A Type-Directed Approach to Program Repair

Alex Reinking and Ruzica Piskac

Yale University

Abstract. Developing enterprise software often requires composing several libraries together with a large body of in-house code. Large APIs introduce a steep learning curve for new developers as a result of their complex object-oriented underpinnings. While the written code in general reflects a programmer's intent, due to evolutions in an API, code can often become ill-typed, yet still syntactically-correct. Such code fragments will no longer compile, and will need to be updated. We describe an algorithm that automatically repairs such errors, and discuss its application to common problems in software engineering.

1 Introduction

While coding, a developer often knows the approximate structure of the expression she is working on, but may yet write code that does not compile because some fragments are not well-typed. Such mistakes occur mainly because modern libraries often evolve into complex application programming interfaces (APIs) that provide a large number of declarations. It is difficult, if not impossible, to learn the specifics of every declaration and its utilization.

In this paper we propose an approach that takes ill-typed expressions and automatically suggests several well-typed corrections. The suggested code snippets follow the structure outlined in the original expression as closely as possible, and are ranked based on their similarity to the original code. This approach can also be seen as code synthesis. In fact, our proposed method extends the synthesis functionality described in [3, 6, 10]. In light of program repair, plain expression synthesis can be seen as a repair of the empty expression.

We have implemented an early prototype of our algorithm, and empirically tested it on synthesis and repair benchmarks. The initial evaluation strongly supports the idea of a graph-based type-directed approach to code repair and snippet synthesis. Compared to the results reported in [3], our approach outperforms on similar benchmarks, sometimes by several orders of magnitude, while still producing high-quality results.

2 Related Work

Our work is largely inspired by two synthesis tools: Prospector [6] and InSynth [3, 4]. Prospector is a tool for synthesizing code snippets containing only unary API methods. The basic synthesis algorithm used in [6] encodes method signatures using a graph. Although we also encode function information in a graph structure, our synthesis graph is more general. As explained in Sec. 4.1, we distinguish nodes into types and functions,

as opposed to just types. In a way, the connections to each function node models its succinct type as described in [3]. While our approach acts as a generalization of both these tools, we significantly extend their capability. Our algorithm can repair ill-typed expressions, as well.

Debugging and locating errors in code [1, 8] play an important role in the process of increasing software reliability. While our approach suggests repairs based only on a given ill-typed expression and its environment, other tools that tackle this problem [2, 5, 7, 9] additionally require test cases, code contracts and/or symbolic execution.

3 Motivating Example: Correcting Multiple Errors

In this section, we show how our algorithm efficiently repairs ill-typed expressions. Sometimes, such expressions might poorly reflect the structure of the desired expression, while still retaining other useful information. This is the case when the correct structure is obscured by passing too many or too few arguments to a function, or by passing them in the wrong order.

The following code fragment attempts to read a compressed file though a buffered stream while using an extensive number of calls to the standard Java API. The developer attempts to instantiate an `InputStream` object:

```
int buffSize = 1024, compLevel = Deflater.BEST_SPEED;
String fileName = "compressed.txt";

InputStream input =
  new BufferedInputStream(buffSize, new DeflaterInputStream(
    new FileInputStream(), compLevel, true)); // error
```

In this example, the single variable assignment contains three errors. First, the constructor for the `FileInputStream` requires at least one argument, yet has received none; second, the `DeflaterInputStream` constructor has been passed too many arguments; and finally, the `BufferedInputStream` has been passed valid arguments, but in the wrong order.

To repair this expression, our algorithm proceeds from the bottom, viewed as a parse tree, up to the top-level. Thus, it begins by correcting the innermost sub-expression: `new FileInputStream()`. From the entire available code, our repair algorithm returns `new FileInputStream(fileName)` as the closest match. To repair code, we consider the visible user-defined values along with the standard libraries, favoring the values that appear closest to the point in the program where the repair was initiated.

After applying this repair, the repair proceeds to correct the `DeflaterInputStream` call. Since all of its arguments are well-typed, the repair will attempt to re-use them while synthesizing a replacement. After searching through the space of possible repairs, the algorithm finds the following snippet:

```
new DeflaterInputStream(new FileInputStream(fileName), new
    Deflater(compLevel, true))
```

Here, the repair wraps the extra arguments in a call to the `Deflater` constructor from the Java API. Notice that even though `Deflater` was not previously present in the expression, our repair algorithm was able to discover it by examining the valid constructor calls for a `DeflaterInputStream`.

Finally, the algorithm rebuilds the overall expression by interchanging the arguments in the top-level expression to arrive at the final, correct result:

```
new BufferedInputStream(new DeflaterInputStream(
  new FileInputStream(fileName),new Deflater(compLevel, true)),
    buffSize);
```

As we discuss in Sec. 5, the whole search and repair takes under a second to complete.

4 The Algorithm

4.1 Synthesis Graph Construction

Our algorithm operates by searching through a data structure we call the *synthesis graph*. Each node of the synthesis graph corresponds to either a value-producing language entity, such as a function, variable, constant, or literal, or to a type in the language. We therefore divide nodes into two sets V_t (type nodes) and V_f (function nodes). Since variables, constants, and literals can be considered functions taking the empty set to their value, they belong to V_f. From every function node, there is an out-edge to the type it produces, and for each distinct type that the function takes as an argument, there is an incoming edge into the function node to the type node. Importantly, this means that a function on three input parameters of the same type will have in-degree exactly one.

In addition, we assign to every edge a cost, which is a subjective measure that guides the search towards desirable traits. Such traits could include smaller expressions or lower memory usage, similar to [4]. The cost of an expression is defined to be an accumulation of the costs of the edges it includes.

4.2 Synthesis Procedure

We now outline the synthesis portion of our algorithm, Algorithm 1. The algorithm takes as input the synthesis graph $G = (V_t \cup V_f, E)$, the type of expression to synthesize τ, and two numbers $C_{\max}$ and N. N is the number of expressions to synthesize, and $C_{\max}$ is an upper bound on the cost of an expression. The synthesis algorithm returns a list of expressions of type τ. The first two steps can be done using Dijkstra's algorithm. The types in V_t' are explored in reverse order to avoid performing expensive recomputations. The loop finds N expressions of type σ with the shortest cost-distance to τ in G' and stores them in snips. This way, GetExpressions is able to reuse these computations without reducing the search space.

Next we describe the GetExpressions procedure, whose task is to find the N best snippets of type τ in G' within a prescribed cost bound C_{now}. The procedure operates recursively, and it checks the snips table to see whether it can reuse the existing computations. To compute candidates for $\tau \in V_t$, the procedure looks at its outgoing

Algorithm 1: Synthesis Algorithm

input : $G = (V_t \cup V_f, E)$, $\tau \in V_t$, $C_{\max}$, N
output: exprs, the list of expressions

```
G' = (V_t' ∪ V_f', E') ⟵ subgraph of G reachable within C_max from τ ;
Sort V_t' in descending distance away from τ ;
snips ⟵ Hash table mapping types to snippets ;
foreach σ ∈ V_t' do
    snips [σ] ⟵ GetExpressions(G', snips, σ, C_max − Dist(σ), N) ;
exprs ⟵ snips [τ] ;
```

Procedure GetExpressions($G' = (V_t' \cup V_f', E')$, snips, τ, C_{now}, N)

```
if τ ∈ Keys(snips) then return snips [τ];
results ← ∅ ;
foreach g ∈ V_f' of the form g : (τ_1 × ··· × τ_k) → τ do
    if Cost(g) > C_now then continue;
    For all i, let s_i ← GetExpressions(G', snips, τ_i, C_now − Cost(g), N) ;
    foreach args ∈ s_1 × ··· × s_k do
        if Cost(g(args)) ⩽ C_now then
            Add g(args) to results;
        while | results | > N do
            Remove the most costly entry from results;
return results
```

neighbors, which are all functions whose output is of type τ. For each function that does not immediately break the cost constraint, GetExpressions attempts to synthesize subexpressions for each of its arguments recursively. This only needs to be done once for each type. Then, for every possible set of arguments to the function, it adds the allowable expressions to the results. Furthermore, it pushes out the worst few results if the size of the set would exceed N.

4.3 Repair Algorithm

Finally, we describe the repair algorithm, Algorithm 2. The key step in our approach is biasing the previously-described synthesis procedures towards the correctly-typed subexpressions of the broken expression. The intuition for this is that the search should be directed to favor those components that the programmer intended to use. To do this, we adjust the `Cost` function used by GetExpressions to assign the lowest possible cost to the well-types subexpressions. Informally, we call these zero-cost subexpressions "reinforced". This lowers the weights of results that contain these expressions, thus improving their ranking among the returned results.

This scheme has a few advantages: first, it will very strongly prefer those expressions that occurred as part of the given incorrect expression; second, in cases where more than one of the same type is required, it will favor using multiple, distinct subex-

pressions; and finally, if no expressions are given, then `Cost` actually remains unchanged.

With this modification in place, the repair algorithm proceeds from the bottom up. For each broken sub-expression in the input, we first reinforce each of its well-typed subexpressions and then initiate a synthesis for the desired type of the current subexpression. If any of its children are ill-typed, we recurse and repair them first.

Notice that this means the repaired subexpressions will also be reinforced. This behavior is desirable because it favors reusing the subexpressions generated once the repair synthesizes a higher level. Additionally, the recursion guarantees that reinforcing a subexpression will not interfere with a synthesis that occurs at the same level as that subexpression. Although this algorithm, as described, returns up to N possible repairs,

Algorithm 2: Repair Algorithm

input : $G = (V_t \cup V_f, E)$, the synthesis graph; expr, the broken expression; $C_{\max}$, the maximum allowable cost; N, the number of repairs to synthesize
output: repairs, a list

```
1 if expr is well-typed then return [expr];
2 Write expr as expr (x_1, ..., x_k) where x_i are its subexpressions of type τ_i ;
3 foreach x ∈ {x_1, ..., x_k} do
4     x ⟵ Repair (G, x, C_max, N) ;  // Replace x with a list of either
      itself or its possible corrections
5 foreach subs ∈ x_1 × ... × x_k do
6     Reinforce all expressions in subs;
7     Add all results of Synthesize (G, τ, C_max, N) to repairs;
8     Clear reinforcements ;
9 repairs ⟵ Best N results in repairs
```

in our preliminary implementation, the first returned result was mostly the correct one, so we speculate that setting N really low might be acceptable in a practical setting.

5 Preliminary Evaluation

We empirically evaluated our approach on benchmarks based on those found in [4]. Table 1 shows the summary of the results. The runtimes were measured on a standard university-supplied computer. For each benchmark, the best of 50 consecutive trials was recorded to account for variance in process scheduling, cache behavior, and JVM warmup. It was not uncommon to see four-to-five-fold speed increases between the best and the worst runtimes of the algorithm. This is due to the delay in program optimization afforded by Oracle's JIT compiler.

It is important to note that these numbers represent a worst-case scenario for our algorithm. Since the full set of Java libraries are rarely imported, the algorithm should run even faster in practice as it will have smaller graphs to search. We imported the whole Java standard library which resulted in a graph of 45,557 nodes and 102,377 edges.

Benchmark	Type	Size	Time (ms)	Nodes	Edges	Rank
`SequenceInputStream`	Synthesis	3	< 1	141	149	1
`SequenceInputStream`	Repair	5	4	–	–	1
`BufferedReader`	Synthesis	3	16	3119	4225	2
`BufferedReader`	Repair	3	18	–	–	1
`AudioClip` (applet)	Synthesis	3	27	6808	9291	2
`InputStreamReader`	Synthesis	2	29	7064	9673	1
`FileInputStream`	Synthesis	2	38	7832	10516	1
`Matcher` (regex)	Synthesis	4	93	14505	24740	1
`InputStream` (from byte array)	Synthesis	2	116	13163	20581	2
`DeflaterInputStream`	Repair	8	380	–	–	1

Table 1. Typical-use runtimes in various benchmarks. "Nodes" and "Edges" refer to the size of the searched subgraph, and "Rank" indicates the correct expression's position among the results. The "size" refers to the number of subexpressions in the output expression. Each test case was initialized with a small environment consisting of five variables, and produced ten results.

These benchmarks show that repair is fast and accurate even in the face of multiple, difficult errors. The compressed stream example in Sec. 3 had several distinct errors: a missing parameter, two parameters transposed, and additional parameters passed to a function that did not accept them. Still, in three calls to the synthesis routine, our algorithm automatically corrected *all three* errors in around a third of a second.

Although it is impossible to test the full range of possible type errors everywhere they might appear in the Java standard library, if these speeds are indeed representative of the whole space of possible errors, then our repair algorithm is sufficiently fast to operate in an interactive setting.

6 Conclusions and Future Directions

We have seen that our algorithm efficiently subsumes the work done in [3,6,10] and extends it to the problem of program repair. Using our novel graph-theoretic approach, we efficiently solve instances of this problem to synthesize a correct expression from the salvageable parts of a broken one. We believe that the algorithm in its current state has two compelling uses. First, it can assist programmers in writing complex expressions. Second, it could be integrated into a compiler to provide enhanced error messages that not only point to errors, but offer ways to correct them. We believe that our algorithm will perform useful and effective repairs that are well-aligned with the developer's intentions, even when the given ill-typed expression requires several steps to repair.

Acknowledgments. We thank Tihomir Gvero and Ivan Kuraj for early discussions about program repair.

References

1. S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 121–130, New York, NY, USA, 2011. ACM.
2. C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.
3. T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.
4. T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 418–423. Springer, 2011.
5. S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso. Minthint: Automated synthesis of repair hints. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 266–276, New York, NY, USA, 2014. ACM.
6. D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, 2005.
7. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 772–781. IEEE / ACM, 2013.
8. Z. Pavlinovic, T. King, and T. Wies. Finding minimum type error sources. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 525–542, New York, NY, USA, 2014. ACM.
9. Y. Pei, Y. Wei, C. A. Furia, M. Nordio, and B. Meyer. Code-based automated program fixing. In P. Alexander, C. S. Pasareanu, and J. G. Hosking, editors, *26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), Lawrence, KS, USA, November 6-10, 2011*, pages 392–395. IEEE, 2011.
10. D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, pages 275–286, 2012.