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.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

- **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 attempts 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 assignment problems. 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 thousands 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.

2. 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

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

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).

```
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`.

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[1]. 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 `len(poly)=1`. The student chose to change the initialization of the `deriv` variable from $[\,]$ 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 feedback-level 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.

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-

[1] `https://www.edx.org/courses/MITx/6.00x/2012_Fall/discussion/forum/600x_ps3_q2/threads/5085f3a27d1d422500000040`

Three different student submissions for `computeDeriv`

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

(a)

```
 1 def computeDeriv(poly):
 2     idx = 1
 3     deriv = list([])
 4     plen = len(poly)
 5     while idx <= plen:
 6         coeff = poly.pop(1)
 7         deriv += [coeff * idx]
 8         idx = idx + 1
 9         if len(poly) < 2:
10             return deriv
```

(b)

```
 1 def computeDeriv(poly):
 2     length = int(len(poly)-1)
 3     i = length
 4     deriv = range(1,length)
 5     if len(poly) == 1:
 6         deriv = [0]
 7     else:
 8         while i >= 0:
 9             new = poly[i] * i
10             i -= 1
11             deriv[i] = new
12     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.

plementation such as the one in Figure 1. Since Python is dynamically typed, the instructor also provides the types of function arguments and return value. In Figure 1, the instructor specifies the type of input argument to be list of integers (`poly_list_int`) by appending the type to the name.

In addition to the reference implementation, the tool needs a description of the kinds of errors students might make. We have designed an error model language EML, which can describe a set of correction rules that denote the potential corrections to errors that students might make. For example, in the student attempt in Figure 2(a), we observe that corrections often involve modifying the return value and the range iteration values. We can specify this information with the following three correction rules:

$$\begin{aligned} \texttt{return}\ a &\rightarrow \texttt{return}\ [0] \\ \texttt{range}(a_1, a_2) &\rightarrow \texttt{range}(a_1 + 1, a_2) \\ a_0 == a_1 &\rightarrow \texttt{False} \end{aligned}$$

The correction rule `return` $a \rightarrow$ `return` $[0]$ states that the expression of a `return` statement can be optionally replaced by $[0]$. The error model for this problem that we use for our experiments is shown in Figure 8, but we will use this simple error model for simplifying the presentation in this section. In later experiments, we also show how only a few tens of incorrect solutions can provide enough information to create an error model that can automatically provide feedback for thousands of incorrect solutions.

The rules define a space of candidate programs which the tool needs to search in order to find one that is equivalent to the reference implementation and that requires minimum number of corrections. We use constraint-based synthesis technology [16, 37, 40] to efficiently search over this large space of programs. Specifically, we use the SKETCH synthesizer that uses a SAT-based algorithm to complete program sketches (programs with holes) so that they meet a given specification. We extend the SKETCH synthesizer with support for *minimize* hole expressions whose values are computed efficiently by using incremental constraint solving. To simplify the presentation, we use a simpler language MPY (*miniPython*) in place of Python to explain the details of our algorithm. In practice, our tool supports a fairly large subset of Python including closures, higher order functions, and list comprehensions.

2.2 Solution Strategy

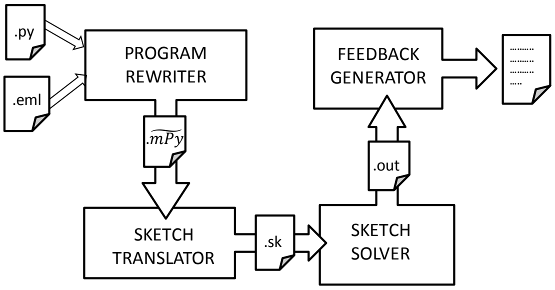


Figure 3. The architecture of our feedback generation tool.

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 $\widetilde{\text{MPY}}$; 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 $\widetilde{\text{MPY}}$ program is translated into a sketch program by the `Sketch Translator`.

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

Figure 4. The resulting $\widetilde{\text{MPy}}$ 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 $\widetilde{\text{MPy}}$ 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 $\widetilde{\text{MPy}}$ language extends the imperative language MPy with expression choices, where the choices are denoted with squiggly 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 $\{\boxed{a_0}, a_1, \cdots, a_n\}$ denotes a choice between expressions $a_0, \cdots, a_n$ 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 grow 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 $\widetilde{\text{MPy}}$ 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 $\widetilde{\text{MPy}}$ 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 `MTList` 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 MTList(len=2,lVals={new MultiType(val=1,flag=INTEGER), new MultiType(val=2,flag=INTEGER)}), flag=LIST)`.

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

Figure 5. The `MultiType` struct for encoding Python types.

The second key aspect of this translation is the translation of expression choices in $\widetilde{\text{MPy}}$. The Sketch construct `??` denotes an unknown integer hole that can be assigned any constant integer value by the synthesizer. The expression choices in $\widetilde{\text{MPy}}$ 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` $\{\boxed{\texttt{deriv}},[0]\}$; (line 5 in Figure 4) is translated to `return modRetVal0(deriv)`, where the `modRetVal0` function is defined as:

```
MultiType modRetVal0(MultiType a){
  if(??) return a; // default choice
  choiceRetVal0 = True; // non-default choice
  MTList list = new MTList(lVals={new
       MultiType(val=0, flag=INTEGER)}, len=1);
  return new MultiType(lst=list, type = LIST);
}
```

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 2^{16} 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. Eml: Error Model Language

In this section, we describe the syntax and semantics of the error model language Eml. An Eml 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

$$
\begin{aligned}
[\![a]\!] &= \{(a,0)\}\\
[\![\{\boxed{\tilde{a}_0},\cdots,\tilde{a}_n\}]\!] &= [\![\tilde{a}_0]\!] \cup \{(a,\ c+1) \mid (a,c) \in [\![\tilde{a}_i]\!]_{0<i\leq n}\}\\
[\![\tilde{a}_0[\tilde{a}_1]]\!] &= \{(a_0[a_1],\ c_0+c_1) \mid (a_i,c_i) \in [\![\tilde{a}_i]\!]_{i\in\{0,1\}}\}\\
[\![\texttt{while}\ \tilde{b} : \tilde{s}]\!] &= \{(\texttt{while}\ b :\ s,\ c_b+c_s) \mid\\
&\qquad\qquad (b,c_b) \in [\![\tilde{b}]\!], (s,c_s) \in [\![\tilde{s}]\!]\}
\end{aligned}
$$

Figure 7. The $[\![\ ]\!]$ function (shown partially) that translates an $\widetilde{\text{MPY}}$ program to a weighted set of MPY programs.

represented succinctly as an $\widetilde{\text{MPY}}$ program element, where $\widetilde{\text{MPY}}$ extends the MPY language with set-exprs (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 $\widetilde{\text{MPY}}$ 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 $\widetilde{\text{MPY}}$ languages

The syntax of MPY and $\widetilde{\text{MPY}}$ languages is shown in Figure 6(a) and Figure 6(b) respectively. The purpose of $\widetilde{\text{MPY}}$ language is to represent a large collection of MPY programs succinctly. The $\widetilde{\text{MPY}}$ 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 $\{\boxed{n_0},\cdots,n_k\}$ represents a weighted set of constant integers where n_0 denotes the default integer value associated with cost 0 and all other integer constants $(n_1,\cdots,n_k)$ 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 $\{\boxed{a_0},a_0+1\}\{\boxed{<},\leq,>,\geq,==,\neq\}\{\boxed{a_1},a_1+1,a_1-1\}$ represents 36 MPY expressions.

Each MPY program in the set of programs represented by an $\widetilde{\text{MPY}}$ 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 $[\![\ ]\!]$ function shown partially in Figure 7, the complete function definition can be found in [36]. The $[\![\ ]\!]$ 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 $\{\boxed{\tilde{a}_0},\cdots,\tilde{a}_n\}$, the function returns the union of the weighted set of MPY expressions corresponding to the default set-expression ($[\![\tilde{a}_0]\!]$) and the weighted set of expressions corresponding to other set-expressions $(\tilde{a}_1,\cdots,\tilde{a}_n)$, 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 $\widetilde{\text{MPY}}$ program. A *correction rule* $\mathcal{C}$ is written as a rewrite rule $L \to R$, where L and R denote a *program element* in MPY and $\widetilde{\text{MPY}}$ 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 $\widetilde{\text{MPY}}$ 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.

$$
\begin{aligned}
\textsc{IndR}:\ v[a] &\rightarrow v[\{a+1,a-1,?a\}]\\
\textsc{InitR}:\ v=n &\rightarrow v=\{n+1,n-1,0\}\\
\textsc{RanR}:\ \texttt{range}(a_0,a_1) &\rightarrow \texttt{range}(\{0,1,a_0-1,a_0+1\},\\
&\qquad\qquad \{a_1+1,a_1-1\})\\
\textsc{CompR}:\ a_0\ op_c\ a_1 &\rightarrow \{\{a_0'-1,?a_0\}\ \widetilde{op}_c\ \{a_1'-1,0,1,?a_1\},\\
&\quad \texttt{True},\texttt{False}\}\\
&\quad \text{where } \widetilde{op}_c=\{<,>,\leq,\geq,==,\neq\}\\
\textsc{RetR}:\ \texttt{return}\ a &\rightarrow \texttt{return}\{[0]\ \texttt{if}\ \texttt{len}(a)==1\ \texttt{else}\ a,\\
&\qquad a[1:]\ \texttt{if}\ (\texttt{len}(a)>1)\ \texttt{else}\ a\}
\end{aligned}
$$

Figure 8. The error model $\mathcal{E}$ for the `computeDeriv` problem.

Example 1. *The error model for the* `computeDeriv` *problem is shown in Figure 8. The* INDR *rewrite rule transforms the list access indices. The* INITR *rule transforms the right hand side of constant initializations. The* RANR *rule transforms 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 $\mathcal{C} : L \to R$ is defined to be well-formed if all tagged sub-terms t' in R have a smaller size syntax tree than that of L.*

The rewrite rule $\mathcal{C}_1 : v[a] \to \{(v[a])'+1\}$ 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 $\mathcal{C}_2 : v[a] \to \{v'[a']+1\}$ is well-formed.

Definition 2. *Well-formed Error Model : An error model $\mathcal{E}$ is defined to be well-formed if all of its constituent rewrite rules $\mathcal{C}_i \in \mathcal{E}$ are well-formed.*

3.3 Transformation with EML

An error model $\mathcal{E}$ is syntactically translated to a function $\mathcal{T}_{\mathcal{E}}$ that transforms an MPY program to an $\widetilde{\text{MPY}}$ program. The $\mathcal{T}_{\mathcal{E}}$ 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 $w_0 \in \widetilde{\text{MPY}}$. For each correction rule $\mathcal{C}_i : L_i \to R_i$ in the error model $\mathcal{E}$, the function contains a

$$
\begin{array}{rcl}
\text{Arith Expr } a & := & n \mid [\,] \mid v \mid a[a] \mid a_0\ op_a\ a_1 \\
 & \mid & [a_1, \cdots, a_n] \mid f(a_0, \cdots, a_n) \\
 & \mid & a_0\ \texttt{if}\ b\ \texttt{else}\ a_1 \\
\text{Arith Op } op_a & := & + \mid - \mid \times \mid / \mid ** \\
\text{Bool Expr } b & := & \texttt{not}\ b \mid a_0\ op_c\ a_1 \mid b_0\ op_b\ b_1 \\
\text{Comp Op } op_c & := & == \mid < \mid > \mid \leq \mid \geq \\
\text{Bool Op } op_b & := & \texttt{and} \mid \texttt{or} \\
\text{Stmt Expr } s & := & v = a \mid s_0; s_1 \mid \texttt{while}\ b :\ s \\
 & \mid & \texttt{if}\ b :\ s_0\ \texttt{else:}\ s_1 \\
 & \mid & \texttt{for}\ a_0\ \texttt{in}\ a_1 :\ s \mid \texttt{return}\ a \\
\text{Func Def. } p & := & \texttt{def}\ f(a_1, \cdots, a_n) :\ s
\end{array}
$$

(a) MPY

$$
\begin{array}{rcl}
\text{Arith set-expr } \tilde{a} & := & a \mid \{\boxed{\tilde{a}_0}, \cdots, \tilde{a}_n\} \mid \tilde{a}[\tilde{a}] \mid \tilde{a}_0\ \widetilde{op}_a\ \tilde{a}_1 \\
 & \mid & [\tilde{a}_0, \cdots, \tilde{a}_n] \mid \tilde{f}(\tilde{a}_0, \cdots, \tilde{a}_n) \\
\text{set-op } \widetilde{op}_x & := & op_a \mid \{\boxed{\widetilde{op}_{x_0}}, \cdots, \widetilde{op}_{x_n}\} \\
\text{Bool set-expr } \tilde{b} & := & b \mid \{\boxed{\tilde{b}_0}, \cdots, \tilde{b}_n\} \mid \texttt{not}\ \tilde{b} \mid \tilde{a}_0\ \widetilde{op}_c\ \tilde{a}_1 \mid \tilde{b}_0\ \widetilde{op}_b\ \tilde{b}_1 \\
\text{Stmt set-expr } \tilde{s} & := & s \mid \{\boxed{\tilde{s}_0}, \cdots, \tilde{s}_n\} \mid \tilde{v} := \tilde{a} \mid \tilde{s}_0; \tilde{s}_1 \\
 & \mid & \texttt{while}\ \tilde{b} :\ \tilde{s} \mid \texttt{for}\ \tilde{a}_0\ \texttt{in}\ \tilde{a}_1 :\ \tilde{s} \\
 & \mid & \texttt{if}\ \tilde{b} :\ \tilde{s}_0\ \texttt{else} :\ \tilde{s}_1 \mid \texttt{return}\ \tilde{a} \\
\text{Func Def } \tilde{p} & := & \texttt{def}\ f(a_1, \cdots, a_n)\ \tilde{s}
\end{array}
$$

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

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

`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 $w_i \in \widetilde{\text{MPY}}$ as defined by the right hand side R_i of the rule after calling the $\mathcal{T}_\mathcal{E}$ function on each of its tagged sub-terms t'. Finally, the method returns the set of all transformed terms $\{\boxed{w_0}, \cdots, w_n\}$.

Example 2. *Consider an error model $\mathcal{E}_1$ consisting of the following three correction rules:*

$$
\begin{array}{rcl}
\mathcal{C}_1 :\ v[a] & \rightarrow & v[\{a-1, a+1\}] \\
\mathcal{C}_2 :\ a_0\ op_c\ a_1 & \rightarrow & \{a_0' - 1, 0\}\ op_c\ \{a_1' - 1, 0\} \\
\mathcal{C}_3 :\ v[a] & \rightarrow & ?v[a]
\end{array}
$$

The transformation function $\mathcal{T}_{\mathcal{E}_1}$ for the error model $\mathcal{E}_1$ is shown in Figure 9.

$$
\begin{array}{l}
\mathcal{T}_{\mathcal{E}_1}(w : \text{MPY}) : \widetilde{\text{MPY}} \quad = \\
\textbf{let } w_0 = w[t \rightarrow \mathcal{T}_{\mathcal{E}_1}(t)] \textbf{ in } \ (*\ t : \text{a sub-term of } w\ *) \\
\textbf{let } w_1 = \textbf{Match } w \textbf{ with} \\
\qquad\qquad v[a] \rightarrow v[\{a+1, a-1\}] \textbf{ in} \\
\textbf{let } w_2 = \textbf{Match } w \textbf{ with} \\
\qquad\qquad a_0\ op_c\ a_1 \rightarrow \{\mathcal{T}_{\mathcal{E}_1}(a_0) - 1, 0\}\ op_c \\
\qquad\qquad\qquad\qquad\qquad \{\mathcal{T}_{\mathcal{E}_1}(a_1) - 1, 0\} \textbf{ in} \\
\{\boxed{w_0}, w_1, w_2\}
\end{array}
$$

Figure 9. The $\mathcal{T}_{\mathcal{E}_1}$ method for error model $\mathcal{E}_1$.

The recursive steps of application of $\mathcal{T}_{\mathcal{E}_1}$ function on expression $(x[i] < y[j])$ are shown in Figure 10. This example illustrates two interesting features of the transformation function:

- **Nested Transformations :** Once a rewrite rule $L \rightarrow R$ is applied to transform a program element matching L to R, the instructor may want to apply another rewrite rule on only a few sub-terms of R. For example, she may want to avoid transforming the sub-terms which have already been transformed by some other correction rule. The EML language facilitates making such distinction between the sub-terms for performing nested corrections using the $'$ (prime) operator. Only the sub-terms in R that are tagged with the prime operator are visited for applying further transformations (using the $\mathcal{T}_\mathcal{E}$ function recursively on its tagged sub-terms t'), whereas the non-tagged sub-terms are not transformed any further. After applying the rewrite rule $\mathcal{C}_2$ in the example, the sub-terms $x[i]$ and $y[j]$ are further transformed by applying rewrite rules $\mathcal{C}_1$ and $\mathcal{C}_3$.
- **Ambiguous Transformations :** While transforming a program using an error model, it may happen that there are multiple rewrite rules that pattern match the program element w. After applying rewrite rule $\mathcal{C}_2$ in the example, there are two rewrite rules $\mathcal{C}_1$ and $\mathcal{C}_3$ that pattern match the terms $x[i]$ and $y[j]$. After applying one of these rules ($\mathcal{C}_1$ or $\mathcal{C}_3$) to an expression $v[a]$, we cannot apply the other rule to the transformed expression. In such ambiguous cases, the $\mathcal{T}_\mathcal{E}$ function creates a separate copy of the transformed program element (w_i) for each ambiguous choice and then performs the set union of all such elements to obtain the transformed program element. This semantics of handling ambiguity of rewrite rules also matches naturally with the intent of the instructor. If the instructor wanted to perform both transformations together on array accesses, she could have provided a combined rewrite rule such as $v[a] \rightarrow ?v[\{a+1, a-1\}]$.

Theorem 1. *Given a well-formed error model $\mathcal{E}$, the transformation function $\mathcal{T}_\mathcal{E}$ always terminates.*

Proof. From the definition of well-formed error model, each of its constituent rewrite rule is also well-formed. Hence, each application of a rewrite rule reduces the size of the syntax tree of terms that are required to be visited further for transformation by $\mathcal{T}_\mathcal{E}$. Therefore, the $\mathcal{T}_\mathcal{E}$ function terminates in a finite number of steps. □

4. Constraint-based Solving of $\widetilde{\text{MPY}}$ programs

In the previous section, we saw the transformation of an MPY program to an $\widetilde{\text{MPY}}$ program based on an error model. We now present the translation of an $\widetilde{\text{MPY}}$ program into a SKETCH program [37].

4.1 Translation of $\widetilde{\text{MPY}}$ programs to SKETCH

The $\widetilde{\text{MPY}}$ programs are translated to 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 $\widetilde{\text{MPY}}$ to SKETCH, and (ii) the translation of set-expr choices in $\widetilde{\text{MPY}}$ to SKETCH functions.

$$\mathcal{T}(x[i] < y[j]) \equiv \{\boxed{\mathcal{T}(x[i]) < \mathcal{T}(y[j])}, \{\mathcal{T}(x[i]) - 1, 0\} < \{\mathcal{T}(y[j]) - 1, 0\}\}$$

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

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

$$\mathcal{T}(x) \equiv \{\boxed{x}\} \qquad \mathcal{T}(i) \equiv \{\boxed{i}\} \qquad \mathcal{T}(y) \equiv \{\boxed{y}\} \qquad \mathcal{T}(j) \equiv \{\boxed{j}\}$$

`Therefore, after substitution the result is:`

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

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

Figure 10. Application of $\mathcal{T}_{\mathcal{E}_1}$ (abbreviated $\mathcal{T}$) on expression $(x[i] < y[j])$.

Handling dynamic typing of $\widetilde{\textrm{MPY}}$ variables The dynamic typing in $\widetilde{\textrm{MPY}}$ is handled using a `MultiType` variable as described in Section 2.3. The $\widetilde{\textrm{MPY}}$ 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 `assignMT(a, b)`, where the `assignMT` function assigns `MultiType` `b` to `a`. Similarly, the binary add expression `(a + b)` is translated to `binOpMT(a, b, ADD_OP)` that in turn calls the function `addMT(a,b)` to add `a` and `b` as shown in Figure 11.

```
MultiType addMT(MultiType a, MultiType b){
  assert a.flag == b.flag; // same types can be added
  if(a.flag == INTEGER)    // add for integers
   return new MultiType(val=a.val+b.val, flag =
        INTEGER);
  if(a.flag == LIST){      // add for lists
    int newLen = a.lst.len + b.lst.len;
    MultiType[newLen] newLVals = a.lst.lVals;
    for(int i=0; i<b.lst.len; i++)
      newLVals[i+a.lst.len] = b.lst.lVals[i];
  return new MultiType(lst = new
       MTList(lVals=newLVals, len=newLen),
       flag=LIST);}
  ··· ···
}
```

Figure 11. The `addMT` function for adding two `MultiType` `a` and `b`.

Translation of $\widetilde{\textrm{MPY}}$ set-expressions The set-expressions in $\widetilde{\textrm{MPY}}$ are translated to functions in SKETCH. The function bodies obtained by the application of translation function (Φ) on some of the interesting $\widetilde{\textrm{MPY}}$ constructs are shown in Figure 12. The SKETCH construct ?? (called *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 MPY expression such as $\{a\}$ are translated simply to the corresponding expression itself. A set-expression of the form $\{\boxed{\tilde{a}_0}, \cdots, \tilde{a}_n\}$ is translated recursively to the `if` expression :`if` $(??)\ \Phi(\tilde{a}_0)$ `else` $\Phi(\{\tilde{a}_1, \cdots, \tilde{a}_n\})$, 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 $(\tilde{a}_1, \cdots, \tilde{a}_n)$. The set-expressions of the form $\{\tilde{a}_0, \cdots, \tilde{a}_n\}$ are similarly translated but with an additional statement for setting a fresh variable $\mathtt{choice}_\mathtt{k}$ if the synthesizer selects the non-default choice $\tilde{a}_0$.

$$\begin{array}{rcl}
\Phi(\{a\}) & = & a \\
\Phi(\{\boxed{\tilde{a}_0}, \cdots, \tilde{a}_n\}) & = & \texttt{if } (??)\ \Phi(\tilde{a}_0) \texttt{ else } \Phi(\{\tilde{a}_1, \cdots, \tilde{a}_n\}) \\
\Phi(\{\tilde{a}_0, \cdots, \tilde{a}_n\}) & = & \texttt{if } (??)\ \{\mathtt{choice}_\mathtt{k} = \mathtt{True}; \Phi(\tilde{a}_0)\} \\
 & & \texttt{else } \Phi(\{\tilde{a}_1, \cdots, \tilde{a}_n\}) \\
\Phi(\tilde{a}_0[\tilde{a}_1]) & = & \Phi(\tilde{a}_0)[\Phi(\tilde{a}_1)] \\
\Phi(\tilde{a}_0 = \tilde{a}_1) & = & \Phi(\tilde{a}_0) := \Phi(\tilde{a}_1)
\end{array}$$

Figure 12. The translation rules (shown partially) for converting $\widetilde{\textrm{MPY}}$ set-exprs to corresponding SKETCH function bodies.

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 `if` expression choices occurring on the left hand side are desugared to individual assignments. For example, the left hand side expression `if` $(??)\ x$ `else` $y := 10$ is desugared to `if` $(??)\ x := 10$ `else` $y := 10$. The infix operators in $\widetilde{\textrm{MPY}}$ are first translated to function calls and are then translated to sketch using the translation for set-function expressions. The remaining $\widetilde{\textrm{MPY}}$ 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 parmeterized 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 SKETCH primitive types such as `int`, `bit` and arrays. Therefore, after the translation of $\widetilde{\textrm{MPY}}$ programs to SKETCH programs, we need additional driver functions to integrate the functions over `MultiType` input arguments and return value to the corresponding functions over SKETCH 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 $\widetilde{\text{MPY}}$ 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(choice`$_{\mathtt{k}}$`) totalCost++;` and the statement `minimize(totalCost)`, which tells the synthesizer to compute a solution to the Boolean variables `choice`$_{\mathtt{k}}$ that minimizes the `totalCost` variable.

4.2 CEGISMIN: Incremental Solving for the Minimize holes

Algorithm 1 `CEGISMIN` Algorithm for Minimize expression

```
$\sigma_0 \leftarrow \sigma_{\mathtt{random}}$,   $i \leftarrow 0$,   $\Phi_0 \leftarrow \Phi$,   $\phi_p \leftarrow$ null
while (True)
    $i \leftarrow i + 1$
    $\Phi_i \leftarrow$ Synth($\sigma_{i-1}, \Phi_{i-1}$)                    ▷ Synthesis Phase
    if ($\Phi_i$ = UNSAT)                                        ▷ Synthesis Fails
        if ($\Phi_{\mathtt{prev}}$ = null) return UNSAT_SKETCH
        else return PE(P,$\phi_p$)
    choose $\phi \in \Phi_i$
    $\sigma_i \leftarrow$ Verify($\phi$)                              ▷ Verification Phase
    if ($\sigma_i$ = null)                                   ▷ Verification Succeeds
        (minHole, minHoleValue) $\leftarrow$ getMinHoleValue($\phi$)
        $\phi_p \leftarrow \phi$
        $\Phi_i \leftarrow \Phi_i \cup$ {encode(minHole < 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 σ and the sketch constraint store is denoted by Φ. Initially, the input state σ_0 is assigned a random input state value and the constraint store Φ_0 is assigned the constraint set obtained from the sketch program. The variable ϕ_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 Φ_{i-1} to Φ_i by removing behaviors from Φ_{i-1} that do not conform to the input state σ_{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 σ_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 ϕ, stores the current satisfiable hole solution ϕ in ϕ_p, and adds an additional constraint `{minHole<minHoleVal}` to the constraint set Φ_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 ϕ.

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 can not 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[2] on PEX4FUN that were based on loop-over-arrays and dynamic programming from an AP level exam[3]. 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 `l`, return a tuple consisting of every other element of `l`.
- `compDeriv-6.00` : Compute the derivative of a polynomial `poly`, where the coefficients of `poly` are represented as a list.
- `evalPoly-6.00` : 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 $\mathtt{m}^{\mathtt{n}}$ using only the multiplication operator, where `m` and `n` are integers.
- `recurPower-6.00x` : Compute the value $\mathtt{m}^{\mathtt{n}}$ 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.

[2] `http://pexforfun.com/learnbeginningprogramming`

[3] 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(C#) : 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(C#) : 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 (C#) : 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 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 attempts (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 solution 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 program 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 significant fraction of the problems require 3 and 4 coordinated corrections, and to provide feedback on such attempts, we need a technology 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 benchmark 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 comparison to the traditional test-cases based feedback techniques that test the programs over a few dozens of test-cases, our tool typically performs the equivalence check over more than 10^6 inputs. Programs that print the output to console (e.g. compBal-stdin) pose an interesting 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 heuristics 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 corresponding 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 context 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 unrollings and recursion depth as well as manually provide specialized 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 features, most notably classes and objects, which are required for providing feedback on problems from later weeks of the class.

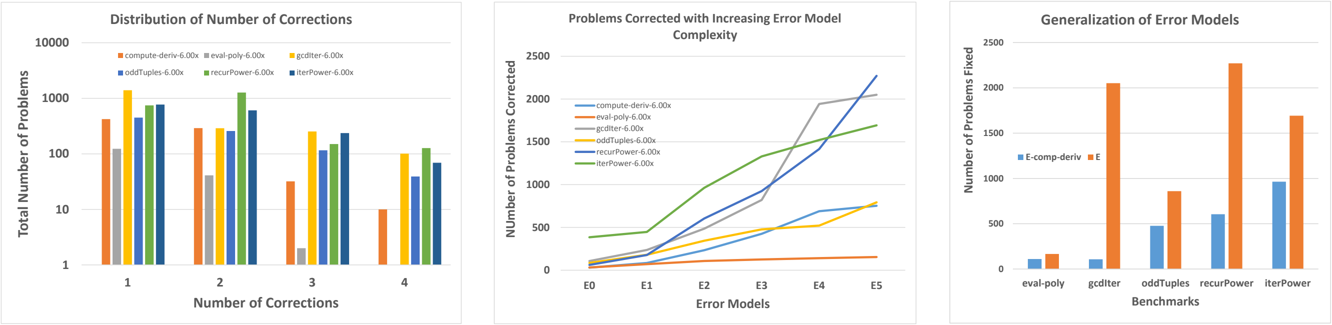
```python
def evaluatePoly(poly, x):
    result = 0
    for i in list(poly):
        result += i*x**poly.index(i)
    return result
```

(a) an evalPoly solution

```python
def getGuessedWord(secretWord, lettersGuessed):
    for letter in lettersGuessed:
        secretWord = secretWord.replace(letter, '_')
    return secretWord
```

(b) a hangman2-str solution

Figure 13. An example of big conceptual error for a student's attempt for (a) evalPoly and (b) hangman2-str problems.



(a) (b) (c)

Figure 14. (a) The number of incorrect problem that require different number of corrections (in log scale), (b) the number of problems corrected with adding rules to the error models, and (c) the performance of compute-deriv error model on other problems.

Benchmark	Median (LOC)	Total Attempts	Syntax Errors	Test Set	Correct	Incorrect Attempts	Generated Feedback	Average Time(in s)	Median Time(in s)
prodBySum-6.00	5	1056	16	1040	772	268	218 (81.3%)	2.49s	2.53s
oddTuples-6.00	6	2386	1040	1346	1002	344	185 (53.8%)	2.65s	2.54s
compDeriv-6.00	12	144	20	124	21	103	88 (85.4%)	12.95s	4.9s
evalPoly-6.00	10	144	23	121	108	13	6 (46.1%)	3.35s	3.01s
compBal-stdin-6.00	18	170	32	138	86	52	17 (32.7%)	29.57s	14.30s
compDeriv-6.00x	13	4146	1134	3012	2094	918	753 (82.1%)	12.42s	6.32s
evalPoly-6.00x	15	4698	1004	3694	3153	541	167 (30.9%)	4.78s	4.19s
oddTuples-6.00x	10	10985	5047	5938	4182	1756	860 (48.9%)	4.14s	3.77s
iterPower-6.00x	11	8982	3792	5190	2315	2875	1693 (58.9%)	3.58s	3.46s
recurPower-6.00x	10	8879	3395	5484	2546	2938	2271 (77.3%)	10.59s	5.88s
iterGCD-6.00x	12	6934	3732	3202	214	2988	2052 (68.7%)	17.13s	9.52s
hangman1-str-6.00x	13	2148	942	1206	855	351	171 (48.7%)	9.08s	6.43s
hangman2-str-6.00x	14	1746	410	1336	1118	218	98 (44.9%)	22.09s	18.98s
stock-market-I(C#)	20	52	11	41	19	22	16 (72.3%)	7.54s	5.23s
stock-market-II(C#)	24	51	8	43	19	24	14 (58.3%)	11.16s	10.28s
restaurant rush (C#)	15	124	38	86	20	66	41 (62.1%)	8.78s	8.19s

Table 1. The percentage of student attempts corrected and the time taken for correction for the benchmark problems.

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 and compares them 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.

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 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 construct a *concise and precise* error model. This enables us to model more sophisticated transformations such as introducing new program statements, replacing LHS of assignments etc. in our error model. Our approach also supports minimal cost changes to student's programs where each error in the model is associated with a certain cost, unlike the earlier mentioned approaches.

Mutation-based program repair [10] performs mutations repeatedly to statements in a buggy program in order of their suspiciousness until the program becomes correct. The large state space of mutants (10^{12}) makes this approach infeasible. Our approach uses a symbolic search for exploring correct solutions over this large set. There are also some genetic programming approaches that exploit redundancy present in other parts of the code for fixing faults [5, 14]. These techniques are not applicable in our setting as such redundancy is not present in introductory programming problems.

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 EdX team for their support and for providing us the `6.00x` data. We thank the anonymous reviewers for their valuable feedback. We also thank the UROP students Tuan Andrew Nguyen, Xola Ntumy, 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

[1] A. Adam and J.-P. H. Laurent. LAURA, A System to Debug Student Programs. *Artif. Intell.*, 15(1-2):75–122, 1980.

[2] U. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In *IJCAI*, 2013.

[3] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In *IJCAI*, 2013.

[4] E. Andersen, S. Gulwani, and Z. Popovic. A trace-based framework for analyzing and synthesizing educational progressions. In *CHI*, 2013.

[5] A. Arcuri. On the automation of fixing software bugs. In *ICSE Companion*, 2008.

[6] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *POPL*, 2003.

[7] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylent: a word processor with a crowd inside. In *UIST*, 2010.

[8] R. Bodík, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *POPL*, 2010.

[9] C. Daly. Roboprof and an introductory computer programming course. ITiCSE, 1999.

[10] V. Debroy and W. Wong. Using mutation to automatically suggest fixes for faulty programs. In *ICST*, 2010.

[11] C. Douce, D. Livingstone, and J. Orwell. Automatic test-based assessment of programming: A review. *J. Educ. Resour. Comput.*, 5(3), Sept. 2005.

[12] P. Ertmer, J. Richardson, B. Belland, D. Camin, P. Connolly, G. Coulthard, K. Lei, and C. Mong. Using peer feedback to enhance the quality of student online postings: An exploratory study. *Journal of Computer-Mediated Communication*, 12(2):412–433, 2007.

[13] R. G. Farrell, J. R. Anderson, and B. J. Reiser. An interactive computer-based tutor for lisp. In *AAAI*, 1984.

[14] S. Forrest, T. Nguyen, W. Weimer, and C. L. Goues. A genetic programming approach to automated software repair. In *GECCO*, 2009.

[15] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *STTT*, 8(3):229–247, 2006.

[16] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, 2008.

[17] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[18] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. In *CACM*, 2012.

[19] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Data representation synthesis. In *PLDI*, 2011.

[20] P. Hawkins, A. Aiken, K. Fisher, M. C. Rinard, and M. Sagiv. Concurrent data representation synthesis. In *PLDI*, 2012.

[21] J. B. Hext and J. W. Winings. An automatic grading scheme for simple programming exercises. *Commun. ACM*, 12(5), May 1969.

[22] D. Jackson and M. Usher. Grading student programs using assyst. SIGCSE, 1997.

[23] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, pages 226–238, 2005.

[24] W. L. Johnson and E. Soloway. Proust: Knowledge-based program understanding. *IEEE Trans. Software Eng.*, 11(3):267–275, 1985.

[25] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, 2011.

[26] U. Junker. QUICKXPLAIN: preferred explanations and relaxations for over-constrained problems. In *AAAI*, 2004.

[27] R. Könighofer and R. P. Bloem. Automated error localization and correction for imperative programs. In *FMCAD*, 2011.

[28] C. Kulkarni and S. R. Klemmer. Learning design wisdom by augmenting physical studio critique with online self-assessment. Technical report, Stanford University, 2012.

[29] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. PLDI, 2010.

[30] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. Turkit: human computation algorithms on mechanical turk. In *UIST*, 2010.

[31] W. R. Murray. Automatic program debugging for intelligent tutoring systems. *Computational Intelligence*, 3:1–16, 1987.

[32] W. Sack, E. Soloway, and P. Weingrad. From PROUST to CHIRON: Its design as iterative engineering: Intermediate results are important! In *In J.H. Larkin and R.W. Chabay (Eds.), Computer-Assisted Instruction and Intelligent Tutoring Systems: Shared Goals and Complementary Approaches.*, pages 239–274, 1992.

[33] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5, 2012.

[34] R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, 2011.

[35] R. Singh, S. Gulwani, and S. K. Rajamani. Automatically generating algebra problems. In *AAAI*, 2012.

[36] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated semantic grading of programs. *CoRR*, abs/1204.1751, 2012.

[37] A. Solar-Lezama. *Program Synthesis By Sketching*. PhD thesis, EECS Dept., UC Berkeley, 2008.

[38] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

[39] E. Soloway, B. P. Woolf, E. Rubin, and P. Barth. Meno-II: An Intelligent Tutoring System for Novice Programmers. In *IJCAI*, 1981.

[40] S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. *POPL*, 2010.

[41] S. S. Staber, B. Jobstmann, and R. P. Bloem. Finding and fixing faults. In *Correct Hardware Design and Verification Methods*, Lecture notes in computer science, pages 35 – 49, 2005.

[42] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[43] D. S. Weld, E. Adar, L. Chilton, R. Hoffmann, and E. Horvitz. Personalized online education - a crowdsourcing challenge. In *Workshops at the Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[44] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28:183–200, 2002.