Extended Abstract: Type-Directed Program Repair

Alexander Reinking ∗
Yale University
New Haven, CT 06520
alexander.reinking@yale.edu

PROBLEM AND MOTIVATION
Programmers working in modern object-oriented languages often need to compose several large, complicated libraries together with in-house code while developing enterprise software. As these libraries evolve, legacy code that depends on deprecated declarations sometimes breaks. Many different transformations can break legacy code. For example, changing the name or signature of a function, changing the type hierarchy, or removing a deprecated declaration will leave behind expressions that are well-formed, but ill-typed. While such code will still reflect the programmer’s intent, it will no longer compile, and will need correction.

Due to the size of these libraries, it is also common for a programmer to call a function incorrectly, or be unsure how to construct a new object. This is particularly common outside of an IDE setting. Ill-typed code of this variety also reflects the programmer’s intent, as the components she wishes to compose are all present, while the composition itself is malformed.

We have designed an algorithm to compute plausible corrections to this class of well-formed, but ill-typed expressions. While the theoretical underpinnings apply to any statically typed language, we have chosen Java as the experimental target, and have provided an example implementation as a plugin to the Oracle Java Compiler.

BACKGROUND AND RELATED WORK
Most related to our work are the tools InSynth [1] and Prospector [2]. Both tools attempt to synthesize single expressions of a given type, and although their approaches differ, our tool subsumes both. InSynth introduced the notion of a “succinct type” of a function, which is a duplicate-free version of the function’s signature. Prospector introduced the “jungloid graph”, which consists of transformations from one type to another. We introduce the “synthesis graph”, which combines these two ideas by encoding how types are constructed from functions, and which types are necessary to call such functions. While Prospector can only transform one type to another, the synthesis graph allows us to generate passages of code that depend on multiple input types. Unlike InSynth, the synthesis graph allows us to generate code based on known snippets, not just compositions of functions.

Our main focuses are synthesis, and the repair of broken expressions. Existing tools for code repair take different approaches or operate on a larger scale. Our algorithm operates using a highly optimized search through a special graph structure, rather than by constructing an input to an SMT solver or theorem prover. As a result, it requires only the broken expression and the symbols visible at that program point as input. Similar tools ask for pre- and post-conditions [6], sometimes along with a test harness [6, 4, 3]. These extra requirements can be difficult for new programmers to understand or supply. Our tool does not require any input from the user, it takes only the ill-typed expression and aims to repair it from the type information alone.

The slow performance of existing repair tools is staggering. Many of these tools spend on the order of twenty minutes or more [4] to produce small snippets of code whose correctness is reliant on the programmer’s understanding of the problem. A mistake in the inputs on the part of the programmer would be costly, as this wastes time producing incorrect results. By contrast, our tool has shown millisecond runtimes in practice. We believe that our tool scales well; when the entire Java Standard Library is imported, the synthesis graph approaches fifty thousand nodes and twice as many edges. Even in this situation, it produces results in fewer than one hundred milliseconds, and runs even faster in a more typical setting.

APPROACH AND UNIQUENESS
The main data structure employed by our algorithm is the synthesis graph. This graph encodes the types and function signatures visible at the time the algorithm is called. The type nodes are connected to the functions whose return values match. Each function node is in turn connected to the types in its succinct signature. To build up expressions, the algorithm runs Dijkstra’s algorithm to expand a neighborhood around a desired type in the synthesis graph, and searches through that neighborhood for trees that correspond to well-formed expressions. Each function is weighted by its cost of inclusion into an expression, and a metric over the weights of each function ranks each candidate. When the search completes, the results are returned in order according to that metric. The synthesis process can be seen as repairing an empty expression of a given type.

To return quality results, we introduced two different systems of weights: a naive scheme that blindly assigns weights proportional to the number of arguments the function takes, and a stochastic scheme. This stochastic scheme was computed by analyzing the frequency of functions in an extensive available online corpus.

∗This was a joint project with my undergraduate supervisor Ruzica Piskac
In summary, the repair algorithm takes a non-empty expression, and proceeds from the innermost sub-expressions up to the top level. Whenever it encounters a well-typed sub-expression, it adds it to a list of reinforced expressions. This list biases the metric towards reusing the well-formed parts of the original, broken expression. When the algorithm finds a broken sub-expression, it synthesizes a new expression of the expected type with all the correct sub-expressions reinforced. The results are possible corrections for the current sub-expression and contribute to the reconstruction of the top-level expression.

RESULTS AND CONTRIBUTIONS
We are currently in the process of experimentally evaluating our algorithm. Our preliminary implementation [5] has produced promising results. As an illustration, our tool simultaneously introduced a missing parameter, wrapped extra parameters in a temporary object, and interchanged parameters that appeared in the wrong order all across multiple nested function calls.

In summary, the contributions of this project are:

- A novel approach to code repair based on type constraints.
- An efficient encoding of type constraints via the synthesis graph data structure.
- A compiler plugin that automatically suggests a list of corrections for ill-typed expressions.

REFERENCES