Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments

JUNHO LEE, Korea University, Republic of Korea
DOWON SONG, Korea University, Republic of Korea
SUNBEOM SO, Korea University, Republic of Korea
HAKJOO OH∗, Korea University, Republic of Korea

We present FixML, a system for automatically generating feedback on logical errors in functional programming assignments. As functional languages have been gaining popularity, the number of students enrolling functional programming courses has increased significantly. However, the quality of feedback, in particular for logical errors, is hardly satisfying. To provide personalized feedback on logical errors, we present a new error-correction algorithm for functional languages, which combines statistical error-localization and type-directed program synthesis enhanced with components reduction and search space pruning using symbolic execution. We implemented our algorithm in a tool, called FixML, and evaluated it with 497 students’ submissions from 13 exercises, including not only introductory but also more advanced problems. Our experimental results show that our tool effectively corrects various and complex errors: it fixed 43% of the 497 submissions in 5.4 seconds on average and managed to fix a hard-to-find error in a large submission, consisting of 154 lines. We also performed user study with 18 undergraduate students and confirmed that our system actually helps students to better understand their programming errors.

CCS Concepts: • Software and its engineering → Automatic programming: Functional languages;

Additional Key Words and Phrases: Automated Program Repair, Program Synthesis

ACM Reference Format:

1 INTRODUCTION

Motivation. The motivation for this work originated from an undergraduate course on functional programming taught by the authors over the last few years. As functional languages have been gaining popularity, the number of students enrolling the course has increased significantly. The quality of feedback, however, hardly satisfied the increased demands. Because most students have no experience in functional languages, they often have more difficulty with various programming

∗Corresponding author
errors than learning other languages such as Java or Python. However, assisting students to resolve those errors is challenging. In particular, providing personalized feedback on logical errors was most difficult as it requires instructors to manually figure out the cause of the errors, unlike syntax or type errors for which a wealth of tool support already exists [Chen and Erwig 2014; Lerner et al. 2007; Pavlinovic et al. 2014, 2015; Seidel et al. 2017; Wu et al. 2017; Wu and Chen 2017; Zhang et al. 2017].

Providing simple, non-personalized feedback on logical errors was not much helpful for students. Since it is hard to give advice on a one-to-one basis to many students, one standard way of giving feedback on logical errors is to provide failing testcases. Unfortunately, debugging is a time-consuming and error-prone task even for the experts on programming, so students frequently fail to fix bugs by themselves even with the given testcases. Another method of feedback is to offer the answer code. However, it is still troublesome since there are many possible ways to implement a program for the specific problem; the given answers cannot help students to understand their mistakes in this case. Hence, providing failing testcases and answers is not sufficient for students who need more guided feedback.

Goal. The goal of this paper is to develop an automated system, called FixML, for generating personalized feedback on logical errors in functional programming assignments. Given a student’s program that has logical errors as well as input-output testcases and a correct implementation from an instructor, FixML automatically repairs the student’s program by correcting buggy parts. The error location and an appropriate correction are presented to students, assisting them to better identify and understand their mistakes. Our target language is OCaml and we aim to provide feedback on not only introductory but also more challenging programming exercises.

Approach. To achieve this goal, we present a new error-correction algorithm for functional language. The algorithm combines statistical error-localization and enhanced type-directed program synthesis. First, it identifies the error locations and scores them with statistical reasoning on the results from dynamic executions. Secondly, the algorithm uses search-based program synthesis to replace the erroneous expression by a correct one that satisfies all of the given testcases. However, there remains a major scalability challenge for generating feedback on complex programs. Since we target a realistic subset of OCaml to deal with various types of problems, the search space for synthesis explodes. We address this challenge with an enhanced type-directed search algorithm. Due to the scalability issue, prior work on functional program synthesis [Feser et al. 2015; Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016] use type-directed search techniques. Although the technique can reduce the search space substantially, type-directed search alone was still unsuitable to repair nontrivial student submissions. Our algorithm enhances type-directed enumerative search with two techniques: components reduction and search space pruning. Because enumerating all components is too large to search entirely, we reduce the components by deducing semantically redundant variables via data-flow analysis and using syntax components extracted from a correct implementation provided by instructor. To prune the search space more effectively, we combine type-directed enumeration with symbolic execution to detect the partial programs that are well-typed but functionally inconsistent. We formalize our algorithm for a subset of OCaml but the general idea is applicable to other functional languages as well.

Results. Our experimental results demonstrate that our approach effectively provides personalized feedback on variety of programming exercises. We evaluated our tool, FixML, with 497 students’ programs from 13 problems with different levels. Overall, FixML successfully repaired 43% of the submissions in 5.4 seconds on average. Most notably, FixML was able to accurately
identify and fix an error in a large submission (154 lines, Appendix C). Furthermore, we show that FixML is actually helpful for students by conducting user study with 18 undergraduate students.

**Contributions.** This paper makes the following contributions:

- We present FixML, the first system for automatically diagnosing and correcting logical errors in functional programming exercises. While related existing work on functional languages has mostly focused on type errors (e.g. [Seidel et al. 2017; Wu et al. 2017; Wu and Chen 2017; Zhang et al. 2017]), our system aims to provide personalized feedback on logical errors.
- We present a novel error-correction algorithm that combines statistical error-localization and enhanced type-directed enumerative synthesis. We show that combining these techniques is a key to providing useful feedback on a wide range of nontrivial programs.
- We provide extensive evaluation results with real student submissions and user study. Our implementation and 497 benchmark programs are publicly available.¹

2 OVERVIEW

2.1 Motivating Examples

Let us demonstrate FixML with three functional programming problems from an undergraduate Programming Languages course and StackOverflow. According to our experience, most students had difficulty in solving these problems on their own, often implemented erroneous programs, and required help from human instructors to correct the errors.

As inputs, FixML takes three components (Fig. 4): a student’s program that has a logical error, a set of input-output testcases, which exposes the error, and a correct implementation. We assume that the last two components are provided from an instructor. Then, FixML generates a program that passes all the testcases, by automatically fixing erroneous parts of the student’s program.

**Example 1 (Differentiation).** Consider the problem of implementing a function that symbolically differentiates algebraic expressions. The expressions are defined in OCaml datatype as follows:

```ocaml
1 type var = string
2 type aexp = CONST of int | VAR of var | SUM of aexp list | TIMES of aexp list | POWER of var * int
```

The goal of the problem is to define the function `diff: aexp * var -> aexp`, which takes an expression and a variable name as input and produces another expression obtained by differentiating the input expression with respect to the given variable. For example, the expression \(x^2 + 1\) is represented by `SUM [POWER ("x", 2); CONST 1]` and differentiating it with respect to \(x\) gives \(2x\) which can be represented by `TIMES [CONST 2; VAR "x"]`. The main objective of this problem is to familiarize students with the use of inductive datatypes and recursive functions, both of which are essential in ML-style functional programming.

A student submitted an incorrect implementation shown in Fig. 1. The major part of the program is correct; for example, the student properly handles both the base and inductive cases for TIMES and SUM. However, the implementation has an error at line 10; when the base variable \(y\) is not equal to the variable \(x\) given as argument, the result should be `CONST 0`, not `POWER(y, n)` (e.g. differentiating \(y^2\) with respect to \(x\) produces 0).

FixML is able to automatically generate the feedback in 1 second. Given the incorrect submission in Fig. 1, a correct implementation by an instructor, and a set of test-cases that expose the error of the submission, FixML pinpoints the erroneous part (i.e. `POWER(y, n)` at line 10) and replaces the

¹https://github.com/kupl/FixML
let rec diff : aexp * var -> aexp = fun (e,x) ->
  match e with
  | CONST n -> CONST 0
  | VAR y -> if (x = y) then CONST 1 else CONST 0
  | POWER (y,n) ->
    if (x = y) then
      if (n = 0) then CONST 0
      else if (n = 1) then CONST 1
      else TIMES [CONST n; POWER (x,n-1)]
    else POWER (y,n)
    (* Feedback: Replace "POWER(y,n)" by "CONST 0" *)
  | TIMES l ->
    if (List.length l = 0) then CONST 0
    else if (List.length l = 1) then diff (List.hd l, x)
    else SUM [TIMES (diff (List.hd l,x) :: List.tl l);
               TIMES [List.hd l; diff (TIMES (List.tl l),x)]]
  | SUM l ->
    if (List.length l = 0) then SUM []
    else if (List.length l = 1) then diff (List.hd l,x)
    else SUM [diff (List.hd l,x); diff (SUM (List.tl l),x)]

Fig. 1. Incorrect implementation for Example 1 (Differentiation)

expression by CONST 0. In Fig. 1, the generated feedback is annotated as a comment at line 10. The
instructor’s solution for this problem is presented in Appendix A.

FixML can correct diverse types of logical errors. For example, consider the incorrect program
written by another student:

let rec diff (e,x) =
  match e with
  | SUM [] -> SUM []
  | SUM (h::[]) -> CONST 0 (*) Feedback: Replace "CONST 0" by "diff (h,x)" *)
  | SUM (h::t) -> SUM [diff (h,x); diff (SUM t,x)] | ...

In this case, the program is incorrect as it does not differentiate the head element h at line 4. With
the same solution and testcases, FixML corrects the error by replacing the expression CONST 0 by
diff (h,x) in 0.7 seconds.

Note that the two student submissions are substantially different from the instructor’s solution. For example, in Appendix A, the instructor implemented the SUM case using List.map as follows (Problem #13):

let rec diff (e,x) =
  match e with
  | SUM l -> SUM (List.map (fun e -> diff (e,x)) l) | ...

We observed that, for nontrivial programming assignments, students use many different ways of
implementing the required functionality (see Section 5.5). The development of FixML was motivated
by the difference between the solution and submissions, which makes it difficult for students to
identify and correct the errors in their own programs.

Furthermore, FixML can introduce more complex expressions such as conditional expressions. For example, consider the following code:

let rec diff (e,x) =
  match e with
let rec natmul n1 n2 = match n1 with |
| ZERO -> ZERO | SUCC ZERO -> n2
| SUCC n1' -> SUCC (match n2 with |
| ZERO -> ZERO | SUCC ZERO
| SUCC n2' -> SUCC (natmul n1' (natmul n1 n2'))

(* Feedback: Replace line 5–8 (left) by "(natmul n2 (natmul n1' n2))" *)

Fig. 2. Incorrect implementation for Example 2 (Natural Numbers)

| Var v -> CONST 1
| (* Feedback: Replace "Const 1" by "Const (if (x=v) then 1 else 0)" *) | ...

The program has an error at line 3, where the student missed the case when the variable v equals to the input variable x. FixML correctly identified this error and corrected the expression 1 by if var=str then 1 else 0 in 2.1 seconds.

**Example 2 (Natural Numbers).** The next problem is to implement functions that add and multiply user-defined natural numbers. The natural number can be defined in datatype as follows:

type nat = ZERO | SUCC of nat

For instance, (SUCC (SUCC ZERO)) denotes 2. The goal of the problem is to define two functions natadd: nat -> nat -> nat and natmul: nat -> nat -> nat, which take two natural numbers as input and produce their addition and multiplication, respectively. For example, natadd (SUCC (SUCC ZERO)) (SUCC ZERO) and natmul (SUCC (SUCC ZERO)) (SUCC ZERO) should produce (SUCC (SUCC (SUCC ZERO))) and (SUCC (SUCC ZERO)), respectively.

Fig. 2 shows an erroneous program written by a student, where natadd is correct but natmul has a big conceptual error. Note that multiplication is inductively defined with addition as follows:

\[
\begin{align*}
n_1 \times n_2 = \begin{cases} 
0 & n_1 = 0 \\
1 + (n_1 - 1) \times n_2 & n_1 \neq 0 
\end{cases}
\end{align*}
\]

However, the student could not conceive this equation and implemented the wrong codes at lines 5–8, where a substantial modification is needed to correct the program. Impressively, FixML replaced the four lines (5–8) by the expression (natadd n2 (natmul n1' n2)) in 22 seconds.

**Example 3 (Append).** The last problem came from StackOverflow\(^2\), where the goal is to write append: 'a list -> 'a list -> 'a list which appends the first list to the second list while removing duplicated elements. For instance, given two lists [4;5;6;7] and [1;2;3;4], the function should output [1;2;3;4;5;6;7]:

append [4;5;6;7] [1;2;3;4] = [1;2;3;4;5;6;7].

To solve the problem, one student defined the function append in Fig. 3. She first defined two helper functions: **find** and **help_append**. The function **find** takes an element and a list, and returns true iff the list contains the element. The purpose of the function **help_append** is to append elements of the list l1 to the list l2 if they are not in l2. Next, she defined the function **append** using

\(^2\)https://stackoverflow.com/questions/10271711/ocaml-append-list-to-another-list-without-duplicated, accessed 16-April-2018
let rec find e l =
  match l with
  | [] -> false
  | h::t -> if (h=e) then true else find e t

let rec help_append_list l1 l2 =
  match l1 with
  | [] -> l2
  | h::t ->
    if (find h l2 = false) then (help_append_list t (l2@[h]))
    else (help_append_list t l2)
  let append l1 l2 = help_append_list l1 l2

(* Feedback: Replace "l2" by "help_append_list l2 []" *)

Fig. 3. Incorrection implementation for Example 3 (Append)

help_append_list. Note that the implementation works correctly if the list l2 has no duplicated elements; for example, it produces the correct result for the example lists above. However, it does not work when the members of l2 are redundant. For example, given the lists [4;5;6;7] and [1;2;4;4], the program outputs [1;2;4;4;5;6;7], where 4 is duplicated. The student observed this faulty behavior but failed to diagnose and fix the error for herself.

FixML repairs the program in 43 seconds by removing duplicated elements in l2 before appending. The failure point of the original program is that it does not attempt to remove duplicates in l2. FixML precisely captures this root cause and modifies (help_append_list l1 l2) at line 13 into (help_append_list l1 (help_append_list l2 [])), where duplicates in l2 are removed by applying help_append_list to l2 and [] (help_append_list correctly checks duplicates in the first argument list). We remark that this feedback is exactly the same as the manual feedback provided by a human in the post and agreed upon by several others without any objections.

2.2 How FixML Works

Fig. 4 illustrates how FixML works. Given a student’s incorrect program, a set of testcases, and a correct implementation from an instructor, FixML repairs the student code by using statistical error
Localization and enhanced type-directed synthesis. Let us illustrate our algorithm using a simple example. Assume that we are given the following buggy (left) and correct (right) implementations of the factorial function:

```ml
1 let rec factorial n = 1 let rec factorial n =  
2 match n with 2 if (n=0) then 1  
3 | 0 -> 1 3 else n * factorial (n-1)  
4 | x -> 2 (* error *)  
```

as well as a set of input-output testcases, \{0 \mapsto 1, 3 \mapsto 6, 4 \mapsto 24\}.

**Statistical Error Localization.** FixML begins with localizing the error and produces a ranked list of partial programs. The error-localization procedure works in the three steps. First, we classify the testcases into positive (\(P\)) and negative (\(N\)) testcases: \(P = \{0 \mapsto 1\}\) and \(N = \{3 \mapsto 6, 4 \mapsto 24\}\), such that the buggy program correctly works for \(P\) but not for \(N\). Second, we collect subexpressions of the buggy program that are evaluated with the negative testcases. In our example, these subexpressions are: \(S = \{2, n, \text{match } n \text{ with } | 0 \rightarrow 1 | x \rightarrow 2\}\). Then, we generate a set of partial programs by replacing each expression in \(S\) by a hole, producing the following three partial programs:

```ml
(1) let rec factorial n = match n with | 0 -> 1 | x -> ?
(2) let rec factorial n = match ? with | 0 -> 1 | x -> 2
(3) let rec factorial n = ?
```

Finally, we rank the partial programs based on the observation that subexpressions that are less involved with the positive cases \(P\) are more likely to be erroneous. As a result, our algorithm attempts to repair the partial program (1) with the highest priority, as the subexpression 2 is never evaluated under the positive testcases.

**Enhanced Type-Directed Synthesis.** Once we generate a set of partial programs, the next step is to synthesize corrections, i.e., filling in the holes in partial programs. Our correction-synthesis algorithm basically performs enumerative search, which attempts to replace holes in partial programs by considering all the possible expressions according to the language grammar in increasing size. We enhance this naive algorithm with three techniques.

First, we use type-directed search [Feser et al. 2015; Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016], where ill-typed expressions do not get enumerated. For example, the type of holes in our running example is integer, so we do not attempt to instantiate the holes with boolean expressions. However, type-directed search alone was not scalable enough beyond problems at introductory level, which motivated us to develop the following techniques.

Second, we reduce variable components using a data-flow analysis that identifies semantically redundant variables. For example, given a partial program (\(\text{let factorial n = match n with } | 0 \rightarrow 1 | x \rightarrow ?\)), our synthesizer uses only the variable \(x\) to instantiate the hole ?, and does not use the variable \(n\) because \(n\) is semantically equivalent to \(x\) at the location of the hole. In this way, we can accelerate the synthesis procedure without losing chances of generating repairs. In addition, we reduce syntax components using a simple heuristic that only searches for syntax patterns used in a correct program provided from an instructor. For example, given a teacher’s program (\(\text{let factorial n = if } (n=0) \text{ then } 1 \text{ else } n \times \text{factorial } (n-1)\)), the pattern (\(\text{if } ? \text{ then } ? \text{ else } ?\)) is used during the synthesis process, but the synthesizer does not use other syntax components such as \text{match}.

Third, using symbolic execution, we aim to prune away partial programs that are well-typed but will eventually fail to satisfy the given testcases. We run a partial program symbolically and
generate a constraint on its input and output relationship. If the constraint does not hold for any given testcase, we conclude that the partial program cannot be a solution. For example, suppose a testcase \{3 \mapsto 6\} and a partial program \(\text{let } \text{factorial} \ n \ = \ \text{match } n \ \text{with} \ | \ 0 \ \rightarrow \ 1 \ | \ x \ \rightarrow \ x \ \times \ x \ \times \ ?\) are given. Continuing search starting from the partial program would eventually fail, as the symbolic execution produces the unsatisfiable constraint \(3 \times 3 \times ? = 6\).

3 PROBLEM DEFINITION

Language. FixML targets a significant subset of OCaml. In particular, it has been designed to handle a variety of functional programming exercises including not only introductory but also more advanced problems. For example, FixML is expressive enough to cover all programming exercises used in a senior-level Programming Languages course taught by the authors of this paper.

The full language is given in Appendix B but, for brevity, we illustrate our approach with a small language in Fig. 5, where \(E\) and \(U\) denote labelled and unlabelled expressions. Expressions include variable \((x)\), function definition \((\lambda x. E)\), function application \((E_1 \ E_2)\), let-expression \((\text{let } x = E_1 \text{ in } E_2)\), recursive function definition \((\text{letrec } f(x) = E_1 \text{ in } E_2)\), user-defined type constructor \((c(E_1, \ldots, E_k))\), where \(c\) is the constructor name, and pattern-matching \((\text{match } E \text{ with } \overline{p_i \rightarrow E_i^k})\), where \(\overline{p_i \rightarrow E_i^k}\) denotes \(p_1 \rightarrow E_1 \ | \ \cdots \ | \ p_k \rightarrow E_k\). An exceptional syntactic element is a hole (\(\square\)), which is a placeholder that will get replaced by a complete expression by our algorithm. Initial programs do not contain holes. A pattern \((p)\) is a user-defined type constructor with variable arguments \((c(x_1, \ldots, x_k))\). Types \((\tau)\) include user-defined algebraic data types \((T)\), function types \((\tau_1 \rightarrow \tau_2)\), and type variable \((\alpha)\).

Let \(\mathcal{E}[E] : Env \rightarrow Val\) be the evaluation function, where an environment \(\rho \in Env = Id \rightarrow Val\) maps variables (\(Id\)) to values (\(Val\)) and values are user-defined values (\(Cnstr\)), functions (\(Closure\)), and recursive functions (\(RecClosure\)): \(Val = Cnstr + Closure + RecClosure\), where \(Cnstr = Id \times Val^*\) (constructor name and values), \(Closure = Id \times E \times Env\) (argument, body, and environment), and \(RecClosure = Id \times Id \times E \times Env\) (function name, argument, body, and environment). Semantics is standard and defined only for expressions without holes.

The Correction Problem. Our error-correction problem is defined with the four components:

\[(x, E, E_c, T)\]

where \(x\) is an input variable, \(E\) is a program written by a student, \(E_c\) is a correct implementation provided by an instructor, and \(T \in Val \times Val\) is a set of input-output testcases. We assume that functions are not the final value, \(x\) is the only variable that appears free in \(E\) and \(E_c\) (i.e., \(FV(E) = FV(E_c) = \{x\}\)), and the program \(E\) is incorrect with respect to the testcases:

\[\exists (v_1, v_0) \in T. \mathcal{E}[E]\left[\{x \mapsto v_1\}\right] \neq v_0.\]
We compute the score of suspicious subexpression size where a candidate \( e \). Then, our goal is to find a program \( E' \) that works correctly for all testcases:

\[
\forall (v_1, v_o) \in T \cdot E[E'[x \mapsto v_1]] = v_o.
\]

Note that our aim is not to simply return the correct implementation \( E_c \) but instead generates \( E' \) from \( E \). Our algorithm works even without \( E_c \), but we require it for extracting common syntax components (Section 4.3), which increases the efficiency of our algorithm.

### 4 ERROR CORRECTION ALGORITHM

In this section, we describe our error-correction algorithm (Algorithm 1). Section 4.1 explains the error localization procedure, which identifies a set of suspicious expressions that may contain errors. Our algorithm attempts to replace each of these expressions by correct one until it finds a program that passes all testcases. In Sections 4.2, 4.3, and 4.4, we formalize this correction process as a state-search procedure that is able to find a correction efficiently by combining type-directed enumeration, components reduction, and pruning by symbolic execution.

#### 4.1 Statistical Error Localization

The algorithm begins with localizing errors. To locate errors in a program, we adapt a well-known statistical technique [Jones et al. 2002] for functional programs. Given a program \( E \) which is buggy with respect to testcases \( T \), our technique produces a set of pairs, where each pair \( (E_o, s) \) consists of a holed program \( E_o \) and its score \( s \). \( E_o \) is obtained from \( E \) by substituting a suspicious subexpression of \( E \) with a hole (\( \square \)). The score \( s \) measures how suspicious the substituted expression is.

The localization procedure takes the input variable \( x \), buggy program \( E \), and testcases \( T \). It first classifies the testcases into positive (\( P \)) and negative (\( N \)) testcases. We say a testcase \( (v_i, v_o) \in T \) is positive if the program \( E \) with \( v_i \) correctly evaluates to \( v_o \). Otherwise, the testcase is called negative. The sets \( P \) and \( N \) are defined as follows:

\[
P = \{(v_i, v_o) \in T \mid E[E'[x \mapsto v_i]] = v_o\}, \quad N = \{(v_i, v_o) \in T \mid E[E'[x \mapsto v_i]] \neq v_o\}.
\]

Next, we collect subexpressions of \( E \) that are evaluated under the program execution with negative testcases. Let \( L \) be the set of all subexpressions of the program \( E \). We assume a predicate \( V : S \times T \rightarrow \{0, 1\} \) such that, given a subexpression \( e \) \( \in E \) and a testcase \( t \in T \), \( V(e, t) \) is 1 iff the expression \( e \) is evaluated when \( E \) runs under the testcase \( t \). Then, our localization procedure produces the following set of candidates for repair:

\[
C = \{(e^l, \text{hole}(E, l), \text{score}(e^l)) \mid e^l \in S \land \exists t \in N. \ V(e^l, t) = 1\}.
\tag{1}
\]

A candidate \( (e, E_o, s) \in C \) includes a potentially erroneous subexpression \( e \) of \( E \), a holed expression \( E_o \), and the score of \( e \). The function \( \text{hole}(E, l) \) replaces the subexpression \( e \), whose label is \( l \), of \( E \) by a hole, which is inductively defined in a straightforward way: e.g.,

\[
\text{hole}(x^l, l') = \begin{cases} 
\square & \text{if } l = l', \\
 x^l & \text{if } l \neq l'.
\end{cases}
\]

We compute the score of suspicious subexpression \( e \) as follows (lower is more suspicious):

\[
\text{score}(e) = \text{size}(e) + \frac{\sum_{p \in P} V(e, p) \cdot \sum_{t \in T} V(e, t)}{\sum_{t \in T} V(e, t)} \times \frac{\sum_{(e^l, s) \in C} \text{size}(e^l)}{|C|}
\]

where \( \text{size}(e) \) denotes the size of \( e \), which is heuristically estimated based on Occam’s razor. The idea is that we would like to repair as small a subexpression as possible (\( \text{size}(e) \)) and we presume that subexpressions that are less involved in positive test runs are more likely to be erroneous

\[
\left(\frac{\sum_{p \in P} V(e, p)}{\sum_{t \in T} V(e, t)}\right). \quad \text{The last term } \left(\frac{\sum_{(e^l, s) \in C} \text{size}(e^l)}{|C|}\right) \text{ is the normalization factor that scales the second term in accordance with the first term.}
\]
\[ x \in \Sigma(l) \quad S = \text{unify}(\Gamma(l), \Gamma(l)(x)) \]
\[
\begin{align*}
\langle l^1, \Gamma \rangle \rightsquigarrow_{\Sigma, \Omega} (x^1, S(\Gamma), S(\Gamma)) \\
\langle l^1, \Gamma \rangle \rightsquigarrow_{\Sigma, \Omega} (\langle \lambda x, \Box \rangle l^1, \Gamma[l_1 \mapsto \tau_1, \Gamma[l_2 \mapsto (\Gamma(l))(x \mapsto \tau_1)] \rangle) \\
\langle l^1, \Gamma \rangle \rightsquigarrow_{\Sigma, \Omega} (\langle \lambda x, \Box \rangle l^1, \Gamma[l_1 \mapsto \alpha, \alpha \neq \tau_1, \Gamma[l_2 \mapsto (\Gamma(l))(x \mapsto \alpha)] \rangle) \\
\langle l^1, \Gamma \rangle \rightsquigarrow_{\Sigma, \Omega} (\langle \lambda x, \Box \rangle l^1, \Gamma[l_1 \mapsto \alpha, \alpha \neq \tau_1, \Gamma[l_2 \mapsto (\Gamma(l))(x \mapsto \alpha)] \rangle) \\
\end{align*}
\]
\[
\text{Fig. 6. Type-directed transition relation.}
\]
4.2 Type-Directed Enumerative Search

After locating errors, the algorithm attempts to fix them by synthesizing correct expressions for holes in partial programs. To do so, it basically performs enumerative search, exhaustively generating all the possible expressions according to the language grammar in increasing size. However, this simple approach does not work as our language (Appendix B) is too large. To make it practical, we use three techniques: type-directed enumeration, components reduction, and pruning via symbolic execution.

We first describe type-directed enumerative search, which is defined by the transition system:

\[(S, \rightsquigarrow, I, F)\]

where \(S\) is a set of states, \((\rightsquigarrow) \subseteq S \times S\) is a transition relation, \(I \subseteq S\) is a set of initial states, and \(F \subseteq S\) is a set of final states.

**States.** A state \(s \in S\) is a triple \((E, \Upsilon, \Gamma)\), where \(E\) is a partial program that may have holes and \(\Upsilon\) and \(\Gamma\) are type environments for holes and variables. \(\Upsilon : \text{Hole} \rightarrow \text{Type}\) is a type environment that maps holes to their types. \(\Gamma : \text{Hole} \rightarrow (\text{Id} \rightarrow \text{Type})\) stores types of variables for each hole of the program. For example, when \(E_1 = (\lambda x. x + 1\text{ if } x \text{ is } \text{int})\), \(\Upsilon\) and \(\Gamma\) store the following information: \(\Upsilon = [\{\_\rightarrow \text{int}\}]\), \(\Gamma = [\{i \mapsto x \mapsto \text{int}\}]\). The type of hole \(i\) is \text{int} and variable \(x\) has type \text{int} at the hole \(l\). An important role of \(\Gamma\) is to keep track of variables that can appear free inside each hole. For example, when \(E_2 = (\lambda x. x = \text{int}\text{ if } y \text{ is } \text{int})\), \(\Gamma\) stores the information: \(\Gamma = [i_1 \mapsto [], i_2 \mapsto [x \mapsto \alpha, y \mapsto \alpha]]\), which indicates that no variables and variables \([x, y]\) can be used without definitions at the first and second holes, respectively. We write \(\text{dom}(\Gamma(l))\) for the set of variables available at the location \(l\): e.g., in \(E_2\), \(\text{dom}(\Gamma(i_1)) = \emptyset\) and \(\text{dom}(\Gamma(i_2)) = [x, y]\). In our algorithm, maintaining these two type environments enables to perform an efficient type-directed search.

**Initial and Final States.** The set \(I\) of initial states is defined as follows:

\[I = \{(E_0, \Upsilon_0, \Gamma_0) | \lambda E_0, \Upsilon_0, \Gamma_0 \in C, \text{ where } E_0 \text{ are initial type environments for } E_0\}\]

where \(C\) is the set of repair candidates in (1) that results from the localization procedure. \(\Upsilon\) and \(\Gamma\) can be easily obtained by running a standard type inference algorithm with a slight modification to assign a fresh type variable \((\alpha)\) to each hole in \(E_0\). The goal of search is to find a final state \((E, \Upsilon, \Gamma)\) such that \(E\) does not have any holes and is correct with respect to the given testcases: \(F = \{(E, \_\_\_) \in S | \nu(\nu_l, \nu_i) \in T. E[\nu]\}[\{x \mapsto \nu_l\}] = \nu_i\).

**Transition Relation.** Given a state \(s\), transition relation \((\rightsquigarrow) \subseteq S \times S\) determines the next states of \(s\). Fig. 6 defines the transition relation as a set of inference rules. In the definition, we assume the standard unification algorithm, \(\text{unify} : \tau \times \tau \times \text{Subst} \rightarrow \text{Subst}\), and the subscripts \(\Sigma\) and \(\Omega\) represent the syntax and variable components that we are allowed to use during synthesis. For the moment, we assume that the algorithm simply uses all syntax components of the language:

\[\Omega = \{\lambda x. \Box, \Box \Box, \text{letrec } f(x) = \Box \text{ in } \Box, \text{match } \Box \text{ with } p_1 \rightarrow \Box | \cdots | p_k \rightarrow \Box\}\]

and all available variables, i.e., \(\Sigma = \lambda l. \text{dom}(\Gamma(l))\).

Note that state transitions between states occur in a type-directed manner; ill-typed partial programs never appear during the transitions. For example, the rule

\[\frac{\lambda x. \Box \in \Omega \quad \Upsilon(l_1) = \tau_1 \rightarrow \tau_2 \quad \text{new } l_2}{(\Box^{l_1}, \Upsilon, \Gamma) \rightsquigarrow_{\Sigma, \Omega} \langle (\lambda x. \Box^{l_2})^{l_1}, \Upsilon[l_2 \mapsto \tau_2], \Gamma[l_2 \mapsto (\Gamma(l_1))[x \mapsto \tau_1]\rangle}\]

says that the hole \(\Box^{l_1}\) is able to get replaced by \(\lambda x. \Box^{l_2}\) only when the hole \(\Box^{l_1}\) is of a function type (i.e. \(\Upsilon(l_1) = \tau_1 \rightarrow \tau_2\) for some type \(\tau_1\) and \(\tau_2\)). The type environments \(\Upsilon\) and \(\Gamma\) are updated...
We further accelerate enumerative search by reducing the program components (i.e., workset element \( W \) variable components (line 12) and update variable-type information for the newly introduced hole \( \gamma \), \( \Gamma \) and \( \Gamma \) are initial environments for \( E \) ) according to: 

\[
\text{cost}(\langle E, \gamma, \Gamma \rangle, e, s) = \text{size}(E) + s.
\]

Intuitively, we would like to produce a minimal correction (size\( (E) \)) and make minimal changes from the original incorrect program \( s \), the suspicious score obtained in Section 4.1. If the chosen program \( E \) does not have any holes (line 6), we check whether \( E \) is a final state (line 7). If a final state is found, it is returned and the algorithm terminates (line 8). Otherwise, we check if the current program \( E \) is consistent with the given testcases (line 10). If so, we determine the variable components (line 12) and update \( W \) so that it includes all the next states according to the transition relation. During the algorithm, we exclude potentially non-terminating programs using three heuristics. Firstly, we do not allow recursive calls in every branch of a program, e.g., \( \text{letrec } f \ x = \text{match } x \text{ with } p_1 \rightarrow f \square \quad \text{or } p_2 \rightarrow f \square \). Secondly, we do not allow recursive calls with unchanging arguments, e.g., \( \text{letrec } f \ x = \text{match } x \text{ with } p_1 \rightarrow f \ x \quad \text{or } p_2 \rightarrow \square \). Finally, when we evaluate the candidate program (line 7), we abort its execution if it runs exceeding a predetermined time limit.

### 4.3 Reducing Program Components

We further accelerate enumerative search by reducing the program components (i.e., \( \Sigma \) and \( \Omega \) in Fig. 6). In particular, we reduce the variable components (\( \Sigma \)) by running a data-flow analysis that identifies semantically equivalent variables. To reduce the syntax components (\( \Omega \)), we focus on the components used in the correct implementation \( E_c \) provided by an instructor.

**Motivation.** One major factor that degrades the performance of enumerative search is the large set of program variables the algorithm is allowed to use during the synthesis process. Our
algorithm does not use all available variables but carefully chooses a subset of them without losing expressiveness. Consider the following code snippet:

```plaintext
type nat = ZERO | SUCC of nat
let rec id n =
  match n with
  | ZERO -> ZERO
  | SUCC n' -> (* hole with label l *)
```

At line 5, although two variables \( n \) and \( n' \) can be used by the synthesis procedure, we do not use \( n \) because \( n \) can be expressed in terms of \( n' \), i.e., \( n = \text{SUCC} \ n' \) at the hole. This way, our algorithm avoids to enumerate all available variables, increasing the efficiency significantly, but doing so does not lose any chance of finding a solution.

**Data-Flow Analysis.** To do so, we designed a simple data-flow analysis that identifies semantically equivalent variables (or patterns). The goal of analysis is to produce the following map:

\[
M : \text{Lab} \rightarrow \varphi((\text{Id} + \text{Pat}) \times (\text{Id} + \text{Pat})).
\]

Given a label \( l \in \text{Lab} \), \( M(l) \) stores the set of equivalent variables (or patterns) right before evaluating the \( l \)-labelled subexpression. For example, in the code snippet above, \( M(l) = \{(n, \text{SUCC} \ n')\} \), meaning that \( n \) and \( \text{SUCC} \ n' \) always have the same value right before the label \( l \). Formally, we write \( M \models E \) for the case that \( M \) is correct with respect to the expression \( E \). The correctness specification is given in Fig. 7. The specification can be easily translated to a top-down data-flow analysis algorithm. For example, when \( E = (E_1^l, E_2^l)^l \), we propagate the equivalence relation at \( l \) to both \( l_1 \) (i.e. \( M(l) \subseteq M(l_1) \)) and \( l_2 \) (i.e. \( M(l) \subseteq M(l_2) \)). Facts are killed and generated at the function definitions, \( \text{let}-\text{binding}, \) and \( \text{match} \). For example, consider \( E = (\text{let} \ x = E_1^l \text{ in } E_2^l)^l \). Starting from the facts \( M(l) \) at \( l \), we update the relationships by removing old facts \( (\text{kill}(M(l), \{x\})) \), generating new facts \( (\text{gen}(x, E_1^l)) \), and closuring the result. The \( \text{gen} \) and \( \text{kill} \) functions are responsible for generating and removing facts according to the semantics of expressions.

**Using the Analysis Result.** With the analysis, we refine the variable components \( (\Sigma) \) as follows. Given the current partial program \( E \) during the search procedure (line 11 in Algorithm 1), we analyze the program and obtain the information \( M_E \) about equivalent variables. With \( M_E \), our algorithm uses the following sets of variables for each label \( l \):

\[
\text{VarComponents}(E, \Gamma) = \lambda l. \text{dom}(\Gamma(l)) \setminus \{x \in \text{Id} \mid (\_, x) \in M_E(l)\}.
\]

That is, we use variables if they do not appear in constructors (i.e. \( x \in \text{Id} \)) and they can be replaced by other variables (i.e. \( (\_, x) \in M_E(l) \)). Let us justify the second condition. For each pair \( (v_1, v_2) \in M(l) \), we have the three possible cases:

- **When** \((v_1, v_2) = (y, x)\): It is fine to exclude the variable \( x \), because we can express \( x \) by a more recently defined variable \( y \).
- **When** \((v_1, v_2) = (c(x_1, \ldots, x_k), x)\): It is also fine to exclude the variable \( x \), because we can express \( x \) using the variables \( x_1, \ldots, x_k \) with the constructor \( c \).
- **When** \((v_1, v_2) = (x, c(x_1, \ldots, x_k))\): We should not remove the variables \( x_1, \ldots, x_k \), because they cannot be expressed by \( x \). Instead, we can remove \( x \), as we do in the second case. However, we also use \( x \) during the synthesis process, considering the readability of synthesized results (more recently defined variables are more likely to be used later).
The final component of our algorithm is pruning based on symbolic execution.

Syntax Components. In addition to reducing variable components, we reduce syntax components as well. Note that our full language in Appendix B is too large to perform exhaustive search. For instance, the language has 35 different syntactic cases for expressions. To reduce the search space more effectively, we used a simple heuristic that only searches through the syntax components that appear in the correct implementation provided by an instructor. Given the correct implementation $E_c$, we extract its components using $E : \text{Exp} \rightarrow \wp(\text{Exp})$ as follows:

\[
E(x) = \emptyset \quad E(\lambda x. E) = \{\lambda : \Box \} \cup E(E) \quad E(\text{let } x = E_1 \text{ in } E_2) = \{\Box \} \cup E(E_1) \cup E(E_2) \\
E(\text{letrec } f(x) = E_1 \text{ in } E_2) = \{\text{letrec } f(x) = \Box \} \cup E(E_1) \cup E(E_2) \\
E(\text{match } E \text{ with } p_1 \rightarrow E_1 \mid \cdots \mid p_k \rightarrow E_k) = \{\text{match } \Box \text{ with } p_1 \rightarrow \Box \mid \cdots \mid p_k \rightarrow \Box \} \cup E(E) \cup \bigcup_{i=1}^k E(E_i)
\]

Our algorithm uses the function SyntaxComponents($E_c$) = $E(E_c)$ at line 2 of Algorithm 1.

4.4 Pruning by Symbolic Execution

The final component of our algorithm is pruning based on symbolic execution.

Motivation. Consider the following partial program:

```
1  let rec id n =
2      match n with
3      | ZERO -> Succ ?
4      | Succ n' -> Succ n'
```

Suppose that we have an input-output testcase \{ZERO \mapsto ZERO\}, which means that the function `id` on ZERO must evaluate to ZERO. Note that although the partial program has no type error, it will never match the given input-output behavior because `Succ e` cannot be equal to ZERO regardless of the expression `e`. We detect such semantic discrepancies by running a symbolic executor on the partial programs encountered during the search algorithm.

Symbolic Execution Rules. We designed a symbolic executor that efficiently detects partial programs that are inconsistent with respect to the given testcases. The semantics for symbolic

\[
M \models x^I \\
M \models \Box^I \\
M \models (E_1, E_2)^I \quad \text{iff } M(l) \subseteq M(l_1) \land M(l) \subseteq M(l_2) \\
M \models (\lambda x. E)^I \quad \text{iff } \text{kill}(M(l), \{x\}) \subseteq M(l_1) \\
M \models (c(E_1, \ldots, E_k))^I \quad \text{iff } \forall 1 \leq i \leq k. M(l) \subseteq M(l_i) \\
M \models (\text{let } x = E_1 \text{ in } E_2)^I \quad \text{iff } M(l) \subseteq M(l_1) \land \text{kill}(M(l), \{x\}) \cup \text{gen}(x, E_1)^* \subseteq M(l_2) \\
M \models (\text{letrec } f(x) = E_1 \text{ in } E_2)^I \quad \text{iff } M(l) \subseteq M(l_1) \land \text{kill}(M(l), \{f\}) \subseteq M(l_2) \\
M \models (\text{match } E_0^I \text{ with } p_1 \rightarrow E_1^I \mid \cdots \mid p_k \rightarrow E_k^I)^I \quad \text{iff } M(l) \subseteq M(l_0) \land \forall 1 \leq i \leq k. \text{kill}(M(l), \{p_i\}) \cup \text{gen}(p_i, E_0)^* \subseteq M(l_i)
\]

Fig. 7. Data-flow analysis for components reduction
which can be checked using an off-the-shelf SMT solver.

Automatic Error Correction for Functional Programming Assignments

(i.e. FixML ϕ first generate the following formula

\[ \rho \vdash E_1 \Rightarrow \nu_1 \quad \ldots \quad \rho \vdash E_k \Rightarrow \nu_k \]

Given a partial program \( E \) and a set \( T \) of testcases, we first generate the following formula \( \phi_{E,T} \) by running the symbolic executor on \( E \) for each testcase \((v_i, v_o) \in T : \phi_{E,T} = (v_i \rightarrow v_o) \) equal \( (S(E)[x \mapsto v_i], v_o) \), where the function equal relates the input and output values and produces a formula as follows:

\[
\begin{align*}
\text{equal}(\beta, c(\nu_1, \ldots, \nu_k)) &= (\beta = c(\nu_1, \ldots, \nu_k)) \quad (\beta \in \mathbb{S}) \\
\text{equal}(c(\nu_1, \ldots, \nu_k), c'(\nu'_1, \ldots, \nu'_k)) &= (c = c') \land \bigwedge_{1 \leq i \leq k} \text{equal}(\nu_i, \nu'_i) \\
\text{equal}(\_, \_) &= \text{false}
\end{align*}
\]

Then, we prune out the program \( E \) if and only if \( \phi_{E,T} \) is unsatisfiable:

\[
\text{Inconsistent}(E, T) \iff \phi_{E,T} \text{ is unsatisfiable}
\]

which can be checked using an off-the-shelf SMT solver.

5 EVALUATION

In this section, we evaluate FrXML aiming at answering the following research questions:

- **Effectiveness:** How effectively can FrXML correct real student submissions? Is it applicable to a wide range of functional programming assignments? Is it efficient enough to provide quick feedback to students?
Table 1. Performance of FixML on 497 student submissions. #P reports the number of target programs for each problem. #T reports the number of testcases used for each problem. LOC reports the number of lines in submissions averaged over all submissions as well as the LOCs of the smallest and largest submissions. Time reports average time took for localizing and fixing errors in seconds. Fix Rate reports the ratio of successfully fixed programs by FixML.

<table>
<thead>
<tr>
<th>No</th>
<th>Problem Description</th>
<th>#P</th>
<th>#T</th>
<th>LOC (min-max)</th>
<th>Time</th>
<th>Fix Rate (#Fix)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Filtering elements satisfying a predicate in a list</td>
<td>3</td>
<td>10</td>
<td>6 (6-7)</td>
<td>13.0</td>
<td>100% (3)</td>
</tr>
<tr>
<td>2</td>
<td>Finding a maximum element in a list</td>
<td>32</td>
<td>10</td>
<td>8 (4-14)</td>
<td>0.2</td>
<td>100% (32)</td>
</tr>
<tr>
<td>3</td>
<td>Mirroring a binary tree</td>
<td>9</td>
<td>10</td>
<td>10 (9-14)</td>
<td>0.1</td>
<td>89% (8)</td>
</tr>
<tr>
<td>4</td>
<td>Checking membership in a binary tree</td>
<td>15</td>
<td>17</td>
<td>11 (9-18)</td>
<td>5.2</td>
<td>80% (12)</td>
</tr>
<tr>
<td>5</td>
<td>Computing $\sum_{j=k}^{i} f(i)$ for $j$, $k$, and $f$</td>
<td>23</td>
<td>11</td>
<td>5 (2-9)</td>
<td>4.2</td>
<td>78% (18)</td>
</tr>
<tr>
<td>6</td>
<td>Adding and multiplying user-defined natural numbers</td>
<td>34</td>
<td>10</td>
<td>20 (10-50)</td>
<td>20.6</td>
<td>59% (20)</td>
</tr>
<tr>
<td>7</td>
<td>Finding the number of ways of coin-changes</td>
<td>9</td>
<td>10</td>
<td>21 (6-35)</td>
<td>2.6</td>
<td>44% (4)</td>
</tr>
<tr>
<td>8</td>
<td>Composing functions</td>
<td>28</td>
<td>12</td>
<td>7 (1-19)</td>
<td>5.5</td>
<td>43% (12)</td>
</tr>
<tr>
<td>9</td>
<td>Implementing a leftist heap using a priority queue</td>
<td>20</td>
<td>13</td>
<td>43 (33-72)</td>
<td>2.6</td>
<td>40% (8)</td>
</tr>
<tr>
<td>10</td>
<td>Evaluating expressions and propositional formulas</td>
<td>101</td>
<td>17</td>
<td>32 (17-90)</td>
<td>1.2</td>
<td>39% (39)</td>
</tr>
<tr>
<td>11</td>
<td>Adding numbers in user-defined number system</td>
<td>14</td>
<td>10</td>
<td>52 (19-138)</td>
<td>7.0</td>
<td>36% (5)</td>
</tr>
<tr>
<td>12</td>
<td>Deciding lambda terms are well-formed or not</td>
<td>86</td>
<td>11</td>
<td>28 (13-79)</td>
<td>1.3</td>
<td>26% (22)</td>
</tr>
<tr>
<td>13</td>
<td>Differentiating algebraic expressions</td>
<td>123</td>
<td>17</td>
<td>36 (14-154)</td>
<td>11.4</td>
<td>25% (31)</td>
</tr>
</tbody>
</table>

|                  | Total / Average                                         | 497| 158| 27 (2-154) | 5.4   | 43% (214)       |

- **Helpfulness**: How helpful is FixML for students? How often do students produce logical errors in functional programming problems? How difficult is it to correct those errors by themselves? Does FixML help students to better understand the errors?
- **Utility of Techniques**: How useful is the combination of various techniques in Section 4? Does it significantly outperform simpler approaches?

FixML is written with ~6400 lines of OCaml code. We used the Alt-Ergo Zero SMT solver\(^3\) to check the satisfiability of constraints generated by symbolic execution (Section 4.4). All experiments were done on an iMac with Intel i5 CPU and 16GB memory.

### 5.1 Effectiveness

**Benchmarks.** We have evaluated the effectiveness of FixML with 497 programs with logical errors collected from 13 functional programming assignments used in a Programming Languages course over the last few years. Those programs have logical errors only, not syntax or type errors.

The problem descriptions are presented in Table 1. We classify the benchmark problems into introductory (#1–#5), intermediate (#6–#9), and advanced (#10–#13) problems. The problems at introductory-level let students to experience basic functional programming concepts such as recursion, inductive datatypes, and high-order functions. Solving intermediate problems needs relatively more challenging programming skills. The advanced problems require familiarity in programming language concepts or algorithmic skills. For example, problem #10 asks to write a simple interpreter for a language with arithmetic expressions and propositional formulas and problem #12 requires to understand the concept of lambda calculus.

Note that our benchmark problems are comparatively more challenging than those typically used in prior work. Previous feedback generation systems mostly focus on problems at introductory-level

---

3[http://cubicle.lri.fr/alt-ergo-zero](http://cubicle.lri.fr/alt-ergo-zero)
Automatic Error Correction for Functional Programming Assignments

onl, such as programming tasks that manipulate integers or arrays (e.g. reversing numbers, finding the $k$th largest element, etc) [Gulwani et al. 2018; Singh et al. 2013; Wang et al. 2018]. In this paper, we aim for more sizable programs up to 100 lines of code.

In Table 1, we report the number of incorrect submissions we could collect for each problem (#P), the number of testcases (#T), (average, smallest, largest) LOCs of the submissions (LOC), the average time took to generate corrections (including localization), and the ratio of successfully fixed submissions for each problem with respect to the entire programs (Fix Rate).

Results. The results in Table 1 indicate that FixML is powerful and capable of fixing logical errors in real student submissions. In summary, FixML successfully fixed 214 out of 497 submissions in 5.4 seconds on average. For introductory-level problems (#1–#5), FixML fixed most (89%, 73/82) of the submissions in 2.5 seconds on average. For intermediate-level problems (#6–#9), the fix rate was 48% (44/91) and the average time for fixing was 11.6 seconds. For problems at advanced level (#10–#13), FixML was able to correct 30% (97/324) in 4.8 seconds on average. Although the fix rates decrease in problems at advanced-level, the results are still impressive, considering the size and complexity of the benchmark programs. For example, FixML managed to accurately localize and repair an error in the largest submission (154 lines) presented in Appendix C.

In Table 1, all the generated patches are correct; the patched programs are semantically equivalent to the reference programs. In test-based repair, generated patches are likely to overfit the given testcases [Smith et al. 2015]. In this work, we avoided this issue by manually refining the testcases until the repaired programs become semantically equivalent to the reference implementation. We started with 130 testcases (10 testcases for each problem), which resulted in 58 overfitted patches (27%), and increased the set to 158 testcases through the refinement process. The more detailed discussion is included in Section 5.4.

Moreover, we have manually checked the quality of the patches. Even though the repaired programs are semantically correct, they might not be ideal for various reasons (e.g., readability, optimality, etc.). We found that 25 out of 214 are not ideal. However, 23 of them could be easily transformed into ideal ones via simple postprocessing. For example, consider the function `mem` (Problem 4 in Table 1) written by a student, which is supposed to check whether `n` is stored in the given binary tree (`tree`):

```ml
let rec mem (n:int) (tree: btree) : bool =
match tree with
| Empty -> false
| Node (m, left, right) ->
  if n = m then true
  else if n < m then (mem n left) (* Feedback : Replace "n < m" by "(mem n left)" *)
  else mem n right
```

The program has an error at line 6, where the student misunderstood that a binary search tree, not a plain binary tree, was given. FixML corrected the conditional expression ($n < m$) by $(mem n left)$, which checks the membership in both left and right subtrees. The patch, however, is undesirable since the $(mem n left)$ is redundant. In this case, the fix can be modified through post-processing, which replaces "$if (mem n left) then mem n left else mem n right$" by "$(mem n left) || (mem n right)$" based on the fact that the conditional expression is syntactically equivalent to the true branch. The remaining two cases were not ideal because of the imprecision of error localization. We describe this case in the first paragraph ("Accurate Localization") of Section 5.4.

4We manually checked the correctness of the 214 fixes.

5.2 Helpfulness

To assess how helpful FixML is for students, we have conducted user study with 18 undergraduate students who took the Programming Language course taught by the authors.

**Scenario.** The user study proceeded for one hour and consisted of three parts. First, we asked the students to solve the three programming problems not included in the problem set of the course:

1. **Problem 1 (append):** Write a function, which takes two lists and appends the first list to the second one while removing duplicating elements (Example 3 in Section 2.1).
2. **Problem 2 (uniq):** Write a function, which removes duplicated elements from a given list so that the list contains unique elements. For instance, given input \([5;6;5;4]\) the expected output is \([5;6;4]\).
3. **Problem 3 (dropWhile):** Write a function, which takes a predicate and a list, and removes the elements of the list until an element on which the predicate is false is found. For example, \(\text{dropWhile} (\text{fun } x \rightarrow x \mod 2 = 0) [2;4;7;6;9]\) outputs \([7;6;9]\).

Because solving all of the problems from scratch with the limited time is a bit challenging, we provided templates for each problem. The templates include helper functions and students are allowed to use them to complete the programs. For example, we used the following template for Problem 2:

```ocaml
let rec uniq_help (l:int list) (n:int) : int list =  
  match l with  
  | [] -> [] | h::t -> if n = h then uniq_help t n else h::(uniq_help t n)  
let rec uniq (x:int list) : int list =  
  match x with  
  | [] -> (* Blank *)  
  | h::t -> (* Blank *)
```

where the job of the students is to replace the two blanks by appropriate expressions. We collected the answers by students and counted the number of incorrect submissions for each problem.

In the second part of the user study, students were asked to fix their errors by themselves using counterexamples provided by the instructor. The last task is for students to use FixML and evaluate it based on how useful FixML is to better understand their mistakes. The survey questions used for tool evaluation are in Table 2.

**Results.** Fig. 9 summarizes the results from the first and second parts of user study. Fig. 9a shows that most of the students felt difficulty in solving the problems, even though templates...
Table 2. Questions used in the survey and corresponding responses from 18 undergraduates. Q1 targeted 12 students who had succeeded in correcting their errors by themselves for at least one problem. Q2 targeted 12 students who failed to fix their errors by themselves for at least one problem. Q3 targeted all 18 students. \((n)\) shows the number of actual responses.

<table>
<thead>
<tr>
<th>Question</th>
<th>Yes</th>
<th>Neutral</th>
<th>No</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1. Compared to your own corrections, does FixML generate better corrections?</td>
<td>67% (8)</td>
<td>33% (4)</td>
<td>0% (0)</td>
</tr>
<tr>
<td>Q2. Does FixML help to understand your mistakes when you cannot resolve them by yourself?</td>
<td>50% (6)</td>
<td>42% (5)</td>
<td>8% (1)</td>
</tr>
<tr>
<td>Q3. Is FixML overall useful in learning functional programming?</td>
<td>72% (13)</td>
<td>28% (5)</td>
<td>0% (0)</td>
</tr>
</tbody>
</table>

were given. In Problem 1 (append), 94% (17/18) of the students failed to correctly implement the desired functionality. Even in the best case (dropWhile), 44% (8/18) of the students submitted buggy programs. All submitted programs are free of syntax and type errors and only contain logical errors.

Fig. 9b shows that providing error-triggering testcases only does not much help students to figure out their mistakes. Most students failed to fix their errors by themselves using the provided testcases, although we allowed them to use additional time as long as they requested. At best, 45% of the students were able to correct their errors (Problem 2 (uniq), 5/11). On the other hand, FixML was able to correct most of the errors introduced by the students. Remarkably, FixML successfully corrected all of the errors for Problem 1 (append).

The results in Table 2 shows that FixML can be overall useful for students. We let students to experience FixML and evaluate how helpful it is in assisting diagnosing and understanding their mistakes. We asked three questions that aim to assess the following aspects: (Q1) qualities of the feedback by FixML; (Q2) helpfulness in understanding their mistakes; and (Q3) overall usefulness in learning functional programming. For Q3, we asked students to additionally justify their responses.

For Q1, we targeted 12 students who succeeded in correcting their errors by themselves for at least one problem. All of them recognized that FixML provides corrections of better or similar qualities compared to their own ones. Indeed, from further inspections of the students’ corrections, we found that the students’ corrections are often verbose (e.g. defining multiple redundant functions that have the same functionality) and introduced new errors as they are likely to produce corrections overfit to the testcases given by the instructor. On the other hand, the FixML-generated corrections were minimal and easy-to-understand.

For Q2, we targeted 12 students who failed to fix their errors by themselves for at least one problem. Among 12 students, only one responded that FixML did not help to understand her mistakes, while the rest 11 answered that they were able to understand the reasons for their mistakes based on FixML’s corrections.

For Q3, we targeted all 18 students. 72% of them agreed that FixML is definitely useful for learning functional programming with the following comments: “FixML can play a role as an automated teacher by pinpointing failure points and providing corresponding corrections”, “Students can immediately resolve their troubles thanks to the realtime-feedback of FixML”, “FixML’s corrections are concise and valuable for enhancing programming skills”, etc. The 28% of students (5 students) in Q3 overall acknowledged the usefulness of FixML, but responded ‘Neutral’ due to the following reasons. Two of them worried that students are likely to become dependent on FixML without trying to solve the problems on their own. Other two students replied that they may be able to fix
errors by themselves in most cases without FixML. The last one responded that FixML may not be able to help students who have difficulties in designing algorithms at early stages.

5.3 Utility of Techniques

FixML could not be successful without the combination of the techniques presented in Section 4. In this section, we justify this claim by evaluating the impact of major techniques used in FixML.

In Fig. 10, the line labelled ‘Ours’ depicts the cumulative repair time of FixML when we applied all techniques (statistical error localization, type-directed search, component reduction, and pruning by symbolic execution). ‘Comp’ represents the performance of FixML except the symbolic execution (statistical error localization, component reduction, type-directed search). ‘Type’ shows the result for FixML with statistical error localization and type-directed search, which may provide a glimpse of the current state-of-the-art functional program synthesizers [Feser et al. 2015; Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016] whose key technology is type-based pruning. ‘Localize’ shows the performance of the algorithm that uses statistical error localization and plain enumerative search without type-based pruning. ‘Enum’ represents the performance of FixML when it performs plain enumerative search but does not perform statistical reasoning in localizing errors (i.e., the suspicious score in Section 4.1 is computed as score(e) = size(e)). For each submission, time limit for repair was one-minute.

The results show that the combination of all techniques (‘Ours’) remarkably improves the performance of the type-directed search (‘Type’) on both effectiveness (i.e. the number of successful fixes) and efficiency (i.e. the repair time when compared for the same number of benchmarks). Specifically, while ‘Type’ corrected only 160 submissions, ‘Ours’ corrected 214 submissions. In terms of efficiency, while ‘Type’ spent 579 seconds for fixing the first 160 submissions, ‘Ours’ do so for the same set of submissions in 65 seconds.

The competitiveness of ‘Ours’ over ‘Type’ is noteworthy, because the type-directed search itself is already well-tuned compared to ‘Localize’. ‘Localize’ was able to repair only 118 submissions and it took 796 seconds to do so, whereas ‘Type’ only required 50 seconds to fix the same 118 submissions.

Additionally, the result shows that both component reduction and symbolic execution have significant impacts on the performance. By comparing ‘Comp’ to ‘Type’ in Fig. 10, component reduction made it possible to fix 45 more submissions and only spent 78 seconds to fix 160 patches, whereas ‘Type’ spent 579 seconds to fix the same number of programs. Even though the component
reduction remarkably speed up the tool, symbolic execution can still enhance FixML more. ‘Ours’ repaired 214 submissions while ‘Comp’ corrected 205 submissions. Furthermore, ‘Ours’ took 706 seconds to fix 205 programs in contrast to ‘Comp’ which required 1052 seconds.

Also, the difference between ‘Localize’ and ‘Enum’ tells that applying statistical error localization is useful for accelerating the repair time. ‘Enum’ repaired 113 programs, while ‘Localize’ repaired 118 programs. Furthermore, while ‘Enum’ spent 884 seconds for repairing 113 programs, ‘Localize’ only spent 524 seconds for repairing the same set of the programs. Note that, since our localization procedure is currently based on simple statistical reasoning, there still exists a lot of room for improvements. We will discuss more details in Section 5.4

5.4 Discussion 1: Limitations and Future Work

In experiments, a number limitations of FixML has been identified. Accordingly, we plan to improve FixML in the following directions.

**Accurate Localization.** More accurate error-localization is needed to produce better corrections. For example, consider the filter function (Problem 1 in Table 1) written by a student:

```ml
let rec filter pred lst =
    let rec check l r =
        match l with
        | [] -> r | h::t -> if pred h then (check t r)@[h] else check t r
    in check lst []
```

The student implemented filter by defining a helper function check and invoking it. Function check has a bug at line 4, where it appends the head (h) at the end of the list. As a result, for example, filter (fun x -> x mod 2 = 0) [1;2;3;4] produces [4;2], not [2;4]. To fix the bug, the expression (check t r)@[h] should be replaced by h::(check t r). However, FixML pointed the call expression (check lst []) at line 5 as the error location and fixed the error by replacing the expression by (check (check lst []) []). Although this is a semantically correct fix, it might not be ideal feedback for students.

**Fixing Multiple Errors.** Currently, FixML focuses on fixing programs with a single error location. Thus, it fails to produce corrections for the following submission (Problem 11 in Table 1):

```ml
let rec eval f =
    match f with
    | True -> true | False -> false
    | Not e -> if e = True then false else true
    | AndAlso (e1, e2) -> if e1 = True && e2 = True then true else false
    | OrElse (e1, e2) -> if e1 = False && e2 = False then false else true
    | Imply (e1, e2) -> if e1 = True && e2 = False then false else true | ...
```

where eval takes a propositional formula f and intends to evaluate its truth value using recursion. However, the implementation is buggy as it consistently missing recursive calls in all necessary places; the expressions True, False, e, e1, and e2 at lines 4–7 must be replaced by true, false, eval e, eval e1, and eval e2, respectively. We plan to enable FixML to iteratively fix similar bugs in multiple locations.

**Automatic Testcase Generation.** To improve usefulness, we realized that FixML should be combined with automatic testcase generation, as its effectiveness depends on the quality of given testcases. For example, consider the function sigma (Problem 5 in Table 1) written by a student:

```ml
let rec sigma f a b =
    if f a != f b then f b + sigma f a (b-1) else f b
```
Given \( f, a, \) and \( b, \) \( \sigma f a b \) should compute \( \sum_{i=a}^{b} f(i). \) The implementation works correctly as long as \( f \) is injective (e.g., \( f = \text{fun} \ x \to x*x \)). However, it becomes incorrect when, for example, \( f \) is \( (\text{fun} \ x \to x \mod 3). \) The bug is at line 2, where the condition \( f a \neq f b \) must be replaced by \( a \neq b. \) At an early stage of this work, FixML was not able to find this error, as such a counterexample was missing in our testcases. We plan to automatically generate high-quality testcases, which then enables FixML not to miss tricky bugs.

**Scalability.** Although we have improved scalability significantly, FixML still needs more advanced program synthesis techniques to generate complex repairs. For instance, FixML could not fix the student’s submission below (Problem 13 in Table 1):

```ocaml
let rec diff (e, x)=
    match e with
    | Times lst ->
      (match lst with
        | [] -> Const 0
        | hd::tl -> Times [diff (hd,x); diff (Times tl,x)] |...
```

where a desirable fix is to replace \( \text{Times}[\text{diff}(hd,x); \text{diff}(\text{Times} tl,x)] \) (i.e., \( fg' = f'g' \)) at line 6 with \( \text{Sum}[\text{Times} [[\text{diff}(hd,x)]; \text{Times}(\text{hd}; \text{diff}(\text{Times} tl,x))]] \) (i.e., \( fg' = f'g + fg' \)). However, FixML failed to produce the correction even after 15 minutes. We plan to develop more powerful synthesis techniques to generate sizable repairs.

### 5.5 Discussion 2: Similarity between Correct and Incorrect Submissions

A recent trend in automatic feedback generation systems is data-driven approach [Gulwani et al. 2018; Pu et al. 2016; Wang et al. 2018], where a number of correct implementations that are similar to a given buggy program are exploited to find appropriate candidates for corrections. Readers might wonder FixML can benefit from this promising and complementary approach. To answer this question, we analyzed similarities between correct and incorrect programs submitted for our 13 benchmark problems in Table 1.

The analysis result in Fig. 11 implies that although the data-driven approach can be effective for introductory-level problems, it would be unsuitable for more complex programs. In this study, we checked whether we could find any similar programs for each incorrect submission using Moss [Schleimer et al. 2003], a system for quantifying similarities between programs.\(^5\) For each

---

\(^5\) For OCaml, we used the implementation available at [https://github.com/Chris00/ocaml-moss](https://github.com/Chris00/ocaml-moss)
erroneous submission, we provided Moss with the buggy program and a set of correct submissions submitted for the same problem. Then, Moss ranks the correct submissions based on the similarity score with respect to the buggy program. We repeated this process for each incorrect submission $P$, and stored $(P, s)$, where $s$ is the score of the correct program with the highest similarity. In total, we used 385 incorrect programs and 4350 correct programs. Fig. 11 depicts the data obtained from this study, where for each $(P, s)$, the x-axis shows the size of $P$ and y-axis reports $s$. The result indicates that we can hardly find similar correct submissions as the program size increases. Specifically, for the programs larger than 20 lines (266 among 385), which include most of our intermediate and advanced benchmarks, 85% (226/266) of them had highest similarity scores less than 40. According to our experience, the similarity score 40 means that the two programs are substantially different from each other. That is, we were frequently unable to find useful correct programs similar to a given incorrect program.

6 RELATED WORK

In this section, we survey recent researches that are closely related to our work.

**Debugging functional programs.** For functional languages, prior works have focused on localizing and fixing type errors [Chen and Erwig 2014; Lerner et al. 2007; Pavlinovic et al. 2014, 2015; Seidel et al. 2017; Wu et al. 2017; Wu and Chen 2017; Zhang et al. 2017]. For example, Seidel et al. [2017] recently proposed a data-driven approach to localize type errors in OCaml programs. Wu et al. [2017]; Wu and Chen [2017] studied how type errors are fixed and presented Learnskell for producing user-friendly feedback on type errors. Pavlinovic et al. [2014, 2015] presented SMT-based techniques for localizing type errors. Chen and Erwig [2014] presented a method for fixing type errors by generating possible changes and checking the them using type checker. Zhang et al. [2017] present a technique for localizing errors detected by static analyses such as type system. Unlike these works, our goal is to localize and fix logical errors in functional programs. The technique by Kneuss et al. [2015] is able to repair logical error of functional programs but it requires user-provided specifications.

**Automated feedback generation.** Recently, a large amount of work has been devoted to providing feedback on logical errors [D’Antoni et al. 2016; Gulwani et al. 2018; Kim et al. 2016; Pu et al. 2016; Singh et al. 2013; Wang et al. 2018], type errors [Seidel et al. 2017; Wu et al. 2017; Wu and Chen 2017], syntax errors [Bhatia et al. 2018; Gupta et al. 2017], and performance problems [Gulwani et al. 2014] in students’ programs. AutoGrader [Singh et al. 2013], which repairs students’ programs using predetermined correction rules provided by instructors, has inspired recent advances in automatic feedback generation. QLOSE [D’Antoni et al. 2016] aims to generate repairs that are optimal in that the resulting repaired program is most similar to the original buggy program. APEX [Kim et al. 2016] provides explanations for root causes of logical errors in students’ programs, but does not attempt to repair the errors. Recent data-driven approach, which exploits a number of correct programs to find appropriate candidates for corrections, has shown to be effective to introductory-level programming problems [Gulwani et al. 2018; Pu et al. 2016; Wang et al. 2018]. Our work lies in this line of research but our goal and techniques are different from those in existing works: 1) FixML is the first system aiming for fixing logical errors in functional programs while most feedback generation systems are for imperative languages, and 2) we present an error-correction algorithm based on statistical localization and enhanced type-directed synthesis, which is able to accurately produce useful feedback beyond introductory-level problems.

---

6 Among the 497 incorrect submissions in Table 1, we failed to run Moss on 112 programs.
Program synthesis. Program synthesis has been used in many application domains such as string-processing [Gulwani 2011; Kini and Gulwani 2015], data processing [Feng et al. 2017a], complex APIs [Feng et al. 2017b], database [Yaghmazadeh et al. 2017], and many others. Our work uses program synthesis for generating feedback on functional programs.

In particular, our work leverages recent advances in functional program synthesis, where type system has played a key role for accelerating functional program synthesizers by pruning ill-typed programs from the search space [Feser et al. 2015; Frankle et al. 2016; Osera and Zdancewic 2015; Polikarpova et al. 2016]. However, in our experience, the type-based pruning alone was not enough to be scalable, hence we applied two additional techniques: reduction of semantically redundant components and pruning search space using symbolic execution. There are also some other functional program synthesizers. Escher [Albarghouthi et al. 2013] performs heuristic goal-directed search, where a final solution is constructed by combining subsolutions with if-then-else statement. The proposed approach is hard to be generalized into a significant subset of OCaml that we target. Leon [Kneuss et al. 2013] aims to synthesize recursive functional programs verified with proofs, where specifications should be given as predicates in first-order formula unlike ours.

Techniques used by So and Oh [2017] and Balog et al. [2017] would help to improve performance of our system. So and Oh [2017] use abstract interpretation for pruning search space in the context of imperative program synthesis. Balog et al. [2017] use deep learning to optimize cost models used in search-based synthesis. We believe that using abstract interpretation and machine learning can significantly improve our results, which we leave as future work.

Error localization. The researchers in [Jose and Majumdar 2011] proposed a localization method based on MAX-SAT problem solving. However, the proposed method may not be directly applicable to our setting, because it is hard to generate constraints on high-order functions precisely and statically. Instead, we adapt a well-known statistical fault localization method [Ball et al. 2003; Griesmayer et al. 2007; Groce et al. 2006; Jones et al. 2002; Renieris and Reiss 2003] for functional programs.

Automatic program repair. Aside from the researches for repairing students’ programs, there also have been many recent researches for fixing bugs in large software, e.g., [Forrest et al. 2009; Kim et al. 2013; Kneuss et al. 2015; Könighofer and Bloem 2011; Le Goues et al. 2012; Long and Rinard 2016; Nguyen et al. 2013; Weimer et al. 2009]. The techniques for repairing students’ programs and developers’ software are complementary; while the former can generate relatively complicated patches for small programs, the latter usually produces small patches for large programs. For example, SemFix [Nguyen et al. 2013] focuses on fixing righthand-side of assignments or branch conditions for scalability.

7 CONCLUSION
In this paper, we presented FixML, the first system for automatically diagnosing and correcting logical errors in functional programming assignments. We presented the new error-correction algorithm that combines statistical error-localization, type-directed enumeration, automatic component reductions, and pruning via symbolic execution. We have demonstrated the effectiveness of our approach with 497 students’ submissions and the usefulness with 18 undergraduate students.

ACKNOWLEDGMENTS
We thank Jaehyun Lim for his help in preparing the user study. This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT (2017M3C4A7068175). This work
was also supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09.

A SOLUTION PROGRAM (EXAMPLE 1 IN SECTION 2.1)

1 type aexp =
2 | Const of int
3 | Var of string
4 | Power of string * int
5 | Times of aexp list
6 | Sum of aexp list
7
8 let rec diff : aexp * string -> aexp = fun (e, x) ->
9 match e with
10 | Const n -> Const 0
11 | Var a -> if (a <> x) then Const 0 else Const 1
12 | Power (a, n) -> if (a <> x) then Const 0 else Times [Const n; Power (a, n-1)]
13 | Times l ->
14 begin match l with
15 | [] -> Const 0
16 | hd::tl -> Sum [Times ((diff (hd, x))::tl); Times [hd; diff (Times tl, x)]]
17 end
18 | Sum l -> Sum (List.map (fun e -> diff (e,x)) l)

B LANGUAGE

The full language of FixML is as follows:

\[
E ::= () | n | x | true | false | str | \lambda x. E | E_1 + E_2 | E_1 - E_2 | E_1 \times E_2 | E_1 \mod E_2 | -E | \not E | E_1 \| E_2 | E_1 \&\& E_2 | E_1 < E_2 | E_1 > E_2 | E_1 \leq E_2 | E_1 \geq E_2 | E_1 = E_2 | E_1 <> E_2 | E_1 \_\_\_ E_2 | \text{raise } E | (E_1, \ldots, E_k) | \text{[E}_1; \ldots; E_k]\n\]

\[
\tau ::= \text{unit} | \text{int} | \text{bool} | \text{string} | \text{exn} | \rho_1 \rightarrow \rho_2 | \rho \text{ list} | T | (\rho_1 \ldots \rho_k) | \alpha
\]

\[
\rho ::= n | x | true | false | p_1 :: p_2 | [p_1; \ldots; p_k] | (p_1, \ldots, p_k) | c(p_1, \ldots, p_k) | p_1 | \ldots | p_k | _
\]

C FEEDBACK ON A LARGE PROGRAM

FixML was able to accurately identify and fix the error at line 103 in the following student program for problem #13 in 3.4 seconds. Although the implementation was on a program with 154 lines, we shorten it for better readability.

1 type aexp = ...
2 type env = (string * int * int) list
3
4 let rec diff : aexp * string -> aexp = fun (aexp, x) ->
5 match env flag ->
6 match env with
begin match hd with
| (x, c, p) ->
  if (flag = 0 && c = 0) then deployEnv tl flag
  else if (x = "const" && flag = 1 && c = 1) then deployEnv tl flag
  else if (p = 0) then (CONST c)::(deployEnv tl flag)
  else if (c = 1 && p = 1) then (VAR x)::(deployEnv tl flag)
  else if (p = 1) then TIMES[CONST c; VAR x]::(deployEnv tl flag)
  else TIMES [CONST c; POWER(x, p)]::(deployEnv tl flag)
end
| [] -> [] in
let rec updateEnv : (string * int * int) -> env -> int -> env
= fun elem env flag ->
match env with
| (hd::tl) ->
  begin match hd with
  | (x, c, p) ->
    begin match elem with
    | (x2, c2, p2) ->
      if (flag = 0) then
        if (x = x2 && p = p2) then (x, (c + c2), p)::tl
        else hd::(updateEnv elem tl flag)
      else
        if (x = x2) then (x, (c*c2), (p + p2))::tl
        else hd::(updateEnv elem tl flag)
    end
  end
end
| [] -> elem::[] in
let rec doDiff : aexp * string -> aexp
= fun (aexp, x) ->
match aexp with
| CONST _ -> CONST 0
| VAR v ->
  if (x = v) then CONST 1
  else CONST 0
| POWER (v, p) ->
  if (p = 0) then CONST 0
  else if (x = v) then TIMES ((CONST p)::POWER (v, p-1)::[]) 
  else TIMES 0
| TIMES lst ->
  begin match lst with
  | (hd::[]) -> doDiff (hd, x)
  | (hd::tl) ->
    let diff_hd = doDiff(hd, x) in
    let diff_tl = doDiff((TIMES tl), x) in
    begin match (hd, diff_hd, tl, diff_tl) with
    | (CONST p, CONST s, [CONST r], CONST q) -> CONST (p*q + r*s)
    | (CONST p, _, _, CONST q) ->
      if (diff_hd = CONST 0 || tl = [CONST 0]) then CONST (p*q)
      else SUM [CONST(p*q); TIMES(diff_hd::tl)]
    | (_, CONST s, [CONST r], _) ->
if (hd = CONST 0 || diff_tl = CONST 0) then CONST (r*s)
else SUM [TIMES [hd; diff_tl]; CONST(r*s)]
| _ ->
if (hd = CONST 0 || diff_tl = CONST 0) then TIMES(diff_hd::tl)
else if (tl = [CONST 0] || diff_hd = CONST 0) then TIMES [hd; diff_tl]
else SUM [TIMES [hd; diff_tl]; TIMES (diff_hd::tl)]

end
| [] -> CONST 0
end
| SUM lst -> SUM(List.map (fun aexp -> doDiff(aexp, x)) lst) in
let rec simplify : aexp -> env -> int -> aexp list
= fun aexp env flag ->
match aexp with
| SUM lst ->
begin match lst with
| (CONST c)::tl -> simplify (SUM tl) (updateEnv ("const", c, 0) env 0) 0
| (VAR x)::tl -> simplify (SUM tl) (updateEnv (x, 1, 1) env 0) 0
| (POWER (x, p))::tl -> simplify (SUM tl) (updateEnv (x, 1, p) env 0) 0
| (SUM lst)::tl -> simplify (SUM (List.append lst tl)) env 0
| (TIMES lst)::tl ->
(
  let l = simplify (TIMES lst) [] 1 in
  match l with
  | h::t ->
    if (t = []) then List.append l (simplify (SUM tl) env 0)
    else List.append (TIMES l::[]) (simplify (SUM tl) env 0)
  | [] -> []
)
| [] -> deployEnv env 0
end
| TIMES lst ->
begin match lst with
| (CONST c)::tl -> simplify (TIMES tl) (updateEnv ("const", c, 0) env 1) 1
| (VAR x)::tl -> simplify (TIMES tl) (updateEnv (x, 1, 1) env 1) 1
| (POWER (x, p))::tl -> simplify (TIMES tl) (updateEnv (x, 1, p) env 1) 1
| (SUM lst)::tl ->
(
  let l = simplify (SUM lst) [] 0 in
  match l with
  | h::t ->
    if (t = []) then List.append l (simplify (TIMES tl) env 1)
    else List.append (SUM l::[]) (simplify (TIMES tl) env 1)
  | [] -> [] (* Feedback : Replace [] by ((Sum lst) :: tl) *)
)
| (TIMES lst)::tl -> simplify (TIMES (List.append lst tl)) env 1
| [] -> deployEnv env 1
end in
let result = doDiff (aexp, x) in
match result with
| SUM _ -> SUM (simplify result [] 0)
| TIMES _ -> TIMES (simplify result [] 1)
| _ -> result
REFERENCES


Manu Jose and Rupak Majumdar. 2011. Cause Clue Clauses: Error Localization Using Maximum Satisfiability. SIGPLAN Not. 46, 6 (June 2011), 437–446. DOI: http://dx.doi.org/10.1145/1993136.1993550


