# Searching for Type-Error Messages

Benjamin S. Lerner  Matthew Flower  Dan Grossman  Craig Chambers
University of Washington
{blerner, mflower, djg, chambers}@cs.washington.edu

## Abstract

Advanced type systems often need some form of type inference to reduce the burden of explicit typing, but type inference often leads to poor error messages for ill-typed programs. This work pursues a new approach to constructing compilers and presenting type-error messages in which the type-checker itself does not produce the messages. Instead, it is an oracle for a search procedure that finds similar programs that do type-check. Our two-fold goal is to improve error messages while simplifying compiler construction.

Our primary implementation and evaluation is for Caml, a language with full type inference. We also present a prototype for C++ template functions, where type instantiation is implicit. A key extension is making our approach robust even when the program has multiple independent type errors.

***Categories and Subject Descriptors*** D.2.5 [*Software Engineering*]: Testing and Debugging; D.3.4 [*Programming Languages*]: Compilers

***General Terms*** Languages, Design

***Keywords*** Type-Checking, Type-Inference, Error Messages, Objective Caml, Seminal

## 1. Introduction

Advanced type systems play an increasingly important role in many modern programming languages. They have widely touted benefits for detecting errors, expressing invariants, and enforcing abstractions. They are central to many languages that originated in the research community (such as ML and Haskell), and features such as parametric polymorphism are quickly gaining popularity in more commercially popular languages (such as Java and C#).

Unfortunately, when types get complicated, it can be burdensome to require programmers to write down explicit types (e.g., on variables, function arguments, and type applications), so modern languages typically provide some form of *type inference*. In particular, for the languages considered in this paper, Caml requires almost no type annotations and C++ often allows implicit instantiation of template functions. While inference is amazingly convenient for programs that typecheck, it often leads to inscrutable error messages for ill-typed programs. In particular, the type-checker often reports error locations that are far from the simplest source of the problem, an issue that has been acknowledged for decades [22].

Improving type-error messages would positively affect many programmers and would make type-system advances more widely embraced, but addressing the problem can be a thankless task. Producing good messages during inference is an engineering challenge that can lead to a larger, slower (even for code that type-checks), less maintainable, and potentially buggier compiler. Researchers developing novel uses for types generally focus on programs that type-check, but bad messages for those that do not can hinder adoption of the ideas. Researching the error-message problem itself leads to a difficult evaluation question: if an ill-typed program is "semantically meaningless," how can we evaluate whether one error message is better than another?

We are pursuing a new approach to producing better type-error messages in the face of inference. Our approach requires *no change* to a compiler's existing type-checker. Instead, a *search procedure* looks for a program close to the original one that does type-check. The search procedure has no knowledge of type-system specifics; it simply uses the existing type-checker as an oracle to see if a change type-checks, which in turn guides further search. The messages complement the conventional approach of reporting a type error by reporting changes that lead to type-correct programs. For example, a message might say that `f(x,y)` (in the original code) does not type-check but `f(y,x)` (at the same location) does.

This general approach has several advantages. (1) By not changing the type-checker, it imposes no burden to compiler correctness or compile-time efficiency for well-typed programs. The computational cost of searching should be measured against the speed of the human writing the program. (2) It can free the type-checker implementor from worrying about error messages, which often requires maintaining extra state or traversing the program in a less convenient order. (3) It can produce better error *locations* because it can search for a small change independently of how type inference works. (4) It can produce more concise and useful error *messages* because the compiler architecture makes it easy for the compiler writer to "try a slightly different expression" at a relevant location. In short, we believe searching for error messages is a rare win-win situation where we can simplify compiler construction (by decoupling error-message generation from type-checking) while improving the programmers' experience (by producing better messages).

In a recent workshop paper [16] we described an initial prototype of our approach for Caml, which we called SEMINAL (for Searching for Error Messages IN Advanced Languages). In extending this work to make it practical and evaluate its effectiveness, we have (1) developed an extended search procedure that works even when a function has multiple independent type errors, (2) compared the error-message quality of our system to the very mature Caml type-checker on a large corpus of automatically collected programs, and (3) implemented a second prototype for C++ template-function errors. Our quantitative results for Caml show that we do better than the underlying type-checker and that support for multiple independent errors is important. Our preliminary experience with C++ suggests that the type-checker's messages (specif-

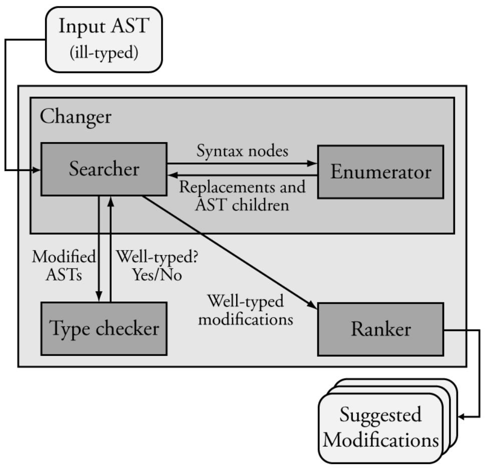


**Figure 1.** The compiler architecture for our approach

ically `gcc`) can be so bad that any complementary technique is useful. Adapting the approach to C++ raises some new technical challenges, but overall the two systems are reassuringly similar.

The rest of this paper is organized as follows. Section 2 presents a complete description of our approach, proceeding in stages to present more sophisticated versions of our search. Section 3 describes our empirical evaluation on Caml files we collected automatically from students. Section 4 describes our prototype for C++ templates, emphasizing the differences due to the language and the platform. Section 5 discusses related work. Section 6 concludes.

## 2. Our Approach

This section explains how we produce type-error messages using a search procedure that does not require changing the type-checker. For specificity, we describe our Caml system and use Caml examples, though the approach generalizes.

The algorithm takes as input an untyped abstract-syntax tree that does not type-check. That is, the new compiler code sits between parsing and type-checking and is bypassed entirely for files that type-check. The output is a ranked list of error messages (though we often present only one to the programmer). Figure 1 shows how the main components of our system interact. The *changer* produces modified untyped abstract-syntax trees and uses the *type-checker* to see which ones type-check. These results guide subsequent search. Those that succeed in type-checking are also passed to the *ranker*, which sorts them.

To explain how the changer and ranker work in more detail, we proceed by describing increasingly more sophisticated techniques in our search procedure. Section 2.1 considers only changes that "remove" parts of the program, which suffices to understand our compiler architecture and how our approach can produce better error-message locations. Section 2.2 adds "constructive changes," which lets the system produce error messages that can sometimes identify the particular mistake in the program. Section 2.3 adds changes that identify when the problem is that a well-typed subexpression is used incorrectly. Finally and most importantly, Section 2.4 modifies the search procedure so that it can find a precise type-error location if the input has multiple independent errors.

### 2.1 Top-down Removal

To understand the top-down nature of our search procedure, consider the ill-typed Caml program on the left of Figure 2. The function `map2` correctly creates a list by applying `f` to corresponding elements of the two lists `aList` and `bList`. However, it expects `f` to take two arguments in curried form but the use of `map2` in the binding of `lst` provides a function expecting one pair argument. Hence this expression does not type-check and the type-checker reports an error. However, its decision to report an error for the addition expression `x + y` makes sense only to someone familiar with unification-based type inference and even then it requires nontrivial reasoning and "manual type-checking" to determine the true source of the error. The type-checker's error message is thus (1) nonlocal (the location is not where the error was made), (2) inaccurate (the error has nothing to do with addition), and (3) misleading (no change at that location will make the program type-check).

The searcher begins with the abstract syntax for the entire ill-typed file and descends recursively to find small subexpressions[1] that could be changed in some way to produce a type-correct program. The searcher works by constructing variations of the input program and calling the type-checker to see if the changes are successful. The searcher first tests increasingly-long prefixes of the top-level definitions to find the first top-level definition that has a type error. In our example, the searcher finds that while the definition of `map2` type-checks by itself, the definitions of `map2` and `lst` together do not. It does not examine the third top-level binding because the second one already causes a type error. After localizing the type error to the second top-level binding, the searcher proceeds to examine its initializer expression, i.e.,

```
map2 (fun (x, y) -> x + y) [1;2;3] [4;5;6]
```

The searcher "removes" this entire expression, by which we actually mean it *replaces* it with a "wildcard" expression that will always type-check in any context, and discovers that the modified program now type-checks. In presenting error messages, we write `[[...]]` for the wildcard replacement expression. For communicating with the type-checker, we use `raise Foo` (where `Foo` is any exception) as the wildcard, since it is always legal and introduces no constraints on type-checking.[2]

After learning that removing the entire initializer expression eliminates the type error, the searcher attempts to localize the type error by examining each of its subexpressions in turn and recursively testing whether removing that subexpression alone is sufficient to remove the type error. In this example, removing either `map2` or `(fun (x, y) -> x + y)` removes the type error, but removing `[1;2;3]` or `[4;5;6]` does not. Further recursion on the two successful subexpressions does not identify any smaller subexpressions whose removal eliminates the type error, so the searcher is done. In particular, no change to `x + y` (the expression identified by the type-checker) can produce a well-typed program.

The searcher has identified *two* candidate error locations, both of which are less misleading than the one found by Caml's type-checker. Our ranker prefers changes closer to the leaves in the abstract-syntax tree, but in this case the two suggestions are tied by this metric. Therefore, the ranker would present both, favoring the removal of `(fun (x, y) -> x + y)` since there is a heuristic for preferring the expression on the right in a function application. The next section explains that our system actually produces a much more precise message for this example.

Another example where search outperforms the type-checker is `let x = e1 in e2` where `e1` type-checks but does not have the type the programmer intended and `e2` uses `x` many times. The type-checker reports an error at a use of `x` but fixing this error only leads to another error at another use. Assuming `e1` is smaller than `e2`, our algorithm will suggest removing `e1` (or a subexpression therein).

[1] The searcher also tries changing members of other syntax categories (e.g., patterns and bindings), but our explanation focuses on expressions.

[2] In other languages, the lack of an expression with any type could complicate top-down search; see Section 4 for how we deal with C++.

```
(* List.combine : 'a list       -> 'b list -> ('a * 'b) list *)
(* List.map     : ('a -> 'b)    -> 'a list -> 'b list        *)
(* List.filter  : ('a -> bool) -> 'a list -> 'a list         *)

let map2 f aList bList =
    List.map (fun (a, b) -> f a b)
             (List.combine aList bList)

let lst = map2 (fun (x, y) -> x + y) [1;2;3] [4;5;6]

let ans = List.filter (fun x -> x==0) lst
```

Type-checker:

```
The expression x+y has type int
but is here used with type 'a->'b
```

Our approach:

```
Try replacing fun (x, y) -> x + y
with fun x y -> x + y
of type int -> int -> int
within context let lst =
  map2 (fun x y -> x + y)
        [1;2;3] [4;5;6]
```

**Figure 2.** A program where the approach outlined in Sections 2.1 and 2.2 produces a better error message than the underlying type-checker. (The actual type-checker prints line/column information instead of expressions, but that is an orthogonal issue.)

## 2.2 Constructive Changes

For each expression for which its removal succeeds in type-checking, we also try more specific modifications. What modifications we try depends on the kind of expression encountered (such as function application, conditional expression, pattern-match, etc.) but does not depend on any type information. For function definitions that take a single tupled argument (such as `(fun (x, y) -> x + y)`), we try among other things changing them to take curried arguments (e.g., `(fun x y -> x + y)`), which for our example leads to an ideal result. The type reported in our message is simply what the type-checker gave to `(fun x y -> x + y)`; all we do is print it in case it is helpful. In our example, the algorithm also tries several other constructive changes, such as permuting the arguments to `map2`, uncurrying the arguments to `map2`, and adding arguments to `(fun (x, y) -> x + y)`, but these are unsuccessful.

Because our ranker prefers constructive changes to removals, the error message in Figure 2 is ranked highest for the example. Though we have experimented with various principled ranking measures such as tree-edit distance, we have found simple heuristics such as this one suffice.

***Creating Constructive Changes*** The quality of error messages often depends on whether the algorithm attempts a constructive change that concisely summarizes the reason the program does not type-check. Therefore, it is important to have an architecture that makes it easy for the compiler writer to define changes and try them efficiently, and it is important to create an effective methodology for thinking of changes.

In our experience, the best approach when defining constructive changes for a kind of syntax node is to exploit both "straightforward tree manipulations" (e.g., swapping the positions of children) and *ad hoc* knowledge of the language (e.g., that currying versus tupling can lead to errors). Figure 3 describes some of the changes we try, including examples of both sorts. (For additional examples, see our prior work [16].) A key feature of our approach is that trying something unusual is a local decision that does not pollute type-checking (many of the changes would be extremely awkward in the type-checker) and has no more cost beyond additional calls to the type-checker. Special cases are encouraged rather than discouraged. For example, to the type-checker, `:=` is just another function (with infix syntax), but it can be misused in ways worthy of special cases.

***Modular Implementation*** To manage the various changes, we decompose the changer into a *searcher* and an *enumerator*. The role of the enumerator is to take a syntax node and return a "list of things to try" (though we refine this notion below). The enumerator is essentially a giant case expression that matches on the sort of node it is given and produces a list of modifications. It also returns the syntax children for the purpose of recursive descent. The role of the searcher is to manage a worklist of changes to try and where in the overall abstract-syntax tree a change is being attempted (so that it can "plug in the change" and call the type-checker).

By separating out the enumerator, adding a new constructive change typically requires only a few lines of code in the compiler and never requires modifying the core search procedure. One could even imagine an open framework where programmers could add possible changes (especially since it does not threaten compiler correctness), but we leave this to future work.

***More Efficient Search*** Keeping the number of changes tractable is necessary for efficiency. For example, trying all permutations of function arguments is exponential, so we should not try them all indiscriminately. Therefore, the enumerator/searcher interface is not just a large flat list of changes to try. Rather, the enumerator produces a structured collection of changes where some changes are attempted only if other changes succeed or fail. For example, we can try changing `(e1,e2,e3)` to `(raise Foo, raise Foo, raise Foo)`, to test whether any 3-tuple would type-check in this location, and follow up with trying permutations `(e1,e3,e2)`, `(e2,e3,e1)`, etc., only if this succeeds. The follow-up changes are also computed lazily so that we reduce the amount of syntax created as well as reducing calls to the type-checker. For more details and more sophisticated examples, see our prior work [16].

## 2.3 Adaptation to Context

Consider the code `if e1 e2 then e3 else e4`, where `e1` has type `string->string` and `e2` has type `string`. The algorithm as presented thus far will give the sort of concise and local message that compiler writers do not mind and programmers do:

```
Try replacing e1 with [[...]]
of type string->bool
within context if e1 e2 then e3 else e4
```

This suggestion belies the fact that the function call `e1 e2` *does* type-check—it just does not type-check within its parent, which expects a `bool` at that location. It focuses on `e1` rather than `e1 e2`; the ranker preferred it to the larger change of replacing all of `e1 e2`.

Fortunately, improving the algorithm amounts to adding just one more attempted change that is as general as removal but is ranked more highly: see if the expression would type-check if its result type were not constrained by its parent in the syntax tree. We call this change *adaptation* (to context). In Caml, the simplest way to test this is to replace `e` with `adapt e` where we define:[3]

```
let adapt x = raise Foo
```

[3] `(e; raise Foo)` works just as well.

| Example syntax node | Example change | Description |
|---|---|---|
| `f a1 a2 a3` | `f a1 a3` | Remove an argument from a function call |
| `f a1 a2 a3` | `f a1 [[...]] a2 a3` | Add an argument to a function call |
| `f a1 a2 a3` | `f a3 a2 a1` | Reorder arguments in a function call |
| `f a1 a2 a3` | `f (a1 a2 a3)` | Reassociate to make a nested call |
| `f a1 a2 a3` | `f (a1,a2,a3)` | Put call-arguments in a tuple |
| `f (a1, a2, a3)` | `f a1 a2 a3` | Curry arguments instead of tupling |
| `e1.fld := e2` | `e1.fld <- e2` | Replace reference-update with field-update |
| `[e1, e2, e3]` | `[e1; e2; e3]` | Make an $n$-element list, not a 1-element list of an $n$-tuple |
| `let f x = e1 in e2` | `let rec f x = e1 in e2` | Make a function recursive |

**Figure 3.** A small sample of constructive changes: some are systematic, such as rearranging a function call's arguments; others are specific to idiosyncrasies of Caml's concrete syntax.

The function adapt has type $\alpha \to \beta$. Other languages may not have functions of such a general type, which could require a more complicated syntax-transformation to achieve the same effect.

The one place where adaptation is not just another constructive change is the ranker. We prefer adapting *larger* expressions, whereas with other changes we prefer modifying *smaller* expressions. This preference is intuitive since it finds a place high in the syntax tree where a type constraint was unsolvable. It is also necessary for our example. After all, adapting e1 also succeeds, which is only a bit more useful than the message without adaptation.

Overall, the ranker prefers adaptation to removal, but prefers constructive changes to adaptation.

## 2.4 Triage for Multiple Errors

The algorithm as presented thus far makes a fundamental assumption that the program has one type error. If there is more than one, the only changes likely to succeed involve removing an ancestor in the syntax-tree that includes all the errors. Even within one top-level function, this result can be terrible. Consider:

```
let x = 3 + true in
... (* many lines of correct code *) ...
4 + "hi"
... (* many more lines of correct code *) ...
```

Suggesting this entire code fragment be replaced does not help the programmer find the errors. In contrast, the conventional type-checker's messages do not suffer from this problem: the type-checker reports the first error it encounters.

Extending our algorithm to find multiple small changes simultaneously seems daunting especially since the number of errors is unknown. It is also of questionable worth since we expect programmers will often fix one error and recompile. Therefore, we extend our algorithm to recursively search for good error messages in one subtree while simultaneously removing one or more other problematic subtrees. We call this approach *triage* because it ignores some other parts of the program to focus on one problem. In this way, we recover the specificity the type-checker has by reporting one error while still leveraging our algorithm's ability to search for effective locations and constructive changes.

Consider first a simple scenario where the function application e1 e2 e3 e4 does not type-check, some of the four subexpressions are large, removing the entire application succeeds in eliminating the type error, and removing no single one of the four subexpressions succeeds. The changer would deem the suggested change too large and enter *triage mode* to find a more specific error in this portion of the program. As before, the changer recursively searches (i.e., *focuses on*) each subexpression e1, ..., e4 in turn, but in triage mode it does so in a context where some of the other $e_i$ are also removed. By focusing on one subexpression and removing some sibling subexpressions (thereby removing their type constraints), we aim to find better error messages for the focused-on subexpression.

Consider focusing on e1. Which of e2, e3, e4 should also be removed such that some modification of e1 can be type-checked? One approach removes them all, searching within e1 in the context

```
e1 [[...]] [[...]] [[...]]
```

to find an e1' such that e1' [[...]] [[...]] [[...]] type-checks, but removing all $n-1$ other expressions may leave e1 less constrained than necessary. Another approach exhaustively computes minimal subsets of the $n-1$ expressions to remove with which e1 can be fixed, but this is potentially exponential. We currently use an algorithm between these extremes. We cumulatively remove the other $n-1$ expressions one at a time in some order, test each context to see if it permits any fix for e1, and recur with the first one that succeeds. So for this example we would try the following contexts in sequence:[4]

```
1.  [[...]] e2      e3      e4
2.  [[...]] e2      e3      [[...]]
3.  [[...]] e2      [[...]] [[...]]
4.  [[...]] [[...]] [[...]] [[...]]
```

For example, if (1) failed but (2) succeeded, then we would recur on e1 in the context e1 e2 e3 [[...]]. (The first failure implies e4 overconstrains the typechecker; the second success implies some fix exists for e1—at the very least, it can be removed.) We use the same procedure to find contexts to use for focusing on e2, ..., e4. The results of the four recursive searches are passed to the ranker, which in addition to its preference for small changes has a preference for removing fewer of the other $n-1$ expressions.

While Section 3 demonstrates that triage is important, our intuition is that the details of the algorithm for removing other expressions are less important. There are many variations we could try, but our current procedure has proven effective.

***Handling Binding Occurrences*** The example above did not involve an expression with variable bindings, such as a pattern-match expression. For such expressions, we cannot delete the binding occurrences and expect other subexpressions to type-check since they may now refer to unbound variables. We therefore refine the triage search for such expressions to include *phases*.

As an example, we describe how triage works for the pattern match in Figure 4. The first phase focuses on the scrutinee (x,y), type-checking it in a context where the patterns and corresponding expressions have been removed. If this does not type-check, the changer would just recur on it in the reduced context and not proceed to subsequent phases, skipping searching the patterns and expressions. In this example, the scrutinee does type-check, so we proceed. In the second phase, we add the patterns, type-checking the scrutinee and the patterns in a context where the corresponding

[4] We need not actually try the first one since we already know it failed nor the last one since it must always succeed.

Original expression:

```
(* val x : int      *)
(* val y : 'a list *)
match (x, y) with
  0, [] -> []
| n, [] -> n
| _, 5  -> 5 + "hi"
```

First phase: scrutinee only

```
match (x, y) with
  _ -> [[...]]
```

Second phase: patterns

```
match (x, y) with
  0, [] -> [[...]]
| n, [] -> [[...]]
| _, 5  -> [[...]]
```

Third phase: entire expression

```
match (x, y) with
  0, [] -> []
| n, [] -> n
| _, 5  -> 5 + "hi"
```

**Figure 4.** A pattern match with several type errors, and the three phases that triage attempts to isolate them.

expressions have been removed. If the combination of the scrutinee and the patterns type-checks, then we would proceed to the third phase, where we would perform triage on the corresponding expressions. In this example, the patterns do *not* type-check. With a triage procedure similar to that for subexpressions (trying each pattern with a greedy subset of others that work), we can report:

```
Your code has several type errors. If you ignore
the surrounding code, try replacing 5 with _ in
  match (x, y) with
    0, [] -> [[...]]
  | n, _  -> [[...]]
  | _, _  -> [[...]]
```

Triage is crucial for reaching this result because two of the triaged expressions fail to type-check (the first two branches have incompatible types and `5+"hi"` is always ill-typed). In this example, having an algorithm that is robust to multiple errors avoids telling the programmer to remove an entire match expression, which is almost never useful.

***Integrating Triage and Search*** Extending the architecture in Figure 1 to support triage is nontrivial because, unlike adding new constructive changes or adaptation, it affects the core search procedure. As described above, the searcher maintains a mode for each syntax node of interest and switches from *regular* to *triage* mode when the only regular suggestion for a node with a nontrivial number of descendents is removing it. We have a second enumerator for triage mode since the work for ordering the changes into phases and the greedy algorithm for deciding what branches to prune are quite different from the regular changes. When the searcher recurs for each focused subexpression in triage it again uses regular mode, which can in turn invoke additional triage on its subtrees.

The ranker prefers triaged solutions least of all. When comparing different triaged solutions, it prefers small changes. The details of the ranking are less important than the nontrivial question of how to present results using triage to the programmer. We first print only the small change found as a result of triage and a message indicating triage occurred (i.e., that other type errors remain, meaning the change will *not* make the program type-check by itself). We also print a representation of which parts of the program were triaged, but our experience is this information is only occasionally useful.

## 3. Empirical Evaluation for Caml

To evaluate our approach's effectiveness, we collected Caml programs from students and manually inspected the error messages from (1) the underlying type-checker, (2) our approach without triage, and (3) our approach with triage. The results show that:

- For most programs, our approach and the type-checker produce messages of comparable quality (suggesting it is reasonable to pursue our approach in lieu of type-checker messages as a way to simplify compiler construction).
- For a significant number of programs, our approach produces a message that is at a better location or significantly more accurate in identifying the problem (suggesting that our approach adds value even for compilers that already have good type-checker messages).
- Triage provides a significant improvement over the system with triage disabled.

Section 3.1 discusses our methodology, which tries to minimize the inevitable subjectivity of the evaluation. Section 3.2 presents overall statistics. Section 3.3 presents some real examples where our approach works well.

### 3.1 Methodology

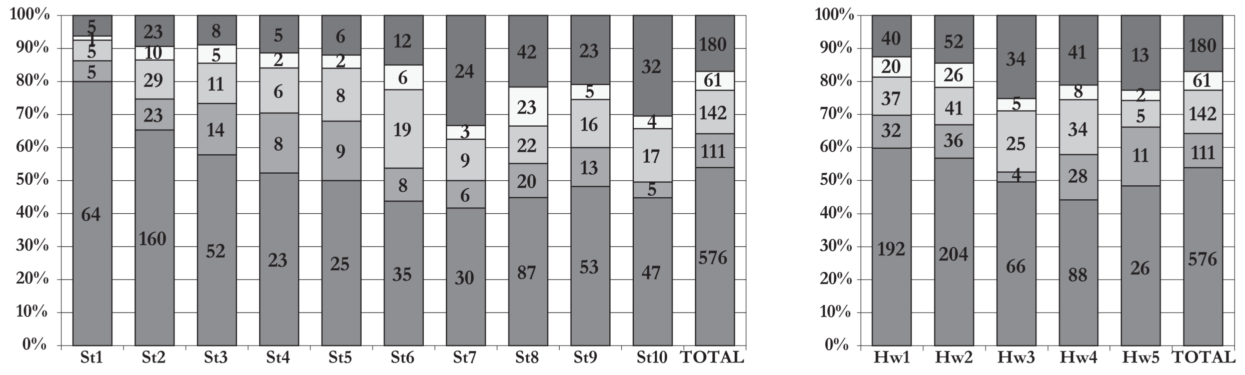
***Data Collection:*** Students in one of the author's courses were asked to volunteer to use a modified version of the Caml compiler that stored to the local disk (1) any file passed to it that did not type-check and (2) a timestamp for the file. This compiler did *not* have our approach implemented; it was just a data collector. Students then emailed the collected files to us. So volunteers opted in twice: when they installed the modified compiler and when they sent us results. The saved files had comments obfuscated (e.g., `(* hi mom *)` replaced with `(* XX XXX *)`) since we thought this might encourage participation.

The course was a graduate-level programming-languages course for part-time students with at least two years professional software-development experience. We believe this population is particularly interesting: they were not beginner programmers but they were new to Caml. We analyzed 5 homework assignments, each requiring 100–200 lines of code. We plan to make the assignments and data available. 10 of 44 class members participated.

***Analysis:*** When manually analyzing a file and the quality of the three error messages, it is necessary to identify the actual problems. Doing so is inherently subjective, so we simply removed any files for which it was unclear. Two factors make this process less subjective. First, using the time-sorted files we can see what the programmers actually changed in the near-future. Second, the code is for a specific programming task; we are not trying to determine the purpose of arbitrary code.

When a collection of files in time sequence all have the same problems, we count them only once (choosing a representative file from the equivalence class). This quotienting of results is important for two reasons. First, it normalizes for different programmers' behavior since some programmers tend to try recompiling much more often than others. Second, the programmers were getting only the type-checker's messages. So in cases where our approach is superior one would expect a misleading type-checking message to lead to several files with the same problem while the programmer discovers the misdiagnosis. Counting each of these files separately would make our approach look better.

A second potential source of subjectivity is determining whether an error message accurately describes one of the real type-checking problems. However, the program *location* identified by a message is not subjective, so we separately counted whether a message identified a good location and whether a message described the problem at that location correctly. Considering only location strictly increases the number of good results for each of the three error messages, but only slightly, so due to space constraints we have chosen only to report the theoretically more subjective measurements.



(a) Results separated by programmer

(b) Results separated by assignment

**Figure 5.** Results separated by programmer and by homework assignment; programmer experience increases for higher-numbered assignments. Moving from bottom to top, the stacked bars represent files where (1) both approaches (type-checker and our approach) produce an equally good message even without triage, (2) both approaches produce an equally good message but triage is necessary, (3) our approach is better even without triage, (4) our approach is better but triage is necessary, and (5) our approach is inferior to the type-checker.

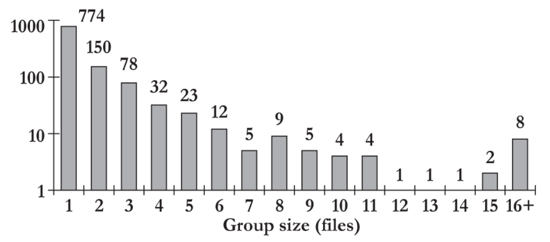


**Figure 6.** Size of groups of files that are the same problem. In results, only one file from each group is counted. Note log-scale.

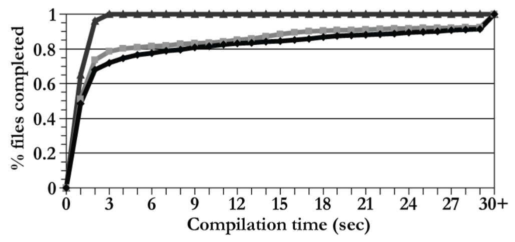


**Figure 7.** Cumulative distribution of time to run our prototype on the analyzed files. The bottom curve is for our full tool. The middle curve disables one constructive change with a performance bug. The top curve disables triage.

***Validity:*** In addition to the subjective evaluation described above, we can identify some other threats to validity. Because we wrote the homework assignments and taught Caml programming, we could have implicitly encouraged a programming style or designed homeworks for which one error-message style is better. There could also have been bias in the self-selection of participants.

On the other hand, we should not discount our overall conclusion that our approach is usually comparable to the type-checker's messages and is often better. Our prototype is still preliminary compared to the mature type-checker which has been used and refined for many years. We also did *not* allow ourselves to modify the implementation after we started analyzing the files. The data suggests a few new constructive changes we can easily add to our tool. This feedback and the ease of exploiting it is a benefit of our approach and would, of course, significantly improve our results.

## 3.2 Aggregated Results

We start by presenting our primary results and then provide some additional data to describe our set of sample files and the efficiency of our prototype. For each file we analyze (recalling that for a group of files in time-sequence with the same error we analyze only one), we consider three messages (from the type-checker, from our approach, and from our approach with triage disabled). We then place the file in one of these five categories:

1. Our approach produces a message comparable in quality to the type-checker, and triage is unnecessary to achieve this "tie."
2. Our approach produces a message comparable in quality to the type-checker, and triage is necessary to achieve this "tie."
3. Our approach produces a message better than the type-checker, and triage is unnecessary to achieve this "win."
4. Our approach produces a message better than the type-checker, and triage is necessary to achieve this "win."
5. The type-checker produces a message better than our approach.

As the TOTAL bars in Figure 5(a) and Figure 5(b) show, our approach is better (the sum of categories 3 and 4) 19% of the time and the underlying type-checker is better 17% of the time. Adding together categories 1–4, our approach is no worse 83% of the time, supporting our claim that one could feasibly not rely on the type-checker's messages at all.

Triage increases by 44% the files for which our approach produces a better result (category 4 divided by category 3) and increases the ties by 19% (category 2 divide by category 1). In all, triage improves our results for 16% of the files (sum of categories 2 and 4). Qualitatively, in the cases where triage helps, it helps a lot—without it, the messages are typically much worse than those from the type-checker.

Not shown in the graphs is a distinction between ties where both approaches produce a good message and ties where both approaches produce a bad message. We deemed 9% of the files "ties where no approach was very helpful," suggesting that type errors could still be improved, though this number is inflated by input programs too nonsensical for a reasonable message.

Figure 5(a) separates the results by programmer to see if personal coding style might affect the results. Figure 5(b) separates the

```
(* List.mem : 'a -> 'a list -> bool *)
let add str lst = if List.mem str lst
                  then lst
                  else str::lst

(* ... in a function later in the file ... *)
        add vList1 s
```

Type-checker:

```
   The expression s has type string
   but is here used with type string list list
```

Our approach:

```
   Try replacing add vList1 s
   with add s vList1
   ...
```

**Figure 8.** Example where our approach finds a better message.

results by homework assignment; programmers are more familiar with Caml on later homeworks. While there is significant variation, sample sizes are moderate and we can draw no firm conclusions.

Figure 6 shows how big the equivalence classes of time-ordered files with the same problem are. While most equivalence classes are very small, we would have been over-counting by having hundreds more data points had we not chosen representatives from each class. In all, we analyzed 1075 files from a total of 2122 collected.

Finally, Figure 7 presents a cumulative distribution function of the time our prototype took to run on the analyzed files. The full approach completed in less than 4 seconds on over 75% of files and less than 30 seconds on over 90% of files.[5] We conclude that the prototype is already usually efficient enough to replace the conventional approach and is almost always a complementary technique worth using while puzzling over the type-checker's message. More importantly, our collected files now give us our first opportunity to debug the performance of our system. (Recall we made no changes to the system after we started the manual analysis.) The additional data in Figure 7 suggest improving performance will be easy. First, about one third of the files that take longer than 4 seconds do so because of a single performance bug in a single constructive change that has to do with reparenthesizing nested match-expressions. Second, without triage—the newest part of our system and therefore the part that we have optimized least—not a single file takes longer than 4 seconds and over 95% take less than 2 seconds.

### 3.3 Examples

In analyzing the files we collected from real programmers, we found many compelling situations where our approach produced better locations, produced more accurate messages, or used triage to avoid a poor message. Here we share a few examples. [6]

Figure 8 shows the definition and subsequent use of a function `add` intended as a simple utility for adding a string to a list. The programmer passed the arguments to `add` in the wrong order, but `add` actually has the polymorphic type `'a -> 'a list -> 'a list`, so the first argument of type `string list` constrains the second to have type `string list list`. Hence the type-checker's error message is in a reasonable location, but only programmers used to the idiosyncrasies of polymorphic types and unification-based type inference find this message intuitive.

[5] Qualitatively, time is not correlated with size of the file; the search quickly descends into a small portion of the file.

[6] As in Figure 2, we modify the type-checker's messages to use expressions rather than line numbers.

```
type move = For of int * (move list) | (* ... *)

let rec loop movelist x y dir acc =
   (* ... some other local bindings ... *)
   match movelist with
     [] -> acc
   | For(moves, lst)::tl ->
     let rec finalLst index searchLst =
       if (index = (moves-1))
       then []
       else (List.nth searchLst)
            ::(finalLst (index+1) searchLst)
     in loop  (finalLst 0 lst) x y dir acc
   | (*... branches for other kinds of moves ...*)
```

Type-checker:

```
   The expression (finalLst 0 lst) has type
   (int -> move) list
   but is here used with type move list
```

Our approach:

```
   Try replacing (List.nth searchLst)
   with (List.nth searchLst [[...]])
   ...
```

**Figure 9.** Example where our approach finds a better location.

Figure 9 shows a larger example where our approach finds a better location. The code, extracted from an attempt to write a small-step interpreter for a simple Logo-like language, defines a recursive local helper function `finalLst` that is supposed to return a `move list`. However, the call to `List.nth` has one too few arguments, which is not in itself an error because this is just a partial application of a curried function. In fact, `finalLst` still type-checks, but as a function that returns a list of functions (type `(int -> move) list`). Hence the type-checker reports no error until the result of a call to `finalLst` is passed to the outer function `loop`, which due to its pattern-match expression needs a `move list`. Our top-down search finds small suggestions both in the body of `finalLst` and its use, but only in the former is there a constructive change (adding an argument to the call to `List.nth`), which becomes the top-ranked suggestion.

Most examples with triage are too large to be shown in their entirety, but we can describe a scenario where it is necessary for doing as well as the type-checker. Suppose the programmer calls `print` in each branch of a match-expression where the actual standard-library function is called `print_string`. The underlying type-checker works well, finding one use of `print` and reporting it as an unbound variable. Our approach without triage is terrible, suggesting removal of the entire match expression. With triage, we determine that the match-expression type-checks except for the expressions in the branches and that one of the branches can be fixed by removing a call to `print`. In fact, because removing `print` works but replacing it with `adapt print` does not, we can conclude that `print` is an unbound variable. Improving our tool's presentation to report "unbound variable" when removing a variable works but adapting it does not would be straightforward and is independent of the search algorithm.

There are coding practices for which our approach is ill-suited. As an extreme example, consider again `let x = e1 in e2` where `x` is used many times in `e2`. Savvy ML programmers who change `e1` to have a different type may rely on the type-checker to find all the uses of `x`. Our approach will instead suggest changing `e1` (perhaps back to what it was), which is clearly not desired.

## 4. Prototype for C++ Template Functions

While our approach is ideal for an ML-like language (type inference creates a need, and polymorphic types for expressions like `raise Foo` aid the implementation), the benefits extend beyond strongly typed functional languages. As a proof of concept, we have developed a prototype for producing error messages for C++ programs that use template functions, which are common because template functions pervade the Standard Template Library (STL). This section explains our focus on template functions (Section 4.1), describes how we modify our approach for the C++ language (Section 4.2), and sketches our prototype (Section 4.3).

### 4.1 The Problem with Templates

C++ does not generally suffer from nonlocal type-error messages because all variables and functions have explicit, monomorphic types, so we see insufficient need to change how most type-error messages are produced. Template functions, however, are fertile ground: they take type parameters the caller need not specify, and some type-checking of template bodies is delayed until each use.

Misuse by a client often manifests itself as a type error several layers deep in template calls. As a simple example, consider this template function in the commonly used `gcc` extension to the STL:

```
template <class _Operation1, class _Operation2>
compose1(const _Operation1& __fn1,
         const _Operation2& __fn2) {
  return unary_compose<_Operation1,_Operation2>
         (__fn1, __fn2);
}
```

The function is a polymorphic wrapper around a template class (`unary_compose`) for representing the composition of two "functors."[7] Not seen here is that arguments to `compose1` must have particular STL types or an error arises in the body of `unary_compose`. Compilers like `gcc` do report the location of the outermost template call, but the programmer must still manually dissect the error and call-chain to determine how to change the client code.

Continuing our example, consider the C++ code in Figure 10. The intended effect is to initialize the elements of `outv` with a function of the `inv` values. Using the higher-order features of the STL keeps `myFun` concise and at a high level of abstraction; we generally encourage code in this style. It is also *almost* type-correct. The problem is `compose1` expects to receive two functors, and `labs` is merely a function pointer. The STL provides a function `ptr_fun` to convert a function pointer to a functor; the fix is to "adapt" `labs` by writing `ptr_fun(labs)`. The use of functors is not universal, however; other places in template libraries require function pointers. In short, this sort of error is simple, easy to make, and (with the right diagnosis) easy to fix.

The error messages reported by `gcc` for this error are shown in Figure 11. The first eight lines do describe the problem, but in terms of a bad template call several layers deep in the `functional` library. The remaining lines describe a "no template found" error that is a cascading error resulting from the first problem. As inscrutable as these messages may seem, we have chosen an example where they are tractably small. If we had made the same mistake for an operation over `vector<vector<long> >` instead of `vector<long>` (and vectors of vectors are certainly a realistic use of the STL), the messages would have been over twice as long.

Our approach, of course, produces an error message indicating that changing `labs` to `ptr_fun(labs)` eliminates the typing errors in `myFun`. Because `ptr_fun` is just a library function, it is an *ad hoc* constructive change to try wrapping function pointers with it. Trying this change is justified by its usefulness.

[7] This STL term for classes representing function closures should not be confused with the functors of ML or category theory.

```
#include <algorithm> // for transform
#include <vector> // for vector
#include <functional> // for multiplies, bind1st, ptr_fun
#include <ext/functional> // for compose1
#include <cmath> // for labs
using namespace std;
using namespace __gnu_cxx;

// compute outv[i] = labs(5 * inv[i])
void myFun(vector<long>& inv, vector<long>& outv){
 transform(inv.begin(), inv.end(), outv.begin(),
           compose1(bind1st(multiplies<long>(),5),
                    (labs))); // type error
}
```

**Figure 10.** An STL client with a type error

### 4.2 Our Approach in C++

The algorithm we use for Caml largely works for C++, but there are interesting differences in how we perform top-down search, what constructive changes we try, how we do adaptation, and how we call the type-checker. We highlight the most important differences.

***Top-Down Search and Adaptation*** Because C++ is explicitly typed, we do not find it necessary to perform search over the entire program. Rather, we consider only a function containing a template-call error[8] and any template functions it may call (transitively). Within a function body, removing statements and initializer expressions is trivial because we can just delete them.

However, removing or adapting an expression proves more difficult because we do not have the convenience of expressions like `raise Foo` (which can have any type) or `adapt` (which can take any expression and have any return type). Ironically, we can (ab)use a template function to create expressions of the right type:

```
template <class A, class B>
B magicFun(A x) { for ( ; ; ) ; } ;
```

Now, "removing an expression `e`" can mean replacing it with `magicFun(0)` and adapting it can mean replacing it with `magicFun(e)`. Unfortunately, using `magicFun` is imperfect because C++, for deep reasons involving ambiguity and overloading, does not have full inference. So in many contexts, `magicFun(0)` or `magicFun(e)` will not type-check because an appropriate return type cannot be resolved.

As a result, our changer often resorts to alternate techniques to determine which subexpressions contain typing errors. Most simply, we can hoist expressions (e.g., replace `e0(e1,e2);` with `e0; e1; e2;`), taking some care to circumvent odd C++ restrictions (e.g., function names cannot appear as top-level expressions).[9] Full hoisting is sometimes avoidable with additional C++ chicanery. For example, given `x = e`, we can retain the constraint that `e` has a type compatible with `x` by replacing the assignment with `magicFunAssign(x,e)` where we define:

```
template <class A, class B>
A magicFunAssign(A& x, B e) {
  x = adapt<A>(e); return x;
}
```

[8] Simple processing of the error message identifies the location, and the abstract-syntax tree indicates that the location contains a template call.

[9] It suffices to wrap the expressions with a `void`-returning variant of `magicFun`.

```
/usr/lib/gcc/i686-pc-cygwin/3.4.4/include/c++/ext/functional: In instantiation of
    ‘__gnu_cxx::unary_compose<std::binder1st<std::multiplies<long int> >, long int ()(long int)>’:
../tester2.cpp:9:   instantiated from here
/usr/lib/gcc/i686-pc-cygwin/3.4.4/include/c++/ext/functional:128: error: ‘long int ()(long int)’ is not a class, struct, or union type
/usr/lib/gcc/i686-pc-cygwin/3.4.4/include/c++/ext/functional:136: error: ‘long int ()(long int)’ is not a class, struct, or union type
/usr/lib/gcc/i686-pc-cygwin/3.4.4/include/c++/ext/functional:131: error: field
    ‘__gnu_cxx::unary_compose<std::binder1st<std::multiplies<long int> >, long int ()(long int)>::_M_fn2’
    invalidly declared function type
/usr/lib/gcc/i686-pc-cygwin/3.4.4/include/c++/bits/stl_algo.h: In function
    ‘_OutputIterator std::transform(_InputIterator, _InputIterator, _OutputIterator, _UnaryOperation)
    [with _InputIterator = __gnu_cxx::__normal_iterator<long int*, std::vector<long int, std::allocator<long int> > >,
     _OutputIterator = __gnu_cxx::__normal_iterator<long int*, std::vector<long int, std::allocator<long int> > >,
     _UnaryOperation = __gnu_cxx::unary_compose<std::binder1st<std::multiplies<long int> >, long int ()(long int)>]’:
../tester2.cpp:9:   instantiated from here
/usr/lib/gcc/i686-pc-cygwin/3.4.4/include/c++/bits/stl_algo.h:789: error: no match for call to
    ‘(__gnu_cxx::unary_compose<std::binder1st<std::multiplies<long int> >, long int ()(long int)>) (long int&)’
```

**Figure 11.** The error message from gcc for Figure 10, with additional linebreaks to improve readability.

***Constructive Changes*** As expected, each language has different common errors worth detecting. For C++, we try several language-level changes (e.g., switching `e.f` to `e->f` and vice-versa), as well as changes specific to the STL (such as adding or removing `ptr_fun`). Other changes are just like in Caml, such as rearranging arguments at a call-site.

***Calling the Type-Checker*** As explained below, instead of modifying gcc, we print each change in concrete syntax and call the type-checker. Because C++ is prone to cascading errors during compilation, we focus solely on the first error and specifically on the line it cites as "instantiated from here." However, we maintain all the reported error lines, so that we can properly evaluate whether a modification succeeded or not: if it eliminates some errors while introducing no new ones, it is a success. Partially implicit in this definition is a notion of triage: we ignore any later erroneous code, so that we can focus on suggestions for one statement.

### 4.3 Implementation

Our prototype is implemented as a plugin for the Eclipse 3.2 IDE, using the CDT 3.1 plugin (which provides an abstract syntax tree for C++) and gcc (since we assume the type-checker's error messages are in a particular format). We take an abstract syntax tree, change it, and use the CDT to print it to a file, which we then pass to gcc. While this architecture makes for an expedient prototype, the process and I/O overhead of printing and parsing the program for each change is too high for interactive use. On the other hand, by using an IDE we have the ability to perform our algorithm in the background. Moreover, our presentation of error messages uses Eclipse's support for "problem markers" and "quick fixes." That is, we provide a marker in the user interface that brings up a menu item, such as, "replace this expression by wrapping it in `ptr_fun`."

## 5. Related Work

This work extends and evaluates the approach advocated in our recent workshop publication [16]. We discuss our relation to that work before discussing other approaches to improving type-checker error messages and other related techniques.

### 5.1 SEMINAL

We have extended our prior work in the following important ways:

- Our prior work had no way to give good error messages when the same function had multiple independent type errors. Triage (Section 2.4) overcomes this fundamental problem, and our results demonstrate that triage is significant in practice.
- Our approach of adapting expressions to their context (Section 2.3) is new, can sometimes lead to much more accurate messages, and requires a different approach to ranking since a large expression may need only a small adaptation.
- Our prior work had no evaluation of the system's effectiveness (only that it was reasonably efficient). Our analysis on actual files (Section 3) is, to our knowledge, the only evaluation of its kind ever completed and encompasses a larger sample set than anything considered by another system.
- Our prototype for C++ template-function instantiation errors is entirely new and requires different techniques than in Caml.

Our prior publication includes more details on the implementation of the SEMINAL system for Caml, particularly the interaction between the searcher and the enumerator.

### 5.2 Other type-checker approaches

While SEMINAL was hardly the first project to attempt to address the nonlocal and unintuitive nature of ML-style type-error messages, all prior approaches of which we are aware made pervasive changes to the type-checker. Doing so makes compiler construction and the efficient compilation of well-typed programs more difficult. Such changes take many forms, including trying different program-traversal orders (thereby changing the order of the underlying unification problem) [15, 17], maintaining slice information to record what expressions affect what parts of the unification problem [5, 10, 21], and providing an interactive environment revealing the internals of the type-inference algorithm [4, 20]. Heeren's recent summary [11] of these systems and others [1, 2, 7, 8, 9, 13, 22] provides more details.

The approach most similar to ours is from McAdam [18]. He enriches the type-checker with type-level "linear morphisms" that can be used to make a program type-check that otherwise would not. For example, a morphism from $\tau_1 \to \tau_2 \to \tau_3$ to $\tau_2 \to \tau_1 \to \tau_3$ would allow `f x y` to type-check even if it should be `f y x`. The resulting error message can then show what morphisms were used. While this clearly has commonalities with our constructive changes, it has the same nonlocal problems as the conventional approach (it never searches for other locations) and presents messages in terms of types that the programmer never wrote.

For C++ template errors, there are *ad hoc* tools available that post-process error messages to make them more readable [24]. The process largely involves regular-expression matching to trim out namespaces and other irrelevant features of error messages. We view the existence of such tools as evidence of a problem rather than as a complete solution. Unlike our tool, the result is still in terms of types and does not analyze the source program.

### 5.3 Other work

Parsing errors can also be nonlocal and unintuitive. Structurally, the problems are different because parsing takes a token stream instead of an abstract syntax tree. Nonetheless, the approach by Burke and Fisher [3] is similar in that an error-finder separate from the parser tries single-token changes (by directly changing the parse-stack) and seeing which ones lead to more progress in parsing.

Also note that a compiler may often treat as a type error what the programmer views as a parsing error. For example, in Caml `[1,2,3]` is a list holding one triple whereas `[1;2;3]` is a list of three integers. Our tool presents a good message in this case because it always tries replacing a list with one tuple with a list holding the tuple's elements. In the tool this is a syntax-tree transformation, but the actual error message is in concrete syntax, giving the illusion of suggesting a parsing change.

The dynamic testing technique of delta-debugging is another example of automatic search to produce a concise error. Recent work has focused on changing dynamic program state [6]. Older work on changing the program code worked by applying a subset of changes between two versions [23] under the assumption that the new version had a bug the old version did not. In our setting, it is much less clear this assumption is valid or that consulting a previous version of the code would be fruitful.

Ongoing work on adding "concepts" to C++ [12, 19] aims to make type-checking for templates more modular. This work acknowledges that nonlocal errors and long messages (hundreds or thousands of lines) are real problems with templates. It is mentioned that preliminary experience with concepts shows they can improve the situation. In terms of the example in Figure 10, using concepts in the STL would probably lead to the correct error location (in the client), but not make the fix any more intuitive. Concepts add additional static checking via more-explicit type information (template writers indicating requirements about template parameters), so less-annotated code would remain problematic.

## 6. Conclusions

By decoupling type-checking from the generation of type-error messages, we have pursued a novel approach to circumventing the tension among type inference, efficient type-checking, and quality error messages. We have developed proofs-of-concept in two quite different settings and carefully evaluated the effectiveness of one of the systems. Our approach often finds good messages that describe different program locations and are more concise than the conventional approach of producing error messages in the type-checker. Being robust when programs have an unknown number of independent type errors proved to be an important advance.

There are several directions for future work beyond the obvious steps of attacking new languages with inference and improving the systems we have. First, we would like to integrate our systems into IDEs and focus on the user-interface issues that are clearly important for error-message quality. Second, a controlled user study could bolster our conclusions. Third, an open system where programmers could describe new search strategies or constructive changes to try would increase usefulness. In particular, when embedding a domain-specific language in a general-purpose language, the domain-specific language implementor would like to define error messages in higher-level terms [14] and our approach should make this easier than it has been.

## References

[1] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4):17–30, 1993.

[2] K. Bernstein and E. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, 1995.

[3] M. G. Burke and G. A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9(2):164–197, 1987.

[4] O. Chitil, F. Huch, and A. Simon. Typeview: a tool for understanding type errors. In M. Mohnen and P. Koopman, editors, *12th International Workshop on Implementation of Functional Languages*, Aachner Informatik-Berichte, 2000.

[5] V. Choppella and C. T. Haynes. Diagnosis of ill-typed programs. Technical Report 426, Indiana University, December 1994.

[6] H. Cleve and A. Zeller. Locating causes of program failures. In *27th International Conference on Software Engineering*, 2005.

[7] D. Duggan. Correct type explanation. In *ACM SIGPLAN Workshop on ML*, 1998.

[8] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, July 1996.

[9] M. Gandhe, G. Venkatesh, and A. Sanyal. Correcting errors in the Curry system. In *16th Conference on Foundations of Software Technology and Theoretical Computer Science*, 1996.

[10] C. Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. *Science of Computer Programming*, 50(1-3):189–224, 2004.

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

[12] J. Järvi, D. Gregor, J. Willcock, A. Lumsdaine, and J. Siek. Algorithm specialization in generic programming: challenges of constrained generics in C++. In *ACM Conference on Programming Language Design and Implementation*, 2006.

[13] Y. Jun, G. Michaelson, and P. Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.

[14] S. Krishnamurthi, Y.-D. Erlich, and M. Felleisen. Expressing structural properties as language constructs. In *European Symposium on Programming*, 1999.

[15] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, 1998.

[16] B. Lerner, D. Grossman, and C. Chambers. Seminal: Searching for ML type-error messages. In *ACM SIGPLAN Workshop on ML*, 2006.

[17] B. J. McAdam. On the unification of substitutions in type inference. In K. Hammond, A. J. Davie, and C. Clack, editors, *Implementation of Functional Languages*, volume 1595 of *LNCS*, 1998.

[18] B. J. McAdam. *Repairing Type Errors in Functional Programs*. PhD thesis, Laboratory for Foundatations of Computer Science, The University of Edinburgh, 2001.

[19] G. D. Reis and B. Stroustrup. Specifying C++ concepts. In *33rd ACM Symposium on Principles of Programming Languages*, 2006.

[20] P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *ACM SIGPLAN Workshop on Haskell*, 2003.

[21] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology*, 10(1):5–55, 2001.

[22] M. Wand. Finding the source of type errors. In *13th ACM Symposium on Principles of Programming Languages*, 1986.

[23] A. Zeller. Yesterday, my program worked. today, it does not. Why? In *7th European Software Engineering Conference and 7th ACM Symposium on the Foundations of Software Engineering*, 1999.

[24] L. Zolman. STLFilt: An STL error message decryptor for C++. http://www.bdsoft.com/tools/stlfilt.html, 2005.