

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 Examples

3.1 Code Correction: Single Error

The following simple example, adapted from [3], illustrates how our algorithm handles an expression that does not compile.

```
String body = "email.txt"; String sig = "sign.txt";
SequenceInputStream seqStream = new SequenceInputStream(body, sig); // error
```

Here, the developer has given the `SequenceInputStream` constructor two strings, even though none of its overloads accept those arguments. Still, we can recognize the intent of the ill-typed expression. To preserve this intent, our algorithm constructs a code snippet that maintains the relative position of the original arguments. The corrected expression should favor reusing the well-typed sub-expressions, such as the `body` and `sig` variables in their corresponding sub-expression trees. Our algorithm finds all such constructible expressions from the visible declarations in the current scope. The results are well-typed and ranked according to a metric that favors resemblance to the starting expression.

In this case, the returned expression with the highest score is `new SequenceInputStream(new FileInputStream(body), new FileInputStream(sig))`, which precisely reflects the programmer's intent.

3.2 Code Correction: Multiple Errors

Sometimes, an ill-typed expression 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 uses an extensive number of calls to the standard Java API with the intention to read a compressed file through a buffered stream. The developer attempts to instantiate an `InputStream` object:

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

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

In this example, the single expression contains three errors. First, the `FileInputStream` constructor requires at least one argument; second, the `DeflaterInputStream` constructor has

been passed too many arguments; and finally, the `BufferedInputStream` has been passed valid arguments 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 correcting the sub-expression `new FileInputStream()`. The algorithm draws on the environment to produce `new FileInputStream(fileName)` as the closest match.

After applying this repair, it proceeds to correct the `DeflaterInputStream`. Since all of its arguments are now well-typed, the repair will attempt to re-use them while synthesizing a replacement. The algorithm wraps the extra arguments in a call to the `Deflater` constructor from the Java API.

Finally, it 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);
```

Our goal with these examples is to show how our algorithm efficiently searches through a large number of possible repair expressions, and guides that procedure according to an appropriate metric that values shorter distances between the given expression and the corrected expression.

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 each 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, the edges are weighted by their cost of inclusion, a subjective measure that guides the search towards desirable traits, such as smaller expressions or lower memory usage.

4.2 Synthesis Procedure

We now outline the synthesis portion of our algorithm, Algorithm 1. The algorithm takes as input the synthesis graph $G = (V_t \cup V_f, E)$, a type τ , a cost limit C_{max} , and the number of expressions to synthesize N . It returns a list of expressions of type τ . The first two steps can be done using Dijkstra's algorithm. Nodes V_t' are sorted in reverse order so as to cache at the latest possible opportunity. This also means that in every run of the loop σ is as close as possible to τ . If we create an expression of type τ

Algorithm 1: Synthesis Algorithm

input : $G = (V_t \cup V_f, E)$, the synthesis graph; $\tau \in V_t$, the type to synthesize; C_{max} , the maximum allowable cost; N , the number of snippets to synthesize

output: exprs, the list of expressions

- 1 $G' = (V'_t \cup V'_f, E') \leftarrow$ subgraph of G reachable within C_{max} from τ ;
- 2 Sort V'_t in descending distance away from τ ;
- 3 snips \leftarrow Hash table mapping types to snippets ;
- 4 **foreach** $\sigma \in V'_t$ **do**
- 5 snips $[\sigma] \leftarrow$ GetExpressions(G' , snips, σ , $C_{max} - \text{Dist}(\sigma)$, N) ;
- 6 exprs \leftarrow snips $[\tau]$;

that involves σ , then the rest of the expression may cost at most $C_{max} - \text{Dist}(\sigma)$. The procedure GetExpressions uses snips to avoid performing expensive recomputations.

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

Procedure GetExpressions($G' = (V'_t \cup V'_f, E')$, snips, τ , C_{now} , N)

- 1 **if** $\tau \in \text{Keys}(\text{snips})$ **then return** snips $[\tau]$;
- 2 results $\leftarrow \emptyset$;
- 3 **foreach** $g \in V'_f$ of the form $g : (\tau_1 \times \dots \times \tau_k) \rightarrow \tau$ **do**
- 4 **if** $\text{Cost}(g) > C_{now}$ **then continue**;
- 5 For all i , let $s_i \leftarrow$ GetExpressions(G' , snips, τ_i , $C_{now} - \text{Cost}(g)$, N) ;
- 6 **foreach** $\text{args} \in s_1 \times \dots \times s_k$ **do**
- 7 **if** $\text{Cost}(g(\text{args})) \leq C_{now}$ **then**
- 8 Add $g(\text{args})$ to results;
- 9 **while** $|\text{results}| > N$ **do**
- 10 Remove the most costly entry from results;
- 11 **return** results

recursively, and it checks to see whether the computation has been completed before by referring to the snips table. To compute candidates for $\tau \in V_t$, the procedure looks at its outgoing neighbors, which are all functions whose output is of type τ . For each function that would not immediately break the cost constraint, GetExpressions attempts to satisfy all of its argument types 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, and 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 those subexpressions of the broken expression that are correctly-typed. The intuition for this is that the search

should be directed to favor those components that the programmer wanted to use. To do this, we adjust the Cost function used by GetExpressions to scale down its results by a factor of 2^n , where n is the number of expressions “reinforced” by the repair procedure that appear in the snippet. As a practical matter, we say we have “reinforced” an expression when we add it to a list of reinforced expressions.

This scheme has a few distinct 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 subexpressions among those reinforced; and finally, if no expressions are reinforced, 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 : $G = (V_i \cup V_f, E)$, 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** $[\text{expr}]$;
- 2 Write expr as $\text{expr}(x_1, \dots, x_k)$ where x_i are its subexpressions of type τ_i ;
- 3 **foreach** $x \in \{x_1, \dots, x_k\}$ **do**
- 4 $x \leftarrow \text{Repair}(G, x, C_{max}, N)$; // Replace x with a list of either itself
 or its possible corrections
- 5 **foreach** $\text{subs} \in x_1 \times \dots \times x_k$ **do**
- 6 Reinforce all expressions in subs ;
- 7 Add all results of Synthesize (G, τ, C_{max}, N) to repairs ;
- 8 Clear reinforcements ;
- 9 $\text{repairs} \leftarrow$ 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 some typical-use benchmarks. 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 section 3.2 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.

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.