Automated feedback generation for introductory programming assignments

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>As Published</td>
<td><a href="http://dx.doi.org/10.1145/2499370.2462195">http://dx.doi.org/10.1145/2499370.2462195</a></td>
</tr>
<tr>
<td>Publisher</td>
<td>Association for Computing Machinery</td>
</tr>
<tr>
<td>Version</td>
<td>Author’s final manuscript</td>
</tr>
<tr>
<td>Accessed</td>
<td>Wed Jan 02 14:31:00 EST 2019</td>
</tr>
<tr>
<td>Citable Link</td>
<td><a href="http://hdl.handle.net/1721.1/90851">http://hdl.handle.net/1721.1/90851</a></td>
</tr>
<tr>
<td>Terms of Use</td>
<td>Creative Commons Attribution-Noncommercial-Share Alike</td>
</tr>
<tr>
<td>Detailed Terms</td>
<td><a href="http://creativecommons.org/licenses/by-nc-sa/4.0/">http://creativecommons.org/licenses/by-nc-sa/4.0/</a></td>
</tr>
</tbody>
</table>
Automated Feedback Generation for Introductory Programming Assignments

Rishabh Singh
MIT CSAIL, Cambridge, MA
rishabh@csail.mit.edu

Sumit Gulwani
Microsoft Research, Redmond, WA
sumitg@microsoft.com

Armando Solar-Lezama
MIT CSAIL, Cambridge, MA
asolar@csail.mit.edu

Abstract
We present a new method for automatically providing feedback for introductory programming problems. In order to use this method, we need a reference implementation of the assignment, and an error model consisting of potential corrections to errors that students might make. Using this information, the system automatically derives minimal corrections to student’s incorrect solutions, providing them with a measure of exactly how incorrect a given solution was, as well as feedback about what they did wrong.

We introduce a simple language for describing error models in terms of correction rules, and formally define a rule-directed translation strategy that reduces the problem of finding minimal corrections in an incorrect program to the problem of synthesizing a correct program from a sketch. We have evaluated our system on thousands of real student attempts obtained from the Introduction to Programming course at MIT (6.00) and MITx (6.00x). Our results show that relatively simple error models can correct on average 64% of all incorrect submissions in our benchmark set.

Categories and Subject Descriptors D.1.2 [Programming Techniques]: Automatic Programming; I.2.2 [Artificial Intelligence]: Program Synthesis

Keywords Automated Grading; Computer-Aided Education; Program Synthesis

1. Introduction

There has been a lot of interest recently in making quality education more accessible to students worldwide using information technology. Several education initiatives such as EdX, Coursera, and Udacity are racing to provide online courses on various college-level subjects ranging from computer science to psychology. These courses, also called massive open online courses (MOOC), are typically taken by thousands of students worldwide, and present many interesting scalability challenges. Specifically, this paper addresses the challenge of providing personalized feedback for programming assignments in introductory programming courses.

The two methods most commonly used by MOOCs to provide feedback on programming problems are: (i) test-case based feedback and (ii) peer-feedback [12]. In test-case based feedback, the student program is run on a set of test cases and the failing test cases are reported back to the student. This is also how the 6.00x course (Introduction to Computer Science and Programming) offered by MITx currently provides feedback for Python programming exercises. The feedback of failing test cases is however not ideal; especially for beginner programmers who find it difficult to map the failing test cases to errors in their code. This is reflected by the number of students who post their submissions on the discussion board to seek help from instructors and other students after struggling for hours to correct the mistakes themselves. In fact, for the classroom version of the Introduction to Programming course (6.00) taught at MIT, the teaching assistants are required to manually go through each student submission and provide qualitative feedback describing exactly what is wrong with the submission and how to correct it. This manual feedback by teaching assistants is simply prohibitive for the number of students in the online class setting.

The second approach of peer-feedback is being suggested as a potential solution to this problem [43]. For example in 6.00x, students routinely answer each other’s questions on the discussion forums. This kind of peer-feedback is helpful, but it is not without problems. For example, we observed several instances where students had to wait for hours to get any feedback, and in some cases the feedback provided was too general or incomplete, and even wrong. Some courses have experimented with more sophisticated peer evaluation techniques [28] and there is an emerging research area that builds on recent results in crowd-powered systems [7, 30] to provide more structure and better incentives for improving the feedback quality. However, peer-feedback has some inherent limitations, such as the time it takes to receive quality feedback and the potential for inaccuracies in feedback, especially when a majority of the students are themselves struggling to learn the material.

In this paper, we present an automated technique to provide feedback for introductory programming assignments. The approach leverages program synthesis technology to automatically determine minimal fixes to the student’s solution that will make it match the behavior of a reference solution written by the instructor. This technology makes it possible to provide students with precise feedback about what they did wrong and how to correct their mistakes. The problem of providing automatic feedback appears to be related to the problem of automated bug fixing, but it differs from it in following two significant respects:

- The complete specification is known. An important challenge in automatic debugging is that there is no way to know whether a fix is addressing the root cause of a problem, or simply masking it and potentially introducing new errors. Usually the best one can do is check a candidate fix against a test suite or a partial specification [14]. While providing feedback on the other hand, the solution to the problem is known, and it is safe to assume that the instructor already wrote a correct reference implementation for the problem.
• **Errors are predictable.** In a homework assignment, everyone is solving the same problem after having attended the same lectures, so errors tend to follow predictable patterns. This makes it possible to use a model-based feedback approach, where the potential fixes are guided by a model of the kinds of errors students typically make for a given problem.

These simplifying assumptions, however, introduce their own set of challenges. For example, since the complete specification is known, the tool now needs to reason about the equivalence of the student solution with the reference implementation. Also, in order to take advantage of the predictability of errors, the tool needs to be parameterized with models that describe the classes of errors. And finally, these programs can be expected to have higher density of errors than production code, so techniques which attempt to correct bugs one path at a time [25] will not work for many of these problems that require coordinated fixes in multiple places.

Our feedback generation technique handles all of these challenges. The tool can reason about the semantic equivalence of student programs with reference implementations written in a fairly large subset of Python, so the instructor does not need to learn a new formalism to write specifications. The tool also provides an error model language that can be used to write an error model: a very high-level description of potential corrections to errors that students might make in the solution. When the system encounters an incorrect solution by a student, it symbolically explores the space of all possible combinations of corrections allowed by the error model and finds a correct solution requiring a minimal set of corrections.

We have evaluated our approach on thousands of student solutions on programming problems obtained from the 6.00x submissions and discussion boards, and from the 6.00 class submissions. These problems constitute a major portion of first month of assignments and discussion boards, and from the 6.00 class submissions. Our tool can successfully provide feedback on over 64% of the incorrect solutions.

This paper makes the following key contributions:

• We show that the problem of providing automated feedback for introductory programming assignments can be framed as a synthesis problem. Our reduction uses a constraint-based mechanism to model Python’s dynamic typing and supports complex Python constructs such as closures, higher-order functions, and list comprehensions.

• We define a high-level error model language EML that can be used to provide correction rules to be used for providing feedback. We also show that a small set of such rules is sufficient to correct the majority of incorrect solutions written by students.

• We report the successful evaluation of our technique on thousands of real student attempts obtained from 6.00 and 6.00x classes, as well as from PEX4FUN website. Our tool can provide feedback on 64% of all submitted solutions that are incorrect in about 10 seconds on average.

1. **Overview of the approach**

In order to illustrate the key ideas behind our approach, consider the problem of computing the derivative of a polynomial whose coefficients are represented as a list of integers. This problem is taken from week 3 problem set of 6.00x (PS3: Derivatives). Given the input list poly, the problem asks students to write the function `computeDeriv` that computes a list poly' such that

\[
\text{poly}' = \begin{cases} 
\{ i \times \text{poly}[i] \ | \ 0 < i < \text{len}(\text{poly}) \} & \text{if len(poly)} > 1 \\
\{ 0 \} & \text{if len(poly)} = 1
\end{cases}
\]

For example, if the input list poly is [2, −3, 1, 4] (denoting \( f(x) = 4x^3 + x^2 - 3x + 2 \)), the `computeDeriv` function should return \([-3, 2, 12]\) (denoting the derivative \( f'(x) = 12x^2 + 2x - 3 \)). The reference implementation for the `computeDeriv` function is shown in Figure 1. This problem teaches concepts of conditionals and iteration over lists. For this problem, students struggled with many low-level Python semantics issues such as the list indexing and iteration bounds. In addition, they also struggled with conceptual issues such as missing the corner case of handling lists consisting of single element (denoting constant function).

One challenge in providing feedback for student submissions is that a given problem can be solved by using many different algorithms. Figure 2 shows three very different student submissions for the `computeDeriv` problem, together with the feedback generated by our tool for each submission. The student submission shown in Figure 2(a) is taken from the 6.00x discussion forum. The student posted the code in the forum seeking help and received two responses. The first response asked the student to look for the first if-block return value, and the second response said that the code should return \([0]\) instead of empty list for the first if statement. There are many different ways to modify the code to return \([0]\) for the case \(\text{len}(\text{poly})=1\). The student chose to change the initialization of the `deriv` variable from \([1]\) to the list \([0]\). The problem with this modification is that the result will now have an additional 0 in front of the output list for all input lists (which is undesirable for lists of length greater than 1). The student then posted the query again on the forum on how to remove the leading 0 from result, but unfortunately this time did not get any more response.

Our tool generates the feedback shown in Figure 2(d) for the student program in about 40 seconds. During these 40 seconds, the tool searches over more than 10^7 candidate fixes and finds the fix that requires minimum number of corrections. There are three problems with the student code: first it should return \([0]\) in line 5 as was suggested in the forum but wasn’t specified how to make the change, second the if block should be removed in line 7, and third that the loop iteration should start from index 1 instead of 0 in line 6. The generated feedback consists of four pieces of information (shown in bold in the figure for emphasis):

- the location of the error denoted by the line number.
- the problematic expression in the line.
- the sub-expression which needs to be modified.
- the new modified value of the sub-expression.

The feedback generator is parameterized with a feedforward parameter to generate feedback consisting of different combinations of the four kinds of information, depending on how much information the instructor is willing to provide to the student.

1. **2.1 Workflow**

In order to provide the level of feedback described above, the tool needs some information from the instructor. First, the tool needs to know what the problem is that the students are supposed to solve. The instructor provides this information by writing a reference im-

```python
1 def computeDeriv_list_int(poly_list_int):
  result = []
  for i in range(len(poly_list_int)):
    result += [i * poly_list_int[i]]
  if len(poly_list_int) == 1:
    return result  # return [0]
  else:
    return result[1:]  # remove the leading 0
```

Figure 1. The reference implementation for `computeDeriv`. 

https://www.edx.org/courses/MITx/6.08x/2012.Fall/discussion/forum/600x.ps3.q2/threads/5085f3a27d1d422500000040
Three different student submissions for computeDeriv

```python
1 def computeDeriv(poly):
2     deriv = []
3     if len(poly) == 1:
4         return deriv
5     for e in range(0, len(poly)):
6         if poly[e] == 0:
7             deriv += [coeff * idx]
8         else:
9             deriv.append(poly[e]*e)
10    return deriv
```

(a)

```python
1 def computeDeriv(poly):
2     deriv = []
3     if len(poly) == 1:
4         return deriv
5     for e in range(0, len(poly)):
6         if poly[e] == 0:
7             deriv += [coeff * idx]
8         else:
9             deriv.append(poly[e]*e)
10    return deriv
```

(b)

```python
1 def computeDeriv(poly):
2     deriv = []
3     if len(poly) == 1:
4         return deriv
5     for e in range(0, len(poly)):
6         if poly[e] == 0:
7             deriv += [coeff * idx]
8         else:
9             deriv.append(poly[e]*e)
10    return deriv
```

(c)

Feedback generated by our Tool

The program requires 3 changes:

- In the return statement return deriv in line 5, replace deriv by [0].
- In the comparison expression (poly[e] == 0) in line 7, change (poly[e] == 0) to False.
- In the expression range(0, len(poly)) in line 6, increment 0 by 1.

(d)

The program requires 1 change:

- In the function computeDeriv, add the base case at the top to return [0] for len(poly)=1.

(e)

The program requires 2 changes:

- In the expression range(1, length) in line 4, increment length by 1.
- In the comparison expression (i >= 0) in line 8, change operator => to >=.

(f)

Figure 2. Three very different student submissions ((a), (b), and (c)) for the computeDeriv problem and the corresponding feedback generated by our tool ((d), (e), and (f)) for each one of them using the same reference implementation.

The program requires 3 changes:

- In the return statement return deriv in line 5, replace deriv by [0].
- In the comparison expression (poly[e] == 0) in line 7, change (poly[e] == 0) to False.
- In the expression range(0, len(poly)) in line 6, increment 0 by 1.

The program requires 1 change:

- In the function computeDeriv, add the base case at the top to return [0] for len(poly)=1.

The program requires 2 changes:

- In the expression range(1, length) in line 4, increment length by 1.
- In the comparison expression (i >= 0) in line 8, change operator => to >=.

Figure 3. The architecture of our feedback generation tool.

The architecture of our tool is shown in Figure 3. The solution strategy to find minimal corrections to a student’s solution is based on a two-phase translation to the Sketch synthesis language. In the first phase, the Program Rewriter uses the correction rules to translate the solution into a language we call MPY; this language provides us with a concise notation to describe sets of MPY candidate programs, together with a cost model to reflect the number of corrections associated with each program in this set. In the second phase, this MPY program is translated into a sketch program by the Sketch Translator.
Figure 4. The resulting MPY program after applying correction rules to program in Figure 2(a).

In the case of example in Figure 2(a), the Program Rewriter produces the MPY program shown in Figure 4 using the correction rules from Section 2.1. This program includes all the possible corrections induced by the correction rules in the model.

The MPY language extends the imperative language MP with expression choices, where the choices are denoted with square brackets. Whenever there are multiple choices for an expression or a statement, the zero-cost choice, the one that will leave the expression unchanged, is boxed. For example, the expression choice \([a_0, a_1, \ldots, a_n]\) denotes a choice between expressions \(a_0, \ldots, a_n\), where \(a_0\) denotes the zero-cost default choice.

For this simple program, the three correction rules induce a space of 32 different candidate programs. This candidate space is fairly small, but the number of candidate programs grows exponentially with the number of correction places in the program and with the number of correction choices in the rules. The error model that we use in our experiments induces a space of more than \(10^{12}\) candidate programs for some of the benchmark problems. In order to search this large space efficiently, the program is translated to a sketch by the Sketch Translator.

2.3 Synthesizing Corrections with Sketch

The Sketch [37] synthesis system allows programmers to write programs while leaving fragments of it unspecified as holes; the contents of these holes are filled up automatically by the synthesizer such that the program conforms to a specification provided in terms of a reference implementation. The synthesizer uses the CEGIS algorithm [38] to efficiently compute the values for holes and uses bounded symbolic verification techniques for performing equivalence check of the two implementations.

There are two key aspects in the translation of an MPY program to a Sketch program. The first aspect is specific to the Python language. Sketch supports high-level features such as closures and higher-order functions which simplifies the translation, but it is statically typed whereas MPY programs (like Python) are dynamically typed. The translation models the dynamically typed variables and operations over them using struct types in Sketch in a way similar to the union types. The second aspect of the translation is the modeling of set-expressions in MPY using ?? (holes) in Sketch, which is language independent.

The dynamic variable types in the MPY language are modeled using the MultiType struct defined in Figure 5. The MultiType struct consists of a flag field that denotes the dynamic type of a variable and currently supports the following set of types: [INTEGER, BOOL, TYPE, LIST, TUPLE, STRING, DICTIONARY]. The val and bval fields store the value of an integer and a Boolean variable respectively, whereas the str, tup, dict, and lst fields store the value of string, tuple, dictionary, and list variables respectively. The MultiType struct consists of a field len that denotes the length of the list and a field lVals of type array of MultiType that stores the list elements. For example, the integer value 5 is represented as the value MultiType(val=5, flag=INTEGER) and the list [1,2] is represented as the value MultiType(lst=new MultiType(val=1, flag=INTEGER), new MultiType(val=2, flag=INTEGER)), flag=LIST).

The second key aspect of this translation is the translation of expression choices in MPY. The Sketch construct ?? denotes an unknown integer hole that can be assigned any constant integer value by the synthesizer. The expression choices in MPY are translated to functions in Sketch that based on the unknown hole values return either the default expression or one of the other expression choices. Each such function is associated with a unique Boolean choice variable, which is set by the function whenever it returns a non-default expression choice. For example, the set-statement return \([\text{deriv}[0]]\); (line 5 in Figure 4) is translated to return modRetVal0(deriv), where the modRetVal0 function is defined as:

```plaintext
struct MultiType{
    int val, flag;
    struct MTList{
        bit bval;
        MTString str; MTuple tup;
        MultiType[len] lVals;
        MTDict dict; MTList lst;
    }
}
```

Figure 5. The MultiType struct for encoding Python types.

The translation phase also generates a Sketch harness that compares the outputs of the translated student and reference implementations on all inputs of a bounded size. For example in case of the computeDeriv function, with bounds of \(n = 4\) for both the number of integer bits and the maximum length of input list, the harness matches the output of the two implementations for more than \(10^6\) different input values as opposed to 10 test-cases used in 6.00x. The harness also defines a variable totalCost as a function of choice variables that computes the total number of corrections performed in the original program, and asserts that the value of totalCost should be minimized. The synthesizer then solves this minimization problem efficiently using an incremental solving algorithm CEGISMIN described in Section 4.2.

After the synthesizer finds a solution, the Feedback Generator extracts the choices made by the synthesizer and uses them to generate the corresponding feedback in natural language. For this example, the tool generates the feedback shown in Figure 2(d) in less than 40 seconds.

3. EMIL: Error Model Language

In this section, we describe the syntax and semantics of the error model language EMIL. An EMIL error model consists of a set of rewrite rules that captures the potential corrections for mistakes that students might make in their solutions. We define the rewrite rules over a simple Python-like imperative language MPY. A rewrite rule transforms a program element in MPY to a set of weighted MPY program elements. This weighted set of MPY program elements is
represented succinctly as an \( \tilde{M} \) program element, where \( \tilde{M} \) extends the MPy language with set-exps (sets of expressions) and set-stmts (sets of statements). The weight associated with a program element in this set denotes the cost of performing the corresponding correction. An error model transforms an MPy program to an MPy program by recursively applying the rewrite rules. We show that this transformation is deterministic and is guaranteed to terminate on well-formed error models.

3.1 MPY and MPy languages

The syntax of MPY and MPy languages is shown in Figure 6(a) and Figure 6(b) respectively. The purpose of MPy language is to represent a large collection of MPy programs succinctly. The MPy language consists of set-expressions (\( \tilde{a} \) and \( \tilde{b} \)) and set-statements (\( \tilde{s} \)) that represent a weighted set of corresponding MPy expressions and statements respectively. For example, the set expression \( \{n_0, \ldots, n_k\} \) represents a weighted set of constant integers where \( n_0 \) denotes the default integer value associated with cost 0 and all other integer constants \( (n_1, \ldots, n_k) \) are associated with cost 1. The sets of composite expressions are represented succinctly in terms of sets of their constituent sub-expressions. For example, the composite expression \( \{(a_0, a_0 + 1) \mid \langle <, >, \langle, \rangle, \leq, \geq, =, =\rangle\} \{(a_1, a_1 + 1, a_1 - 1)\} \) represents 36 MPy expressions.

Each MPy program in the set of programs represented by an MPy program is associated with a cost (weight) that denotes the number of modifications performed in the original program to obtain the transformed program. This cost allows the tool to search for corrections that require minimum number of modifications. The weighted set of MPy programs is defined using the \( \tilde{M} \) function shown partially in Figure 7, the complete function definition can be found in [36]. The \( \tilde{M} \) function on MPy expressions such as \( a \) returns a singleton set \( \{(a, 0)\} \) consisting of the corresponding expression associated with cost 0. On set-expressions of the form \( [\tilde{a}_0, \ldots, \tilde{a}_n] \), the function returns the union of the weighted set of MPy expressions corresponding to the default set-expression \( [n_0] \) and the weighted set of expressions corresponding to other set-expressions \( (\tilde{a}_0, \ldots, \tilde{a}_n) \), where each expression in \( [\tilde{a}_i] \) is associated with an additional cost of 1. On composite expressions, the function computes the weighted set recursively by taking the cross-product of weighted sets of its constituent sub-expressions and adding their corresponding costs. For example, the weighted set for composite expression \( \tilde{x}[\tilde{y}] \) consists of an expression \( x_i[y_j] \) associated with cost \( c_{x_i} + c_{y_j} \) for each \( (x_i, c_{x_i}) \in [\tilde{x}] \) and \( (y_j, c_{y_j}) \in [\tilde{y}] \).

3.2 Syntax of EML

An EML error model consists of a set of correction rules that are used to transform an MPy program to an MPy program. A correction rule \( C \) is written as a rewrite rule \( L \rightarrow R \), where \( L \) and \( R \) denote a program element in MPy and MPy respectively. A program element can either be a term, an expression, a statement, a method or the program itself. The left hand side \( L \) denotes an MPy program element that is pattern matched to be transformed to an MPy program element denoted by the right hand side \( R \). The left hand side of the rule can use free variables whereas the right hand side can only refer to the variables present in the left hand side. The language also supports a special prime operator that can be used to tag sub-expressions in \( R \) that are further transformed recursively using the error model. The rules use a shorthand notation \( a \) in the right hand side to denote the set of all variables that are of the same type as the type of expression \( a \) and are in scope at the corresponding program location. We assume each correction rule is associated with cost 1, but it can be easily extended to different costs to account for different severity of mistakes.

Example 1. The error model for the computeDeriv problem is shown in Figure 8. The INDR rewrite rule transforms the list access indices. The INIT rewrite rule transforms the right hand side of constant initializations. The RAN rewrite rules transform the arguments for the range function; similar rules are defined in the model for other range functions that take one and three arguments. The COMPR rule transforms the operands and operator of the comparisons. The RETR rule adds the two common corner cases of returning 0 when the length of input list is 1, and the case of deleting the first list element before returning the list. Note that these rewrite rules define the corrections that can be performed optionally; the zero cost (default) case of not correcting a program element is added automatically as described in Section 3.3.

Definition 1. Well-formed Rewrite Rule : A rewrite rule \( R : L \rightarrow R \) is defined to be well-formed if all tagged sub-terms \( v \) in \( R \) have a smaller syntax tree than that of \( L \).

The rewrite rule \( C_1 : v[a] \rightarrow (v[a])' + 1 \) is not a well-formed rewrite rule as the size of the tagged sub-term \( v[a] \) of \( R \) is the same as that of the left hand side \( L \). On the other hand, the rewrite rule \( C_2 : v[a] \rightarrow (v[a']) + 1 \) is well-formed.

Definition 2. Well-formed Error Model : An error model \( E \) is defined to be well-formed if all of its constituent rewrite rules \( C_i \in E \) are well-formed.

3.3 Transformation with EML

An error model \( E \) is syntactically translated to a function \( T_E \) that transforms an MPy program to an MPy program. The \( T_E \) function first traverses the program element \( w \) in the default way, i.e. no transformation happens at this level of the syntax tree, and the function is called recursively on all of its top-level sub-terms \( t \) to obtain the transformed element \( w_0 \in \text{MPy} \). For each correction rule \( C_i : L_i \rightarrow R_i \) in the error model \( E \), the function contains a...
Arith Expr \( a \) := \( n \mid [] \mid v \mid a[a] \mid a_0 \circ p_c \ a_1 \mid \{a_1, \ldots, a_n\} \mid f(a_0, \ldots, a_n) \mid a_0 \text{ if } b \text{ else } a_1 \)

Arith Op \( op_a \) := \(+\mid -\mid \times \mid \div \mid \ast\)

Bool Op \( op_c \) := \( \& \mid \mid \mid \lnot \mid \ast \)

Comp Op \( op_p \) := \( \Rightarrow \mid | \mid \circ \mid \ast \circ \mid \ast \ast \mid \ast \circ \ast \mid \ast \ast \circ \ast \mid \ast \ast \ast \ast \)

Stmt Expr \( s \) := \( v = a \mid s_0 ; s_1 \mid \text{ while } b : s \mid \text{ if } b : s_0 \text{ else } s_1 \mid \text{ for } a_0 \text{ in } a_1 : s \mid \text{ return } a \)

Func Def. \( \text{ p} \) := \( \text{ def } f(a_1, \ldots, a_n) : s \)

(a) \( \text{ MPY} \)

Arith set-exp \( \tilde{a} \) := \( a \mid \{\tilde{a}_0, \ldots, \tilde{a}_n\} \mid \tilde{a}[a] \mid \tilde{a}_0 \circ p_c \tilde{a}_1 \mid \{\tilde{a}_0, \ldots, \tilde{a}_n\} \mid f(\tilde{a}_0, \ldots, \tilde{a}_n) \mid \tilde{a}_0 \circ p_c \tilde{a}_1 \mid \tilde{b}_0 \circ p_c \tilde{b}_1 \)

Bool set-exp \( \tilde{b} \) := \( b \mid \{\tilde{b}_0, \ldots, \tilde{b}_n\} \mid \lnot \tilde{b} \mid \tilde{a}_0 \circ p_c \tilde{a}_1 \mid \tilde{b}_0 \circ p_c \tilde{b}_1 \)

Stmt set-exp \( \tilde{s} \) := \( s \mid \{\tilde{s}_0, \ldots, \tilde{s}_n\} \mid \tilde{v} : = \tilde{a} \mid \tilde{s}_0 ; \tilde{s}_1 \mid \text{ while } \tilde{b} : \tilde{s} \mid \text{ for } \tilde{a}_0 \text{ in } \tilde{a}_1 : \tilde{s} \mid \text{ if } \tilde{b} : \tilde{s}_0 \text{ else } \tilde{s}_1 \mid \text{ return } \tilde{a} \)

Func Def \( \tilde{\text{ p}} \) := \( \text{ def } f(a_1, \ldots, a_n) \tilde{s} \)

(b) \( \text{ MPY} \)

Match expression that matches the term \( w \) with the left hand side of the rule \( L_i \) (with appropriate unification of the free variables in \( L_i \)). If the match succeeds, it is transformed to a term \( w_i \) in \( \text{ MPY} \) as defined by the right hand side \( R_i \) of the rule after calling the \( T_E \) function on each of its tagged sub-terms \( t' \). Finally, the method returns the set of all transformed terms \( \{w_0, \ldots, w_n\} \).

\textbf{Example 2.} Consider an error model \( E_1 \) consisting of the following three correction rules:

\begin{align*}
C_1 : & \quad v[a] \rightarrow v\{a-1, a+1\} \\
C_2 : & \quad a_0 \circ p_c \ a_1 \rightarrow \{a_0 - 1, 0\} \circ p_c \{a_1 - 1, 0\} \\
C_3 : & \quad v[a] \rightarrow ?v[a]
\end{align*}

The transformation function \( T_{E_1} \) for the error model \( E_1 \) is shown in Figure 9.

\[ T_{E_1}(w : \text{ MPY}) : \overline{\text{MPY}} = \]

\textbf{Figure 6.} The syntax for (a) \( \text{ MPY} \) and (b) \( \overline{\text{MPY}} \) languages.

\textbf{4. Constraint-based Solving of \( \overline{\text{MPY}} \) programs}

In the previous section, we saw the transformation of an \( \text{ MPY} \) program into an error model. We now present the translation of an \( \text{ MPY} \) program into a \textit{SKETCH} program [37].

\textbf{4.1 Translation of \( \text{ MPY} \) programs to \textit{SKETCH}}

The \( \overline{\text{MPY}} \) programs are translated to \textit{SKETCH} programs for efficient constraint-based solving for minimal corrections to the student solutions. The two main aspects of the translation include: (i) the translation of Python-like constructs in \( \overline{\text{MPY}} \) to \textit{SKETCH}, and (ii) the translation of set-exp choices in \( \overline{\text{MPY}} \) to \textit{SKETCH} functions.
\[ T(x[i] < y[j]) \equiv \{ T(x[i]) < T(y[j]) \}
\]

\[ T(x[i]) \equiv \{ T(x[i]) \} x[i+1, i-1], y[i]\]

\[ T(y[j]) \equiv \{ T(y[j]) \} y[j+1, j-1], x[j]\]

\[ T(x) \equiv \{ x \} \quad T(i) \equiv \{ i \} \quad T(y) \equiv \{ y \} \quad T(j) \equiv \{ j \} \]

Therefore, after substitution the result is:

\[ T(x[i] < y[j]) \equiv \{ x[i+1, i-1], y[i] \} < \{ y[j+1, j-1], x[j] \}
\]

\[ \{ x[i+1, i-1], y[i] \} < \{ y[j+1, j-1], x[j] \} - 1,0 \}
\]

Figure 10. Application of \( T_\varepsilon \) (abbreviated \( T \)) on expression \( [x[i] < y[j]] \).

**Handling dynamic typing of \( \tilde{M}P \) variables** The dynamic typing in \( \tilde{M}P \) is handled using a MultiType variable as described in Section 2.3. The \( \tilde{M}P \) expressions and statements are transformed to SKETCH functions that perform the corresponding transformations over MultiType. For example, the Python statement \((a = b)\) is translated to \( \text{assignMT}(a, b) \), where the \( \text{assignMT} \) function assigns MultiType \( b \) to \( a \). Similarly, the binary add expression \((a + b)\) is translated to \( \text{binopMT}(a, b, \text{ADD_OP}) \) that in turn calls the function \( \text{addMT}(a, b) \) to add \( a \) and \( b \) as shown in Figure 11.

```python
1 | MultiType addMT(MultiType a, MultiType b){
2 | assert a.flag == b.flag; // same types can be added
3 | if(a.flag == INTEGER) // add for integers
4 | return new MultiType(val=a.val+b.val, flag = INTEGER);
5 | if(a.flag == LIST){ // add for lists
6 | int newLen = a.list.len + b.list.len;
7 | MultiType[newLen] newLVals = a.list.LVals;
8 | for(int i=0; i<b.list.len; i++)
9 | newLVals[i+a.list.len] = b.list.LVals[i];
10 | return new MultiType(list = new MList(LVals=newLVals, len=newLen),
11 | flag=LIST);}
12 |}
```

Figure 11. The \( \text{addMT} \) function for adding two MultiType \( a \) and \( b \).

**Translation of \( \tilde{M}P \) set-expressions** The set-expressions in \( \tilde{M}P \) are translated to functions in SKETCH. The function bodies obtained by the application of translation function \( \Phi \) on some of the interesting \( \tilde{M}P \) constructs are shown in Figure 12. The SKETCH construct \( ?? \) (called \( \text{hole} \)) is a placeholder for a constant value, which is filled up by the SKETCH synthesizer while solving the constraints to satisfy the given specification.

The singleton sets consisting of an \( \tilde{M}P \) expression such as \( \{a\} \) are translated simply to the corresponding expression itself. A set-expression of the form \( \{\tilde{a}_0, \tilde{a}_1, \ldots, \tilde{a}_n\} \) is translated recursively to the if expression \( \text{if}(??) \Phi(\tilde{a}_0) \text{ else } \Phi(\{\tilde{a}_1, \ldots, \tilde{a}_n\}) \), which means that the synthesizer can optionally select the default set-expression \( \Phi(\tilde{a}_0) \) (by choosing ?? to be true) or select one of the other choices \( \{\tilde{a}_1, \ldots, \tilde{a}_n\} \). The set-expressions of the form

\[ \Phi(\{\tilde{a}_0\}) = \ a \\
\Phi(\{\tilde{a}_0, \ldots, \tilde{a}_n\}) = \text{if}(??) \Phi(\tilde{a}_0) \text{ else } \Phi(\{\tilde{a}_1, \ldots, \tilde{a}_n\}) \\
\Phi(\{\tilde{a}_0, \ldots, \tilde{a}_n\}) = \text{if}(??) \text{ choice}_k = \text{true} \text{ then } \Phi(\tilde{a}_0) \text{ else } \Phi(\{\tilde{a}_1, \ldots, \tilde{a}_n\}) \\
\Phi(\tilde{a}_0) = \tilde{a}_1 \Rightarrow \Phi(\tilde{a}_0) := \Phi(\tilde{a}_1) \\
\{\tilde{a}_0, \ldots, \tilde{a}_n\} \} \] are similarly translated but with an additional statement for setting a fresh variable \( \text{choice}_k \) if the synthesizer selects the non-default choice \( \tilde{a}_0 \).

The translation rules for the assignment statements \( \tilde{a}_0 := \tilde{a}_1 \) results in if expressions on both left and right sides of the assignment. The \( \text{if} \) expression choices occurring on the left hand side are desugared to individual assignments. For example, the left hand side expression \( \text{if}(??) x \text{ else } y := 10 \) is desugared to \( \text{if}(??) x := 10 \text{ else } y := 10 \). The infix operators in \( \tilde{M}P \) are first translated to function calls and are then translated to sketch using the translation for set-function expressions. The remaining \( \tilde{M}P \) expressions are similarly translated recursively and the translation can be found in more detail in [36].

**Translating function calls** The translation of function calls for recursive problems and for problems that require writing a function that uses other sub-functions is parameterized by three options: 1) use the student’s implementation of sub-functions, 2) use the teacher’s implementation of sub-functions, and 3) treat the sub-functions as uninterpreted functions.

**Generating the driver functions** The SKETCH synthesizer supports the equivalence checking of functions whose input arguments and return values are over \( \tilde{S}KETCH \) primitive types such as \( \text{int} \), \( \text{bit} \) and \( \text{arrays} \). Therefore, after the translation of \( \tilde{M}P \) programs to \( \tilde{S}KETCH \) programs, we need additional driver functions to integrate the functions over MultiType input arguments and return value to the corresponding functions over \( \tilde{S}KETCH \) primitive types. The driver functions first converts the input arguments over primitive types to corresponding MultiType variables using library functions.
such as computeMTFromInt, and then calls the translated MPY function with the MultiType variables. The returned MultiType value is translated back to primitive types using library functions such as computeIntFromMT. The driver function for student’s programs also consists of additional statements of the form if(choicek) totalCost++; and the statement minimize(totalCost), which tells the synthesizer to compute a solution to the Boolean variables choicek that minimizes the total cost variable.

4.2 CEGISMIN: Incremental Solving for the Minimize holes

Algorithm 1 CEGISMIN Algorithm for Minimize expression

1: σ0 ← σrandom
2: while (True)
3: i ← i + 1
4: Φi ← Synth(σi−1, Φi−1) \( \triangleright \) Synthesis Phase
5: if (Φi = UNSAT) \( \triangleright \) Synthesis Fails
6: if (Ψprev = null) return UNSAT\_SKETCH
7: else return PE(Ψp, Φi)
8: choose \( \phi \in \Phi_i \)
9: \( \sigma_i \leftarrow \text{Verify}(\phi) \) \( \triangleright \) Verification Phase
10: if (\( \sigma_i = \text{null} \)) \( \triangleright \) Verification Succeeds
11: \( (\text{minHole, minHoleValue} \leftarrow \text{getMinHoleValue}(\phi)) \)
12: \( \Phi_i \leftarrow \Phi_i \cup \{\text{encode(minHole} < \text{minHoleVal}\}) \)

We extend the CEGIS algorithm in SKETCH [37] to obtain the CEGISMIN algorithm shown in Algorithm 1 for efficiently solving sketches that include a minimize hole expression. The input state of the sketch program is denoted by \( \sigma \) and the sketch constraint store is denoted by \( \Phi \). Initially, the input state \( \sigma_0 \) is assigned a random input state value and the constraint store \( \Phi_0 \) is assigned the constraint set obtained from the sketch program. The variable \( \Phi_p \) stores the previous satisfiable hole values and is initialized to null. In each iteration of the loop, the synthesizer first performs the inductive synthesis phase where it shrinks the constraints set \( \Phi_{i-1} \) to \( \Phi_i \) by removing behaviors from \( \Phi_{i-1} \) that do not conform to the input state \( \sigma_{i-1} \). If the constraint set becomes unsatisfiable, it either returns the sketch completed with hole values from the previous solution if one exists, otherwise it returns UNSAT. On the other hand, if the constraint set is satisfiable, then it first chooses a conforming assignment to the hole values and goes into the verification phase where it tries to verify the completed sketch. If the verifier fails, it returns a counter-example input state \( \sigma_i \) and the synthesis-verification loop is repeated. If the verification phase succeeds, instead of returning the result as is done in the CEGIS algorithm, the CEGISMIN algorithm computes the value of minHole from the constraint set \( \phi \), stores the current satisfiable hole solution \( \phi \) in \( \Phi_p \), and adds an additional constraint \( \{\text{minHole} \leq \text{minHoleVal}\} \) to the constraint set \( \Phi_i \). The synthesis-verification loop is then repeated with this additional constraint to find a conforming value for the minHole variable that is smaller than the current value in \( \phi \).

4.3 Mapping SKETCH solution to generate feedback

Each correction rule in the error model is associated with a feedback message, e.g., the correction rule for variable initialization \( v = n \rightarrow v = \{n + 1\} \) in the computeDeriv error model is associated with the message “Increment the right hand side of the initialization by 1”. After the SKETCH synthesizer finds a solution to the constraints, the tool maps back the values of unknown integer holes to their corresponding expression choices. These expression choices are then mapped to natural language feedback using the messages associated with the corresponding correction rules, together with the line numbers. If the synthesizer returns UNSAT, the tool reports that the student solution cannot be fixed.

5. Implementation and Experiments

We now briefly describe some of the implementation details of the tool, and then describe the experiments we performed to evaluate our tool over the benchmark problems.

5.1 Implementation

The tool’s frontend is implemented in Python itself and uses the Python ast module to convert a Python program to a SKETCH program. The backend system that solves the sketch is implemented as a wrapper over the SKETCH system that is extended with the CEGISMIN algorithm. The feedback generator, implemented in Python, parses the output generated by the backend system and translates it to corresponding high level feedback in natural language. Error models in our tool are currently written in terms of rewrite rules over the Python AST. In addition to the Python tool, we also have a prototype for the C# language, which we built on top of the Microsoft Roslyn compiler framework. The C# prototype supports a smaller subset of the language relative to the Python tool but nevertheless it was useful in helping us evaluate the potential of our technique on a different language.

5.2 Benchmarks

We created our benchmark set with problems taken from the Introduction to Programming course at MIT (6.00) and the EdX version of the class (6.00x) offered in 2012. Our benchmark set includes most problems from the first four weeks of the course. We only excluded (i) a problem that required more detailed floating point reasoning than what we currently handle, (ii) a problem that required file i/o which we currently do not model, and (iii) a handful of trivial finger exercises. To evaluate the applicability to C#, we created a few programming exercises on PEX4FUN that were based on loop-over-arrays and dynamic programming from an AP level exam. A brief description of each benchmark problem follows:

- prodBySum-6.00: Compute the product of two numbers \( m \) and \( n \) using only the sum operator.
- oddTuples-6.00: Given a tuple \( t \), return a tuple consisting of every other element of \( t \).
- compDeriv-6.00: Given a tuple \( t \), return a tuple consisting of every other element of \( t \).
- evalPoly-6.00: Compute the value of a polynomial (represented as a list).
- evalPoly-6.00x: Compute the value of a polynomial (represented as a list) at a given value \( x \).
- compBal-stdin-6.00: Print the values of monthly installment necessary to purchase a car in one year, where the inputs car price and interest rate compounded monthly are provided from stdin.
- compDeriv-6.00x: compDeriv problem from the EdX class.
- evalPoly-6.00x: evalPoly problem from the EdX class.
- oddTuples-6.00x: oddTuples problem from the EdX class.
- iterPower-6.00x: Compute the value \( m^n \) using only the multiplication operator, where \( m \) and \( n \) are integers.
- recurPower-6.00x: Compute the value \( m^n \) using recursion.
- iterGCD-6.00x: Compute the greatest common divisor (gcd) of two integers \( m \) and \( n \) using an iterative algorithm.
- hangman1-str-6.00x: Given a string secretWord and a list of guessed letters lettersGuessed, return True if all letters of secretWord are in lettersGuessed, and False otherwise.

\[http://pexforfun.com/learnbeginningprogramming\]

\[^2\] AP exams allow high school students in the US to earn college level credit.
• hangman2-str-6.00x: Given a string secretWord and a list of
guessed letters lettersGuessed, return a string where all letters
of secretWord that have not been guessed yet (i.e. not present
in lettersGuessed) are replaced by the letter ‘.’.
• stock-market-I(#): Given a list of stock prices, check if the
stock is stable, i.e. if the price of stock has changed by more
than $10 in consecutive days on less than 3 occasions over the
duration.
• stock-market-II(#): Given a list of stock prices and a start
and end day, check if the maximum and minimum stock prices
over the duration from start and end day is less than $20.
• restaurant rush (#): A variant of maximum contiguous
subset sum problem.

5.3 Experiments
We now present various experiments we performed to evaluate our
tool on the benchmark problems.

Performance Table 1 shows the number of student attempts cor-
corrected for each benchmark problem as well as the time taken by
the tool to provide the feedback. The experiments were performed
on a 2.4GHz Intel Xeon CPU with 16 cores and 16GB RAM. The
experiments were performed with bounds of 4 bits for input integer
values and maximum length 4 for input lists. For each benchmark
problem, we first removed the student attempts with syntax errors
to get the Test Set on which we ran our tool. We then separated the
attempts which were correct to measure the effectiveness of the tool
on the incorrect attempts. The tool was able to provide appropriate
corrections as feedback for 64% of all incorrect student attempts
in around 10 seconds on average. The remaining 36% of incorrect
student attempts on which the tool could not provide feedback fall
in one of the following categories:

• Completely incorrect solutions: We observed many student
attempts that were empty or performing trivial computations
such as printing strings and variables.
• Big conceptual errors: A common error we found in the case
of eval-poly-6.00x was that a large fraction of incorrect at-
ttempts (260/541) were using the list function index to get the
index of a value in the list (e.g. see Figure 13(a)), whereas
the index function returns the index of first occurrence of the
value in the list. Another example of this class of error for the
hangman2-str problem in shown in Figure 13(b), where the so-
lution replaces the guessed letters in the secretWord by ‘
’ instead of replacing the letters that are not yet guessed.
The correction of some other errors in this class involves introducing
new program statements or moving statements from one pro-
gram location to another. These errors can not be corrected with
the application of a set of local correction rules.
• Unimplemented features: Our implementation currently lacks
a few of the complex Python features such as pattern matching
on list enumerate function and lambda functions.
• Timeout: In our experiments, we found less than 5% of the
student attempts timed out (set as 4 minutes).

Number of Corrections The number of student submissions that
require different number of corrections are shown in Figure 14(a)
(on a logarithmic scale). We observe from the figure that a signif-
ificant fraction of the problems require 3 and 4 coordinated correc-
tions, and to provide feedback on such attempts, we need a technol-
gy like ours that can symbolically encode the outcome of different
corrections on all input values.

Repetitive Mistakes In this experiment, we check our hypothesis
that students make similar mistakes while solving a given problem.
The graph in Figure 14(b) shows the number of student attempts
corrected as more rules are added to the error models of the bench-
mark problems. As can be seen from the figure, adding a single rule
to the error model can lead to correction of hundreds of attempts.
This validates our hypothesis that different students indeed make
similar mistakes when solving a given problem.

Generalization of Error Models In this experiment, we check
the hypothesis that the correction rules generalize across problems
of similar kind. The result of running the compute-deriv error
model on other benchmark problems is shown in Figure 14(c). As
expected, it does not perform as well as the problem-specific error
models, but it still fixes a fraction of the incorrect attempts and
can be useful as a good starting point to specialize the error model
further by adding more problem-specific rules.

6. Capabilities and Limitations
Our tool supports a fairly large subset of Python types and language
features, and can currently provide feedback on a large fraction
(64%) of student submissions in our benchmark set. In compar-
ison to the traditional test-cases based feedback techniques that test
the programs over a few dozens of test-cases, our tool typically per-
forms the equivalence check over more than 10^6 inputs. Programs
that print the output to console (e.g. compBal-stdin) pose an inter-
esting challenge for test-cases based feedback tools. Since beginner
students typically print some extra text and values in addition to the
desired outputs, the traditional tools need to employ various heuris-
tics to discard some of the output text to match the desired output.
Our tool lets instructors provide correction rules that can optionally
drop some of the print expressions in the program, and then the tool
finds the required print expressions to eliminate so that a student is
not penalized much for printing additional values.

Now we briefly describe some of the limitations of our tool.
One limitation of the tool is in providing feedback on student
attempts that have big conceptual errors (see Section 5.3), which
can not be fixed by application of a set of local rewrite rules.
Correcting such programs typically requires a large global rewrite
of the student solution, and providing feedback in such cases is
an open question. Another limitation of our tool is that it does not
take into account structural requirements in the problem statement
since it focuses only on functional equivalence. For example, some
of the assignments explicitly ask students to use bisection search or
recursion, but our tool can not distinguish between two functionally
equivalent solutions, e.g. it can not distinguish between a bubble
sort and a merge sort implementation of the sorting problem.

For some problems, the feedback generated by the tool is too
low-level. For example, a suggestion provided by the tool in Figure
2(d) is to replace the expression poly[e]=0 by False, whereas
a higher level feedback would be a suggestion to remove the cor-
responding block inside the comparison. Deriving the high-level
feedback from the low-level suggestions is mostly an engineering
problem as it requires specializing the message based on the con-
text of the correction.

The scalability of the technique also presents a limitation. For
some problems that use large constant values, the tool currently
replaces them with smaller teacher-provided constant values such
that the correct program behavior is maintained. We also currently
need to specify bounds for the input size, the number of loop un-
rollings and recursion depth as well as manually provide special-
ized error models for each problem. The problem of discovering
these optimizations automatically by mining them from the large
corpus of datasets is also an interesting research question. Our tool
also currently does not support some of the Python language fea-
tures, most notably classes and objects, which are required for pro-
viding feedback on problems from later weeks of the class.
7. Related Work

In this section, we describe several related work to our technique from the areas of automated programming tutors, automated program repair, fault localization, automated debugging, automated grading, and program synthesis.

7.1 AI based programming tutors

There has been a lot of work done in the AI community for building automated tutors for helping novice programmers learn programming by providing feedback about semantic errors. These tutoring systems can be categorized into the following two major classes:

**Code-based matching approaches**: LAURA [1] converts teacher’s and student’s program into a graph based representation by providing feedback about semantic errors. These tutoring systems return feedback about semantic errors. These tutoring systems can be categorized into the following two major classes:

**Intention-based matching approaches**: LISP tutor [13] creates a model of the student goals and updates it dynamically as the student makes edits. The drawback of this approach is that it forces students to write code in a certain pre-defined structure and limits comparisons heuristically by applying program transformations while reporting mismatches as potential bugs. TALUS [31] matches a student’s attempt with a collection of teacher’s algorithms. It first tries to recognize the algorithm used and then tentatively replaces the top-level expressions in the student’s attempt with the recognized algorithm for generating correction feedback. The problem with these approach is that the enumeration of all possible algorithms (with its variants) for covering all corrections is very large and tedious on part of the teacher.
their freedom. MENO-II [39] parses student programs into a deep syntax tree whose nodes are annotated with plan tags. This annotated tree is then matched with the plans obtained from teacher’s solution. PROUST [24], on the other hand, uses a knowledge base of goals and their corresponding plans for implementing them for each programming problem. It first tries to find correspondence of these plans in the student’s code and then performs matching to find discrepancies. CHIRON [32] is its improved version in which the goals and plans in the knowledge base are organized in a hierarchical manner based on their generality and uses machine learning techniques for plan identification in the student code. These approaches require teacher to provide all possible plans a student can use to solve the goals of a given problem and do not perform well if the student’s attempt uses a plan not present in the knowledge base.

Our approach performs semantic equivalence of student’s attempt and teacher’s solution based on exhaustive bounded symbolic verification techniques and makes no assumptions on the algorithms or plans that students can use for solving the problem. Moreover, our approach is modular with respect to error models; the local correction rules are provided in a declarative manner and their complex interactions are handled by the solver itself.

7.2 Automated Program Repair
Könighofer et. al. [27] present an approach for automated error localization and correction of imperative programs. They use model-based diagnosis to localize components that need to be replaced and then use a template-based approach for providing corrections using SMT reasoning. Their fault model only considers the right hand side (RHS) of assignment statements as replaceable components. The approaches in [23, 41] frame the problem of program repair as a game between an environment that provides the inputs and a system that provides correct values for the buggy expressions such that the specification is satisfied. These approaches only support simple corrections (e.g. correcting RHS side of expressions) in the fault model as they aim to repair large programs with arbitrary errors. In our setting, we exploit the fact that we have access to the dataset of previous student mistakes that we can use to solve the problem. Moreover, our approach is modular with respect to error models; the local correction rules are provided in a declarative manner and their complex interactions are handled by the solver itself.

7.3 Automated Debugging and Fault localization
Techniques like Delta Debugging [44] and QuickXplain [26] aim to simplify a failing test case to a minimal test case that still exhibits the same failure. Our approach can be complemented with these techniques to restrict the application of rewrite rules to certain failing parts of the program only. There are many algorithms for fault localization [6, 15] that use the difference between faulty and successful executions of the system to identify potential faulty locations. Jose et. al. [25] recently suggested an approach that uses a MAX-SAT solver to satisfy maximum number of clauses in a formula obtained from a failing test case to compute potential error locations. These approaches, however, only localize faults for a single failing test case and the suggested error location might not be the desired error location, since we are looking for common error locations that cause failure of multiple test cases. Moreover, these techniques provide only a limited set of suggestions (if any) for repairing these faults.

7.4 Computer-aided Education
We believe that formal methods technology can play a big role in revolutionizing education. Recently, it has been applied to multiple aspects of Education including problem generation [2, 4, 35] and solution generation [17]. In this paper, we push the frontier forward to cover another aspect namely automated grading. Recently [3] also applied automated grading to automata constructions and used syntactic edit distance like ours as one of the metrics. Our work differs from theirs in two regards: (a) our corrections for programs (which are much more sophisticated than automata) are teacher-defined, while [3] considers a small pre-defined set of corrections over graphs, and (b) we use the Sketch synthesizer to efficiently navigate the huge search space, while [3] uses brute-force search.

7.5 Automated Grading Approaches
The survey by Douce et. al. [11] presents a nice overview of the systems developed for automated grading of programming assignments over the last forty years. Based on the age of these systems, they classify them into three generations. The first generation systems [21] graded programs by comparing the stored data with the data obtained from program execution, and kept track of running times and grade books. The second generation systems [22] also checked for programming styles such as modularity, complexity, and efficiency in addition to checking for correctness. The third generation tools such as RoboProf [9] combine web technology with more sophisticated testing approaches. All of these approaches are a form of test-cases based grading approach and can produce feedback in terms of failing test inputs, whereas our technique uses program synthesis for generating tailored feedback about the changes required in the student submission to make it correct.

7.6 Program Synthesis
Program synthesis has been used recently for many applications such as synthesis of efficient low-level code [29, 38], data structure manipulations [34], inference of efficient synchronization in concurrent programs [42], snippets of excel macros [18, 33], relational data representations [19, 20] and angelic programming [8]. The Sketch tool [37, 38] takes a partial program and a reference implementation as input and uses constraint-based reasoning to synthesize a complete program that is equivalent to the reference implementation. In general cases, the template of the desired program as well as the reference specification is unknown and puts an additional burden on the users to provide them; in our case we use the student’s solution as the template program and teacher’s solution as the reference implementation. A recent work by Gulwani et. al. [17] also uses program synthesis techniques for automatically synthesizing solutions to ruler/compass based geometry construction problems. Their focus is primarily on finding a solution to a given geometry problem whereas we aim to provide feedback on a given programming exercise solution.

8. Conclusions
In this paper, we presented a new technique of automatically providing feedback for introductory programming assignments that can complement manual and test-cases based techniques. The technique uses an error model describing the potential corrections and constraint-based synthesis to compute minimal corrections to student’s incorrect solutions. We have evaluated our technique on a
large set of benchmarks and it can correct 64% of incorrect solutions in our benchmark set. We believe this technique can provide a basis for providing automated feedback to hundreds of thousands of students learning from online introductory programming courses that are being taught by MITx, Coursera, and Udacity.

9. Acknowledgements

This work is supported in part by National Science Foundation grants Expeditions in Computer Augmented Program Engineering (ExCAPE NSF-1139056) and NSF-1139056, MIT EECS Super UROP program, Microsoft Research and a Microsoft Research PhD fellowship.

We thank Ned Batchelder, Sarina Canelake, Piotr Mitros, and Victor Shnayder from the EdIX team for their support and for providing us the 6.08x data. We thank the anonymous reviewers for their valuable feedback. We also thank the UROP students Tuan Andrew Nguyen, Xola Nymuy, and Eugene Oh for their contributions to this project. Finally, we thank Nikolai Tillmann and Jonathan de Halleux from the Pex4Fun team at Microsoft Research for providing us the C# data for the initial version of our tool.

References