基于分析程序修复的类型错误反馈

Georgios Sakkas
Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
gsakkas@eng.ucsd.edu

Madeline Endres
Computer Science & Engineering
University of Michigan
Ann Arbor, MI, USA
endremad@umich.edu

Benjamin Cosman
Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
blcosman@eng.ucsd.edu

Westley Weimer
Computer Science & Engineering
University of Michigan
Ann Arbor, MI, USA
weimerw@umich.edu

Ranjit Jhala
Computer Science & Engineering
University of California, San Diego
La Jolla, CA, USA
jhala@cs.ucsd.edu

摘要

我们介绍了分析程序修复，这是一种数据驱动的策略，用于通过错误程序的修复来提供类型错误的反馈。我们的策略基于类似错误具有类似修复的观察。因此，我们展示了如何使用错误类型程序对及其固定版本的训练数据集来：（1）通过将训练集中所做的编辑抽象和划分为一组有代表性的 tem 来学习候选修复模板的集合- 盘子；(2) 通过在训练集中使用的修复模板上训练多类分类器，从给定的错误中预测合适的模板；(3) 通过枚举和排序与预测模板匹配的正确（例如类型良好）的术语，从模板合成具体的修复。我们已经在 Rite 中实现了我们的方法：OCaml 程序的类型错误报告工具。我们在来自介绍性编程课程的两个实例的 4,500 个错误类型的 OCaml 程序的语料库中评估了 Rite 的准确性和效率，以及对显示位置的生成错误消息质量的用户研究并且最终修复质量以统计学上显着的方式优于最先进的工具。

CCS Concepts: · **Software and its engineering → General programming languages**; **Automatic programming**; · **Computing methodologies →** *Machine learning*; · **The- ory of computation →** *Abstraction*.

PLDI '20, June 15ś20, 2020, London, UK

ACM ISBN 978-1-4503-7613-6/20/06. . . $15.00
https://doi.org/10.1145/3385412.3386005

关键词: 类型错误反馈，程序综合, 程序修复，机器学习

ACM 引用格式:
Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. 2020. Type Error Feedback via Analytic Program Repair. In *Proceedings of the 41st ACM SIGPLAN Interna- tional Conference on Programming Language Design and Implemen- tation (PLDI '20), June 15ś20, 2020, London, UK.* ACM, New York, NY, USA, 15 pages.
https://doi.org/10.1145/3385412.3386005

1 介绍

具有 Hindley-Milner 风格、基于统一的推理的语言提供静态类型的好处，并且注释开销最小。然而，问题在于程序员必须首先爬上与理解编译器产生的错误消息相关的陡峭的学习曲线。虽然专家通常可以很容易地破译错误，并将它们视为对程序开发和重构的宝贵帮助，但新手通常会感到非常困惑和沮丧，不清楚问题是什么 [41]。由于问题的重要性，一些作者提出了帮助调试类型错误的方法，通常是通过将程序切到有问题的位置 [12, 31]，列举可能的原因 [6, 21]，或者通过使用 MAX-SAT [30]、贝叶斯 [43] 或统计分析 [37] 对可能的位置进行排名。虽然很有价值，但这些方法最多只能帮助定位问题，但学生们仍然对如何修复他们的代码一无所知。

修复作为反馈. 最近的几篇论文对反馈问题提出了一种鼓舞人心的新思路：使用综合技术以修复的形式提供反馈，学生可以应用这些技术来改进他们的代码。这些修复可以通过符号搜索由专家定义的修复模型 [14, 38] 限定的候选程序空间来找到。然而，对于类型错误，候选修复的空间是巨大的。目前还不清楚是否存在一小组维修模型，或者即使存在，它看起来是什么样的。更重要的是，为了扩大规模，我们必须取消专家仔细策划一些候选维修集的要求。

或者，我们可以通过观察相似的程序具有相似的修复来生成修复，即通过从学生对 łclosestž 正确程序的解决方案计算 łdiffsž [11, 42]。但是，这种方法需要类似程序的语料库，其语法树或执行跟踪可用于将每个不正确的程序与用于提供反馈的 łcorrectž 版本进行匹配。具有静态类型错误的程序没有执行痕迹。更重要的是，我们需要一种方法来为新手编写的新程序生成反馈，因此不能依赖于与某些（现有）正确程序的匹配。

分析程序修复. 在这项工作中，我们提出了一种称为分析程序修复的新型错误修复策略，该策略使用监督学习而不是手动制作修复模型或匹配正确代码的语料库。我们的策略基于类似错误具有相似修复的关键洞察力，并通过使用错误类型程序及其固定版本对的训练数据集来实现这一洞察力：（1）通过抽象来学习候选修复模板的集合并将在训练集中所做的编辑划分为一组有代表性的模板；（2）通过在训练集中使用的修复模板上训练多类分类器，从给定的错误中预测合适的模板；（3）通过列举和排序与预测模板匹配的正确（例如类型良好）的术语，从模板合成具体的修复，从而为候选程序生成修复。至关重要的是，我们展示了如何通过抽象词袋（BOAT）表示程序，即作为句法和语义特征的数字向量 [35]，来执行从特定程序到抽象错误的关键抽象。这种抽象让我们可以在高级代码特征上训练预测器，即学习导致错误的特征与其相应修复之间的相关性，允许分析方法在匹配现有程序之外进行泛化。

Rite. 我们已经在 Rite 中实现了我们的方法：OCaml 程序的类型错误报告工具。我们训练（和评估）Rite 是在 4,500 多个错误类型的 OCaml 程序上进行的，这些程序来自两年的介绍性编程课程。给定一个新的错误类型程序，Rite 生成一个潜在解决方案列表，按可能性和编辑距离度量排序。我们以多种方式评估 Rite。首先，我们测量其准确性：我们表明，当考虑前三个模板时，Rite 在 69% 的时间内正确预测了正确的修复模板，而在我们考虑前六名时超过 80%。其次，我们测量它的效率：我们表明 Rite 能够在 70% 的时间内在 20 秒内合成混凝土修复。最后，我们通过对 29 名参与者的用户研究来衡量生成的消息的质量，并表明人类认为 Rite 的编辑位置和最终修复质量都比最先进的 OCaml 修复工具 Seminal 生成的要好[21] 以统计显着的方式。

2 概述

我们首先概述了我们的方法，通过学习新手程序员修复他们程序中的错误所遵循的过程，为错误的程序提出修复建议。

```
1  let rec mul By Digit i l=
2      match l with
3      |[] ->[]
4     |hd :: tl -> ( hd * i) @ mulByDigit i tl
```

```
1  let rec mul By Digit i l=
2      match l with
3      |[] ->[]
4       |hd :: tl -> [ hd * i] @ mulByDigit i tl
```

图1.（上）一个错误类型的 OCaml 程序，它应该将列表的每个元素乘以一个整数。(下)学生的修复版本。

问题. 考虑图 1 顶部显示的程序 mulByDigit，该程序由一名本科生在编程课程中编写。该程序旨在将列表中的所有数字与整数相乘。学生不小心误用了列表追加运算符 (@)，将其应用于一个数字和一个列表而不是两个列表。仍在构建类型检查器如何工作的心智模型的新手经常对编译器的错误消息感到困惑 [26]。因此，新手通常需要很长时间才能找到合适的解决方案，例如图 1 底部所示的解决方案，其中 @ 与包含头部 hd 和 i 相乘的单例列表一起使用。我们的目标是使用程序员如何修复他们程序中的类似错误的历史数据来自动、快速地指导新手想出像上面那样的候选解决方案。

解决方案：分析程序修复. 一种方法是将搜索候选修复视为一个综合问题：将一组（小）编辑合成到程序中，从而产生一个好的（例如类型正确的）编辑。关键的挑战是通过将修复限制在一个有效的可搜索空间来确保综合是易于处理的，并且精确，因此搜索不会错过错误程序的正确修复。在这项工作中，我们提出了一种称为分析程序修复的新策略，它通过将问题分解为三个步骤来实现易于处理和精确的搜索：首先，学习一组广泛使用的修复模板。其次，为每个错误的程序预测要应用的正确修复模板。第三，从预测模板合成候选修复。在本节的其余部分，我们通过描述如何：

1. 通过修复模板抽象地表示修复（2.1），

2. 获得一组带标签的错误类型程序和修复的训练集（2.2），
3. 通过划分训练集来学习一小组候选修复模板（2.3），
4. 通过从训练集中训练多类分类器来预测要应用的适当模板 (2.4)，以及
5. 通过枚举和检查预测模板中的术语来综合修复，为程序员提供本地化的反馈（2.5）.

2.1表示修复

我们的修复概念被定义为在特定程序位置用新的候选表达式替换现有表达式。例如，mulByDigit 程序通过在第 4 行用表达式 [hd * i] 替换 (hd * i) 来修复。我们专注于 AST 级别的替换，因为它们紧凑但表达能力足以表示修复。

通用抽象语法树. 我们通过称为通用抽象语法树（GAST）的抽象修复模板表示不同的可能候选表达式，每个模板对应于许多可能的表达式。 GAST 是从具体的 AST 中分两步获得的。首先，我们抽象出具体的变量、函数和运算符名称。接下来，我们在一定深度 d 修剪 GAST，以仅保留修复的顶级更改。修剪后的子树被替换为空洞，它可以代表我们语言中任何可能的表达。

这些步骤一起确保 GAST 仅包含有关修复结构的信息，而不是变量和函数的具体变化。例如，mulByDigit 示例中的固定 [hd * i] 由表达式 [_ ⊕ _] 的 GAST 表示，其中变量 hd 和 i 被抽象为孔（例如，通过在深度 $d = 2$ 处修剪 GAST） * 由抽象二元运算符 ⊕ 表示。我们的方法类似于 Lerner 等人的方法。 [21]，其中使用了 AST 级别的修改，然而，我们提出的 GAST 代表了更抽象的修复模式。

2.2获取一个固定标签的训练集

以前的工作已经使用专家创建了一组错误类型的程序及其修复版本 [21, 22]，或者手动创建可以产生修复补丁的修复模板 [16] [24, 25]。这些方法很难扩展以产生适合机器学习的数据集。此外，他们没有发现特定类别的新手错误及其修复在实践中的频率。相比之下，我们展示了这样的修复模板可以从一个大型的、自动构建的训练集上学习，这些训练集标有他们的修复标记。我们数据集中的修复表示为学生在错误类型程序中更改的表达式的 AST，以将其转换为正确的解决方案。

交互痕迹. 在[37]之后，我们从交互跟踪中提取了错误程序及其固定版本的标记数据集。通常学生会编写他们的程序的多个版本，直到他们为编程作业找到正确的解决方案。仪器编译器用于捕获学生程序的此类序列（或跟踪）。在这一系列尝试中，第一个类型正确的解决方案被认为是所有先前解决方案的固定版本，因此它们中的每一个都被添加到数据集中。对于每个程序对，我们然后生成它们的抽象语法树 (AST) 的差异，并将在程序正确和错误类型尝试之间变化的最小子树分配为数据集的修复标签。

2.3学习候选修复模板

我们数据集中的每个标记程序都包含一个修复，我们将其抽象为修复模板。例如，对于图 1 中的 mulByDigit 程序，我们得到了候选修复 [hd * i]，因此得到了修复模板 [_ ⊕ _]。然而，可能包含许多不同解决方案的大量带有修复标记的程序数据集可能会引入大量修复模板，这可能不适用于预测用于最终程序修复的正确模板。

因此，我们方法的下一步是学习一组足够小的修复模板，以自动预测将哪个模板应用于给定的错误程序，但仍然涵盖了实践中出现的大多数修复。

对修复程序进行分区. 我们通过对从我们的数据集中获得的所有模板进行分区，然后选择一个 GAST 来表示每个修复模板集中的修复模板来学习一小组合适的修复模板。分区有两个目的。首先，它识别一小组最常见的修复模板，然后允许使用离散分类算法来预测将哪个模板应用于新程序。其次，它允许原则性地删除异常值，因为学生提交的内容通常包含我们不希望用于建议修复的非标准或特殊解决方案。

与以前使用聚类将相似程序组合在一起的修复方法（例如，[11, 42]）不同，我们根据修复相似关系将我们的修复模板集划分为它们的等价类。

2.4通过多分类预测模板

接下来，我们训练可以正确预测错误位置并为给定的错误类型程序修复模板的模型。我们使用这些模型来生成候选表达式作为可能的程序修复。为了降低预测正确修复模板和错误位置的复杂性，我们将这些问题分开并将它们编码为两个不同的监督分类问题。

监督多类分类。 我们建议使用有监督的多类分类问题来预测修复模板。监督学习问题是这样一个问题，给定一个标记的训练集，任务是学习一个函数，将输入准确地映射到输出标签并推广到未来的输入。在分类问题中，我们试图学习的函数将输入映射到一组离散的两个或多个输出标签，称为类。因此，我们将学习函数的任务编码为多类分类 (MCC) 问题，该函数将错误类型程序的子表达式映射到一小组候选修复模板。

特征提取。我们所要训练的用来解决MCC 问题的机器学习模型期望将标记固定长度向量的数据集作为输入。因此我们定义了固定标记程序到固定长度向量的转换。

类似于Seidel等人的[37]，我们定义了一组特征提取函数f1，…,fn，它将程序的子表达式映射为一个数值(或者只是{0,1}编码的布尔属性)。给定一组特征提取函数，我们可以将AST e分解为一组组成它的子表达式{e1，…，en}，然后用n维度的向量表示每个ei[f1(e1)，…,fn(en)]。这种方法在以前的工作[37]中称为抽象术语袋(BOAT)表示。

通过MCC预测模板。我们固定标记的数据集合可以被更新所以标签代表了固定每个位置的相应模板，而这些位置是从划分得来的修复模板的最小集中提取出的。我们之后在最新版的标记的模板数据集上训练了一个深度神经网络（DNN）分类器。

神经网络的优点是将每一类与一个置信值相关联，这个置信值可以解释为：根据模型的估计分布对于给定的输入每一类正确的模型概率。因此，信心得分可以用来对新程序的修复模板预测进行排序，并在合成修复时按降序使用它们。利用机器学习的最新进展，我们使用深度和密集的架构[34]用于更准确的修复模板预测。

错误定位。我们把在一个新程序中寻找错误位置的问题看作是一个二值分类问题。与模板预测问题相比，我们希望学习一个将程序的子表达式映射到表示是否存在错误的二进制输出的函数。因此，这个问题相当于只有两个类的MCC，因此，我们使用类似的神经网络的深度架构。对于给定程序中的每个表达式，学习的模型输出一个置信值，表示需要修正的错误位置的可能性有多大。我们利用这些分数以可信度降序为每个位置合成候选表达式。

2.5 由模板综合反馈

接下来，我们使用典型的程序综合技术来综合会用于给使用者提供反馈的候选表达式。而且，综合是由已预测的修复模板和一组可能的错误位置引导的，且会返回一个排好序的最小修复方法列表作为给使用者的反馈。

程序综合。给定一组位置和给这些位置的候选模板，我们试图解决程序综合的一个问题。对于每个程序位置，我们在语言语法中所有的表达式中遍历寻找一个匹配修复模板且能实现类型检验的候选表达式的小集合。在合成过程中，还使用了不良类型程序的表达式来修剪候选表达式的搜索空间。

复杂位置的综合。很常见的情况是有不止一个位置需要修复。因此我们不只考虑单个错误位置的有续集，还有它的幂集。为了简化，我们考虑把不同的程序位置看做独立的来修复；我们指派一组需要修复的位置的概率因而是它们各自信心得分的乘积。这与最近的多块程序修复[33]的方法不同，后者的修改是相互依赖的。

排序修复。最后我们用两个度量方法来为每个解决方案排序：树编辑距离和字符串编辑距离。之前的任务[11,21,42]就用了这样的度量来考虑最小的改变，即尽可能接近原程序的改变，所以新手程序员可以得到更连贯的反馈。

```
1   let rec mulByDigit i l =
2     match l with
3     | []     -> []
4     | hd::tl -> [v1 * v2] @ mulByDigit i tl
```

图2. mulByDigit程序的一个候选修复

例子。我们由图二看到我们的方法能够用在2.3中讨论的模板进行综合而返回的最小修复（第4行中[v1*v2]）。虽然此解决方案不是我们的实现返回的最高级别解决方案(这将与人工解决方案相同)，但它演示了合成器的相关方面。特别地，这个解决方案有一些抽象的变量，v1和v2。我们的算法建议用户可以用两个不同的变量来替换两个变量，并将整个表达式插入到一个列表中以获得正确的程序。

3 学习修复模板

我们首先介绍我们从由成对错误和固定程序组成的训练数据集中提取有用的修复模板的方法。我们表示那些模板就一种语言而言允许我们以一种新手在实践中使用的各种修复模式所捕获的基本结构的方式简洁地表示修复。但是，为程序对数据集中的每个修复提取单一修复模板会产生太多的模板，从而无法执行准确的预测。因此，我们定义了模板之间的相似关系，用它来将提取的模板划分成一个小但有代表性的集合，这将使训练精确的模型来预测修正更容易。

$$
\begin{array}{lcl}
e & ::= & x \mid \lambda x.e \mid e\,\bar{e} \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & \mid & n \mid b \mid e + e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \langle e, e\rangle \mid \texttt{match}\ e\ \texttt{with}\ \langle x, x\rangle \to e \\
 & \mid & [] \mid e :: e \mid \texttt{match}\ e\ \texttt{with} \begin{cases} [] \to e \\ x :: x \to e \end{cases} \\
n & ::= & 0, 1, -1, \ldots \\
b & ::= & \texttt{true} \mid \texttt{false} \\
t & ::= & \alpha \mid \texttt{bool} \mid \texttt{int} \mid t \to t \mid t \times t \mid [t]
\end{array}
$$

图3. λ^{ML} 的句法

$$
\begin{array}{lcl}
e & ::= & \_ \mid \hat{x} \mid \lambda\hat{x}.e \mid \hat{x}\,\bar{e} \mid \texttt{let}\ \hat{x} = e\ \texttt{in}\ e \\
 & \mid & \hat{n} \mid e \oplus e \mid \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
 & \mid & \langle e, e\rangle \mid \texttt{match}\ e\ \texttt{with}\ \langle \hat{x}, \hat{x}\rangle \to e \\
 & \mid & [] \mid e :: e \mid \texttt{match}\ e\ \texttt{with} \begin{cases} [] \to e \\ \hat{x} :: \hat{x} \to e \end{cases}
\end{array}
$$

图4. λ^{RTL} 的句法

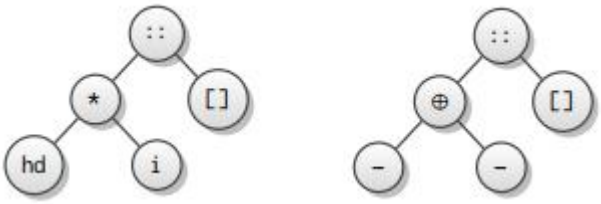
3.1 表示用户修复

修复模板语言。图4描述了我们修复模板的语言，λ^{RTL}，这是一个使用整数、布尔型、有序对、列表的λ积分，用句法抽象表单扩展了我们核心的ML语言λ^{ML}（图3）：

1. 名为$\hat{x}$的抽象变量被用来表示函数、变量、绑定器（即$\hat{x}$表示λ^{RTL}中一个未知变量名）中变量的出现。
2. 抽象文字值$\hat{n}$可以表示任何整数、浮点数、布尔值、字符或字符串;
3. 抽象算子⊕同样表示未知一元或二元操作符;
4. 通配符表达式_用于表示λ^{RTL}中的任何表达式，即程序洞。

回想在2.1中，我们将修复定义为在特定程序位置用新的候选表达式替换表达式。因此，我们使用候选表达式来表示修复模板。

概括AST。通用抽象语法树(Generic Abstract Syntax Tree, GAST)是来自λ^{RTL}的一个术语，它表示来自λ^{ML}的许多可能的表达式。GAST是从核心语言λ^{ML}上的标准AST抽象出来的，使用的是抽象函数，该函数将表达式e^{ML}(大于λ^{ML})和深度d作为输入，并返回表达式e^{RTL}(大于λ^{RTL})，即带有所有变量的GAST，从深度大于d开始的e^{ML}的字面量和操作符以及所有子表达式都被删除并替换为空洞_。



(a) AST修复 (b) GAST模板

图5.（左）图1例子的修复和（右）那个修复的一个可能的模板

例子。会议我们在图1 mulByDigit程序的例子。第4行中表达式[hd * i]代替了(hd * i)，因此它是用户的修复，其AST在图5a中给出。给定这个AST和一个深度为2的输入，抽象的输出会成为图5b中的GAST，其中操作符*被一个凑向操作符⊕所替换，以及在深度2中的子术语hd和i被抽象为通配符表达式_。因此，这个λ^{RTL}式[_ ⊕ _]表示了一个潜在的对于mulByDigit的修复模板。

3.2 从一个数据集中提取修复模板

我们的方法通过从程序对的训练集中获取一组修复模板来实现修复的完全自动化提取。给定数据集中的程序对(perr, pfix)，我们为在perr中更改的每个位置提取一个唯一的修复pfix。我们使用表达式级diff[20]函数来实现这一点。回想一下，我们的修复是表达式的替代，因此我们将这些提取的更改抽象为我们的修复模板。

上下文的修复。继Felleisen等人的[8]之后，让C作为一个在程序中出现表达式e的上下文p，即将程序p用e替换为空洞_。我们写p = C[e]，这意味着如果我们用原来的表达式e填补这个空缺，我们就得到了原来的程序p。在这种方式下，diff找到一个最小的(以节点数量计)表达式的替换形式efix，以替代perr中的表达式eerr，这样perr = Cperr[efix] 和Cperr[efix]=pfix。可能会有几个这样的表达式，diff返回所有这样的更改。

例子。如果f x重写为g x,上下文就是C=_x继而修复就是g，因为C[g]=g。如果f x重写为(f x)+1，上下文就是C=_，继而修复就是整个表达式(f x)+1，因此C[(fx)+1]=(fx)+1。（即使f x出现在最初的和修复的程序中，我们认为应用程序表达式f x -，而不是f或x -被+操作符取代。）

3.3 分区的模板

程序在λ^{ML}强制进行类似的修复，例如更改变量名，以具有相同的GAST。我们的下一步是定义程序修复相似度的概念。我们的定义支持形成一组小型但广泛适用的修复模板。这个小的集合被用来训练修复预测器。

GAST相似性。当根节点相同且它们的子树（如果有的话）可排成成对相似时两个GAST是相似的。例如，x+3和7-y产生相似的GAST $\hat{x} \oplus \hat{n}$和$\hat{n} \oplus \hat{x}$，其中根节点都是抽象二进制操作数，一个孩子是一个抽象文字，另一个孩子是一个抽象变量。

分区。GAST相似性定义了一个自反的、对称的和传递的关系，因此是等价关系。我们现在可以将划分定义为我们提取的修复模板的所有可能等价类的计算w.r.t. GAST相似性。每个类可以由几个成员表达式组成，其中任何一个都可以被视为类的代表。然后，每个代表都可以作为一个修复模板来对类型化不良的程序进行修复。

例如$\hat{x} \oplus \hat{n}$和$\hat{n} \oplus \hat{x}$在同一个类，其中任何一个都可以作为代表。第5节中的修复算法在使用这个模板修复错误程序时将从本质上考虑这两种情况。

最后，我们的划分算法根据它们在数据集中的成员表达式频率返回N等价类的顶部。N是算法的一个参数，它被选择为尽可能小，而N类顶部代表了数据集足够大的一部分。

4 预测修复算法

给定一组候选模板，我们的下一个任务是训练一个模型，当给定一个(错误的)程序时，该模型可以预测该程序中的每个位置使用哪个模板。为此，我们定义了一个函数预测，它以(1)特征提取函数特征，(2)数据集作为输入程序对的数据集(perr，pfix)，以及(3)一个修复模板t的列表。它作为输出返回一个fix-template-predictor，对于一个错误的程序，它返回可能的修复的位置，以及在这些位置应用的模板。

我们使用三个辅助函数构建预测，它们执行每一个高级步骤。首先，提取函数从程序对数据集中提取特征和标签。接下来，这些特征向量被分组并输入到train中，train产生两个模型，即LModel和TModel,分别用于错误定位和预测修复模板。最后，rank获取一个新的(错误的)程序的特征，并查

询训练过的模型，以返回可能的修复位置和相应的修复模板。

接下来，我们将描述图6中的关键数据类型、三个关键步骤的实现，以及如何将它们组合起来生成预测算法。

信心，数据，和标签。如图6所示，我们将EMap a定义为从表达式e到类型a的值的映射，将TMap C定义为从模板T到此类值的映射。例如，TMap C是从模板T到它们的置信度分数C的映射。Data表示用于训练我们的预测模型的特征向量，而Label B是用于训练的数据集标签，Label C是输出置信度分数。最后,一对是一对程序(perr,pfix)。

特性和预测。我们将特征定义为一个函数，它为输入程序e的每个子表达式生成特征向量数据。这些特征向量以地图EMap Data的形式给出，它将输入程序e的所有子表达式映射到其特征向量数据。

预测器是从我们的算法中学习的固定模板预测器，该算法用于为输入程序生成置信值映射e。具体来说，它们返回一个映射EMap(标签C)，该映射EMap将输入程序p的每个子表达式与一个信心评分标签C关联起来。

体系结构。首先，提取函数以特征提取函数的特征、模板列表作为输入[T]以及一个单独的程序对Pair并生成一个错误输入程序的所有子表达式的特征向量的和布尔标签组成的地图EMap (Data × Label B)。然后将所有的特征向量Data和标签Label B累积成一个列表，作为输入进行训练，用于训练LModel和TModel，分别用于预测错误位置和修复模板。接下来，两个经过训练的模型LModel和TModel，以及来自一个新的、以前没有见过的程序的数据，可以被输入到排序中。这产生一个预测器，它可以用来将新程序的子表达式映射到可能的错误位置和修复模板。

4.1 体系结构和标签提取

我们用来预测固定模板和错误位置的机器学习算法期望固定长度的特征向量数据作为它们的输入。但是，我们希望通过 λ^{ML} 修复可变大小的程序。因此，我们使用提取函数将程序转换为特征向量。

继Seidel等人[37]之后，我们选择将程序建模为一组特征向量，其中每个元素对应于程序中的一个子表达式。
因此，给定一个错误的程序perr，我们首先将其分解为组成它的子表达式，然后将每个子表达式转换为单个特征向量，即Feature perr ::EMap Data。我们只考虑最小类型错误片中的表达式。我们在这里展示了使用的五个主要特性类别。

本地句法功能。这些特征描述了每个表达式e的句法种类。换句话说，对e的每个产生式规则由图3中我们引入了一个特征，如果表达式是用该生成构建的，则启用(设置为1)，否则禁用(设置为0)。

C	$\doteq$	$\{r \in \mathbb{R} \mid 0 \le r \le 1\}$
$\mathcal{B}$	$\doteq$	$\{b \in \mathbb{R} \mid b = 0 \vee b = 1\}$
T	$\doteq$	e^{RTL}
EMap a	$\doteq$	$e \to a$
TMap a	$\doteq$	$\mathsf{T} \to a$
Data	$\doteq$	$[C]$
Label a	$\doteq$	$a \times \mathsf{TMap}\ a$
Pair	$\doteq$	$e \times e$
DataSet	$\doteq$	[Pair]
Features	$\doteq$	$e \to \mathsf{EMap\ Data}$
Predictor	$\doteq$	$e \to \mathsf{EMap\ (Label}\ C)$
abstract	:	$e \to \mathsf{T}$
diff	:	$\mathsf{Pair} \to [e]$
extract	:	$\mathsf{Features} \to [\mathsf{T}] \to \mathsf{Pair} \to \mathsf{EMap\ (Data} \times \mathsf{Label}\ \mathcal{B})$
train	:	$[\mathsf{Data} \times \mathsf{Label}\ \mathcal{B}] \to \mathsf{LModel} \times \mathsf{TModel}$
rank	:	$\mathsf{LModel} \to \mathsf{TModel} \to \mathsf{Data} \to \mathsf{Label}\ C$
predict	:	$\mathsf{Features} \to [\mathsf{T}] \to \mathsf{DataSet} \to \mathsf{Predictor}$

图6。用于将程序对转换为特征向量和模板标签的高级API。

语境句法特征。表达式出现的上下文对于正确预测错误源和修复模板至关重要因此，我们引入了上下文特征，它与局部句法特征相似，但描述了一个表达式的父级和子级。例如，Is-[]-C1特性将描述表达式的第一个子级是否为[]。这与语言模型中使用的n元文法相似[9，15] 。

表达式大小。我们还包括一个表示每个表达式大小的特征，即它包含多少子表达式。这使得模型能够了解到更接近叶子的表达式比更接近根的表达式更有可能被修复。 键入功能。我们正在尝试修复的程序是不可类型化的，但是从类型检查器派生的部分类型仍然可以为模型提供有用的信息。因此，我们在表示中包含类型特征$\to \cdot, \cdot \times \cdot$, 和 $[\cdot]$，有无限可能的类型，但我们必须有一组有限的特征。我们为每个抽象类型构造函数添加特性，这些特性描述给定类型是否使用该构造函数。例如，整形$\to$ 整形$\to$ 布尔将启用$\cdot \to$, 整形和布尔特性。

我们为父表达式和子表达式添加这些特性来总结上下文，但也为当前表达式添加这些特性，因为表达式的类型在语法上并不总是清晰的

类型错误片。我们希望区分可以修复错误和不能修复错误的更改。因此，我们计算程序的最小类型错误片（例如[12，40]）（即，导致错误的表达式集），如果程序包含多个类型错误，我们计算每个错误的最小片。然后我们有一个后处理步骤，它丢弃所有不包括在这些切片中的表达式 。标签：
回想一下，我们使用了两个预测模型，用于错误定位的L模型和用于预测修复模板的T模型。因此，我们需要两组与每个特征向量相关的标签，由标签B给出。L模型使用集合[数据× B]进行训练，而T模型使用集合[数据× TMap容器B]进行训练。

对于将错误程序perr转化为pfix程序的每个子表达式，L模型的B型标签都设置为true。来自于perr的一个标签TMap B映射到用于修复它的修复模板T。TMap B将所有子表达式与作为提取输入的固定数量的模板T相关联。因此，出于模板预测的目的，TMap B可以被视为表示用于修复每个子表达式的固定模板的固定长度布尔向量。此向量最多有一个槽设置

为true，表示模板修复的是perr，这些标签是使用微分和抽象提取的，在其中提取模板的方式类似ğ3.2.

4.2培训预测模型

我们使用训练函数的目标是在给定一个训练集[数据]的情况下训练两个独立的分类器×标签B]标记的样品。LModel预测错误位置，TModel预测新输入程序perr的修复模板。关键的是，我们要求错误定位分类器输出一个置信度C，表示子表达式是被修复错误的概率。我们还要求fix模板分类器为每个fix模板输出一个置信度C，该置信度度量分类器是否确定该模板可用于修复输入程序perr的相关位置。我们考虑一个标准的学习算法来生成我们的模型：神经网络。对神经网络的全面介绍超出了这项工作的范围[13，28]。

神经网络。我们使用的模型是一种称为多层感知器的神经网络。多层感知器可以表示为一个有向无环图，它的节点被安排在由加权边完全连接的层中。第一层对应于输入特征，最后一层对应于输出特征。内部节点的输出是传递给非线性函数（称为激活函数）的前一层的加权输出之和。 层的数量、每层的节点数量以及层与层之间的连接构成了系统的体系结构神经网络。在这项工作中，我们使用了相对较深的神经网络网络（DNN）。我们可以将一个DNN-LModel训练成一个二值分类器，它可以预测一个位置是否在程序perr中并且是否需要修复 。

Algorithm 1 Predicting Templates Algorithm

Input: Feature Extraction Functions F, Fix Templates Ts, Program Pair Dataset D
Output: Predictor Pr

procedure PREDICT(F, Ts, D)
  $D_{ML} \leftarrow \emptyset$
  for all $p_{err} \times p_{fix} \in D$ **do**
    $d \leftarrow$ EXTRACT(F, Ts, $p_{err} \times p_{fix}$)
    $D_{ML} \leftarrow D_{ML} \cup$ INSLICE(p_{err}, d)
  $Models \leftarrow$ TRAIN(D_{ML})
  $Data \leftarrow \lambda p.$ INSLICE(p, EXTRACT(F, Ts, $p \times p$))
  $Pr \leftarrow \lambda p.$ MAP($\lambda \tilde{p}.$ RANK($Models$, $\tilde{p}[0]$), $Data(p)$)
  return Pr

多类DNN。虽然上述模型足以进行误差定位，但在模板预测的情况下，我们必须从两个以上的类中进行选择。我们再次使用DNN作为模板预测TModel，但是我们调整输出层从N个节点对应N个选择的模板类。对于用神经网络解决的多类分类问题，通常在输出层使用softmax函数[5，10]。Softmax将概率分配给每个类，这些概率加起来必须为1。这个额外的约束加速了训练。

4.3预测修复模板

我们的最终目标是能够精确地指出一个错误程序的哪些部分应该被修复，以及应该使用哪些修复模板来达到这个目的。因此，predict函数使用预测所有子表达式的置信度C的排序作为错误位置，并预测每个修复模板的置信度TMap C。我们在这里展示我们怎样把在图6中的高级API结合起来，为一个新程序生成一个最终的置信度列表p。算法1给出了我们的D高级预测算法。

预测算法。我们的算法首先从程序对数据集提取机器学习适用的数据集D_{ML} 对于中的每个程序对p, Extract返回从错误程序的子表达式到特征和标签的映射。那么，InSlice只保留类型错误切片中的表达式，并计算为相应特征向量和标签向量的列表，该列表将添加到数据集D_{ML}。这个数据集被Train函数用来生成我们的预测结果model、 i、 e.L模型和T模型 在这一点上，我们要生成一个新的预测未知程序p. 我们使用Extract对图像进行特征提取，并使用InSlice限制中的表达式p's类型错误片。结果如下：数据集p

然后将秩应用于由数据集p生成的所有子表达式它将创建一个EMap（标签C）类型的映射，该映射将表达式与置信度分数相关联 。我们将秩应用于每个与类型错误切片中的表达式相对应的特征向量p. 这些向量是p∈ 数据集p的第一要素, 数据集p 属于数据类型×标签B。最后，预测器Pr 返回，第5节中的合成算法通过他们的自信得分关联子表达式。

训练数据集。从我们的评价来说，我们使用从以前用于相关工作〔35, 37〕本科编程语言大学课程收集的OCAML数据集，。它由错误的程序及其后续的修正组成，分为两部分；2014年春季（SP14）和2015年秋季班（FA15）。家庭作业要求学生编写23个不同的程序，演示一系列函数式编程习惯用法，例如高阶函数和（多态）代数数据类型。

Algorithm 2 Local Repair Algorithm

Input: Language Grammar λ^{ML}, Program P, Template T, Repair Location L, Max Repair Depth D
Output: Local Repairs R

procedure REPAIR(λ^{ML}, P, T, L, D)
  $R \leftarrow \emptyset$
  for all $d \in [1 \ldots D]$ **do**
    $\tilde{\alpha} \leftarrow$ NONTERMINALSAT(T, d)
    for all $\alpha \in$ RANKNONTERMINALS($\tilde{\alpha}$, P, L) **do**
      if ISHOLE(α) **then**
        $Q \leftarrow$ GRAMMARRULES(λ^{ML})
        $\tilde{\beta} \leftarrow \{\beta \mid (\alpha, \beta) \in Q\}$
        for all $\beta \in$ RANKRULES($\tilde{\beta}$, T) **do**
          $\hat{T} \leftarrow$ APPLYRULE(T, (α, β))
          $R \leftarrow R \cup \{\hat{T}\}$
      else
        for all $\sigma \in$ GETTERMINALS(P, L, λ^{ML}) **do**
          $\hat{T} \leftarrow$ REPLACENODE(T, α, σ)
          $R \leftarrow R \cup \{\hat{T}\}$
  return R

特征提取。Rite用程序中每个表达式的449个特征表示程序：45个局部语法特征、315个上下文特征、88个键入特征和1个表达式大小特征。对于上下文特征，我们提取每个表达式的前4个子（从左到右）的局部句法特征。此外，我们从其父节点提取这些特征，然后再提取两个以上的父节点。对于输入特性，我们支持int、float、chars、string和用户定义的expr。为每个表达式及其上下文提取这些特征。

5.2排序错误位置

错误位置置信度。回想第4节，对于程序类型错误片中的每个子表达式，L模型为它生成一个作为错误位置的置信度C，T模型为修复模板生成一个置信度C。我们的合成算法

根据置信度C对所有程序位置进行排序。对于置信度降序排列的所有位置，使用算法2使用固定模板生成局部修复。固定模板按置信度降序排列。然后从返回的局部修复列表中删除表达式R替换给定程序位置处的表达式。该过程将尝试其余的修复、模板和位置，直到找到类型正确的程序。

在[21]之后，我们允许我们的最终局部修复有程序孔或抽象变量x在其中。然而，算法2将优先考虑完整解。在类型检查具体解决方案时，抽象RTL术语可以有任何类型，类似于OCaml提高exn。

多个错误位置。实际上，经常需要修复多个程序位置。因此，我们扩展了上述方法来修复具有多个错误的程序。因此，我们扩展了上述方法来修复具有多个错误的程序。让所有位置L的置信度得分为C.在我们的错误定位模型LModel-be的类型错误切片中(l1, c1), . . . , (lk,ck), li是一个程序位置,ci是它的错误信心分数。为了简单起见，我们假设ci他们是独立的。因此，所有的位置都需要固定的概率是乘积。因此，我们使用位置集({}、{、}，{、、}，等等。)，根据他们的信心分数的结果进行排名。对于一个给定的集合，我们对每个区域独立地使用算法2并应用所有可能的本地组合修复，再次寻找一个类型正确的解决方案。

6 评估

我们在Rite中实现了分析程序修复：一个修复OCaml纯函数子集类型错误的系统。接下来，我们将介绍我们的实现和评估，其中涉及三个问题：

RQ1：Rite预测的维修有多准确(ğ 6.1) •
RQ2：Rite如何高效地进行修复(ğ 6.2) •
RQ3:Rite的错误消息有多有用(ğ 6.3) •
RQ4:Rite的模板修复有多精确(ğ 6.4)

数据集清理。我们使用**diff**将修复提取为程序对上的表达式替换。将**diff**用于此数据集的缺点是，一些学生可能在编译之间进行了许多可能不相关的更改；在某个时刻，“修复”变成了“重写”。这些重写可能导致无意义的修复模板和错误位置。当一个程序中发生变化的子表达式的分数比平均值高出一个以上的标准差时，我们会丢弃这些异常值，从而建立一个40%的差异阈值。我们还丢弃了在5个或5个以上位置发生更改的程序，注意到即使是最先进的多位置修复技术也无法复制这种“修复”[33]。丢弃的更改约占每个数据集的32%，为SP14留下2475个程序对，为FA15留下2177个程序对。自始至终，我们使用SP14作为训练集，使用FA15作为测试集。

基于DNN的分类器。RITE的模板预测使用了一个基于多层神经网络DNN的分类器，该分类器有三个完全连接的512个神经元隐藏层。神经元使用校正线性单位（ReLU）作为其激活函数[27]。使用提前停止[13]对DNN进行训练：当训练集的一小部分的准确度在经过一定的时间段（在我们的实现中为5个时间段）后没有提高时，停止训练。我们将epoch的最大数目设置为200，使用了Adam优化器[17]，这是一种收敛速度更快的随机梯度下降的变体。

6.1 RQ1:准确性

大多数开发人员在回到手动调试之前会考虑五到六条建议[18，29]。因此，我们将RITE的准确性考虑到前六个固定模板预测，即检查前N个预测模板中是否有任何一个实际对应于用户的编辑。这些预测的模板并没有显示给用户；它们仅用于指导具体修复的合成，然后呈现给用户。

基线。我们将RITE的基于DNN的预测器与两个基线分类器进行比较：一个随机分类器返回从SP14训练数据集中学习的50个模板中均匀随机选择的模板，另一个流行分类器按降序返回训练集中最流行的模板。我们还比较了决策树（DTree）和在SP14数据上训练的SVM分类器，因为这是两种最常见的学习算法[13]

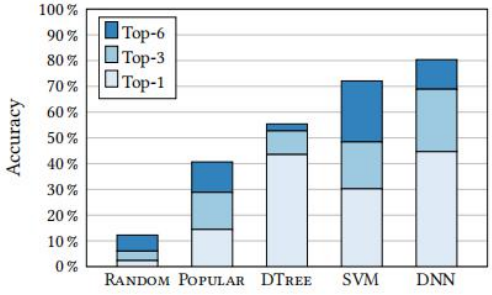


图7 .使用了最流行的50个模板的模板预测分类器结果。由于我们的合成算法考虑了很多模板在下降到不同位置之前的情况，所以我们给出了最多的6个预测结果。

结果：预测的准确性。图7显示了我们的模板预测实验的精度结果。Y轴描述了错误子项（位置）的分数，实际修复是top-K预测修复之一。随机选择模板的朴素基线达到了2%的Top-1精度（12%的Top-6），而流行的分类器达到了14%的Top-1精度（41%的Top-6）。事实上，即使只有DNN的第一个预测优于前6名的随机和流行预测，随机分类器的低性能是如预期的。流行的分类器表现得更好：一些任务在SP14和FA15两个之间共享，而不同的学生组在每个季度解决这些问题，他们犯的新手错误似乎有一个模式。因此，SP14中最流行的“补丁”（以及相关的模板）在FA15中也很流行。

我们还观察到，DTree实现了接近DNN的Top-1精度（即44%对45%），但随着预测的增加（即Top-6，55%对些模板。图8显示了从SP14训练集中获得的前30个模板的矩阵，这些模板在FA15数据集上进行了测试。请注意，大多数模板的预测都是正确的，只有少数模板
经常被另一个模板预测失误。例如，我们看到需要模板20的程序(让$\hat{z}$ = 匹配$\hat{t}$ 与$(\hat{x},\hat{y}) \rightarrow \hat{a}$)在需要修正的情况下，总是用模板11（$(\hat{x},\hat{y} = \hat{t}\ in(\_,\_)$）。我们观察到这些模板仍然非常相似，它们都有一个顶级let来处理元组$\hat{t}$。80%）没有得到改善。另一方面，支持向量机在前1精度上做得很差（即30%对45%），但在预测更多的情况下做得更好。

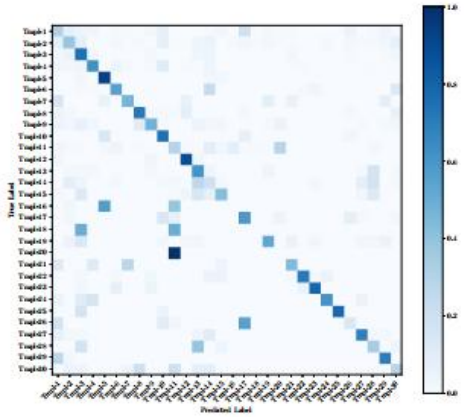


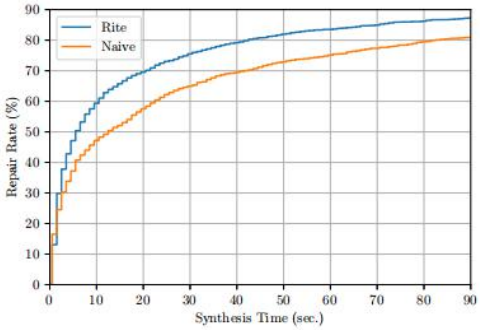
图8.前30个模板的混淆矩阵。热图中较粗的部分显示了经常被另一个模板错误预测的模板。对角线越粗，我们的预测就越准确。

结果：模板"混乱".每个位置最高预测的混淆矩阵显示了我们的模型混淆了哪些模板。图8显示了从SP14训练集中获得的前30个模板的矩阵，这些模板在FA15数据集上进行了测试。请注意，大多数模板的预测都是正确的，只有少数模板经常被另一个模板预测失误。例如，我们看到需要模板20的程序(让$\hat{z}$ = 匹配$\hat{t}$ 与$(\hat{x},\hat{y}) \rightarrow \hat{a}$)在需要修正的情况下，总是用模板11（$(\hat{x},\hat{y} = \hat{t}\ in(\_,\_)$）。我们观察到这些模板仍然非常相似，它们都有一个顶级let来处理元组$\hat{t}$。

RITE学习程序特征和修复模板之间的相关性，比Naive基线的准确率高出近2倍，比其他复杂的学习算法高出8%。通过将程序抽象为特性，RITE能够概括跨年份和不同类型的程序

6.2 RQ2：效率

接下来，我们通过测量它能够为多少程序生成(良好类型的)修复来评估RITE的效率。我们把合成器的时间限制在90秒。(通常，过程是不可确定的，我们推测较长的超时时间将降低对新手的实际可用性。)回想一下，修复综合算法是由修复模板预测指导的。我们通过将RITE与Naive基线实现进行比较来评估其效率，后者在给出预测的修复位置后，试图从微不足道的"洞"模板中合成一个修复。



图表9.在给定时间内可以修复的测试集的比例。

图表9显示累积分布函数。RITE和Naive的修复速度随合成时间而变化。我们观察到，使用预测模板进行合成，RITE可以在20秒内为将近70%的程序生成类型正确的修复，这比Naive基线高出近12个点。我们还观察到，在20秒以上的时间里，Rite比Naive成功修复的程序多10%左右。虽然Naive方法仍然能够相对快速地合成类型良好的修复，但我们将看到这些修复的质量要远远低于从预测模板生成的修复(6.4)。

6.3 RQ3：有用性

主要的结果是由RITE生成的基于修复的错误消息是否对新手有用。为了评估RITE的修复质量，我们对29名参与者进行了一项在线研究。每个参与者都被要求根据最先进的基线(开创性[21])评估程序修复的质量和它们的位置。对于每个程序，除了两次修复之外，参与者还会看到原始的类型不良的程序，以及标准OCAMI编译器的错误消息和程序原始作者意图的简短描述。由这项实验我们发现在一个具有统计学意义的方式中由RITE生成的编辑位置以即最终修复质量都优于SEMINAL的。

用户研究设置。研究参与者从两个公立研究机构（加利福尼亚大学，圣地亚哥和密歇根大学）以及在推特上发布的广告招募。参与者应评估至少5/10次刺激的质量以及给出可理解的bug描述。实验花了大概25分钟。参与者借助输入一个亚马逊Echo语音助手的图画来补偿。有29个可靠的参与者。我们从我们的数据集中合成了修复的1834个程序中随机选择了21个错误程序的语料库来创建刺激。在这个数据集中每个参与者被展示10个随机选择的有bug的程序，以及两个候选的修复程序:一个由RITE生成，一个由SEMINAI生成。对于这两种算法，我们使用返回的排名最高的解决方案。参与者一直都不知道哪个工具生成了哪个候选路径。参与者接下来被要求从1到5的Likert量表评估每个候选修复的质量而且被要求对每个修复的编辑位置的质量进行二进制评估。我们还收集了自我报告的编程和ocam特有经验评估，以及影响每个参与者修复质量主观判断的定性数据评估因素。从29个参与者中，我们收集了554个补丁质量评估，其中Rite和SEMINAL生成的修复各占277。

结果。在一个统计学意义的方式中，人们感觉RITE的错误位置和最终修复的质量都比其他由SEMINAL生成的更高（各为p=0.030和p=0.024）。关于错误位置，我们发现人们81.6的时候认同RITE识别出的编辑位置，而74.0%的时候认

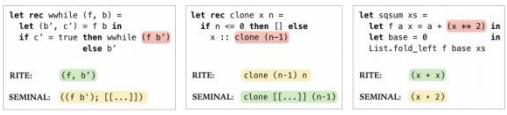
同SEMINAL识别的那些。对于最终修复而言，人们同样相比于SEMINAL所生成的更偏向于RITE的路径。特别地，RITE的修复质量达到了2.41/5，同时SEMINAL的修复质量只有一个2.11/5的平均值，RITE比SEMINAL提升14%(p=0.030)，这表明一个具有统计学意义相比SEMINAL的提升。

质量上的比较。我们考虑了几个案例研究，在这些案例中，人类对RITE和SEMINAL修复的评价存在统计学意义上的差异。图表10a的任务当中(f,b)应该在有值v0,...,vn存在的地方返回x，例如：b=v0,x=vn，以及对每个在0和n-2之间的i我们有fvi=(vi+1,true)和f(vw,false)。在图表b中的任务是返回一个x的n个副本的列表。图表10c的任务是返回xs列表中数字方阵的总和人们认为RITE的修复对图表10a和10c的项目更好。在两个案例中在，Rite都找到了一个解决方案，即类型检查和符合问题的语义规范。然而SEMINAL发现一个修复要么是不完整的(10a)要么语义错误(10 c)。另一方面，在10b,因为第二个参数应该是n-1，所以RITE效果更差。事实上,RITE的第二等级修复是正确的，但它根据编辑距离与第一个相等。人们感觉在具有统计学意义的方式中RITE的编辑位置和最终修复质量都比SEMINAL生成的一个先进OCAMI修复工具要好。

6.4 模板随质量的影响

最后，我们试图评估RITE的模板指导方法是否真的是其有效性的核心。要这样做，就像在6.2之中，我们比较了使用从预测模板合成的RITE错误消息和由返回第一个类型良好的术语的NAIVE合成器生成的错误消息的结果(即从简单的"洞"模板合成)。

用户研究设置。对于这个用户研究，我们运用在6.3中随机选择的一个有20个有bug的程序。对当中每个项目我们生成3个信息：用RITE的，用SEMINAL的，以及在RITE预测的同一个位置用NAIVE方法。我们接下来将它们随机化然后将被报告的工具信息的位置的顺序遮盖，且要求三个专家（这份论文的作者，还未看见任何一个工具对任何一个例子产生的结果）用"很好"，"还好"，"不好"三者之一来评价这些信息。



（a）RITE(4.5/5)的12个响应比SEMINAL(1.1/5)好 p=0.002

（b）RITE(1.5/5)的18个响应比SEMINAL(4.1/5)差 p=0.0002

（c）RITE(4.8/5)的17个响应比SEMINAL(1.2/5)好 p=0.0003

图10.三个错误的程序由RITE和SEMINAL在红色错误位置生成修复

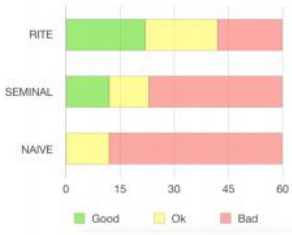


图11.评价由RITE，SEMINAL和NAIVE枚举生成的错误。

结果。图表11总结了评价的结果。因为20个程序中每个都收到了3个评价，因此每个工具总共有60个评价。RITE有20个"很好"，20个"还好"，18个"不好"具有压倒性优势；SEMINAL紧随其后，仅有12个"很好"，11个"还好"，以及37个"不好"；同时NAIVE没有收到了0个"很好"，12个"还好"的评价和一个凄惨的有48个"不好"分数的结果。平均下来（"不好"=0，"还好"=0.5，"很好"=1），RITE打分为0.53，SEMINAL为0.30，NAIVE为仅仅0.1。我们的评级协议kappa是0.54，这被认为是"温和的协议"。由预测模板生成的修复与那些专家偏爱的枚举（SEMINAL）或NAIVE枚举具有高很多的质量。

7 相关工作

有大量关于自动修复或修补程序的文献:我们关注的是最密切相关的工作，即为新手错误提供反馈。

基于例子的反馈。最近的工作使用了反例来说明程序是如何出错的，例如类型错误[36]或一般正确性属性，其中生成的输入显示出与参考实现或其他正确性oracle[39]的差异。与之相反，我们为怎样解决错误提供反馈。

错误定位。几个作者已经研究过错误定位的问题，即筛选一组与错误有关的位置，常用切分[12,31,40,41],反事实的类型[6],或贝叶斯方法[43]。NATE[37]介绍了BOAT表示，以及展示了它可以用于正确定位。我们目标是超越定位而建议新手能够做出的具体的改变来理解和解决问题。

基于修复模型的反馈。SMINAL[21]枚举了利用一个专家导向的启发式的研究的最小的解决方式。[38]将上述方法推广到一般正确性属性，它还使用一组专家提供的表示可能修复的草案执行符号搜索。与之相反，RITE从一个产生更高质量反馈的数据集学习了一个修复模板。

基于数据集的反馈。CLARA[11]使用代码和执行跟踪来匹配错误的程序，用一个"附近的"通过对特定任务的所有正确答案进行聚类获得的正确解决方式。类似地，SARFGEN[42]关注于产生修复的程序的结构和控制流相似性，通过使用AST向量嵌入来更稳健地计算距离度量(到"附近"正确的程序)。CLARA和SARFGEN是数据驱动的，但都猜测数据集中有一个"接近"正确的样本。相反，RITE有一个更普遍的观点就是相似的错误有相似的修复方式：我们提取我们提取通用的修复模板，可以应用于错误(BOAT向量)相似的任意程序。追踪系统[1]在观念上与我们的是最接近的，除了它专

注于C程序的单线编译错误，可表明基于NLP的类似于序列到序列预测深度神经网络的方式能够有效地建议修复，但这并不能按比例增加能解决的普遍类型的错误。我们已经发现OCAML的相对简单句法结构但丰富的类型结构使token级的序列到序列方法非常不精确(例如，删除违规语句就足以"修复"，但会产生类型不良的OCAML)，这就需要RITE更高层次的语义特征和(习得的)修复模板。

HOPPITY[7]是一个基于深度神经网络的用来解决有bug的JavaScript程序的方式。HOPPITY将程序视为适应于一个产生固定长度嵌入的图表神经网络的图表，这些嵌入接着会用于一个LSTM模型，能够产生一系列程序图表的初级编辑。HOPPITY是少数能跨多数位置修复错误工具的其中之一。但是，它单一地依赖已学习的模型来生成一系列编辑，所以它不能生成返回的可靠JavaScript程序。相反，RITE使用已学习的模型来得到正确的错误位置以及固定模板，然而接下来使用一个合成过程来产生总是类型正确的程序。

GETAFIX[3]和REVISAR[32]是两个更多的系统，用一个过去解决的bug的数据集AST级的不同学习固定的模式。它们对生成表达式和都使用不统一[19]，因此集群固定模式。它们将bug修复聚在一起，以减少候选补丁的搜索空间。当REVISAR[32]使用不统一为每个bug类别提供一个修复模式时，GETAFIX[3]构建了一个模式层次结构，其中还包括要进行编辑的上下文。它们都将前和后表达式对作为它们的修复模式，并且它们使用前表达式作为一种手段来匹配一个新的有bug的程序中的表达式，并将其替换为后表达式。虽然这些方法非常有效，但它们只适用于反复出现的错误类别，例如如何处理空指针异常。另一方面，RITE试图通过使用GAST抽象来更加一般化修复模式，并通过从错误修复语料库中学习到的模型来预测适当的错误位置和修复模式，因此可以应用于各种各样的错误。

PROPHET[23]是另一种技术，它使用一个有固定错误的程序语料库来学习将候选补丁排序的概率模型。补丁是使用一组预定义的转换模式和条件合成生成的。PROPHET使用后勤回归来学习这个模型的参数，并使用超过3500个提取的程序特征来这样做。它还使用错误程序的插装式重新编译以及一些失败的输入测试用例来确定感兴趣的程序位置。尽管这种方法对误差定位具有很高的准确性，但实验结果表明，这种方法要用2小内实现误差定位生成一个有效的候选修复。相比之下，RITE的预训练模型使查找适当的错误位置和可能的修复模板更加健壮。

8 结论

我们提出了分析程序修复，这是一种新的数据驱动方法，可以提供修复作为类型错误的反馈。我们的方法是使用一个病态程序及其固定版本的数据集来学习一组具有代表性的固定模板，通过多类分类，可以准确地预测新病态程序的固定模板。这些模板以易于处理和精确的方式指导程序修复的合成。我们已经在RITE中实现了我们的方法，并使用4500个类型错误的OCaml程序的语料库，从一个编程入门课程的两个实例中得出，RITE在考虑前三个模板时，准确的修复预测率为69%，在考虑前六个模板时，准确的修复预测率超过80%，预测的模板可以让我们在20秒内合成超过70%的测试集的修复。最后，我们对29名参与者进行了一项用户研究，结果表明，RITE的修复质量高于最先进的开创性工具的修复质量，该工具包含了一些专家指导的启发式方法，用于提高修复质量和错误消息。因此，我们的结果证明了数据对于生成更好的错误消息的不合理有效性。

致谢

我们感谢匿名推荐人和我们的引领者王珂为改进论文提出的出色建议。这项工作得到了NSF资助（CCF-1908633，CCF1763674）和空军资助（FA8750-19-2-0006，FA8750-19-1-0501）的支持。

参考文献

[1] Umair Z. Ahmed, Pawan Kumar, Amey Karkare, Purushottam Kar, and Sumit Gulwani. 2018. Compilation error repair: for the student programs, from the student programs. In International Conference on Software Engineering: Software Engineering Education and Training. 78ś87. https://doi.org/10.1145/3183377.3183383 [2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to Represent Programs with Graphs. arXiv:cs.LG/1711.00740 [3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: learning to fix bugs automatically. Proceedings of the ACM on Programming Languages 3, OOPSLA (Oct 2019), 1ś27. https: //doi.org/10.1145/3360585 [4] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning Python Code Suggestion with a Sparse Pointer Network. arXiv:cs.NE/1611.08307 [5] Christopher M. Bishop. 2006. Pattern Recognition and Machine Learning (Information Science and Statistics). Springer-Verlag, Berlin, Heidelberg, 209ś210. [6] Sheng Chen and Martin Erwig. 2014. Counter-factual Typing for Debugging Type Errors. In Principles of Programming Languages (POPL '14). ACM, New York, NY, USA, 583ś594. https://doi.org/10.1145/ 2535838.2535863 [7] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. Hoppity: Laerning Graph Transformations to Detect and Fix Bugs in Programs. In International Conference on Learning Representations. https://openreview.net/forum?id=SJeqs6EFvB [8] Matthias Felleisen and Robert Hieb. 1992. The revised report on the syntactic theories of sequential control and state. Theoretical Computer Science 103, 2 (1992), 235 ś 271. https://doi.org/10.1016/0304-3975(92) 90014-7 [9] Mark Gabel and Zhendong Su. 2010. A Study of the Uniqueness of Source Code. In Foundations of Software Engineering (FSE '10). ACM, New York, NY, USA, 147ś156. https://doi.org/10.1145/1882291.1882315 [10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. Deep Learning. MIT Press, 180ś184. http://www.deeplearningbook.org. [11] Sumit Gulwani,

Ivan Radicek, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. Programming Language Design and Implementation (2018). https://doi.org/10.1145/3192366.3192387 [12] Christian Haack and J B Wells. 2003. Type Error Slicing in Implicitly Typed Higher-Order Languages. In Programming Languages and Systems. Springer Berlin Heidelberg, 284ś301. https://doi.org/10.1007/3- 540-36575-3_20 [13] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. 2009. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Springer New York. https://doi.org/10.1007/978-0-387-84858-7 [14] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In Learning @ Scale. 89ś98. https://doi.org/10.1145/3051457. 3051467 [15] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In International Conference on Software Engineering (ICSE '12). Piscataway, NJ, USA, 837ś847. [16] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In International Conference on Software Engineering. 802ś811. https: //doi.org/10.1109/ICSE.2013.6606626 [17] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. arXiv:cs.LG/1412.6980 [18] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016.
Practitioners' Expectations on Automated Fault Localization. In International Symposium on Software Testing and Analysis. ACM, 165ś176. https://doi.org/10.1145/2931037.2931051 [19] Temur Kutsia, Jordi Levy, and Mateu Villaret. 2011. Anti-Unification for Unranked Terms and Hedges. Journal of Automated Reasoning 52, 219ś234. https://doi.org/10.4230/LIPIcs.RTA.2011.219 [20] Eelco Lempsink. 2009. Generic type-safe diff and patch for families of datatypes. Master's thesis. Universiteit Utrecht. [21] Benjamin S Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. 2007. Searching for Type-error Messages. In Programming Language Design and Implementation. ACM, 425ś434. https://doi.org/10. 1145/1250734.1250783 [22] Calvin Loncaric, Satish Chandra, Cole Schlesinger, and Manu Sridharan. 2016. A practical framework for type inference error explanation. In Object-Oriented Programming, Systems, Languages, and Applications. 781ś799. https://doi.org/10.1145/2983990.2983994 [23] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (St. Petersburg, FL, USA) (POPL '16). Association for Computing Machinery, New York, NY, USA, 298ś312. https://doi.org/10.1145/2837614. 2837617 [24] Matias Martinez, Laurence Duchien, and Martin Monperrus. 2013. Automatically extracting instances of code change patterns with AST analysis. In 2013 IEEE international conference on software maintenance. IEEE, 388ś391. [25] Matias Martinez and Martin Monperrus. 2015. Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering 20, 1 (2015), 176ś205.
[26] Jonathan P. Munson and Elizabeth A. Schilling. 2016. Analyzing Novice Programmers' Response to Compiler Error Messages. J. Comput. Sci. Coll. 31, 3 (Jan. 2016), 53ś61. http://dl.acm.org/citation.cfm? id=2835377.2835386 [27] Vinod Nair and Geoffrey E Hinton. 2010. Rectified linear units improve restricted boltzmann machines. In International Conference on Machine Learning. 807ś814. [28] Michael A Nielsen. 2015. Neural Networks and Deep Learning. Determination Press. [29] Chris Parnin and Alessandro Orso. 2011. Are Automated Debugging Techniques Actually Helping Programmers?. In International Symposium on Software Testing and Analysis. ACM, 199ś209. https: //doi.org/10.1145/2001420.2001445 [30] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding Minimum Type Error Sources. In Object Oriented Programming Systems Languages & Applications. ACM, 525ś542. https://doi.org/10.1145/ 2660193.2660230 [31] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. 2015. Skalpel: A Type Error Slicer for Standard ML. Electron. Notes Theor. Comput. Sci. 312 (24 April 2015), 197ś213. https://doi.org/10.1016/j. entcs.2015.04.012 [32] Reudismam Rolim, Gustavo Soares, Rohit Gheyi, Titus Barik, and Loris D'Antoni. 2018. Learning Quick Fixes from Code Repositories. arXiv:cs.SE/1803.03806 [33] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-hunk Program Repair. In International Conference on Software Engineering. 13ś24. https://doi.org/10.1109/ICSE.2019. 00020 [34] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. Neural Networks 61 (Jan 2015), 85ś117. https://doi.org/10. 1016/j.neunet.2014.09.003
[35] Eric L Seidel and Ranjit Jhala. 2017. A Collection of Novice Interactions with the OCaml Top-Level System. https://doi.org/10.5281/zenodo. 806813 [36] Eric L Seidel, Ranjit Jhala, and Westley Weimer. 2016. Dynamic Witnesses for Static Type Errors (or, Ill-typed Programs Usually Go Wrong). In International Conference on Functional Programming. 228ś 242. https://doi.org/10.1145/2951913.2951915 [37] Eric L. Seidel, Huma Sibghat, Kamalika Chaudhuri, Westley Weimer, and Ranjit Jhala. 2017. Learning to Blame:

Localizing Novice Type Errors with Data-driven Diagnosis. Proc. ACM Program. Lang. 1, OOPSLA, Article 60 (Oct. 2017), 27 pages. https://doi.org/10.1145/3138818 [38] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated feedback generation for introductory programming assignments. Acm Sigplan Notices 48, 6 (2013), 15ś26. [39] Dowon Song, Myungho Lee, and Hakjoo Oh. 2019. Automatic and Scalable Detection of Logical Errors in Functional Programming Assignments. Proc. ACM Program. Lang. 3, OOPSLA, Article 188 (Oct. 2019), 30 pages. https://doi.org/10.1145/3360614 [40] Frank Tip and T B Dinesh. 2001. A Slicing-based Approach for Locating Type Errors. ACM Trans. Softw. Eng. Methodol. 10, 1 (Jan. 2001), 5ś55. https://doi.org/10.1145/366378.366379 [41] Mitchell Wand. 1986. Finding the Source of Type Errors. In Principles of Programming Languages. 38ś43. https://doi.org/10.1145/512644.512648 [42] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-driven Feedback Generation for Introductory Programming Exercises. In Programming Language Design and Implementation. 481ś495. https://doi.org/10.1145/3192366.3192384 [43] Danfeng Zhang and Andrew C Myers. 2014. Toward General Diagnosis of Static Errors. In Principles of Programming Languages. 569ś581. https://doi.org/10.1145/2535838.2535870