软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2021,32(4):1006−1022 [doi: 10.13328/j.cnki.jos.006227]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

一种结构信息增强的代码修改自动转换方法*

曹英魁[1,2], 孙泽宇[1,2], 邹艳珍[1,2], 谢 冰[1,2]

[1](北京大学 信息科学技术学院,北京 100871)

[2](高可信软件技术教育部重点实验室(北京大学),北京 100871)

通讯作者: 谢冰, E-mail: xiebing@sei.pku.edu.cn

摘 要: 在开发过程中,开发人员在进行缺陷修复、版本更新时,常常需要修改多处相似的代码.如何进行自动代码修改已成为软件工程领域的热点研究问题.一种行之有效的方式是:给定一组代码修改示例,通过抽取其中的代码修改模式,辅助相似代码进行自动转换.在现有工作中,基于深度学习的方法取得了一定进展,但在捕获代码间的长程信息依赖关系时,效果不佳.为此,提出了一种结构信息增强的代码修改自动转换方法 ExpTrans.ExpTrans 在解析代码时采用图的形式来表示修改示例,显式地指出了代码中变量之间的依赖关系,同时结合图卷积网络和 Transformer 架构,增强了模型对代码的结构信息和依赖信息的捕获能力,从而提升了代码修改自动转换的准确性.实验结果表明,对比同类型基于深度学习的方法,ExpTrans 在准确率上提升了 11.8%~30.8%;对比基于人工规则的方法,ExpTrans 在修改实例的数量和准确率上均有显著提升.

关键词: 代码变更;代码演化;软件维护;代码生成

中图法分类号: TP311

中文引用格式: 曹英魁,孙泽宇,邹艳珍,谢冰.一种结构信息增强的代码修改自动转换方法,2021,32(4):1006−1022. http://www.jos.org.cn/1000-9825/6227.htm

英文引用格式: Cao YK, Sun ZY, Zou YZ, Xie B. Structurally-enhanced approach for automatic code change transformation. Ruan Jian Xue Bao/Journal of Software, 2021,32(4):1006−1022 (in Chinese). http://www.jos.org.cn/1000-9825/6227.htm

Structurally-enhanced Approach for Automatic Code Change Transformation

CAO Ying-Kui[1,2], SUN Ze-Yu[1,2], ZOU Yan-Zhen[1,2], XIE Bing[1,2]

[1](School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China)

[2](Key Laboratory of High Confidence Software Technologies of Ministry of Education (Peking University), Beijing 100871, China)

Abstract: In software development, developers often need to change or update lots of similar codes. How to perform code transformation automatically has become a research hotspot in software engineering. An effective way is: Extracting the change pattern from a set of similar code changes and apply it to automatic code change transformation. In the related work, deep-learning-based approaches have achieved much progress, but they suffer from the problem of significant long-dependency among code. To address this challenge, an automatic code change transformation method is proposed, namely ExpTrans, enhanced by code structure information. Based on graph-based representations of code changes, ExpTrans is enhanced with structural information of code. ExpTrans labels the dependency among variables in code parsing, adopts the graph-convolution network and transformer structure, so as to capture the long-dependency among code. To evaluate ExpTrans's effectiveness, it is compared with existing learning-based approaches first, the results show that ExpTrans gains 11.8%~30.8% precision increment. Then, ExpTrans is compared with rule-based the approaches, the results show that ExpTrans significantly improves the correct rate of the modified instances.

* 基金项目: 国家杰出青年科学基金(61525201); 国家自然科学基金(61972006)

Foundation item: National Natural Science Fund for Distinguished Young Scholars (61525201); National Natural Science Foundation of China (61972006)

本文由"面向领域的软件系统构造与质量保障"专题特约编辑潘敏学教授、魏峻研究员、崔展齐教授推荐.

收稿时间: 2020-09-13; 修改时间: 2020-10-26; 采用时间: 2020-12-19; jos 在线出版时间: 2021-01-22

Key words: code change; software evolution; software maintenance; code generation

1 引 言

在开发过程中,开发人员常常面临着相似的代码修改任务,而完成这些任务不仅需要耗费开发人员大量的时间和精力,同时,开发人员在完成重复的任务时又容易引入新的错误[1,2].得益于不同形式的版本控制系统,软件项目的演化历史得以完整地记录下来.在这些演化历史中,存在着丰富的代码修改场景和修改方案[3−6].基于已有的相似代码修改,研究人员开始关注于相似代码自动转换方法的研究,即:给定一组示例代码修改,通过对修改示例的修改方式进行抽取和表示,并基于表示结果来实现对相似代码的自动修改.

在早期工作中,代码修改的自动转换多是采用人工特征工程的方式[7−11],即人工地提出规则对修改示例中的修改方案的适配条件、修改过程和修改结果进行限定和说明.然而,这些规则的提出往往基于研究人员对特定语言的专家知识,并需要耗费大量的时间精力来总结、提炼.近来,采用基于机器学习的代码转换方法[12,13]不断涌现出来.常见的做法是采用端到端的翻译模式,将待修改的代码“翻译”为修改后的代码.然而,现有的工作尚未对修改示例的结构信息充分加以使用.一方面,一些现有工作利用翻译模型,直接将修改前的代码“翻译”为修改后的代码.然而,在缺失修改示例信息的条件下,试图训练一个全局的翻译模型来处理代码转换任务,无疑是十分困难的.另一方面,相比于自然语言,代码内的信息存在着更为显著的长程依赖.如图 1 所示,函数调用 fis.close()中的变量名 fis,与最初声明的变量名为同一个变量.现有工作在采用时序模型对代码信息建模时,由于两处代码间隔很远,很难捕获两处变量间的依赖关系.

```
FileInputStream fis = new FileInputStream (new File (dir));
try {
    ...
    fis. close ();
} catch  (IOException  e)  {
    ...
}
```

Fig.1 Long dependency among variable names

图 1 变量名间的长程依赖

针对上述问题,本文提出了一种利用结构信息增强代码转换的方法 ExpTrans.一方面,给定修改示例,方法将修改前和修改后的代码解析为抽象语法树,并寻找其节点间的对应关系.基于节点间的对应关系,方法采用图的形式来表示给定的示例代码修改,以增强模型对代码修改中的结构信息的捕获能力;另一方面,方法通过将图卷积网络和 Transformer 架构[14]有效地结合来增强模型对代码间的依赖信息,尤其是长程依赖关系的捕获能力.

方法整体上遵循 encoder-decoder 的架构设计.其中,编码器分别对待修改代码、修改示例代码以及中间信息进行建模;解码器依据编码器的编码结果,预测转换后的代码.为了保证方法产生的代码的可编译性,方法在预测修改后的代码时,借鉴了 Yin 和 Sun 等人的工作[15,16],以预测指令序列的方式来产生代码.所谓指令,形如$\alpha\rightarrow\beta_1\beta_2$…,其所代表的含义是在代码的抽象语法树中的类型为$\alpha$的节点上,插入子节点,类型分别为$\beta_1$、$\beta_2$….具体地,在产生代码的过程中,方法将维持一棵表示中间结果的抽象语法树,并基于预测的下一条指令,对当前最左的待扩展的节点进行扩充,直至所有的非叶子节点被扩展完毕,所生成的抽象语法树所对应的代码即为转换后的代码.

为了验证方法的有效性,本文在两个数据集上进行了对比实验.第 1 个数据集是 Yin 等人对外开放的 111 724 个 C#代码修改[13].实验结果表明,对比 Yin 的工作(基于深度学习)[13],本文方法在准确率上有 11.8%~30.8%的提升.第 2 个数据集是从 GitHub 上收集整理的 6 组 Java 语言典型的相似代码修改.在该数据集上,分别将本文方法、GenPat 方法[17](基于人工规则)和 ARES 方法[10](基于人工规则)用于自动修改这些代码实例.实验结果表明,在每组数据中,均有实例被 ExpTrans 正确修改,并且在一组数据上的实例全部被 ExpTrans 正确修改.相比于 GenPat 和 ARES 方法的结果,ExpTrans 的结果在准确率和正确修改的实例的数量上,均有大幅度提升.

本文的贡献主要表现在:(1) 提出了一种基于图的代码修改表示方法.相比采用单词序列的表示方式,图结构能够更加准确地表示代码修改过程,增强了方法对代码修改中结构信息的捕获能力.(2) 提出了一种基于Transformer 架构的结构信息增强的代码修改自动转换方法.该方法采用特殊的 copy 指令显式地表示代码间广泛存在的变量声明与使用的依赖关系,增加了模型对代码中长程依赖信息的捕获能力.(3) 本文在两个数据集上开展了对比实验并将数据全部公开(https://github.com/caoyingkui/ExpTrans).相比于现有的机器学习方法,本文方法 ExpTrans 在准确率上提升了 11.8%~30.8%.对比基于人工规则的方法,在修改实例的准确率和正确修改的实例数量上均有显著提升.

本文第 2 节介绍已有的相关工作.第 3 节介绍基于预测指令序列的代码产生方式,以及本文方法的整体框架.第 4 节介绍本文方法的具体实现细节.第 5 节介绍为了验证方法有效性而开展的实验以及实验设置和实验结果.最后,第 6 节对本文工作进行总结.

2 相关工作

如前所述,本文的相关工作分为两类:一类是基于人工特征的方法,另一类是基于深度学习的代码修改自动转换方法.

2.1 基于人工特征的方法

该类工作的主要思路是:基于人工提炼的规则,从给定的修改示例中,抽取一个代码转换“脚本”,以对示例中的修改条件、修改模式和修改过程进行说明和约束,并可以按照脚本中所约定的方式自动地适配到符合条件的代码区域并完成代码转换.在现有的工作中,这种脚本呈现出代码编辑序列[7]、模板[8]和特定领域语言(DSL)[11]等多种形式.

在 Meng 等人提出的 SYDIT 中,利用一组编辑操作序列来表示代码转换的过程[7].序列中的编辑操作共包含 4 种类型:增加、删除、修改和移动一个 AST 节点,同时利用通配符等方式编辑操作中的类型、位置信息以进行泛化.例如,!config.inValid()被表示为!v2.m5().按照顺序依次执行该序列中的操作,即可将代码修改为修改后的代码.LASE 是 Meng 等人提出的另外一种代码转换方法,并依旧利用一组编辑操作序列来表示示例代码中的代码转换[9].与 SYDIT 不同,LASE 是从多个相似修改示例中抽取代码转换.

此外,还有一些现有工作采用模板的形式来表示修改示例中的修改模式.如 Andersen 等人提出的方法spdiff[8],该方法从修改示例中抽取一个单词替换补丁(term replacement patch)用于刻画示例中的代码修改方法,并将一组修改示例中最长的公共子补丁作为该组示例代码修改中的代码转换.针对 LASE 在表示代码修改模式时,无法准确地识别代码语句移动等问题,Dotzler 等人提出方法 ARSE.ARSE 同样是一种基于多示例的代码转换方法[10].然而,与 LASE 不同,ARSE 以模板的方式来表示修改示例中的修改模式.Jiang 等人提出的方法 GENPAT 是一种从单一修改示例中抽取代码转换的方法[17].在该方法中,代码转换最终以树形结构的形式加以表示.在该结构中,每一个节点的信息包括节点 ID 和一组属性值.同时,GENPAT 的表示结果中还包含一组操作,其中的每一个操作为一个元组⟨id,id′⟩,id 和 id′分别代表了修改前、后代码的 AST 中的一个节点并存在对应关系,即节点 id 发生修改后变为节点 id′.

在 Rolim 等人提出的 REFAZER 中,定义了一种特殊的 DSL(领域特定语言)[11].其主要功能是定义对代码 AST 的操作,以及对符合特定修改的 AST 的条件及发生修改的位置和修改类型进行限定.因此,该代码生成任务的目标为:一段基于该 DSL 的代码.该代码的输入为一段修改前的代码,输出为修改后的一段代码.

2.2 基于深度学习的方法

该类工作的主要思路是:给定待修改的代码,利用机器学习模型预测修改后的代码.Tufano 等人提出了基于翻译模型完成代码转化方法[12].具体地,给定一段待修改的代码作为模型的输入,并利用翻译模型直接预测一个 token 序列,作为转换后的代码.同时,该方法也探索了翻译模型适合何种类型的代码转化场景.例如,缺陷修改、代码重构.代码作为一种高度结构化的文本,其功能和可编译性严格依赖于代码具体的 token 序列和 token 间的

相互位置关系.然而,以预测 token 序列方式进行工作的翻译模型,无法从方法层面上保证模型的输出结果是可编译的.为了克服这一问题,在 Yin 等人提出的方法中,以预测指令序列的方式来生成代码,从而能够保证模型输出代码的语法正确性[15].基于此工作机制,Yin 等人提出一种基于学习的代码转换模型[13].该模型的输入,除一段待修改的代码外,同时输入一个修改示例的修改前、后的代码,即通过学习修改示例中的代码转换方式,来指导待修改的代码转换过程.

综上,Yin等人提出的方法是目前现有工作中与本文最为相关的方法.在Yin等人的方法中,利用了LSTM时序模型用于处理代码的文本信息.然而,相比于自然语言,代码存在更为显著的长程依赖关系,然而有研究表明,时序模型在处理信息长程依赖时效果并不佳[14].因此,克服代码的信息长程依赖挑战,是提出本文方法的一个主要的动机.

3 方法框架

为了保证方法生成代码的可编译性,本文借鉴了 Yin 等人的工作[15],采用以预测指令序列的方式来生成代码,而非单词序列.同时,方法在代码指令集合中,增设了特殊的 copy 指令,以显式地刻画代码中变量间的依赖关系.在介绍方法框架之前,本节首先详细介绍方法中基于预测指令序列的代码产生方式.

代码指令.在本文中,所谓指令,指的是形如$\alpha\rightarrow\beta_1\beta_2\ldots$的产生式.其中,$\alpha$为非终结符,$\beta_i$ 为终结符或非终结符.在代码的抽象语法树上,每个非叶子节点均对应一条指令$\alpha\rightarrow\beta_1\beta_2\ldots$,其中,$\alpha$为该节点的语法类型,$\beta_i$ 为其子节点的语法类型(或单词).

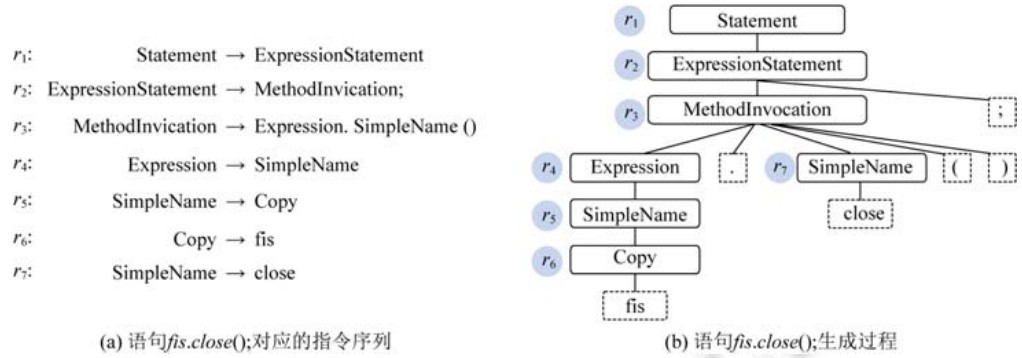
指令集的获取.给定一棵代码抽象语法树,假定其中某个节点 v 的语法类型为α,节点 v 共有 n 个子节点,且其语法类型(或单词)分别为$\beta_1,\ldots,\beta_n$,则依据节点 v,可获取指令 $r:\alpha\rightarrow\beta_1\ldots\beta_n$.按照由上到下、从左至右的顺序依次遍历抽象语法树的所有非叶子节点,即可获取相应的代码指令序列.同时,通过遍历已有数据集中代码的抽象语法树,即可获取指令集$\{r_1,\ldots,r_N\}$.

在代码中,由于开发人员可以采用任何合法的变量名、字符串等,导致代码中存在比自然语言更为显著的OOV(out of vocabulary)问题.因此方法对出现在指令中的单词进行限制和预处理,从而避免所获取的指令集的规模过于庞大,不利于方法进行建模.具体地,在获取指令之前,本文统计了在给定数据集中的代码中出现的所有变量名、字符串、数字常量、字符常量及其出现的次数,只有当出现次数超过阈值的单词,才会将其保留.当代码中某个单词出现次数低于阈值时,本文方法将采用形如type_id的形式进行替换,其中,type表示该单词对应的类型,即变量名、字符串、数字常量或字符常量,id 为其编号.通过这样的处理,能够保证所获取的指令规模在有限的范围内.

此外,本文在指令集中额外增设了一类特殊的copy指令.在代码中,变量声明和使用之间隐含着明确的依赖关系.本文利用增设的 copy 指令,显示地标记变量声明与使用间的这种依赖关系.如在图 1 中,语句"*fis.close*();"中的变量名 fis 是由语句"*FileInputStream fis=new FileInputStream*(*new File*(*dir*));"声明得来.因此,方法在解析*fis.close*();时,采用 copy 指令,以标记该变量名 *fis* 由复制之前定义的 *fis* 而得来.采用 copy 指令主要有两方面的优势:一方面,显式地标记出变量间依赖关系的方式,有利于增强后续模型对变量间的依赖关系的捕获能力.另一方面,copy 指令的复制变量的范围,是代码已定义的变量名集合.因此,方法在后续的预测过程中,所面临的预测空间即为代码已定义的变量名集合.相比于整个单词空间,copy 指令缩小了在预测变量名时的预测空间,从而能够提升方法的准确率.

预测指令序列的代码产生方式.图 2 展示了基于指令序列$[r_1,\ldots,r_7]$生成代码"*fis.close*();"的过程.给定图 2(a)的指令序列,方法首先初始一个只有根节点的抽象语法树,其类型为 Statement.基于第 1 条指令,即 Statement→ExpressionStatement,方法为根节点插入一个类型为 ExpressionStatement 的子节点.如图 2(b)所示,当完成前 3 条指令后,当前的抽象语法树最下层将含有 2 个待扩展的节点,分别是 Expression 和 SimpleName.由于"**.**""("和")"已经是终结符,因此在后续过程中,方法将不会对它们进行扩展.此时,方法将利用下一条指令r_4来扩展最左边的待扩展的节点(非终结符),即左侧的 Expression 节点.按照上述方式,最终将产生一棵完整的抽象语法树.按照从

左至右的方式,获取所有的叶子节点内容,即为所生成的代码.



(a) 语句*fis.close*();对应的指令序列 (b) 语句*fis.close*();生成过程

Fig.2 The rule sequences and generation of *fis.close*();

图 2 语句 *fis.close*();对应的指令序列和生成过程

基于上述方式,本文将以预测指令序列的方式,实现代码修改的自动转换.方法的输入为 x 和 $x_\Delta \to y_\Delta$,其中,x 为待修改的代码,$x_\Delta \to y_\Delta$ 表示一个修改示例(x_Δ 和 y_Δ 分别为修改前、后的代码).最终的输出为 $\bar{y}$, 表示代码转换后的结果.在生成代码 $\bar{y}$ 的过程中,方法将维持一棵表示中间结果的抽象语法树.通过预测下一条指令,并基于该指令对当前的抽象语法树最左边待扩展的节点进行扩展,直至生成最终的代码.本文将生成代码 $\bar{y}$ 的概率形式化地表示为

$$p(\bar{y}) = \prod p(r_i \mid x, x_\Delta \to y_\Delta, r_1, ..., r_{i-1}) \tag{1}$$

其中,$r_1,\ldots,r_i$ 为产生的指令序列.

图 3 展示了本文方法 ExpTrans 的整体框架.

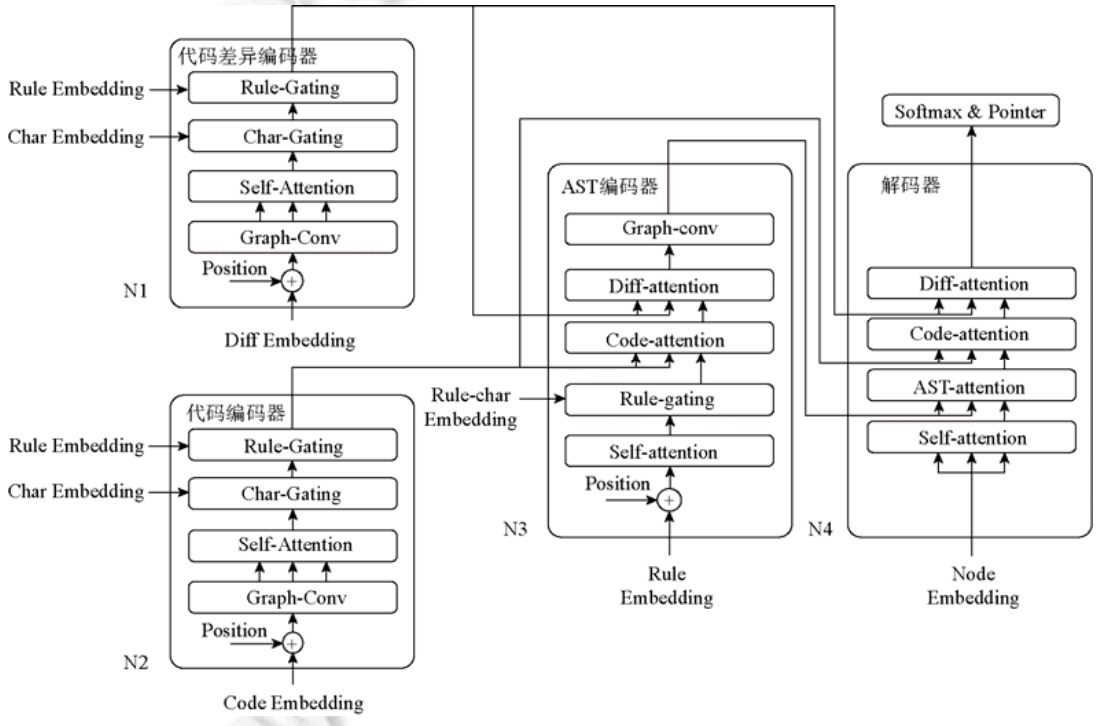


Fig.3 The neural network of ExpTrans

图 3 ExpTrans 的网络结构

方法整体上遵从 encoder-decoder 的架构模式,主要包含 4 个模块:代码差异编码器、代码编码器、AST 编码器和解码器.这 4 个子模块均采用了 Transformer 的多块(block)架构[14],即每一个子模块均由多个结构相同的块组成.例如,在代码差异编码器中,每一个块均由相同的网络结构组成,包含 Graph-conv 层、Self-attention 层、Char-gating 层和 Rule-gating 层,并且按照前一块的输出为后一块的输入的方式进行连接.在每一个块的内部结构中,采用残差连接的方式[18],将相邻的网络层(如 Graph-conv 层与 Self-attention 层之间)进行连接.需要注意的是,图 3 只展示了每个模块的一个块.4 个模块的主要功能如下.

代码差异编码器.方法利用该模块,对输入的修改示例 $x_\Delta \to y_\Delta$ 的信息进行建模.

代码编码器.方法利用该模块,对输入的待修改代码 x 的信息进行建模.

AST 编码器.在生成代码时,方法需要维持一棵表示中间状态的抽象语法树.方法利用该模块对该抽象语法树的信息进行建模,以期在预测代码指令时,能够提供语法树的全局信息.

解码器.在生成代码时,解码器将依据已生成的抽象语法树中待扩展的节点,预测一下指令.具体地,方法以待扩展的节点信息作为查询,输入解码器.基于输入的查询,解码器采用注意力机制来结合代码编码器、代码差异编码器和 AST 编码器的建模信息.然后根据解码器的输出,方法结合 softmax 和指针网络[19]以预测下一条指令.

4 本文方法

给定待修改代码 x 以及修改示例 $x_\Delta \to y_\Delta$,方法的目标是实现代码转换 $\bar{y} = Trans(x, x_\Delta \to y_\Delta)$, 其中, $\bar{y}$ 为转换后的代码.本节将介绍 ExpTrans 中不同模块的具体实现.

4.1 代码差异编码器

方法利用代码差异编码器,对修改示例 $x_\Delta \to y_\Delta$ 的修改方式、代码结构信息进行建模.基于 $x_\Delta \to y_\Delta$,分别将修改前、后代码 x_Δ 和 y_Δ 表示为抽象语法树的形式.按照由上到下、从左至右的顺序,获取两棵语法树所对应的节点序列 $\left[v_1^{(\mathrm{ori})},\ldots,v_{L^{(\mathrm{ori})}}^{(\mathrm{ori})}\right]$ 和 $\left[v_1^{(\mathrm{mod})},\ldots,v_{L^{(\mathrm{mod})}}^{(\mathrm{mod})}\right]$. 然后将两个序列合并成同一个序列并统一地记为 $V=\left[v_1,\ldots,v_{L^{(\mathrm{ori})}},\ldots,v_{L^{(\mathrm{ori})}+L^{(\mathrm{mod})}},\ldots,v_L\right]$, 其中,$L$ 为方法预设的最大长度,前 $L^{(\mathrm{ori})}$ 个节点为修改前的节点序列,随后 $L^{(\mathrm{mod})}$ 个节点为修改后的节点.当修改前、后的节点序列长度小于 L 时,方法利用特殊的占位符号⟨*EMPTY*⟩进行扩充.代码差异编码器将修改示例 $x_\Delta \to y_\Delta$ 建模为节点序列$[v_1,\ldots,v_L]$,代码差异编码器的输出为 $Y^{(diff)}=\left[y_1^{(diff)};\ldots;y_L^{(diff)}\right]^T$, 其中, $y_i^{(\mathrm{diff})} \in \mathbb{R}^H$ 为节点 v_i 的表示向量,H 表示空间纬度(在方法中设定为 128).

4.1.1 代码差异的图表示

修改示例 $x_\Delta \to y_\Delta$,以修改前、后代码分离的形式存在,采用单独对 x_Δ 和 y_Δ 进行建模的方式,将损失修改前、后代码间的关联关系及结构信息.为了更加精确地表示 $x_\Delta \to y_\Delta$ 中所蕴含的代码修改方式,本文将 $x_\Delta \to y_\Delta$ 表示为一个统一的图 $G=\langle V,E\rangle$,其中,节点集合 V 包含 x_Δ 和 y_Δ 所对应的抽象语法树节点的集合,E 为节点间的连边集合.

为了建立 x_Δ 和 y_Δ 节点间的连边关系,本文利用工具 GumTree[20]来获取代码修改前、后节点的对应关系.GumTree 是一种基于代码抽象语法树的代码差异抽取工具.在工作时,GumTree 首先获取修改前、后的代码 x_Δ 和 y_Δ 所对应的抽象语法树 $tree_x$ 和 $tree_y$.然后按照自底向上的顺序,对比 $tree_x$ 和 $tree_y$ 节点间的类型信息和文本信息,并据此计算节点的相似度,从而获取 $tree_x$ 和 $tree_y$ 节点间的最佳匹配关系,并据此能够准确地推导出代码的修改过程.

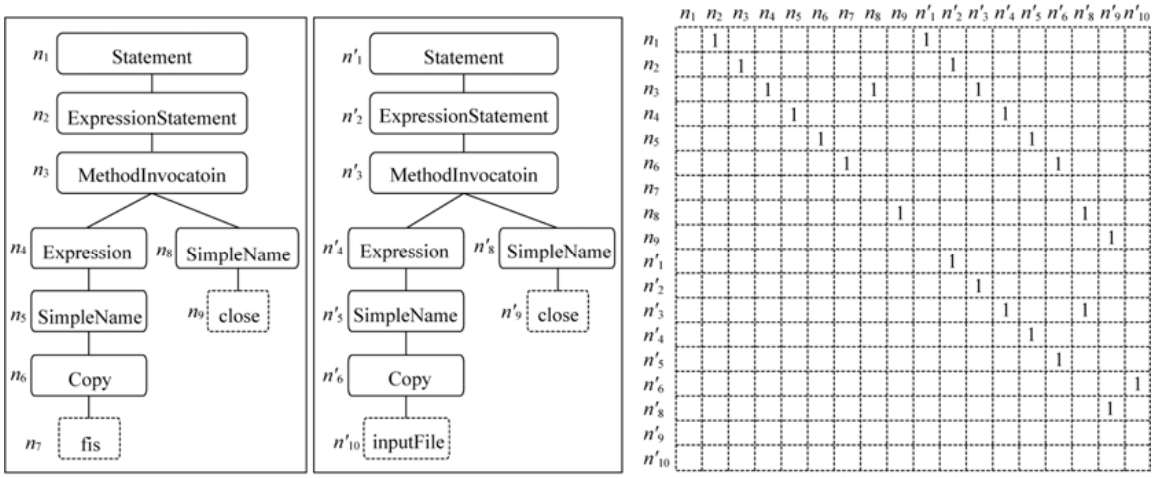
在本文中,当节点 v_j 与节点 v_i 间存在连边,且由 v_j 指向 v_i 时,我们称节点 v_j 为节点 v_i 的父节点.进一步地,v_j 的父节点为 v_i 的祖父节点.结合工具 GumTree 的结果,当节点 v_i 和 v_j 满足下列 3 种关系之一时,则建立一条由 v_j 指向 v_i 的连边,并将该边加入集合 E 中.

(1) 节点 v_i 和 v_j 均为 $tree_x$ 上的节点,且 v_j 为 v_i 的父节点.

(2) 节点 v_i 和 v_j 均为 $tree_y$ 上的节点,且 v_j 为 v_i 的父节点.

(3) 节点 v_i 为 $tree_y$ 上的节点,v_j 为 $tree_x$ 上的节点,并且在 GumTree 的结果中,v_i 和 v_j 存在对应关系.

基于集合 E,可以构造节点间的邻接矩阵 M,当 v_j 为 v_i 的父节点时,$M[i][j]$=1;否则,$M[i][j]$=0.例如,现将代码“*fis.close*();”修改为“*inputFile.close*();”,图 4 给出了采用图结构表示本处代码修改的过程和结果.如图 4(a)所示,首先将修改前、后的代码表示为抽象语法树的形式(这里,省略了‘.’‘(’等符号以方便展示).基于获取的抽象语法树,利用 GumTree 来获取修改前、后代码抽象语法树节点间的对应关系.在图 4(a)中,节点 n_i 与 n_i' 表示两个节点存在对应关系.最终,该处代码修改的图结构被表示为图 4(b)所示的邻接矩阵.



(a) 代码“*fis.close* ();”与“*inputFile.close* ();”对应的抽象法树 (b) 邻接矩阵

Fig.4 The illustration of construction of code change adjacent matrix

图 4 代码差异邻接矩阵建立过程示意图

4.1.2 编码信息的抽取

在计算 $Y^{(diff)}$之前,方法首先获取每个节点不同方面的初始信息.

节点的单词信息.在代码的抽象语法树中,每个非叶子节点均有相应的语法类型,每个叶子节点则对应代码中的一个单词.基于图 G 的节点序列 V,可以得到单词序列$[w_1,\ldots,w_L]$.当 v_i 为非叶子节点时,w_i 为节点对应的语法类型;当 v_i 为叶子节点时,w_i 为节点对应的单词.通过 embedding 方式,为每个单词 w_i 初始一个表示向量 $y_i^{(w)} \in \mathbb{R}^H$, 节点序列$[v_1,\ldots,v_L]$对应的单词信息为 $Y^{(w)}=[y_1^{(w)};\ldots;y_L^{(w)}]$.

节点单词字符信息.有些语义相似的单词存在相同的字符序列.例如,同一词根派生的不同类型的单词.而上述将单词作为整体进行信息编码的方式,将丢弃单词的字符信息.为了在节点表示向量中引入单词的字符信息,本文借鉴了 Sun 等人对单词的字符信息进行编码的方法[16].具体地,基于获取的单词序列,将每个单词 w_i 切分为字符序列$[c_{i,1},\ldots,c_{i,L'}]$,其中,$L'$为模型预设的单词最大长度.类似地,利用 embedding 方式为每个字符 $c_{i,j}$ 初始一个表示向量 $y_{i,j}^{(char)} \in \mathbb{R}^H$, 并利用全连接层,按照公式(2)的方式,计算单词 w_i 的字符表示向量 $y_i^{(char)}$:

$$y_i^{(char)}=W^{(char)}\left[y_{i,1}^{(char)};\ldots;y_{i,L'}^{(char)}\right] \tag{2}$$

其中, $y_i^{(char)} \in \mathbb{R}^H$, $W^{(char)}$为网络参数.据此,节点序列$[v_1,\ldots,v_L]$对应的单词字符信息为 $Y^{(char)}=\left[y_1^{(char)};\ldots;y_L^{(char)}\right]$.

节点指令信息.在抽象语法树中,每个非叶子节点 v_i 均对应一条指令 $r_i:\alpha^{(i)}\rightarrow\beta_1^{(i)}\ldots\beta_{n_i}^{(i)}$, 其中,$\alpha^{(i)}$为节点 v_i 对应的语法类型,所有$\beta^{(i)}$均为节点 v_i 的子节点的语法类型或单词.采用类似公式(2)的方式,方法为每条指令 r_i: $\alpha_i\rightarrow\beta_1^{(i)}\ldots\beta_{n_i}^{(i)}$ 计算表示向量 $y_i^{(rule)} \in \mathbb{R}^H$. 此外,叶子节点所对应的指令,统一采用特殊指令⟨*EMPTY_RULE*⟩进行替代.据此,节点序列$[v_1,\ldots,v_L]$对应的节点指令信息为 $Y^{(rule)}=\left[y_1^{(rule)};\ldots;y_L^{(rule)}\right]$.

位置信息.在 Transformer 的架构中,方法将时序数据(例如代码的节点序列的表示向量)打包成一个向量矩

阵,以使得模型能够并行训练.但这样的处理方式会丢失数据的位置(时序)信息.因此,需要额外添加数据的位置信息.在本文中,采用 Dehghani 的工作做法[21],利用下列公式人为地构造每个节点的位置信息:

$$p_{b,i}[2j]=\sin((i+b)/(10000^{2j/H})) \tag{3}$$

$$p_{b,i}[2j+1]=\cos((i+b)/(10000^{2j/H})) \tag{4}$$

其中, $p_{b,i}\in\mathbb{R}^H$ 表示在第 b 个块中第 i 节点的位置信息.据此,节点序列$[v_1,...,v_L]$对应的位置信息为 $P^{(\mathrm{diff})}=[p_{b,1};\cdots;p_{b,L}]$.

4.1.3 代码差异编码器的网络结构

Graph-conv 层.在将 $x_\Delta\to y_\Delta$ 表示为节点序列后,模型将无法捕获节点间的结构信息.因此本文方法利用一层图卷积层来实现在每个节点的表示向量中引入代码的结构信息.

基于构造的图 $G=\langle V,E\rangle$和邻接矩阵 M,假定当前节点的表示向量矩阵 $F=[f_1;\ldots;f_L]$,方法将按照公式(5)计算每个节点的父节点的表示向量:

$$\left[f_1^{(\mathrm{par})};...;f_L^{(\mathrm{par})}\right]=[f_1;...;f_L]M \tag{5}$$

其中, $f_i^{(\mathrm{par})}\in\mathbb{R}^H$ 为第 i 个节点的父节点表示向量,$[f_1;\ldots;f_L]M^2$ 则为节点的祖父节点表示向量,依次类推.

在卷积时,方法预设了卷积窗口 K,并按照如下方式进行卷积:

$$\mathrm{Conv}(F,M)=f(W^{(\mathrm{conv})}[F;FM;...;FM^{K-1}]) \tag{6}$$

其中,$W^{(\mathrm{conv})}$为卷积参数,f 为 Rule 激活函数.

该层的输入为节点的表示向量和节点的位置信息之和,即 $I=[x_{b,1}+p_{b,1};\ldots;x_{b,L}+p_{b,L}]$,该层的输出为

$$Y^{(\mathrm{conv})}=\mathrm{Conv}(I,M) \tag{7}$$

其中,当 b=1 时(即为第 1 个块),$x_{b,i}$ 表示对应节点的单词信息(即 $y_i^{(\mathrm{w})}$);而对于其他块,$x_{b,i}$ 则为前一块的输出.

Self-attention 层.该层采用了 Transformer 中的 self-attention 注意力机制[14],以捕获代码间的长程依赖关系.

在 Transformer 中,注意力机制被表述为将查询 Q、键值对 K 和 V 映射到输出的过程,其中,Q、K 和 V 均为向量矩阵.输出为对 V 中分量加权求和的结果.而 V 中每个 value 相应的权重将依据 Q 与相应的关键字 K 的匹配程度给出,其加权求和的结果为

$$\mathrm{Attention}(Q,K,V)=\mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{8}$$

其中,d_k 为缩放因子.同时,Transformer 采用 multi-head attention 机制,使得模型能够从不同的表示空间的角度来注意到不同位置的信息,即:

$$\mathrm{Multihead}(Q,K,V)=\mathrm{concat}(head_1,...,head_h)W^O \tag{9}$$

其中, $head_i=\mathrm{Attention}(QW_i^{(Q)},KW_i^{(K)},VW_i^{(V)})$, h 为 head 个数, W^o、$W_i^{(Q)}$、$W_i^{(K)}$ 和 $W_i^{(V)}$ 均为模型参数.

特别地,当 Q、K 和 V 相同时,即形如 Multihead(Q,Q,Q)的形式,则被称为 self-attention.该层实现了 self-attention 注意力机制,该层的输出为

$$Y^{(\mathrm{self})}=Multihead(Y^{(\mathrm{conv})},Y^{(\mathrm{conv})},Y^{(\mathrm{conv})}) \tag{10}$$

Char-gating 层.在该层之前,节点序列被表示为向量矩阵 $Y^{(\mathrm{self})}$,为了在节点 v_i 的表示向量中引入字符信息 $y_i^{(\mathrm{char})}$, 方法在该层中采用门机制,使得节点的表示向量矩阵更新为 $Y^{(\mathrm{self})}$和 $Y^{(\mathrm{char})}$的加权和.

门(gating)机制的目的在于对不同表示空间的表示结果进行综合,形成最终的表示向量.给定表示向量 f_1 和 f_2,门机制的目标是将 f_1 和 f_2 进行加权求和,获取综合 f_1 和 f_2 后的表示向量 $f^{(\mathrm{gate})}$.其中,f_1 和 f_2 的权重按照下列公式进行计算:

$$[\gamma_1,\gamma_2]=\mathrm{weight}(f_1,f_2)=\mathrm{softmax}\{f_1,f_2\} \tag{11}$$

则加权结果 $f^{(\mathrm{gate})}$为

$$f^{(\mathrm{gate})}=\mathrm{gate}(f_1,f_2)=[\gamma_1,\gamma_2]\times[f_1,f_2]^T \tag{12}$$

相似地,当给定向量矩阵 $F_1=\left[f_{1,1};...;f_{1,|F_1|}\right]$ 及 $F_2=\left[f_{2,1},...,f_{2,|F_2|}\right]$, 且满足$|F_1|=|F_2|$时,两者通过门机制的加权结果 Gate($F_1$,$F_2$)为

$$\text{Gate}(F_1,F_2)=[\text{gate}(f_{1,1},f_{2,1});\text{gate}(f_{1,2},f_{2,2});...] \tag{13}$$

为了保证 $Y^{(\text{self})}$依旧为加权后的结果的信息主体,方法首先基于 $Y^{(\text{self})}$通过线性变换的方式获取控制向量 $C^{(\text{self})}$.然后按照下列公式来计算 $Y^{(\text{self})}$和 $Y^{(\text{char})}$的加权结果作为该层的输出:

$$Y^{(\text{char-gate})}=\text{Gate}(Y^{(\text{self})^T}C^{(\text{self})},Y^{(\text{char})^T}C^{(\text{self})}) \tag{14}$$

Rule-gating 层.类似地,该层同样采用门机制,使得在节点的表示向量中融合指令字符信息 $Y^{(\text{rule})}$,该层的输出为

$$Y^{(\text{diff})}=\text{Gate}(Y^{(\text{char-gate})^T}C^{(\text{char-gate})},Y^{(\text{rule})^T}C^{(\text{char-gate})}) \tag{15}$$

其中,$C^{(\text{char})}$为通过 $Y^{(\text{char-gate})}$线性变换得来的控制向量.

最终,代码差异编码器的输出为 $Y^{(\text{diff})}$.

4.2 代码编码器

代码编码器的作用是为了实现对待修改代码 x 的信息建模.如图 3 所示,代码编码器和差异编码器的网络结构是一致的,只是在获取该层的输入方式上有所不同.

给定待修改代码 x,通过将其解析为抽象语法树 $tree_x$,并按照由上到下、从左至右的方式,获取节点序列 $V^{(x)}=[v_1^{(x)},...,v_L^{(x)}]$. 基于 $tree_x$,构造图 $G^{(x)}=\langle V,E\rangle$,其中,$V=V^{(x)}$,$E$ 为 $tree_x$ 中节点的连边集合.此外,方法同样为 $G^{(x)}$构造其节点间的邻接矩阵 $M^{(x)}$.

代码编码器采用与代码差异编码器中相同的数据预处理方式来获取节点序列 $V^{(x)}$对应的单词信息、节点单词字符信息、节点指令信息和位置信息.然后,信息依次经过 Graph-conv 层、Self-attention 层、Char-gating 层和 Rule-gating 层,并记代码编码器的最终输出为 $Y^{(x)}$.

4.3 AST编码器

本文方法在产生代码的过程中,需要维持一个由已产生指令序列生成的抽象语法树.方法以当前抽象语法树的最左非叶子节点(待扩展节点)为查询,预测下一条指令,并利用预测的指令对该节点进行扩充.因此,需要对产生代码过程中的抽象语法树的信息进行编码,以在预测下一条指令时,能够为模型提供抽象语法树的全局视图.在 AST 编码器中,方法利用已产生的指令序列$[r_1,...,r_p]$来表示已产生的抽象语法树.AST 编码器的目标则是:将指令序列最终表示为$[y_1^{(\text{ast})};...;y_P^{(\text{ast})}]$, 其中, $y_i^{(\text{ast})}$ 为指令 r_i 的表示向量,P 为方法预设的最大指令序列长度.

4.3.1 编码信息的提取

指令的初始向量.利用 embedding 方式,为每条指令 r 初始一条表示向量 $\bar{r}$. 因此给定指令序列$[r_1,...,r_R]$, 通过查表的方式,获取指令序列的初始表示向量矩阵 $R^{(\text{init})}=[r_1^{(\text{init})};...;r_P^{(\text{init})}]$.

指令的符号信息.在形如$\alpha\rightarrow\beta_1\beta_2...$的指令表示形式中,符号$\alpha$和$\beta_i$ 对于表达指令的语义信息十分重要.例如,当待扩展的节点的类型为 Statement 时,则下一条预测的指令的非终结符α必须同样为 Statement,以保证代码的合法性.然而,上述将指令作为整体进行编码的方式,将忽略指令涉及的符号信息.为了在指令的表示向量中引入指令符号序列的信息,本文采用了 Sun 的工作方式[16].具体地,给定指令 r_i:$\alpha\rightarrow\beta_1\beta_2...,\alpha$及所有的$\beta_j$均为标准单词集中的一个单词.通过查表的方式,可以获取每个单词对应的表示向量$\alpha^{(w)}$或 $\beta_j^{(w)}$. 类似公式(2),利用全连接层,以$\alpha^{(w)}$和所有 $\beta_j^{(w)}$ 作为输入,并将输出记为 $r_i^{(\text{char})}$. 最后,再次利用全连接层实现如下的编码方式:

$$r_i^{(\text{rule})}=W^{(\text{rule})}[r_i^{(\text{init})},r_i^{(\text{char})},\bar{\alpha}] \tag{16}$$

其中,$W^{(\text{rule})}$为网络参数.指令序列$[r_1,...,r_p]$对应的符号信息为 $R^{(\text{rule})}=[r_1^{(\text{rule})};...;r_P^{(\text{rule})}]$.

位置信息.本文方法同样利用公式(3)和公式(4)来计算指令序列的位置信息 $P^{(\text{ast})}=[p_{b,1}^{(\text{ast})},p_{b,2}^{(\text{ast})},...]$, 其中,

$p_{b,i}^{(\text{ast})}$ 为第 b 个块中的第 i 条指令的位置信息.

4.3.2 AST 编码器的网络结构

Self-attention 层.方法利用一层 Self-attention 层来捕获指令间的依赖关系.该层结构和代码差异编码器中的 Self-attention 层的结构是一致的.该层输入为指令表示向量与位置信息之和,即 $I^{(\text{ast})}=[r_{b,1}+p_{b,1}^{(\text{ast})};\ldots;r_{b,P}+p_{b,P}^{(\text{ast})}]$. 该层的输出为

$$R^{(\text{self})}=Multihead(I^{(\text{ast})},I^{(\text{ast})},I^{(\text{ast})}) \tag{17}$$

当 b=1 时(即为第 1 块),$r_{b,i}$ 表示指令 r_i 的初始向量(即 $r_i^{(init)}$);而在其他块中,$r_{b,i}$ 为前一块的输出.

Rule-gating 层.如上文所述,指令的符号信息对于表示指令的语义十分重要,因此,本文方法同样利用门机制来实现对指令符号信息的获取.与之前的 gating 层相似,首先基于 $R^{(\text{self})}$获取控制变量矩阵 $C^{(\text{r-self})}$,然后按照下列公式进行计算:

$$R^{(\text{gate})}=\text{Gate}(R^{(\text{self})T}C^{(\text{r-self})},R^{(\text{rule})T}C^{(\text{r-self})}) \tag{18}$$

Code-attention 层.代码转换需要依据待修改代码 x 进行转换,因此在后续的代码指令预测过程中,模型需要获取待修改代码的信息.本文方法利用该层来实现在指令序列的编码结果中,引入待修改代码 x 的信息($Y^{(x)}$).该层的输出为

$$R^{(x)}=Multihead(Y^{(x)},R^{(\text{gate})},R^{(\text{gate})}) \tag{19}$$

Diff-attention 层.针对代码 x 进行修改时,所修改的方式依赖于给定修改示例 $x_\Delta \to y_\Delta$.因此,在后续的代码指令预测过程中,模型需要修改示例的差异信息.本文方法利用该层来实现在指令序列的编码结果中,引入 $x_\Delta \to y_\Delta$ 的信息($Y^{(diff)}$).该层的输出为

$$R^{(\text{diff})}=Multihead(Y^{(\text{diff})},R^{(x)},R^{(x)}) \tag{20}$$

Graph-conv 层.由已生成的代码指令可以构造出局部的抽象语法树,而预测的指令均会对应特定的节点,因此,指令(节点)间存在有意义的结构关系.然而,以指令序列的形式表示已生成的抽象语法树,将会遗失指令间的结构信息.因此,本文方法采用一层图卷积层,利用代码的抽象语法树的结构信息来增强指令的编码信息.

依据指令对应节点的邻接关系,可以获得一个指令间的邻接矩阵 M_{p*p}.当 $M[i][j]$=1 时,表示指令 r_j(对应的节点)为指令 r_i (对应的节点)的父节点.基于此,该层的输出为

$$R^{(\text{ast})}=\text{Conv}(R^{(\text{diff})},M) \tag{21}$$

最终 AST 编码器的输出为 $R^{(\text{ast})}$.

4.4 解码器

解码器以当前待扩展的非叶子节点为输入,即 $Q^{(\text{d})}=[q_1;\ldots;q_R]$,其中,$q_i$ 为每个待扩展节点的表示向量.由于解码器仍然遵循多块的设计,因此在第 1 块中,q_i 为节点对应的语法类型或单词信息;而在其他块中,q_i 则为上一个块的输出.解码器的目标是生成查询矩阵 $D^{(\text{query})}=[d_1^{(\text{query})};\ldots;d_R^{(\text{query})}]$, 其中, $d_i^{(\text{query})}$ 为第 i 个节点相应的查询向量,方法将依据 $d_i^{(\text{query})}$ 预测第 i 条指令.解码器首先利用 Self-attention 层来获取待扩展节点间的依赖关系.然后,数据流将依次经过 AST-attention 层、Code-attention 层和 Diff-attention 层,分别用于获取抽象语法树、待修改代码和修改示例的信息.具体地:

Self-attention 层.在上述查询矩阵中,每个查询分量 q_i 均代表抽象语法树中的一个待扩展的节点的信息.在代码的抽象语法树中,不同节点彼此间存在不同程度的依赖关系,因此,我们利用一层 Self-attention 层来捕获查询间(即节点间)的依赖关系,该层的输出为

$$D^{(\text{self})}=Multihead(Q^{(\text{d})},Q^{(\text{d})},Q^{(\text{d})}) \tag{22}$$

Ast-attention 层.本文方法利用该层实现利用已产生的抽象语法树的信息(即 $Y^{(\text{ast})}$)增强查询信息.在该层中,Q=$D^{(\text{self})}$,K=$Y^{(\text{ast})}$,V=$Y^{(\text{ast})}$,输出为

$$D^{(\text{ast})}=Multihead(D^{(\text{self})},Y^{(\text{ast})},Y^{(\text{ast})}) \tag{23}$$

Code-attention 层.本文方法利用该层实现待修改代码 x 的信息(即 $Y^{(x)}$)增强查询信息.在该层中,Q=$D^{(\text{ast})}$,K=$Y^{(\text{x})}$,V=$Y^{(\text{xt})}$.输出为

$$D^{(x)} = Multihead(D^{(ast)}, Y^{(x)}, Y^{(x)}) \tag{24}$$

Diff-attention 层.本文方法利用该层实现利用修改示例 $x_\Delta \to y_\Delta$ 的信息(即 $Y^{(diff)}$)增强查询信息.在该层中,$Q=D^{(x)}$,$K=Y^{(diff)}$,$V=Y^{(diff)}$,输出为

$$D^{(query)} = Multihead(D^{(x)}, Y^{(diff)}, Y^{(diff)}) \tag{25}$$

最终,解码器的输出为 $D^{(query)}$.

4.5 指令预测

在预测下一条指令时,模型的预测范围将来自于两种类型的指令.首先,在我们处理数据时,获取了标准数据指令集 $R=\{r_1,r_2,\ldots,r_N\}$,该指令集能够满足正常的代码生成需要.然而,在代码中变量名定义和使用存在依赖关系.为了捕获这种依赖关系,方法增设了指令 $copy(n)$,表示复制已产生指令中第 n 条指令中的变量名.在预测第 i 条指令时,除了计算指令 r_j 的概率 $p(r_j)$以外,方法还将依据指针网络计算 $p(copy(t))$的概率.因此,最终的第 i 条指令预测结果需要从指令 r_j 和 $copy(t)$中进行选择.本文将采用下列公式所示的门机制来对两种类型的指令进行筛选:

$$p(op) = \begin{cases} g \times p(r_j), & op\text{为指令}r_j \\ (1-g) \times p(copy(t)), & op\text{为}copy(t) \end{cases} \tag{26}$$

其中,g 表示当前预测指令属于 R 的概率,亦即指令为 copy 的概率为 $1-g$.最终预测的指令为 $op=arg\max_{op}p(op)$.

其中,本文将按照公式(27),计算下一条指令为 r_j 的概率 $p(r_j)$:

$$p(r_j) = \text{softmax}(d_i^{(query)} W)[j] \tag{27}$$

在预测第 i 条指令时,copy 指令所复制的变量名来自于前 $i-1$ 条指令中声明的变量,因此,本文方法将按照下列公式来计算 $p(copy(t))$:

$$\xi_t = v^T \tanh(W^{(query)} d_i^{(query)} + W^{(rule)} r_t^{(init)}) \tag{28}$$

$$p(copy(t)) = \frac{\exp \xi_t}{\sum_1^{i-1} \xi_j} \tag{29}$$

其中,$1 \leqslant t \leqslant i-1$, $r_t^{(init)}$ 为已产生的第 t 条指令的表示向量,$W^{(query)}$和 $W^{(rule)}$为网络参数.

5 方法验证

本文开展了两个实验,分别将本文方法与现有的基于深度学习的方法以及基于人工规则的方法进行对比,以验证方法的有效性.

5.1 实验1

该实验通过与现有的基于深度学习的方法进行对比,以验证本文方法的有效性.

(1) 数据集.该实验在 Yin 等人的数据集[13]上来验证本文方法的有效性.在 Yin 等人的工作中,他们从 Github 上收集了 54 个 C#开源项目,然后从这些软件项目的提交历史中抽取并筛选出 111 724 处 C#代码修改,其中,91372/10176/10176 条数据分别用于训练/验证/测试.在该数据集中,数据示例均可表示为$\langle x_i,y_i\rangle$的形式,其中,x_i 为修改前的代码,y_i 为修改后的代码.依据实例$\langle x_i,y_i\rangle$,将其转换为数据$\langle x_i,x_i \to y_i,y_i\rangle$的形式,并按照模型输入要求进行预处理.具体地,首先将 x_i 和 y_i 解析为抽象语法树的形式,并按照由上至下、从左到右的顺序,获取节点序列 $\left[v_1^{(ori)},\ldots,v_{L^{(ori)}}^{(ori)}\right]$ 和 $\left[v_1^{(mod)},\ldots,v_{L^{(mod)}}^{(mod)}\right]$.此外,依据 y_i 对应的抽象语法树,获取指令序列$[r_1,\ldots,r_R]$.其中,修改前的代码节点序列作为代码编码器的输入,长度为 $L^{(ori)}$;将修改前、后代码的节点序列进行拼接,作为代码差异编码器的输入,长度为 $L^{(ori)}+L^{(mod)}$;指令序列作为模型预测目标,长度为 P.表 1 列出了该数据集中数据的 4 种长度的分布情况.如表中第 2 行所示,92.2%的实例的长度 $L^{(ori)}$小于 100,7.7%的实例的长度 $L^{(ori)}$处于 101~200 之间,0.1%的实例的长度 $L^{(ori)}$处于 201~300 之间,最大长度为 239.此外,本文对预处理后的数据中所包含的单词进行统计,共发现 2 931 个独有单词,其中最大字符长度为 70,且 80.3%的单词的字符长度少于 20 个.

(2) 对比方法.Yin 等人的工作是目前与本文工作最为相关的最新工作[13].在该工作中,作者通过表示示例代码中的修改方式,并用于指导代码转换.在该工作中,作者还尝试利用不同的方式来表示代码和修改示例,并验证不同模型的组合的方法效果.

(3) 参数设置.为了获取较优的 Transformer 架构的块数设置,方法分别将块数设为 4、6 和 8,且实验结果表明,当块数为 6 时,模型效果最佳(具体结果见表 3).因此,方法最终将代码差异编码器、代码编码器、AST 编码器和解码器的块数(即 $N1/N2/N3/N4$)设为 6.此外,如表 1 所示,由于所有的代码长度都小于 300,且超过 98%的代码差异长度小于 300,因此代码差异编码器和代码编码器的最大输入长度 L 被设定为 300.同时,所有数据实例的指令长度均小于 200,且最大长度为 185,因此,模型允许最大的指令长度 P 被设定为 200.此外,还有超过 80%的单词的字符≤20,且最大长度为 70,因此,方法将模型允许单词的最大长度 L' 设定为 20,意在保证该值能够覆盖到大多数单词的同时,避免引入过多的占位符而影响模型效果.

(4) 实验过程.在与 Yin 等人的方法进行对比时,本文沿用了他们的实验设置,并利用其数据集的训练集和验证集来训练本文方法.然后,利用测试集来测试本文方法,即方法是否能够将修改前的代码 x_i 正确地转换为 y_i.

(5) 评估标准.本文主要采用准确率来量化方法的效果.给定一条数据$\langle x,x_\Delta\rightarrow y_\Delta,y\rangle$,其中,$x$ 为修改前的代码,$x_\Delta\rightarrow y_\Delta$为修改示例,$y$ 为修改后的代码.若方法的预测结果 $\bar{y}$ 与 y 完全相同,则称 x 被正确地修改.本文将方法的准确率定义为

$$Acc=\frac{\text{正确修改的实例}}{\text{所有的实例}}.$$

(6) 实验结果.表 2 展示了与 Yin 等人方法对比后的实验结果.从实验结果来看,较之 Yin 等人提出的不同的模型,本文方法在准确率上提升了 11.8%~30.8%.在 Yin 等人的工作方法中,他们采用了两个不同的子模块分别用于编码待修改代码的信息和修改示例的信息.同时,他们还尝试了两种不同思路以对信息进行建模,即基于单词序列的方式和基于图结构的方式,并探索了信息建模方式的不同组合的实际效果.仅从 Yin 等人的工作结果来看,基于单词序列模型(如 Seq2Seq-Seq)的结果要比基于图结构模型(如 Graph2Tree-Graph)的结果更优.导致这种结果的原因在于实验数据的特殊性.由于模型输入的修改示例包含了待修改代码的修改结果信息,反而更有利于基于单词序列的模型预测修改结果.

反观本文方法 ExpTrans 的结果,若单独与 Yin 工作中的基于图结构的模型进行对比,ExpTrans 在准确率上提升了 23.4%,这表明,ExpTrans 采用图的形式来表示修改示例,并结合卷积神经网络的方式,能够有效增强模型对代码的结构信息的捕获能力,从而使得方法的准确率有了大幅度的提升.与 Yin 等人基于单词序列的方法进行对比,ExpTrans 同样具有至少 11.8%的提升.这些实验结果表明,本文方法是有效的.

此外,在模型 ExpTrans 中,Transformer 架构的块数、copy 指令和模型的不同模块对代码转换效果均有一定程度的影响.因此,基于 ExpTrans,本文通过修改模型的架构或参数设置,产生了不同的变体,以此探究上述因素在 ExpTrans 转换代码过程中的有效性.表 3 列出这些变体的具体设置及实验结果.

首先,为了寻求模型中 Transformer 架构层数较优的设置,本文借鉴了 Vaswani 等人的工作[14],先后将模型的层数设置为 4、6 和 8,具体的实验结果如表 3 中(A)行所示.从结果可以看出,当层数设置为 4 时,方法的准确率为 67.37%,较之层数为 6 时,下降 4.08%.当层数设置为 8 时,准确率为 64.37%.鉴于上述结果,最终将 ExpTrans 中 Transformer 的层数设置为 6 层.

同时,为了验证 copy 指令的有效性,本文再次将 ExpTrans 应用于 Yin 等人提供的数据集上.所不同的是,在处理该数据集时,取消了 copy 指令.具体的实验结果如表 3 中(B)行所示.如结果显示,当指令集中不含 copy 指令时,模型的准确下降为 60.12%,该结果表明,增设的 copy 指令能够有效地提升模型对代码间长程依赖的捕获能力,并提升方法的准确性.

最后,本文通过分别删除模型中的代码差异编码器以及代码差异编码器和代码编码器中的图卷积层,来验证它们的必要性.具体结果如表 3 中(C)行所示.当删除代码差异编码器时,方法的准确率骤降为 6.36%.该实验结果充分说明,在代码自动转换时,给定一个或一组实例修改的必要性,同时也说明了 ExpTrans 中的代码差异编码

器的有效性.另外,当分别删除代码差异编码器和代码编码器中图卷积层后,模型的准确率分别下降为68.01%和65.69%.在取消图卷积层的情况下,模型实际处理输入数据的效果等同于处理线性的单词序列,失去了代码自身的结构信息.这些结果表明,本文方法采用图卷积的方式能够有效地捕获代码的结构信息,并增强模型的转换能力,提升模型效果.

Table 1 Data length distribution in the first experiment

表 1 实验 1 中数据长度分布

	≤100	101~200	201~300	301~400	>400	最大长度
代码长度($L^{(ori)}$)	102 949(92.2%)	8 574(7.7%)	142(0.1%)	–	–	239
代码差异长度($L^{(ori)}$+$L^{(mod)}$)	55 960(50.1%)	46 839(41.9%)	7 070(6.3%)	1 672(1.5%)	124(0.1%)	464
指令长度	108 929(97.5%)	2 736(2.5%)	–	–	–	185
字符长度	≤5	6~10	11~15	15~20	≥21	最大长度
	437(15.0%)	836(28.5%)	687(23.4%)	404(13.8%)	567(19.3%)	70

Table 2 The comprative results with the work of Yin, *et al*

表 2 与 Yin 等人方法的对比实验结果

	Model	Acc@1 (%)
Yin 的方法	Seq2Seq-Bag-of-Edits Encoder	44.05
	Seq2Seq-Seq Edit Encoder	59.63
	Graph2Tree-Bag-of-Edits Encoder	40.66
	Graph2Tree-Seq Edits Encoder	57.49
	Graph2Tree-Graph Edit Encoder	48.05
本文方法	ExpTrans	71.45

Table 3 Performances of different variations on ExpTrans

表 3 ExpTrans 的不同变体及实验结果

	*N*1/*N*2/*N*3/*N*4	指令集中是否包含 Copy 指令	是否包含代码差异编码器	代码差异编码器中是否包含 Graph-conv 层	代码编码器中是否包含 Graph-conv 层	Acc@1 (%)
ExpTrans	6	是	是	是	是	**71.45**
(A)	4					67.37
	8					64.37
(B)		否				60.12
(C)			否	-	是	6.36
			是	否	是	68.01
			是	是	否	65.69

5.2 实验2

该实验通过与现有的基于人工规则的方法进行对比,来验证本文方法的有效性.

1) 数据集.由于所对比的基于人工规则的方法是针对Java语言的,因此我们又收集了Java语言的代码修改数据集.在收集该数据之前,我们查阅了 Java 语言在不同版本上新增的功能,选出其中 5 种并总结了可能导致的修改模式.表 4 列出了所选出的 Java 功能和相应的修改模式.基于每种功能,构造相应的查询语句,并在 Github 上搜索相关的 commit.依据搜索结果,人工地为每个查询筛选出 10 个具有相应修改模式的代码修改,即为相似代码修改.

2) 对比方法.在本实验中,对比了两种基于人工特征的代码转换方法 GenPat 和 ARES.

(1) GenPat[17].该方法是一种基于单示例的代码转换方法,其通过将给定的单个修改示例解析为抽象语法树,对抽象语法树上的特定节点的属性进行了泛化或限制,并标注抽象语法树的节点的变化过程.在代码转换时,GenPat 将根据抽取的抽象语法树进行代码匹配,并利用标注的节点的变化过程对代码进行转换.

(2) ARES[10].该方法是一种基于多示例的代码转换方法,其利用模板来表示修改示例中的修改模式.在模板中保留了修改示例中的共有部分,而采用通配符的形式对不同的部分进行泛化.在代码转换时,ARES 将依据抽取的模版对代码进行匹配,并利用模版所刻画的修改结果模式,对匹配成功的代码进行修改.

Table 4 The corresponding query sentences and change patterns of different Java features

表 4 不同 Java 功能对应的查询语句和修改模式

版本	功能	查询语句或关键词	修改模式示例

Java 5	EnhancedFor	for loop replaced with enhanced for loop	– for (int i=0; i<names.length; ++i) {
			– String val=getAttributeValueString(names[i]);
			+ for (String name:names) {
			+ String val=getAttributeValueString(name);
			. . .
			}

		replaced while iterators with for iterators	– Iterator it=names.iterator();
			– While(it.hasNext()) {
			– String query=it.next();
			+ for (String name:names) {
			+ String query=name;
			. . .
			}

	Generic type	replace explicit types with diamond operator	– List⟨GroundItem⟩ check=new ArrayList⟨GroundItem⟩(); + List⟨GroundItem⟩ check=new ArrayList⟨⟩();
	ValueOf	replace new Double with Double.valueOf	– retrydelay=new Double(childval); + retrydelay=Double.valueOf(childval);
	StringBuilder	replaced String co ncatenation with StringBuilder	– String score=""; – score +="Love"; + StringBuilder score=new StringBuilder(); + score.append("Love");

Java 7	Try-With-resource	use try-with-resources	– FileInputStream fis=new FileInputStream(new File(dir));
			– try {
			+ try (FileInputStream fis = new FileInputStream(new File(dir))) {
			. . .
			– fis.close();
			}

3) 实验过程.在该实验中,本文按照搜集的 6 组相似代码修改分别进行实验.具体地,给定一组相似修改集合 $P=\{p_1,\ldots,p_{10}\}$,其中,代码修改 $p_i=\langle x_i,y_i\rangle$,$x_i$ 为修改前的代码,y_i 为修改后的代码.为了训练本文方法,我们基于集合 P,进行了如下方式的构造:

$$P'=\{p_{i,j}\mid p_{i,j}=\langle x_i,x_j\rightarrow y_j,y_i\rangle\},$$

其中,$1\leqslant i,j\leqslant 10$,$p_{i,j}$ 所代表的含义是利用 p_j 的修改模式对代码 x_i 进行修改,并且按照如下方式将 P' 分为:

训练集:$\{p_{i,j}\}$,$1\leqslant i\leqslant 10$,$1\leqslant j\leqslant 8$;

验证集:$\{p_{i,j}\}$,$1\leqslant i\leqslant 10$,$j=9$;

测试集:$\{p_{i,j}\}$,$1\leqslant i\leqslant 10$,$j=10$.

在实验时,分别利用 3 种方法对测试集中的实例进行修改.由于 ARES 是一种基于多示例的代码转换方法,因此,我们将 P 作为一组相似代码变更提供给 ARES,以供 ARES 抽取所需的代码修改模式.

参数设置.在该实验中,本文方法中的模型参数沿用了实验 1 中的参数设置.

实验结果.我们将方法的输出结果分为 4 种.

✓ 表示方法的输出结果 $\bar{y}$ 与预期结果 y 完全一致.

○ 表示方法无法从给定的修改示例中抽取出统一的修改模式,导致方法无输出.

⦸ 表示方法抽取的修改模式无法应用到待修改的代码,导致方法无输出.

☒ 表示方法输出的结果 $\bar{y}$ 与预期结果 y 不一致.

表 5 中展示了方法在 6 组数据上的对比实验结果.结果表明,ExpTrans 要明显优于其他两种对比方法.ExpTrans 在所有组别上,均有正确的修改示例,尤其是在第 2 组数据上,ExpTrans 能够将全部的代码实例修改成功.我们人工检查了 ExpTrans 修改错误的实例,发现这些错误实例主要发生在预测同一个函数的多个参数时.其可能的原因是,ExpTrans 在预测函数的参数时,缺乏参数的序列位置信息,从而导致错误的预测结果.例如,一个预期结果为 *byteBufferReadCheck*(*in*,*buf*,11);,而 ExpTrans 的预测结果为 *byteBufferReadCheck*(*in*,**11**,11);,其中,**11** 为错误预测的参数.

进一步地,我们检查了 GenPat 和 ARES 的输出结果,以探究导致两种方法结果不理想的原因.

方法 GenPat 失效的原因在于:(1) 方法依赖于待修改的代码与修改示例代码的结构和语法类型的相似性.例如,“*return new Double*(0.0);”→“*return new Double.valueOf*(0.0);”中的修改方式,由于语法类型不同,导致无法用于修改代码“*val=new Double*(0.0);”.而在本实验所获取的数据中,无法保证相似的代码修改具有相似的结构和语法特征.因此,导致 GenPat 从示例中抽取的修改模式无法适配于待修改的代码(类型⊘).(2) GenPat 需要利用代码中变量的定义信息.然而,由于获取代码修改时,只抽取了发生修改的代码片段,因而无法保证抽取的代码修改中同时包含所有涉及的变量的定义.由于 GenPat 无法获取足够的信息,因此在将抽取的模式匹配待修改的代码时,产生错误匹配,从而导致错误修改(类型⊠).

Table 5 The comprative results with GenPat and ARES

表 5 ExpTrans 方法与 GenPat 和 ARES 方法的对比实验结果

	组 1			组 2			组 3		
	GenPat	ARES	ExpTrans	GenPat	ARES	ExpTrans	GenPat	ARES	ExpTrans
x_1	⊠	⊠	✓	⊠	✓	✓	⊠	○	⊠
x_2	⊠	⊠	⊠	⊠	✓	✓	⊠	○	⊠
x_3	⊘	⊠	⊠	⊠	⊠	✓	⊠	○	✓
x_4	⊠	⊠	✓	⊠	✓	✓	✓	○	✓
x_5	⊠	⊠	✓	⊠	⊠	✓	⊠	○	✓
x_6	⊠	⊠	✓	⊠	⊠	✓	⊠	○	✓
x_7	⊠	⊠	✓	⊠	⊠	✓	⊠	○	✓
x_8	⊘	⊠	✓	⊠	✓	✓	⊠	○	⊠
x_9	⊘	⊠	⊠	⊠	⊠	✓	⊠	○	✓
x_{10}	⊠	⊠	✓	⊠	⊠	✓	✓	○	✓
	组 4			组 5			组 6		
	GenPat	ARES	ExpTrans	GenPat	ARES	ExpTrans	GenPat	ARES	ExpTrans
x_1	⊠	○	✓	⊠	○	✓	✓	⊠	⊠
x_2	⊠	○	⊠	⊠	○	⊠	⊠	⊠	⊠
x_3	⊠	○	⊠	⊠	○	⊠	⊠	⊠	✓
x_4	⊠	○	✓	⊠	○	⊠	⊠	⊠	⊠
x_5	⊠	○	✓	⊠	○	✓	⊠	⊠	⊠
x_6	⊠	○	✓	⊠	○	⊠	✓	⊠	⊠
x_7	⊠	○	✓	⊠	○	⊠	⊠	⊠	⊠
x_8	⊠	○	✓	⊠	○	⊠	⊠	⊠	⊠
x_9	⊠	○	⊠	⊠	○	✓	⊠	⊠	⊠
x_{10}	✓	○	✓	⊠	○	⊠	⊠	⊠	⊠

由于 ARES 方法是一种基于多示例的代码转换方法,因此,当给定一组相似代码修改示例时,ARES 需要对修改示例中的不同部分进行泛化,以使其表示的修改模式能够拟合给定的修改示例.然而,当给定的修改示例的修改语义相似但结构不同时,容易导致方法无法抽取出统一的修改模式.例如,在实验中我们发现,ARES 无法从组别 2、4、6 的数据中抽取统一的修改模式,因此也无法完成对相应代码的修改(类型○).此外,还有一些错误实例是因为 ARES 在抽取的模式中,保留了修改示例所特有的变量名.因此,在将抽取的模式适配到待修改的代码时,所修改的结果中引入了这些变量名,导致修改失败(类型⊠).

5.3 讨 论

方法适用范围.在本文方法中,我们针对输入的待修改代码 x 和修改示例 $x_\Delta \rightarrow y_\Delta$,预设了最大长度.当代码长度超过预设长度时,超过预设长度的代码内容将被截取.这在一定程度上影响了方法能够使用的代码修改场景.但在一些频繁发生的相似修改任务中,例如 API 版本迁移,所修改的代码往往是局部的、简短的,因此,方法预设最大的长度并不会严重制约本文方法的实用性.尽管如此,在未来的工作中,我们仍需尝试提出不同的网络模型以降低修改代码长度对方法的影响.

数据规模.在收集 Java 数据时,依赖于人工对搜索结果的筛选,这限制了本文收集数据的规模和效率,也一定程度地制约了挖掘方法更大的潜力.在未来的工作中,我们将尝试利用自动化的方式来大规模地收集数据,尽量在降低人力代价的情况下,提升方法的效果.

6 总 结

在本文中,我们提出了一种基于深度学习的代码转换方法.通过采用图形式来表示修改示例,并结合卷积网络和Transformer架构,增加了方法捕获代码结构信息的能力.实验结果表明,我们的方法比现有的基于深度学习和基于人工规则的方法,其效果有着较为明显的提升.针对可能影响方法有效性的因素,我们将在未来的工作中,通过提出优化模型和自动化的数据收集方法,来降低这些因素对方法的影响.

References:

[1] Hunter A, Eastwood JD. Does state boredom cause failures of attention? Examining the relations between trait boredom, state boredom, and sustained attention. Experimental Brain Research, 2016, 1–10. [doi:10.1007/s00221-016-4749-7]

[2] Ko AJ, Myers BA. A framework and methodology for studying the causes of software errors in programming systems. Journal of Visual Languages and Computing, 2005,16(1-2):41–84. [doi: 10.1016/j.jvlc.2004.08.003]

[3] Barr ET, Brun Y, Devanbu P, Harman M, Sarro F. The plastic surgery hypothesis. In: Proc. of the 22nd ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering. ACM, 2014. 306–317.

[4] Nguyen HA, Nguyen AT, Nguyen TT, Nguyen TN, Rajan H. A study of repetitiveness of code changes in software evolution. In: Proc. of the 28th IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE Press, 2013. 180–190.

[5] Nguyen TT, Nguyen HA, Pham NH, Al-Kofahi J, Nguyen TN. Recurring bug fixes in object-oriented programs. In: Proc. of the 32nd ACM/IEEE Int'l Conf. on Software Engineering, Volume 1. ACM, 2010. 315–324.

[6] Ray B, Nagappan M, Bird C, Nagappan N, Zimmermann T. The uniqueness of changes: Characteristics and applications. In: Proc. of the 12th Working Conf. on Mining Software Repositories. IEEE Press, 2015. 34–44.

[7] Meng N, Kim M, McKinley KS. Sydit: Creating and applying a program transformation from an example. In: Proc. of the 19th ACM SIGSOFT Symp. and the 13th European Conf. on Foundations of Software Engineering. ACM, 2011. 440–443.

[8] Andersen J, Nguyen AC, Lo D, Lawall JL, Khoo SC. Semantic patch inference. In: Proc. of the 27th IEEE/ACM Int'l Conf. on Automated Software Engineering. IEEE, 2012. 382–385.

[9] Meng N, Kim M, McKinley KS. LASE: Locating and applying systematic edits by learning from examples. In: Proc. of the 2013 Int'l Conf. on Software Engineering. 2013. 502–511.

[10] Dotzler G, Kamp M, Kreutzer P, Philippsen M. More accurate recommendations for method-level changes. In: Proc. of the 11th Joint Meeting on Foundations of Software Engineering. ACM, 2017. 798–808.

[11] Rolim R, Soares G, D'Antoni L, Polozov O, Gulwani S, Gheyi R, Suzuki R, Hartmann B. Learning syntactic program transformations from examples. In: Proc. of the 39th Int'l Conf. on Software Engineering. IEEE Press, 2017. 404–415.

[12] Tufano M, Pantiuchina J, Watson C, Bavota G, Poshyvanyk D. On learning meaningful code changes via neural machine translation. In: Proc. of the 41st Int'l Conf. on Software Engineering. IEEE Press, 2019. 25–36.

[13] Yin P, Neubig G, Allamanis M, Brockschmidt M, Gaunt AL. Learning to represent edits. arXiv Preprint arXiv: 1810.13337. 2018.

[14] Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need. In: Advances in Neural Information Processing Systems. 2017. 5998–6008.

[15] Yin PC, Neubig G. A syntactic neural model for general-purpose code generation. arXiv Preprint arXiv: 1704.01696. 2017.

[16] Sun Z, Zhu Q, Xiong Y, Sun Y, Mou L, Zhang L. Treegen: A tree-based transformer architecture for code generation. In: Proc. of the AAAI. 2020.

[17] Jiang J, Ren L, Xiong Y, *et al*. Inferring program transformations from singular examples via big code. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). IEEE, 2019. 255–266.

[18] He K, Zhang X, Ren S, Sun J. Deep residual learning for image recognition. In: Proc. of the CVPR. 2016. 770–778.

[19] See A, Liu PJ, Manning CD. Get to the point: Summarization with pointer-generator networks. In: Proc. of the ACL. 2017. 1073–1083.

[20] Jean-Rémy F, Morandat F, Blanc X, Martinez M, Monperrus M. Fine-grained and accurate source code differencing. In: Proc. of the 29th ACM/IEEE Int'l Conf. on Automated Software Engineering. ACM, 2014. 313–324.

[21] Dehghani M, Gouws S, Vinyals O, Uszkoreit J, Kaiser Ł. Universal transformers. arXiv Preprint arXiv: 1807.03819. 2018.

曹英魁(1993－),男,博士,主要研究领域为软件工程,软件复用,程序生成.

孙泽宇(1995－),男,博士,CCF 学生会员,主要研究领域为软件工程,软件分析与测试,程序生成.

邹艳珍(1973－),女,博士,副教授,CCF 专业会员,主要研究领域为软件工程,软件复用,知识图谱,智能软件开发.

谢冰(1970－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,形式化方法,软件复用与智能软件开发.