Breiman et al., 1984), to learn the refined strategy if we are given all the error messages that have already been labeled with the necessary information. In particular, for each message, the data has to

tell whether LT or Helium provides an expression or type change suggestion, whether LT suggests that the type annotation is wrong, whether Helium complains a unification failure, and more importantly, whether the message is correct or not. Unfortunately, we are not aware of the existence of any such data sets. Thus we cannot apply any existing supervised machine learning algorithms without first performing the labeling, which is part of what this paper provides.

One could also try to apply an unsupervised learning algorithm to separate all the error messages into correct and incorrect messages. This is indeed a non-trivial research problem. For example, it is unclear what information should be extracted from highly arbitrary compiler error messages (known as *feature extraction* (Narayanan et al., 2012; Caliskan-Islam et al., 2015)) and what information from extracted features should be used (known as *feature selection* (Narayanan et al., 2012; Caliskan-Islam et al., 2015)) for clustering error messages into correct and incorrect. A particular challenge to this approach is that deciding whether an error message is correct requires context information. It is conceivable that one particular error message may be viewed as both correct or incorrect depending on the particular situation. Overall, we can conclude that applying unsupervised learning to separate error messages is infeasible at this time. Nevertheless, the work done in this paper, in particular the labeling of error messages, provides many machine learning research opportunities to help improve type error debugging. It may be interesting, for example, to see if applying some advanced machine learning techniques could discover a combining strategy with a higher accuracy rate than ours.

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, the coding of error messages was performed by one researcher, which could potentially lead to errors. Third, we determine the reference programs by finding the first well-typed programs in sequence after an ill-typed program appears. However, it is possible that in some situations later well-typed programs in the sequences should be used as the reference

programs instead. Additionally, one potential reason Helium performed marginally better than LT overall on ill-typed programs might be because the students writing the programs used Helium as their compiler. However, this should not affect the overall combining result either way.

5. General Strategies for Combining Tools

By reflecting on the process that we used to identify the combining strategies in Section 4, 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 correct error message. Even an oracle cannot improve this situation, as was the case in our running example. See Figures 1 and 2, which show 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. Rule 2 produces 2 misclassified suggestions, rule 3 produces 97, rules 3a and 3b produce 5 and 62, respectively. In total, the original strategy with rules 1, 2, and 3 produces 108 misclassified suggestions and the refined strategy with rules 1, 2, 3a, and 3b produces 78 misclassified suggestions. Together with the 123 cases for which Helium and LT both produce incorrect suggestions, this sums to 231 cases in the original strategy and 201 cases in the refined strategy in which the combination produces an incorrect message. These results can

also be calculated from Figures 6 and 10.

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 four different error categories: cases for which at least one tool produced an expression change suggestion, cases for which LT reported an incorrect type annotation, cases for which Helium report a unification failure, 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. Moreover, the process of searching a good combining strategy needs to go through multiple iterations. Each iteration begins with category analysis and ends with method analysis. If method analysis performs poorly for certain categories, then category analysis should be redone with more refined analyses, and another iteration is performed.

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. 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.

Similarly, method analysis is less effective when two tools have similar performance for a single category, or when categories are large. For example, the original combining strategy including rules 1, 2, and 3 experienced this problem. Specifically, the category three for the original strategy contains 459 cases and Helium performs only slightly better than LT. As a result, we can see that rule 3 accounts for 89.8% (97/108) of all misclassification in the original strategy. Based on the discussions in the beginning of this section, misclassifications account for the performance discrepancy between the result of the combined strategy and the theoretically best result. Thus, rule 3 is the main cause for the 9.7% (89.1%-79.4%) difference between LT/H Oracle and the original strategy.

One potential solution to address this issues was already hinted at in the refined combining strategy presented in Section 4.2: refining the analysis and breaking down the large categories into several smaller categories. In this case study, we performed another iteration of category analysis and method analysis. More specifically, for this category, we further considered the category of cases reported by Helium as unification failures. Since this kind of message is in general not correct for removing type errors, we identify

these programs and suggest to use LT's messages for them. This effort splits the original big category into two smaller categories: the category that represents Helium unification failures and all others. Moreover, since rule 3a for the unification failure category produces a low misclassification rate, and since the misclassification rate for rule 3b is also lowered compared to rule 3, which changed from 21% (97/459) to 16% (62/392), the split is quite effective. This can be witnessed as the total misclassifications have dropped from 108 in the original strategy to 78 in the refined strategy. Consequently, the overall accuracy rate has increased from 79.4% to 82.3%, a difference of about 3%.

Nevertheless, after the refined analysis, the last category is still big. It contains 392 cases. Worse, the misclassification rate is still high. It contains 62 misclassifications, which accounts for 79.5% (62/62+9+2+5) of all misclassifications. This shows that the last category is still the main cause for the precision loss of the refined strategy compared to the LT/H Oracle. Again, the solution to this issue is to further split this category, just as we did for the refined strategy. The last big category might be further refined by considering the complexities of type changes. For example, if one tool suggests to change a subexpression from one type to a similar type while the other tool suggests to change the entire expression from one type to a totally different type, then the suggestion from the former tool may be preferred.

In general, the way to deal with big categories and tools being combined having similar performance for these categories is to refine the analysis. Unfortunately, this finer-grained analysis requires additional information, which may not always be available for the given tools.

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.

When choosing tools for combining, an important factor to consider is the number of categories into which a tool's messages can be classified. If both tools used for combining produce only one category of error messages, then it is hard to derive a combing strategy that outperforms the better of the two original tools. In contrast, if both

tools can generate many categories of error messages and if the tools excel in different situations, then it should be beneficial to combine them according to our guidelines. We have witnessed such a case with LT and Helium.

6. 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 (Lee and Yi, 1998; McAdam, 2002; Lee and Yi, 2000; Yang, 2000), suggesting program fixes (McAdam, 2002), and program slicing techniques to find all program locations involved in type errors (Choppella, 2002; Haack and Wells, 2003; Schilling, 2012; Tip and Dinesh, 2001; Kustanto and Kameyama, 2010). The discussion of technical and behavioral differences among disparate approaches is widely available in Heeren (2005); Wazny (2006) and in our technical report of LT (Chen and Erwig, 2014a). We will therefore instead focus our discussion on the work most closely related to our own, as well as that which was not covered by these summaries.

The most recent approaches to debugging type errors are Chen and Erwig (2014b), Zhang and Myers (2014), and Chen and Erwig (2014c). The idea of Chen and Erwig (2014b) 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 since it does not produce information about type annotations, making it a poor choice when exploring strategies for combining diverse tools. The idea of Zhang and Myers (2014) 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 problems, which is rather low level and makes error messages quite difficult to understand. The approach described in Chen and Erwig (2014c) extends the work in Chen and Erwig (2014b) by taking the programmer's

intentions into account when producing change suggestions. These intentions are elicited in the form of type annotations.

Seminal (Lerner et al., 2007) 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. 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 (Johnson and Walz, 1986) also uses contextual information to help locate faults more accurately. 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.

Erwig (2006) and Jung and Michaelson (2000) have employed visualization approaches to aid the understanding of the type inference process for users. Clerici et al. (2014) presents a type inference system for the graphic language NiMo, which is a visual language similar in some ways to Haskell.

Muşlu et al. (2012) investigated providing programmers with information about the consequences of suggested error fixes. In order to speculatively apply them, this work relies on error suggestions having already been generated. Such an approach could 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. We present several such examples to demonstrate the breadth of areas in which such an approach might be effective. Specht (1990) reviewed two complementary classification techniques which share common decision boundaries but are most suitable in different real world situations . Lawrence et al. (2006) used a probabilistic model based on actual previous user behavior to combine feedback about spreadsheet faults. Much like with type checking, they found that combined approaches tended to outperform individual approaches in the context of spreadsheets. Ceccato et al. (2006) presented a case study of applying three different aspect mining techniques, allowing them to identify the individual strengths of each. They also proposed a set of rules for combining them to

maximize effectiveness. Pelánek et al. (2008) compared techniques for reachability analysis with the explicit goal of finding complementary techniques.

Combined techniques have also been useful in dealing with digital media. Nagy et al. (2000) developed both a top-down and bottom-up approach for analyzing documents, suggested a likely inherently complementary nature between them, and mentioned that a combination of the techniques was part of ongoing work. Song et al. (2005) proposed a video annotation algorithm which made use of multiple complementary predictors.

Finally, Tschannen et al. (2011) presented Eve, a development environment for Eiffel[2], which combined proof-based static verification techniques with random testing-based dynamic techniques .

We have adopted a similar strategy and applied it to the domain of generating type error messages. In addition to identifying an effective combination (LT and Helium), an additional contribution of our work is the extraction of general strategy-creation techniques that can be applied to any combination of (type-)error reporting tools.

7. 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.

In future work we plan to investigate the combination of other type debugging tools and the refinement of the evaluation into 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. We thank VL/HCC 2014 and JVLC reviewers for their constructive feedback. This

[2] `www.eiffel.com`

work is supported by the National Science Foundation under the grants CCF-1219165 and IIS-1314384.

Bibliography

Breiman L, Friedman J, Stone CJ, Olshen RA. Classification and regression trees. CRC press, 1984.

Brown PJ. Error messages: The neglected area of the man/machine interface. Commun ACM 1983;26(4):246–9.

Burnett M. Types and type inference in a visual programming language. In: Visual Languages, 1993., Proceedings 1993 IEEE Symposium on. 1993. p. 238–43.

Caliskan-Islam A, Harang R, Liu A, Narayanan A, Voss C, Yamaguchi F, Greenstadt R. De-anonymizing programmers via code stylometry. In: 24th USENIX Security Symposium (USENIX Security 15). 2015. p. 255–70.

Ceccato M, Marin M, Mens K, Moonen L, Tonella P, Tourwé T. Applying and combining three different aspect mining techniques. Software Quality Journal 2006;14(3):209–31. doi:*path*10.1007/s11219-006-9217-3.

Chen S, Erwig M. Better Type-Error Messages Through Lazy Typing. Technical Report; Oregon State University; 2014a. `http://ir.library.oregonstate.edu/xmlui/handle/1957/58138`.

Chen S, Erwig M. Counter-Factual Typing for Debugging Type Errors. In: ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. 2014b. p. 583–94.

Chen S, Erwig M. Guided Type Debugging. In: Int. Symp. on Functional and Logic Programming. LNCS 8475; 2014c. p. 35–51.

Chen S, Erwig M, Smeltzer K. Let's Hear Both Sides: On Combining Type-Error Reporting Tools. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing. 2014a. p. 145–52.

Chen S, Erwig M, Walkingshaw E. Extending Type Inference to Variational Programs. ACM Trans on Programming Languages and Systems 2014b;36(1):1:1–1:54.

Choppella V. Unification Source-Tracking with Application To Diagnosis of Type Inference. Ph.D. thesis; Indiana University; 2002.

Clerici S, Zoltan C, Prestigiacomo G. Graphical and incremental type inference. a graph transformation approach. Higher-Order and Symbolic Computation 2014;:1–34.

Djang RW, Burnett MM, Chen RD. Static type inference for a first-order declarative visual programming language with inheritance. Journal of Visual Languages & Computing 2000;11(2):191 – 235.

Erwig M. Visual Type Inference. Journal of Visual Languages and Computing 2006;17(2):161–86.

Erwig M, Walkingshaw E. The Choice Calculus: A Representation for Software Variation. ACM Trans on Software Engineering and Methodology 2011;21(1):6:1–6:27.

Haack C, Wells JB. Type error slicing in implicitly typed higher-order languages. In: European Symposium on Programming. 2003. p. 284–301.

Hage J. Helium benchmark programs, (2002-2005). Private communication; 2013.

Hage J, van Keeken P. Neon: A library for language usage analysis. In: Software Language Engineering. volume 5452 of *Lecture Notes in Computer Science*; 2009. p. 35–53.

Hartmann B, MacDougall D, Brandt J, Klemmer SR. What would other programmers do: Suggesting solutions to error messages. In: ACM SIGCHI Conf. on Human Factors in Computing Systems. 2010. p. 1019–28.

Heeren BJ. Top Quality Type Error Messages. Ph.D. thesis; Universiteit Utrecht, The Netherlands; 2005.

Johnson GF, Walz JA. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In: ACM Symp. on Principles of Programming Languages. 1986. p. 44–57.

Jung Y, Michaelson G. A visualisation of polymorphic type checking. Journal of Functional Programming 2000;10:57–75.

Kustanto C, Kameyama Y. Improving error messages in type system. Information and Media Technologies 2010;5(4):1241–54.

Lawrence J, Abraham R, Burnett MM, Erwig M. Sharing Reasoning about Faults in Spreadsheets: An Empirical Study. In: IEEE Int. Symp. on Visual Languages and Human-Centric Computing. 2006. p. 35–42.

Lee O, Yi K. Proofs about a folklore let-polymorphic type inference algorithm. ACM Trans on Programming Languages and Systems 1998;20(4):707–23.

Lee O, Yi K. A Generalized Let-Polymorphic Type Inference Algorithm. Technical Report; Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology; 2000.

Lerner B, Flower M, Grossman D, Chambers C. Searching for type-error messages. In: ACM Int. Conf. on Programming Language Design and Implementation. 2007. p. 425–34.

McAdam BJ. Repairing type errors in functional programs. Ph.D. thesis; University of Edinburgh. College of Science and Engineering. School of Informatics.; 2002.

Milner R. A theory of type polymorphism in programming. Journal of Computer and System Sciences 1978;17:348–75.

Muşlu K, Brun Y, Holmes R, Ernst MD, Notkin D. Speculative analysis of integrated development environment recommendations. In: Proceedings of the ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications. 2012. p. 669–82.

Nagy G, Kanai J, Krishnamoorthy M, Thomas M, Viswanathan M. Two complementary techniques for digitized document analysis. In: ACM Conf. on Document Processing Systems. ACM; 2000. p. 169–76.

Narayanan A, Paskov H, Gong NZ, Bethencourt J, Stefanov E, Shin ECR, Song D. On the feasibility of internet-scale author identification. In: Proceedings of the 2012 IEEE Symposium on Security and Privacy. SP '12; 2012. p. 300–14.

Nienaltowski MH, Pedroni M, Meyer B. Compiler error messages: What can help novices? In: ACM SIGCSE Symp. on Computer Science Education. 2008. p. 168–72.

Pelánek R, Rosecký V, Moravec P. Complementarity of error detection techniques. Electronic Notes in Theoretical Computer Science 2008;220(2):51–65.

Schilling T. Constraint-free type error slicing. In: Trends in Functional Programming. Springer; 2012. p. 1–16.

Song Y, Hua XS, Dai LR, Wang M. Semi-automatic video annotation based on active learning with multiple complementary predictors. In: ACM SIGMM Int. Workshop on Multimedia Information Retrieval. ACM; 2005. p. 97–104.

Specht DF. Probabilistic neural networks and the polynomial adaline as complementary techniques for classification. IEEE Trans on Neural Networks 1990;1(1):111–21.

Tip F, Dinesh TB. A slicing-based approach for locating type errors. ACM Trans on Software Engineering and Methodology 2001;10(1):5–55.

Tschannen J, Furia C, Nordio M, Meyer B. Usable verification of object-oriented programs by combining static and dynamic techniques. In: Software Engineering and Formal Methods. 2011. p. 382–98.

Wand M. Finding the source of type errors. In: ACM Symp. on Principles of Programming Languages. 1986. p. 38–43.

Wazny JR. Type inference and type error diagnosis for Hindley/Milner with extensions. Ph.D. thesis; The University of Melbourne; 2006.

Yang J. Explaining type errors by finding the source of a type conflict. In: Trends in Functional Programming. Intellect Books; 2000. p. 58–66.

Yang J, Michaelson G, Trinder P, Wells JB. Improved type error reporting. In: Int. Workshop on Implementation of Functional Languages. 2000. p. 71–86.

Zhang D, Myers AC. Toward General Diagnosis of Static Errors. In: ACM Symp. on Principles of Programming Languages. 2014. p. 569–81.