Abstract

Developing enterprise software often requires composing several libraries together with a large body of in-house code. Especially in object-oriented languages like Java, these libraries might add hundreds of classes, along with thousands of methods, constants, and functions to the environment, which adds a great deal of complexity for the developer to manage. Modern programming conventions favor composing many “small” objects together to create larger structures, or to delegate functionality or resource acquisition to other modules in a program. While useful, these techniques steepen the learning curve. A developer typically knows the approximate structure of the desired expression. However, often the first attempt at writing that code results in an ill-typed code fragment.

We describe an algorithm and a tool called Winston that automatically repairs code expressions based on the provided almost-correct code. At the core of our algorithm is a graph construction that expresses the relationships between the language’s types and methods. Such approach allows us to synthesize and repair expressions that are biased towards a given criteria by setting the edge weights appropriately. We implemented our algorithm as part of an IntelliJ IDEA plugin that proposes corrections for ill-typed expressions in the Java language. We tested Winston on synthesizing and repairing non-trivial expressions: it successfully returns the desired code snippet in a few milliseconds.

1. Introduction

Software development provides a high degree of freedom and many different approaches can be adopted for writing code. Still, when writing a program, the developer needs to follow the strict rules determined by the programming language. While coding, the developer often knows the approximate structure of the desired expressions but still may write code that does not compile because some fragments are not well-typed. Such mistakes occur mainly because the developer does not know by heart how to choose and properly combine all the necessary declarations visible from the scope. By the term “declarations”, we refer to all the elements visible in the scope, such as variables, functions, and class hierarchy declarations. Moreover, modern libraries often evolve into complex application programming interfaces (APIs) that provide a large number of declarations. For this reason it is difficult, if not impossible, to learn the specifics of every declaration and its utilization. In a typical scenario when code does not compile, the compiler outputs an error message with the expression that is at the source of the error.

Still, on many occasions the written expression reflects the intended structure of the code.

As an illustration, a programmer might expect the Java code

\[
\text{BufferedReader } \br = \text{new BufferedReader("file.txt");}
\]

to open a file called “file.txt”. However, this code will not compile since BufferedReader accepts only a Reader interface implementation as an argument. After consulting the documentation, the programmer corrects the previous line to

\[
\text{BufferedReader } \br = \text{new BufferedReader(new FileReader("file.txt");}
\]

Mistakes like these are common when exploring a new API, or a new language, since classes are often composed in unintuitive ways. A survey conducted on 157 software engineers at Microsoft [LaToza et al. 2006] revealed that a full 56% admit to spending a large amount of time trying to understand code that other people wrote, and 41% agree that the amount of example code adapted for a production setting is a serious problem. We believe that the time spent perusing the API documentation could be spent actually writing the application-specific parts of the code.

In this paper we describe a tool, called Winston, which automatically repairs code expressions based on the hinted structure of the ill-typed code. Winston finds well typed expressions that are as close as possible to the given (potentially) ill-typed expression - we call such an input expression, a backbone expression.

Additionally, Winston can also be seen as a synthesis tool. It extends the functionality described in [Mandelin et al. 2005; Gvero et al. 2013; Perelman et al. 2012]. In the light of the program repair, the synthesis aspect of Winston can be considered as a repair of the empty expression. A user does not need to provide a backbone expression - it is sufficient to declare a variable of an arbitrary type. Based on that type Winston can synthesize corresponding code fragments. The synthesized code has the given type and it can contain user defined values, as well as methods from the API.

Our tool can be applied in interactive scenarios like IDE code completion, to rank expressions based on their similarity to ill-typed code. Ideally, the best suggestions will correct the code while preserving its overall structure. Another possible application of the Winston algorithm is to provide automated repair as a part of the compilation process.

Winston is a tool based on a graph algorithm for synthesizing expressions of a certain type in a programming language. As an algorithm, a synthesis process is generalized to repair ill-typed expressions. We believe that this graph construction will prove to be a useful tool for synthesis and repair in other contexts, as well. Winston ranks expressions based on a system of costs that...
appeal to general principles (eg. the principle of locality), but other contexts might call for different valuation schemes in order to bias the synthesis towards or against certain expressions. For example, one might wish to favor resource acquisition near program entry points, but adjust the costs in the other direction when attempting to synthesize snippets deep inside the class hierarchy.

The research on improving the software development process covers a large number of topics such as an automated program repair [Le Goues et al. 2012; Wei et al. 2010; Pei et al. 2011], enhancements of compiler messages [Burke et al. 1987; Hammond and Rayward-Smith 1984; Lerner et al. 2007], and providing assistance to developers through code inference [Gvero et al. 2013; Mandelin et al. 2005; Kneuss et al. 2013; Kuncak et al. 2013; Perelman et al. 2012]. As a result, a vast number of tools have been created around a common high-level goal of expediting software development. The manner in which such tools operate can be broadly divided into two categories: (1) as automated processes within the compiler (2) or as development assistants that require a certain level of interaction, usually through an IDE interface. Many of the techniques behind these tools such as parsing error recovery by altering the input [Burke et al. 1987], a heuristic search for syntactically correct terms [Perelman et al. 2012], a modification of abstract syntax trees and types [Lerner et al. 2007], and inference of semantically correct code fragments [Kneuss et al. 2013], share common insights.

Motivated by the advances in both the theory of programming languages and techniques that are foundations of tools for software development, our approach addresses the problem of code repair from a new perspective, by providing an algorithm that extends existing methods and incorporates new ideas. Our tool goes beyond the existing line of work in three important ways: (1) the algorithm tries to solve more general code repair problems constrained by the structure of given ill-typed terms, (2) it focuses on repairing programs in as much accurate way as possible, according to the given hint and weight heuristics, while providing useful theoretical guarantees about the utilized repair algorithms, (3) it is suitable for realization as both an interactive and automated software development tool.

The principal contributions of this paper are:

- An especially-efficient synthesis and repair algorithm. Winston outperforms existing tools, sometimes by several orders of magnitude, while still producing high-quality results. From a theoretical standpoint, the Winston algorithm is sound and complete.
- A practical implementation of this algorithm as an IntelliJ IDEA plugin. The code will be made available on the first author’s website.

### 2. Motivating Examples

We first illustrate how Winston works and the nature of repair with the usage of familiar API calls from the Java standard library, reflecting along the way on the differences between our and other tools for code completion and repair. To emphasize the significance in practical software development, our examples are chosen from a set of the real code examples (most of them featured at [http://www.java2s.com/](http://www.java2s.com/)).

#### 2.1 Winston as a Synthesis Tool

Figure 1 is a screenshot demonstrating that Winston provides synthesis as a part of its functionality. In the scenario in Fig. 1 the user has already written some parts of the code, and now she would like to create a regular expression matcher. After declaring a variable, she invokes Winston, which suggests a list of well-typed expressions that would fit at the current program point. Hopefully, the desired code is also among the returned results. It is the case in this particular example, and the user selects the third result, which is then automatically inserted in the program. The number of the results returned by Winston can be easily adjusted in the settings.

Finding useful code snippets is an important aspect of every synthesis tool; it is not enough that the synthesized code can compile – it should also be the code that the user had in mind. It is not always possible to guess correctly the user’s intentions, but as we show in Section 5, Winston is able to suggest the desired code fragment in around a tenth of a second, even with the entire standard library imported.

In Section 3 we explain in details how to generate “most desired” results. The Winston algorithm relies on a system of costs to accurately guess programmers’ intentions. The system of costs is similar to the system of weights used in [Gvero et al. 2013].

This smart auto-completion functionality is present in many tools [Mandelin et al. 2005; Gvero et al. 2013; Perelman et al. 2012]. The novelty present in Winston is in its high performance, flexible graph-theoretic foundations, and auto-correction, which we illustrate in following examples.

#### 2.2 Code Correction: Stream Concatenation

We now show a difference between Winston and InSynth [Gvero et al. 2013]. We take an example that is similar to the example presented in the InSynth paper to illustrate how Winston handles a backbone expression that does not compile.

Consider the following program fragment:

```java
import java.io.*;

public class Main {
    public static void main(String args[]) throws IOException {
        String body = "email.txt";
    }
}
```

Figure 1. Winston suggesting five highest-ranked well-typed expressions synthesized from declarations visible at a given program point
SequenceInputStream (body, sig)
}
}

The developer has declared the variable seqStream, however, the expression assigned to seqStream does not compile. Still, from this backbone expression we can recognize the structure of the intended expression. Our tool should construct an expression that preserves the relative position of the declarations from SequenceInputStream(body, sig). In the resulting expression a SequenceInputStream constructor should be used, with arguments that contain body and sig variables in their corresponding sub-expression trees. Winston finds all such expressions, constructed from the declarations visible in the scope of the backbone expression. The found expressions are well-typed and ranked according to a metric that characterizes the resemblance to the starting backbone expression. The returned expression with the highest rank is

SequenceInputStream seqStream =
    new SequenceInputStream(
        new FileInputStream(body), new FileInputStream(sig))

This expression represents exactly the desired expression, which was derived in only 4 milliseconds.

When we ran InSynth on the same example, but without the backbone expression, the desired expression was ranked as the second highest. Our tool outperformed InSynth on this example, showing that the backbone expression can increase quality of the returned results.

Unlike InSynth, our repair algorithm does not distinguish between value literals (constants) and local variable declarations. This is an additional advantage that our repair algorithm has over InSynth. As an illustration, given the backbone expression:

SequenceInputStream seqStream =
    new SequenceInputStream("email.txt", "sign.txt")

Winston returns, as expected:

SequenceInputStream seqStream =
    new SequenceInputStream(
        new FileInputStream("email.txt"),
        new FileInputStream("sign.txt"))

InSynth is unable to synthesize code snippets with arbitrary literals. From this perspective, our code repair algorithm can be considered an improved synthesis algorithm, because it also tries to incorporate explicitly given literals in its suggestions.

2.3 Use of Coercion Functions

To address the problems that can arise with subtyping, we introduce coercion functions [Tannen et al. 1991] to the graph. They are used for type conversion and can be applied automatically if needed, without the direct intervention from the developer. For every subtyping relation $A <: B$, we introduce a coercion function $c$, such that $c : A \rightarrow B$ is the inclusion mapping. An automated insertion of coercion functions is utilized for the purpose of fixing ill-typed expressions in many modern compilers, but usually in a limited manner (at most one coercion function can be used to fix an ill-typed expression). Our repair algorithm goes beyond this standard and allows more expressive transformations of ill-typed expressions by allowing for an arbitrary number of coercions to be inserted.

Consider the following code, in which the developer declares a byte buffer and wants to construct an expression of the type InputStream by merely hinting the desired type and usage of the declared buffer $b$:

```
String sig = "sign.txt";

SequenceInputStream seqStream =
    new SequenceInputStream(body, sig) // error
}
}

// error
```

To repair the expression that initializes the variable input, Winston inserts a coercion function around a constructor application:

```
InputStream input = new ByteArrayInputStream(b);
```

Winston returns this expression as the highest ranked expression: the expression is well-typed, and follows the simple structure of the backbone expression $b$, with the smallest size. The value was computed in 115 milliseconds. This slightly slower speed represents the added cost of searching through neighborhoods of primitive types, which can be quite large. The upcasting done in this example can be seen as an implicit insertion of a coercion function, which casts ByteArrayInputStream to its superclass, InputStream.

The algorithm returns additional well-typed expressions, such as

```
InputStream input = new ByteArrayInputStream(b, off, len);
```

This expression also correctly repairs the given backbone expression, but it no longer represents a simple coercion function insertion. It is the ByteArrayInputStream overloaded constructor with three arguments. To create this expression, our algorithm considers a broader range of available functions and recursively finds appropriate expressions that fill the places of the missing arguments. Those arguments are synthesized whenever a type-conversion function requires additional parameters.

In general, the Winston algorithm is based on advanced methods for searching and adapting appropriate functions, combined with synthesizing any additional necessary arguments.

2.4 Mutations of Ill-typed Expressions

Sometimes, a developer writes an ill-typed code that poorly reflects the structure of the desired expression. This is usually caused by passing arguments in the wrong order, or passing too many or too few parameters to a function.

Consider the following code that uses an extensive number of calls to the standard Java API library to manipulate streams. The given backbone expression hints the user’s intention to read a file compressed with the ZLIB library through a buffered stream. To read the file, a user needs to instantiate an InputStream object.

```
import java.io.*;
import java.util.zip.*;

public class Main {
    public static void main(String args[])
        throws IOException {
            int off = 8, len = 512, size = 1024;
            byte b[] = args[0].getBytes();
            InputStream input = b; // error
        }
    }
```

This expression also correctly repairs the given backbone expression, but it no longer represents a simple coercion function insertion. It is the ByteArrayInputStream overloaded constructor with three arguments. To create this expression, our algorithm considers a broader range of available functions and recursively finds appropriate expressions that fill the places of the missing arguments. Those arguments are synthesized whenever a type-conversion function requires additional parameters.

In general, the Winston algorithm is based on advanced methods for searching and adapting appropriate functions, combined with synthesizing any additional necessary arguments.
In this example the user wrote arguments of the BufferedInputStream constructor in the wrong order. To correct this error and create a desired expression, Winston algorithm has to change the order of the BufferedInputStream argument in the initial backbone expression. After applying a set of further modifications to the arguments, we derive a new backbone expression: new DeflaterInputStream(new FileInputStream(fileName), new Deflater(compLevel, true)). We recursively continue this procedure until we reach leaves in the Winston graph (cf. Sec. 3). Finally, the algorithm corrects the backbone expression by inserting the Deflater constructor. At the end, Winston suggests the following correction for the given expression:

\[
\text{InputStream input} = \text{new BufferedInputStream(new DeflaterInputStream(new FileInputStream(fileName), new Deflater(compLevel, true)), buffSize);}
\]

Winston derives this code snippet in approximately 380 milliseconds.

These examples show that our algorithm can perform a search through a large number of possible repair expressions, and can guide that procedure according to an appropriate metric that characterizes the distance from the given expression to the corrected expression.

We hope that Winston can indeed 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.

3. Algorithm Description

This section describes an algorithm used to repair ill-typed expressions. The algorithm takes as input an ill-typed expression to be repaired, and two tuning parameters used to guide the internal queries: \( N \), the number of possible subexpressions to consider for any type, and \( L \), the maximum allowable cost for an expression to be returned. These bounds are necessary to avoid entering an infinitely-deep recursion – it is not uncommon for classes to accept instances of themselves – and to control the runtime of the algorithm. Winston has access to the environment via the synthesis graph, which should be precomputed for libraries, and extended as necessary in an IDE setting.

As a subroutine, Winston uses a graph-search-based algorithm for expression synthesis. This portion of the algorithm is of independent value, and can be called directly using our tool.

3.1 Synthesis Graph Construction

The data structure central to Winston is a colored, directed, weighted, simple bipartite graph \( G = (V = V_t \cup V_m, E, w, c) \) between type nodes \( V_t \) and map nodes \( V_m \), henceforth the synthesis graph. A type is any concrete data type, and a map is any language entity that produces exactly one such data type (possibly void) from zero or more types, i.e. it can be modeled by some map \( f : (\tau_1 \times \cdots \times \tau_k) \rightarrow \sigma \). In the synthesis graph, for each map, an incoming edge is drawn from its co-domain, and outgoing edges are drawn from the map to each type in its domain. These relationships capture the notions that each type or map can be constructed from or satisfied by other maps or types, respectively.

The weights on the synthesis graph represent the cost incurred by including the edge as a part of a candidate snippet. These costs can be derived according to simple heuristics, or computed by examining the frequency of usage over a large corpus. It is important only that no negative costs be assigned anywhere in the graph, and that methods that are more important are valued more than those that are not. The algorithm evaluates candidate expressions by considering their cost, not their value, so lower edge costs are considered better. Finally, the synthesis graph is colored according to which language structure added the node to the graph, e.g. constructors, methods, public fields, and the like. These groups color the graph, and are used by the synthesis procedure to decide how to construct the code snippets that ultimately serve as output.

Figure 2 illustrates an example portion of a neighborhood in the graph around BufferedReader, a common Java class. The only valid expression derivable from this small neighborhood is new BufferedReader(new PipedReader()), which is due to the fact that no local variables are included.

![Figure 2](image-url)

Figure 2. A subset of a ball of cost-radius 3 surrounding BufferedReader. The path to PipedReader forms the only valid expression derivable from this graph. The boxed nodes are types, while the oval nodes are maps.

3.2 Type Inference

Since it is often the case that very general types, such as integers and strings, carry additional meaning, the synthesis graph must be extended to include abstract type information that encodes whether a particular argument is, for example, a counter, a file descriptor, or an untyped enumeration, in the case of an integer, or in the case of a string, a URL, a file name, or a regular expression. There are already standard approaches for doing this; Lackwit is such a tool for C language programs [O’Callahan and Jackson 1997]. Once the abstract types have been identified, they can be added to the graph as subtypes of String, int, etc., with a small, but non-zero cost for “downcasting” a normal string or integer to the specific abstract type. This penalty ensures that those expressions that respect the abstract types are preferred over those that don’t.

Although this is technically an optional step in so far as removing it does not pose a challenge to correctness, it serves as a highly-effective heuristic for guiding the search. Even though the returned expressions are always correctly-typed, they might still throw runtime errors; the additional type information recovered by this step reduces the chance of that happening.

3.3 Synthesis Procedure

Using this graph, we can answer quantitative type-inhabitation queries. That is, given a type \( \tau \), we seek expressions of that type that are of low cost. Naturally, the quality of the expressions returned is tied tightly to the costs. Fortunately, as explained in section 1, the relative cost of the expressions matters more than the absolute costs. Thus, the costs act mostly as a heuristic to guide the search for valid expression trees in the graph.

The Synthesize procedure, outlined below, answers these queries by growing a Dijkstra shortest path ball in the graph of a specified size. The parameter, \( L \) acts as a cost limit for the search, and can be tuned to provide a reasonable balance between speed and accuracy,
depending on the chosen valuation scheme. The induced subgraph over those nodes is then recursively searched for type satisfaction by GetExpressions, which is called leaves-first to enable memoization of its results. Since each type is passed a limit equal to the greatest possible remaining cost during a search from τ, the whole (reduced) space of results is considered for each type. Once a type is settled, the result is added to a memoization table to speed up future calls.

The other tuning parameter, N, can be provided to bound both the total number of expressions returned and the number of intermediate results at any given depth by N. Because all of the possible generators are considered regardless, we can be sure that the algorithm does not miss any possibilities during the search that might appear among the N best results.

Procedure Synthesize(G, τ, L, N)

```plaintext
1 \( (V \cup V_m, E, w, c) \leftarrow G \);
2 \( (D_1, D_m) \leftarrow \) nodes distance less than L away from \( \tau \in V_1 \);
3 \( */ D_1, D_m \) ascending by Dijkstra’s */
4 genTable \( \leftarrow \{\} \);
5 snippetTable \( \leftarrow \{\} \);
6 foreach \( f : (\tau_1 \times \cdots \times \tau_k) \rightarrow \sigma \) \( \in D_m \) do
7 \( \) genTable[\( \tau \) \] \( \leftarrow \{ f, w(f) \} \);
8 typeStack \( \leftarrow \) new Stack;
9 foreach \( \sigma \in D_i \) do
10 \( \) push \( \) (\( \sigma, L - \) shortestPathLength(\( \sigma \)) \) on to typeStack;
11 while typeStack is not empty do
12 \( f, w(f) \) \( \leftarrow \) pop typeStack;
13 snippetTable[\( \sigma \) \] \( \leftarrow \) GetExpressions(genTable, snippetTable, \( \sigma, L, N \));
14 return snippetTable[\( \tau \));
```

Note that in Synthesize, all sets are maintained in sorted order using a tree data structure, and genTable and snippetTable are both hash tables of these sets.

Once the results from Synthesize have been obtained, the built up expression can be converted into strings by recursively applying a string formatting function appropriate for the colors of each generator in the expression. The leaves (0-input maps) evaluate to known strings, and fill in the parameters of the maps above them until a complete code snippet is built.

The subroutine GetExpressions goes through each possible generator for a given type and attempts to satisfy its argument types recursively. It checks a memoization table before computing the set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work. The recursion itself is straight-forward: for each function that can generate the requested set of expressions to avoid repeating work.

3.4 Repair Procedure

We are now ready to describe the repair procedure. The key step in our approach is biasing the synthesis towards those subexpressions of the backbone expression that are, in fact, correctly-typed. The reasoning behind this is intuitive: if a user specifies that she would like to create a new type from some specific components, the search should be optimized to favor using those same components.

To do this, we modify the GetExpressions procedure to accept a preference map, \( \text{pref} \), supplied by the repair procedure. The preference map contains a set of subexpressions (indexed by their type) which will be used to bias the search in their favor. We modify GetExpressions so that line 2 initializes \( \text{snips} \) with \( \text{pref}[\tau] \), rather than the empty set. We modify the sets to compare snippet costs after reducing by a factor of \( 2^n \), where \( n \) is the number of expressions in the preference map which occur in each snippet. This does not adjust the base cost, but is used as a heuristic. Since GetExpressions is a subroutine of Synthesize, we must add an argument to its signature in order to forward the preference map to GetExpressions.

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 element of the same type is required, it will favor diversity over homogeneity (ie. if both \( a \) and \( b \) were specified, it will try to use both); and finally, if no subexpressions are supplied, then the modified algorithm degenerates into its original form.

With this modification in place, the repair algorithm proceeds from the bottom up. When we insert an expression in \( \text{pref} \), we will say that we have \( \text{enforced} \) that expression. So, for each broken subexpression in the backbone, we first enforce all of its correctly-typed child expressions and then initiate a synthesis on the current subexpression. If any of its children are not correctly typed, we recurse and repair them first.

Notice that this means the repaired subexpressions will also be enforced. This behavior is desirable since it favors reusing the computations made before when synthesis occurs at a higher level. Additionally, this process only enforces those literals that are contained within a given subexpression, so that two disjoint subexpressions do not interfere while initiating a nested repair.

Procedure Repair(G, expr, L, N)

```plaintext
1 if expr is well-typed then return \{\} ;
2 \( f : (x_1 : \tau_1, \ldots, x_k : \tau_k) : \sigma \leftarrow \text{expr} ;
3 */ S is the preference map */
4 S \( \leftarrow \{\} \);
5 foreach \( x \in \{x_1, \ldots, x_k\} \) do
6 \( (x, S_x) \leftarrow \) Repair(x, L, N);
7 S \( \leftarrow S \cup S_x \);
8 return (\( \text{Synthesize}(G, \sigma, L, N, S) \) )
```

Although Repair as described returns many expressions, none of our tests returned better expressions past the first one, so in
a practical setting, letting \( N = 1 \) is acceptable, and could be considered the “canonical” repair.

### 3.5 Completeness Guarantees

Because the algorithm is limited only by user defined parameters, the search can be tuned to consider every node in the graph. We know already that no type-synthesis results are cached until we have visited the corresponding node at the lowest depth away from the starting type. Thus, results are cached exactly when the cost available for that type is maximized. Since the breadth and depth restrictions can be elided, the full space of possible expressions is reachable. Note, however, that without a depth limit, the algorithm will not terminate, since many copy-constructors or other identity maps often times exist in the environment (e.g. one could synthesize \( f(f(f(\ldots f(a)\ldots))) \) ad infinitum). With a depth limit, it was shown in [Gvero et al. 2013] that the expression synthesis problem corresponds to the type-inhabitation problem, which is known to be PSPACE-complete [Urzyczyn 1997]. Fortunately, the ball-growing step runs in polynomial time (same as Dijkstra’s algorithm), and although we have no proven bounds on the latter step, it has been extremely efficient in practice. Thus, we know that since this problem is PSPACE-complete, there always exists a depth limit larger than the entire graph, but finite such that if an expression exists, it can be found by the algorithm in finite time.

### 4. Implementation

The primary goal of our implementation was to produce a library that can handle these queries, and a tool that provides an interface in an IDE setting. We provide a full implementation of the algorithm described, along with a plugin for IntelliJ IDEA that adds a repair intention, and enhances its autocomplete feature through the Synthesize procedure.

Although the definitions of type and map for our algorithm are language-agnostic, our implementation targets Java for its ease of analysis. In Java, we add type nodes to the graph for each class and for each primitive type. The map nodes are derived from methods, static functions, constructors, local methods, and values (which take void to their own data type), and are colored according to each category.

#### 4.1 System of Costs

For our testing, we devised a simple system of costs for each type of node. (See Table 1) These costs are applied only to the output edges on the map nodes. The weights elsewhere are set to 0. Despite its simplicity, we found that this cost assignment yielded good results in practice, as the desired expressions always appeared very early in our evaluations.

<table>
<thead>
<tr>
<th>Type</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Variables</td>
<td>1/1000</td>
</tr>
<tr>
<td>Up-casts</td>
<td>1/100</td>
</tr>
<tr>
<td>Fields</td>
<td>1</td>
</tr>
<tr>
<td>Methods</td>
<td>1 + ( k/2 )</td>
</tr>
<tr>
<td>Constructors</td>
<td>1 + ( k )</td>
</tr>
</tbody>
</table>

Table 1. Costs for output edges in Java. \( k \) is the number of arguments the associated map takes.

The general intuition is that by locality of reference, local variables should cost next to nothing. Since promoting a type to its supertype has no syntax, and happens automatically in most languages, there should be no penalty for doing so. Fields (and other values) are given a neutral weight of one. General methods are considered the next most expensive since they are able to reuse cheap, existing objects in the environment. Finally, constructors are considered the most expensive since they require allocating and possibly acquiring additional resources.

There is plenty of room for improvement in this cost assignment. For example, this scheme does not take in to account the actual use of each function in practice. In InSynth [Gvero et al. 2013], the authors describe a system of weights that, like this system, were derived by trial and error, but weigh imported methods by their frequency in a corpus. In both InSynth and Winston, the relative ordering of each kind of generator matters more than the specific costs. It is important to note that in this system, many snippets will have the same cost, so some criteria must be used in order to break ties. Without a deterministic tie-breaker, the results might not be deterministic since many ordered set implementations break ties arbitrarily.

Still, we would like to stress that any choice of costs is ultimately arbitrary in that there is no objective metric of a “best” expression. It is always possible that in some situations, a desirable expression might not be returned by Synthesize, since it does not take the semantics of the surrounding code into account directly (i.e. no control flow analysis is performed). We have chosen these weights precisely because they produce appealing results in practice; if instead, one wished to synthesize long, complicated expressions, our algorithm would perform equally well.

#### 4.2 Practical Graph Construction

To construct the graph for the Java target, rather than writing a full parser for the Java language and analyzing the sources, or relying on the JVM’s reflection capabilities to search for method signatures, we opted to write a simpler parser for the output of javap, the standard JDK class file disassembler. The javap tool can write to standard output just the public definitions for every class in a jar file, which affords us the ability to easily filter out certain packages or classes before creating the graph using existing tools. In particular, it was necessary to remove the \texttt{sun.\_\_package} set of packages, since they are incompatible with OpenJDK, and contain implementation details of the Oracle JVM.

During the parse, nodes are added to a graph that has been initialized with all the primitive types as described in Section 3.1. Each class forces a node to be added, and method and field nodes are added to the graph for each such entity in the class definition. When a method references a type that has not yet been seen, it is added to the graph on the spot, and held in an index so when it is encountered, the same node can be used by the construction procedure.

In practice this graph is medium-sized and sparse. Importing the whole standard library beneath the \texttt{java.*} package results in a graph with 38930 nodes, and 88291 edges, or about twice as many edges as nodes. Even so, the cost limits reduce the search space to a far more manageable size. In the case of a BufferedReader, a ball of cost-radius 5.0 consists of merely 6842 nodes and 10592 edges, and takes about a split second to search. In practice, however, the whole graph need not be considered. If the graph is restricted to only those libraries that are already imported, and those that are heavily used – say, \texttt{java.io, java.util, and java.net} – then the ball shrinks to 1041 nodes and 1698 edges, and takes only a few milliseconds to search.

### 5. Evaluation

In this section we present the results of our tool on some typical-use benchmarks. The runtimes were measured on a university-supplied computer equipped with 32GB of RAM, a 4-core Intel Xeon E5-1620 CPU clocked at 3.60 GHz, running Red Hat Enterprise Linux
Server 7.0 on the Oracle JVM version 8u25. The best times of 50 consecutive trials were recorded to account for variance in process scheduling, cache behavior, and, most importantly, JVM warmup. It was not uncommon to see four or five time speed increases between the best and the worst measurements of the algorithm. This is due to the delay in program optimization afforded by the Just-in-Time (JIT) compiler built in to the JVM.

The benchmarks show both that the tool is accurate and fast enough to operate in an interactive setting. Since the full set of Java libraries need not be imported, the algorithm should run far faster in practice, since not too many packages will be imported. Another approach would be to adjust the weights on the graph to bias against crossing package boundaries as was done in [Mandelin et al. 2005]. In our implementation, the IDE plugin loads the same graph as used in the benchmarks, and operates comfortably on-demand as a code completion contributor.

Additionally, these benchmarks show that repair is accurate even in the face of multiple, difficult errors. The example involving the BufferedInputStream and the DeflaterInputStream had several distinct errors: a missing parameter, two parameters transposed, and additional parameters passed to a function that did not accept them that needed to be wrapped in a helper class. In three calls to Synthesize, Repair corrected all three errors in around a quarter 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. Related Work

We next provide an overview of related work on program repair and other algorithms that share similar insights, ideas and techniques to our approach.

**Code Repair.** Debugging and locating errors in the code [Pavlinovic et al. 2014; Chandra et al. 2011] play an important role in the process of increasing software reliability. Once located, some errors can be easily fixed. Recently we have witnessed a number of tools that aim to repair parts of code. MintHint [Kalesswaran et al. 2014] is such a tool that takes a more complete-program approach to code repair. Where we are using user input to guide synthesis of a correctly-typed expression from an incorrectly-typed one, MintHint targets a particular passage that is suspected to carry a bug. MintHint synthesizes some hints in areas that the algorithm considers erroneous, by both symbolically and actually executing the code and comparing its output to test cases.

Automated inference of program fixes and contracts. The goal of the AutoFix project [Wei et al. 2011; Pei et al. 2011] is to fix object-oriented programs according to their contracts. One commonality with Winston is the common goal of inferring code. In [Wei et al. 2011; Pei et al. 2011] program fixes and contracts are generated based on predefined code templates. Rather than filling the predefined code templates, our approach employs smart searches through the space of all expressions that satisfy the constraint for a successful repair. Winston derives code and repairs based only on type information, without characterization of program run-time behavior.

**Syntactic Error Diagnosis and Repair** A related line of work considers a syntactic error diagnosis, as well as a recovery and a repair after such errors are found [Hammond and Rayward-Smith 1984; Degano and Priami 1998]. The main target of these schemes is the parsing phase within the compilation process. Although our algorithm focuses on the repair of expressions that are correctly parsed according to the appropriate language grammar rules, some similarities may be noted. With respect to these techniques and introduced taxonomy, the Winston algorithm may be viewed as a type-checking repair scheme that is global, since it considers all the declarations in the scope for the repair, and interactive, since the developer can choose between multiple offered repair expressions and impact the repair process of subsequent errors. One of the most representative scheme for dealing with syntactic errors in LR and LL parsing is presented in [Burke et al. 1987]. Rather than just trying combinations of insertions, substitutions and deletions on a predefined length of the input after an error is detected, our approach considers the abstract syntax tree of the given expression and tries to encode all valid combinations of insertions in order to produce one or multiple expressions that would successfully type-check.

**Searching for Better Type-error Messages** An interesting work, related to code repairing, has been done on improving type errors and corresponding error messages [McAdam 2001; Lerner et al. 2006, 2007]. Although these approaches use techniques to modify type information [McAdam 2001], as well as the program [Lerner et al. 2007], they target the problem of unintuitive type-checker’s error messages for the purpose of giving better feedback to the developer. Our approach focuses on somewhat more ambitious goal of code repair and it does this by preserving the type-level information as well as the structure of ill-typed expressions. Some techniques used in these approaches are similar to the way backbone expressions are mutated in our approach, like argument addition and reordering [Lerner et al. 2007]. The crucial difference is that those techniques are utilized only after an independent search mechanism determines places for such modifications to be done, while our approach finds all suitable repair expressions according to the whole backbone expression and the weight heuristic.

**Frameworks for Deductive Synthesis and Execution of Specifications.** Frameworks that encompass verification, deductive synthesis and run-time constraint solving were presented in [Kuncak et al. 2013; Kneuss et al. 2013]. These frameworks leverage modern SMT solvers and use various techniques for synthesis and verification which are integrated into an interactive tool with the aim to introduce specifications into the development process of reliable software. Although these frameworks address more general goal of integrating software construction and verification while automat ing multiple aspects of the development process, in contrast to our approach that generates code fragments that do not need to satisfy formal specification, an interesting parallel can be drawn between such frameworks and our approach for code repair. When given a program that contains pieces of incomplete implementation, the aforementioned frameworks may employ techniques that synthesize missing fragments or allow solving them at run-time. Therefore, these frameworks can be viewed as repairing partial implementations that may happen at compile-time or may be deferred to run-time. The domain of both frameworks includes only purely function programs, it is additionally constrained by the expressiveness of the underlying SMT solver theories, verification techniques, and deductive synthesis rules. Winston does not need to address those constraints, however, having only a type constraints results in a simpler code fragments that can be synthesized and repaired.

**Sketching.** Program sketching tries to infer code fragments from the specification given as separate incomplete programs [Solar-Lezama et al. 2007, 2006]. Practicality is achieved by focusing on particular domains that allow algorithms that employ a guided search over the syntax trees of the synthesized program with an a priori determined bound on the syntax tree size. While sketching requires code contracts and a program skeleton, our repair tool needs merely a hint of the resulting expression and uses graph-based techniques to explore the unbounded space of expressions.
that are type-correct and conform to the structure of the backbone expression. Additionally, our approach is driven by externally defined heuristic measures to guide the search towards better solution candidates.

**Program Synthesis.** Winston is a repair tool that can also be used for software synthesis, since its repair algorithm is built on the top of a synthesis algorithm. In addition to standard synthesis tools that we use through the paper as a comparison to Winston, there is a number of synthesis tool with a similar functionality. CodeHint [Galenson et al. 2014] is a new dynamic approach to software synthesis that combines debugging and dynamic execution. CodeHint can also be used in interactive settings, but the main focus of CodeHint is code generation and not code repair. CodeHint can also be used in interactive settings, but the main focus of CodeHint is code generation and not code repair. XSnippet [Tansalarak and Claypool 2006] and SNIFF [Chatterjee et al. 2009] are tools that maintain a repository of code snippets and help programmers to find suitable code. Contrary to this approach, our synthesis algorithm is dynamic as the suggested code snippets are dynamically changing with the program that the user has written up to the point when she invokes our tool.

**Type Inference with Automatic Insertion of Coercions.** Finally, since coercion functions are the basis of our approach to subtyping, let us compare to the related work in that field. Several approaches on using coercions within type checking and type inference were presented in [Mitchell 1984; Luo 2008]. This line of work focuses on type inference with automatic insertion of coercions in the context of functional programming languages. A recent work [Traytel et al. 2011] presents a more sophisticated technique that extends Hindley-Milner type inference with coercive structural subtyping and goes beyond inserting coercions for local expression repairs in the type inference algorithm. The work presents an algorithm that derives subtype constraints from the whole target term, solves them to get a substitution consistent with the partial order on types, and finally applies the substitution to obtain a term that type-checks. Our approach conceptually differs from the aforementioned in multiple dimensions of the repair. The main difference is that our approach searches for any possible expression that is consistent with the types and structure as defined by the backbone expression. The setup for the aforementioned approach requires mappings that define coercions between types, while our approach leverages automatic search to find consistent function applications. While both approaches insert coercions only to function arguments, our approach considers all combinations of term applications that may produce coercion terms with appropriate types, together with any “boarding” arguments and mutations that may occur.

### 7. Conclusions

We have described an algorithm and a tool that performs a type-directed synthesis and repair. The user can give a hint of intended code, or just provide a type of a desired code snippet, and our tool returns a list of valid expressions which correspond to those requirements. In addition, the expressions are ranked by a given system of costs, which capture the user’s intent of finding the most desirable snippet.

We believe that this algorithm has two compelling uses. First, it can assist programmers in writing complex expressions. Second, it could be integrated into a compiler to provide helpful error messages that can adapt the costs to reflect semantic information derived from context (something that is not always possible in an IDE setting). We compared our tool against available, state-of-the-art synthesizers and found that our tool provides a similarly high quality set of results in a far shorter amount of time. Additionally, we tested our repair algorithm on severely broken programs and found that Winston can correct a wide range of errors, even when multiple such errors occur in the same expression.

### References


### Table 2. Typical-use runtimes for Winston in various examples. “Vertices” and “Edges” refer to the number of objects included inside a ball of cost-radius 4.5 around the origin type. The entire Java standard library from rt.jar (excluding the sun. and com.sun. packages) was used to build the graph before running the benchmarks; it consistently took around 5 seconds to load the data set from its serialized form.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Type</th>
<th>Time (ms)</th>
<th>Vertices</th>
<th>Edges</th>
<th>Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>SequenceInputStream</td>
<td>Synthesis</td>
<td>&lt; 1</td>
<td>141</td>
<td>149</td>
<td>1</td>
</tr>
<tr>
<td>FileInputStream</td>
<td>Synthesis</td>
<td>38</td>
<td>7832</td>
<td>10516</td>
<td>1</td>
</tr>
<tr>
<td>InputStreamReader</td>
<td>Synthesis</td>
<td>29</td>
<td>7064</td>
<td>9673</td>
<td>1</td>
</tr>
<tr>
<td>Matcher (regex)</td>
<td>Synthesis</td>
<td>93</td>
<td>14505</td>
<td>24740</td>
<td>1</td>
</tr>
<tr>
<td>InputStream (from byte array)</td>
<td>Synthesis</td>
<td>116</td>
<td>13163</td>
<td>20581</td>
<td>2</td>
</tr>
<tr>
<td>BufferedReader</td>
<td>Repair</td>
<td>18</td>
<td>–</td>
<td>–</td>
<td>1</td>
</tr>
<tr>
<td>SequenceInputStream</td>
<td>Repair</td>
<td>4</td>
<td>–</td>
<td>–</td>
<td>1</td>
</tr>
<tr>
<td>DeflaterInputStream</td>
<td>Repair</td>
<td>380</td>
<td>–</td>
<td>–</td>
<td>1</td>
</tr>
<tr>
<td>BufferedReader</td>
<td>Synthesis</td>
<td>16</td>
<td>3119</td>
<td>4225</td>
<td>2</td>
</tr>
<tr>
<td>AudioClip (applet)</td>
<td>Synthesis</td>
<td>27</td>
<td>6808</td>
<td>9291</td>
<td>2</td>
</tr>
</tbody>
</table>


