软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(6):2515−2534 [doi: 10.13328/j.cnki.jos.007330] [CSTR: 32375.14.jos.007330]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

LLM 赋能的 Datalog 代码翻译技术及增量程序分析框架*

王熙灶[1,2], 沈天琪[1,2], 宾向荣[1,2], 卜 磊[1,3]

[1](计算机软件新技术全国重点实验室 (南京大学), 江苏 南京 210023)

[2](南京大学 计算机学院, 江苏 南京 210023)

[3](南京大学 软件学院, 江苏 南京 210023)

通信作者: 卜磊, E-mail: bulei@nju.edu.cn

摘 要: Datalog 是一种声明式逻辑编程语言, 在不同领域得到了广泛应用. 近年来, 学术界和工业界对 Datalog 的兴趣高涨, 设计并开发了多种 Datalog 引擎和相应方言. 然而, 多方言带来的一个问题是以一种 Datalog 方言实现的代码, 一般而言不能在另一种方言的引擎上执行. 因此, 当采用新 Datalog 引擎时, 需要将现有 Datalog 代码翻译到新方言上. 目前的 Datalog 代码翻译技术可分为人工重写代码和人工设计翻译规则两类, 存在耗时长、大量重复劳动、缺乏灵活性和可拓展性等问题. 提出了一种大语言模型 (LLM) 赋能的 Datalog 代码翻译技术, 利用 LLM 强大的代码理解和生成能力, 通过分治翻译策略、基于少样本提示和思维链提示的提示工程、基于检查-反馈-修复的迭代纠错机制, 可以在不同 Datalog 方言之间实现高精度代码翻译, 减轻开发人员重复开发翻译规则的工作量. 基于此代码翻译技术, 设计并实现了一种通用的基于 Datalog 的声明式增量程序分析框架. 在不同 Datalog 方言对上评估了所提出的 LLM 赋能的 Datalog 代码翻译技术的性能, 评估结果验证了所提代码翻译技术的有效性. 对通用声明式增量程序分析框架进行了实验评估, 验证了基于所提代码翻译技术的增量程序分析的加速效果.

关键词: Datalog; 代码翻译; 大语言模型; 程序分析; 增量分析框架

中图法分类号: TP311

中文引用格式: 王熙灶, 沈天琪, 宾向荣, 卜磊. LLM赋能的Datalog代码翻译技术及增量程序分析框架. 软件学报, 2025, 36(6): 2515–2534. http://www.jos.org.cn/1000-9825/7330.htm

英文引用格式: Wang XZ, Shen TQ, Bin XR, Bu L. LLM-powered Datalog Code Translation and Incremental Program Analysis Framework. Ruan Jian Xue Bao/Journal of Software, 2025, 36(6): 2515–2534 (in Chinese). http://www.jos.org.cn/1000-9825/7330.htm

LLM-powered Datalog Code Translation and Incremental Program Analysis Framework

WANG Xi-Zao[1,2], SHEN Tian-Qi[1,2], BIN Xiang-Rong[1,2], BU Lei[1,3]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](School of Computer Science, Nanjing University, Nanjing 210023, China)

[3](Software Institute, Nanjing University, Nanjing 210023, China)

Abstract: Datalog, a declarative logic programming language, is widely applied in various fields. In recent years, there has been a growing interest in Datalog from both the academic and industrial communities, leading to the design and development of multiple Datalog engines and corresponding dialects. However, one problem brought about by the multiple dialects is that the code implemented in one Datalog dialect generally cannot be executed on the engine of another dialect. Therefore, when a new Datalog engine is adopted, the existing Datalog code needs to be translated into the new dialect. The current Datalog code translation techniques can be classified into two categories: manually rewriting the code and manually designing translation rules, which have problems such as being time-consuming, involving a large amount of repetitive work, and lacking flexibility and scalability. In this study, a Datalog code translation technology

* 基金项目: 国家自然科学基金 (62232008, 62172200); 江苏省前沿引领技术基础研究专项 (BK20202001)
本文由“大模型下的软件质量保障”专题特约编辑王赞教授、王莹副教授、陈碧欢副教授、姚远副教授、张敏灵教授推荐.
收稿时间: 2024-08-26; 修改时间: 2024-10-14; 采用时间: 2024-11-25; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-03-19

© 中国科学院软件研究所 http://www.jos.org.cn

empowered by large language model (LLM) is proposed. By leveraging the powerful code understanding and generation capabilities of LLM, through the divide-and-conquer translation strategy, the prompt engineering based on few-shot and chain-of-thought prompts, and an iterative error-correction mechanism based on check-feedback-repair, high-precision code translation between different Datalog dialects can be achieved, reducing the workload of developers in repeatedly developing translation rules. Based on this code translation technology, a general declarative incremental program analysis framework based on Datalog is designed and implemented. The performance of the proposed LLM-powered Datalog code translation technology is evaluated on different Datalog dialect pairs, and the evaluation results verify the effectiveness of the proposed code translation technology. This study also conducts an experimental evaluation of the general declarative incremental program analysis framework, verifying the speedup effect of incremental program analysis based on the proposed code translation technology.

Key words: Datalog; code translation; large language model (LLM); program analysis; incremental analysis framework

Datalog 是一种声明式逻辑编程语言, 其通过递归拓展了关系代数, 允许定义递归规则. Datalog 为开发人员提供了一个简单的声明式接口, 同时可以在单机和并行设置中提供加快评估速度的通用优化. 得益于 Datalog 语言的声明式特性, 开发人员只需要关注任务描述, 而无需考虑底层实现细节. 在过去 20 年中, Datalog 已被用于表达不同应用领域中的各种任务, 如声明式网络[1]、图分析[2,3]、程序分析[4,5]以及安全[6].

近年来, 学术界和工业界对 Datalog 的兴趣高涨, 设计并开发了多种 Datalog 引擎. 面向不同的需求和应用, 不同的 Datalog 引擎所支持的语法和功能特性也有所不同, 对应不同的 Datalog 方言. 例如, Soufflé[5]是一种针对程序分析任务开发的 Datalog 引擎, 而 LogicBlox[7]则是一种面向商业分析的引擎, 它们对应 Datalog 方言分别为 Soufflé和 LogiQL. 一般而言, 以一种 Datalog 方言实现的代码不能在另一种方言的引擎上执行. 当因为性能优势、新特性等而采用一种新 Datalog 引擎时, 开发人员通常需要对现有的 Datalog 代码进行代码翻译.

目前的 Datalog 代码翻译技术可分为两类. 第 1 类是用新 Datalog 方言人工重写所有代码[8]. Doop[4]是一个基于 Datalog 实现的指针分析框架, 其最初使用 LogiQL 方言, 在后续发展中因 LogicBlox 引擎存在的性能和学术使用限制等原因而迁移到了 Soufflé上, 整个迁移过程是通过全部人工重写的方式完成的, 耗时约 10 人月. 这种方式的优点是通过人工重写可以最大限度发挥新方言的潜力; 缺点是耗时极长, 存在大量重复劳动, 并且不可复用, 不同代码均需要从头开始重新翻译. 第 2 类是人工为特定的源语言和目标语言设计翻译规则[9,10]. Doop 框架中实现的是全量分析算法, 对于被分析代码的任意规模变更, 都需要从头开始全量分析. 沈天琪等人[9]基于被分析代码变更即 Datalog 引擎输入变更的思想, 提出了基于增量 Datalog 引擎 (DDlog[11]) 的增量程序分析框架. 他们通过人工设计 Soufflé到 DDlog 的代码翻译规则, 将 Doop 的 Soufflé实现迁移到了 DDlog 上, 通过代码翻译的方式实现了自动化增量程序分析. 这一类翻译技术的优点是减少了大量重复劳动, 可以将源方言的任意代码自动化地翻译为目标方言代码, 缺点是不同方言之间存在大量琐碎的细节差异, 人工设计翻译规则工作繁琐, 并且所设计的翻译规则也仅适用于一对特定方言, 对于不同方言对需重复开发翻译规则, 缺乏灵活性和可拓展性.

鉴于大语言模型 (large language model, LLM) 强大的代码理解和生成能力, 研究人员开始尝试使用 LLM 来执行代码自动翻译任务[12]. 已有工作主要关注 LLM 在通用编程语言 (如 C++、Java 等) 之间的代码翻译任务, 目前尚无使用 LLM 进行 Datalog 代码翻译的工作. 使用 LLM 进行 Datalog 代码翻译主要面临以下 3 方面挑战. 挑战 1, LLM 上下文长度限制. 现实世界的 Datalog 程序规则复杂, 如 Doop 框架中一种指针分析规则的代码 Token 数可达 57k (使用 OpenAI 提供的分词器 (https://platform.openai.com/tokenizer) 计算 Token 数), 这远超目前主流 LLM 的上下文长度限制, 导致 LLM 无法完成代码翻译任务. 挑战 2, Datalog 代码预训练语料极少, 其代码在 LLMs[13–15] 的训练语料库中所占比例微乎其微. 直接让 LLM 执行 Datalog 代码翻译任务的效果非常不理想. 挑战 3, LLM 存在幻觉问题, 可能会为 Datalog 代码翻译任务生成看似合理但却错误的输出, 直接使用 LLM 的输出结果可能导致翻译后的代码出现错误.

针对上述 3 个挑战, 本文设计并提出了一种 LLM 赋能的 Datalog 代码翻译技术, 可以在不同的 Datalog 方言之间实现高精度的代码翻译, 以减轻开发人员重复开发翻译规则的工作量. 我们提出了分治翻译策略以应对挑战一, 结合少样本提示 (few-shot prompting)[16]和思维链 (chain-of-thought, CoT)[17]的提示工程以应对挑战 2, 基于检查-反馈-修复的迭代纠错机制以应对挑战 3. 基于 LLM 赋能的 Datalog 代码翻译技术, 我们设计并实现了一种通

© 中国科学院软件研究所 http://www.jos.org.cn

用的基于 Datalog 的声明式增量程序分析框架, 旨在为基于不同 Datalog 方言实现的静态分析规则自动提供增量分析的能力. 我们在不同 Datalog 方言对上评估了所提出的 LLM 赋能的 Datalog 代码翻译技术的性能, 评估结果验证了所提代码翻译技术的有效性. 我们也在多组现有程序分析规则和广泛应用的真实世界程序上对声明式增量程序分析框架进行了实验评估, 验证了基于所提代码翻译技术的增量程序分析的加速效果.

总而言之, 本文做出了以下主要贡献.

设计并实现了一种 LLM 赋能的 Datalog 代码翻译技术, 针对使用 LLM 进行 Datalog 代码翻译所面临的 3 个挑战分别提出了相应的解决方案. 基于 LLM 强大的代码理解和生成能力, LLM 赋能的代码翻译技术可以在不同 Datalog 方言之间实现高精度的代码翻译, 减轻开发人员重复开发翻译规则的工作量.

基于 LLM 赋能的 Datalog 代码翻译技术, 设计并实现了一种通用的基于 Datalog 的声明式增量程序分析框架. 该增量分析框架可以复用基于不同 Datalog 方言实现的分析规则, 为其自动提供增量分析能力.

通过对提出的 LLM 赋能的 Datalog 代码翻译技术和声明式增量程序分析框架进行实验评估, 验证了所提 LLM 赋能的 Datalog 代码翻译技术的有效性, 以及基于该翻译技术的增量程序分析框架的加速效果.

本文第 1 节介绍本工作的相关背景知识. 第 2 节介绍所提出的 LLM 赋能的 Datalog 代码翻译技术. 第 3 节介绍声明式增量程序分析框架的设计. 第 4 节通过实验评估展示基于 LLM 的 Datalog 代码翻译技术的有效性和增量分析框架的性能. 第 5 节讨论了本文工作的局限性. 第 6 节回顾相关工作. 第 7 节对本文工作进行总结与展望.

1 背景知识

本文的核心内容为 LLM 赋能的 Datalog 代码翻译技术, 下面就相关概念和基础知识分别予以介绍.

1.1 Datalog

Datalog 是一种声明式逻辑编程语言, 是通用逻辑编程语言 Prolog 的一个图灵不完备子集. Datalog 最初作为一种数据库查询语言而出现, 现在被广泛用于处理结构化数据, 并在人工智能、声明式网络和程序分析等领域中发挥着重要作用. Datalog 提供了一个声明式接口, 允许程序员专注于任务描述 (What) 而非低级细节 (How).

Datalog 的核心语言非常简洁. 一个 Datalog 程序由一组谓词 (predicate) 声明和一组规则 (rule) 定义构成. 谓词用于表示数据. 根据是否为先验定义/规则建立, 谓词可分为外延数据库 (extensional database, EDB, 可视为输入谓词) 和内涵数据库 (intensional database, IDB, 可视为输出谓词) 两类. $P(e_1,\ldots,e_k)$ 表示一个 k 元谓词, 谓词的实例元组被称为事实 (fact). 规则用于描述逻辑, 规则包含规则头和规则体两部分, 规则格式如公式 (1) 所示:

$$A \leftarrow B_1, B_2, \ldots, B_n \tag{1}$$

其中, A 和 B_i 表示谓词, A 为这条规则的规则头, $B_1, B_2, \ldots, B_n$ 的合取构成规则体. Datalog 的逻辑基础源自一阶谓词逻辑, 每条规则都是一个蕴涵式, 例如公式 (1) 中规则对应的蕴涵式为 $B_1 \wedge B_2 \wedge \ldots \wedge B_n \rightarrow A$. 给定一个 Datalog 程序以及一组输入事实, 经评估后会得到一组输出事实. Datalog 通常使用自底向上的评估模型, 它重复应用规则, 直到不再有新的事实产生或达到终止条件.

正如 SQL 是一个语言规范, Transact-SQL、PL/SQL 是其具体实现一样, Datalog 同样也只是一个语言规范, 存在不同的具体实现, 例如 Soufflé[5]、DDlog[11]等, 这些实现被称为 Datalog 引擎, 对应的语言被称为 Datalog 方言. 不同的 Datalog 方言在具体语法上有所差异, 并为了满足不同需求而设计了不同的拓展语言特性. 用一种特定 Datalog 方言编写的代码一般而言不能直接在另一种 Datalog 方言的引擎上运行.

最后介绍 Datalog 在程序分析领域的应用. 最初, 静态分析研究人员使用传统的命令式语言实现具体的程序分析算法, 但这需要研究人员具备强大的工程能力. 大量的开发时间并未用于设计和实现分析算法, 而是用于处理各种细节性的工程问题, 例如极其复杂的数据结构设计、内存管理、并行处理甚至性能调优等. Datalog 语言在这方面有其突出的优势: 一方面, 由于 Datalog 的声明式特性, 这类工程细节问题可以委托给 Datalog 引擎处理, 让研究人员可以专注于分析算法逻辑, 无需过多关心底层实现细节. 另一方面, 对底层 Datalog 引擎的优化可以使得所有基于其实现的分析算法受益. Datalog 语言在程序分析领域的应用可追溯到 2003 年[18]. Doop[4]是一种基于

© 中国科学院软件研究所 http://www.jos.org.cn

Datalog 的指针分析框架, 其使用了目前面向程序分析领域的最先进 Datalog 方言 Soufflé[5].

1.2 代码翻译

代码翻译 (源到源编译或转译) 是一种编程语言处理技术, 它以一种高级编程语言 (如 C++) 的源代码作为输入, 以另一种高级编程语言 (如 Java) 实现的功能等价的程序源代码作为输出. 代码翻译技术一般用于互操作性或将使用过时/已弃用语言编写的代码库移植到现代语言, 以提高软件项目的寿命、可靠性和成本效率. Hinchey 等人[19]的研究显示, 尽管多年来硬件的可靠性一直在提高, 但软件系统的可靠性却并非如此. 事实上, 由于软件工程基础设施的不断发展, 软件经常需要在不同的平台和底层基础设施之间进行迁移以获得收益, 但这通常需要改变软件所使用的编程语言和相关工具链. 因此, 自动化代码翻译技术是这类迁移工作的关键.

在传统的代码翻译技术中, 最重要的一类是基于抽象语法树 (abstract syntax tree, AST) 的代码重写技术. AST 代表了代码的抽象语法结构, 而不包含真实源代码中存在的每个细节, 从而自然的排除了输入源代码中和任务无关的其他内容. 正如 Albrecht 等人[20]和 Balogh 等人[21]在他们工作中展示的, 基于 AST 重写的代码翻译有许多优点. 由于 AST 刻画了代码语义, 它们提供了一种编码关于代码含义信息的内在能力. 这种语义信息对于准确的源到源代码翻译至关重要. 然而, 另一方面, AST 代码翻译技术是高度特定于语言的, 需要付出大量努力来支持新的语言或方言. 开发和维护基于 AST 的代码翻译器是复杂且耗时的, 因为它涉及处理不同编程语言的复杂性. 这要求开发人员具有源语言和目标语言方面的专业知识, 达到能够构建高效、正确的代码翻译器的水平. 此外, 基于 AST 重写生成的翻译通常缺乏可读性, 不尊重目标语言约定, 并且需要手动修改才能正常工作.

近年来, 随着机器学习特别是深度学习技术的快速发展, 基于学习的代码翻译技术被广泛研究. 这种技术利用机器学习模型从大量的代码数据中学习编程语言的语法和语义规则, 从而实现更加智能和灵活的代码翻译. 与传统方法相比, 基于学习的翻译技术能够更好地理解和处理代码中的上下文信息, 提供可读性更好的翻译结果. Roziere 等人[22]利用无监督机器翻译的方法来训练完全无监督的神经代码翻译器, 并证明它可以在 C++、Java 和 Python 之间高精度地翻译函数. 最近几年, LLM 在自然语言处理领域取得了显著的进展. 这些模型通过在大规模代码数据集上预训练, 能够捕捉到编程语言的复杂模式和结构, 实现对源代码的深入理解. LLM 的强大能力也引起了研究人员和开发人员在代码翻译任务上的兴趣[12]. 在代码翻译任务上中, LLM 可以识别源代码的功能意图, 并将其映射到目标语言的等效表达, 从而实现高质量代码翻译. LLM 在代码翻译领域中的主要优势在于基于语言理解能力带来的源代码上下文感知和意图理解, 强大的代码生成能力, 通常更具灵活性与可拓展性. 基于 LLM 的代码翻译技术主要有两种方式: 1) 使用语义等价代码数据集对基础模型进行训练或微调, 这对数据集的质量和规模均提出了相当高的要求; 2) 通过提示工程进行上下文学习, 这需要具有较强代码能力的基础模型以及精心设计的提示工程. 基于机器学习/LLM 的代码翻译技术的缺点是存在不确定性和幻觉等问题.

2 LLM 赋能的 Datalog 代码翻译技术

随着人工智能技术的飞速发展, LLM 在代码理解和生成方面展现出了前所未有的能力. 这些模型不仅能够理解复杂的代码结构, 还能生成符合逻辑的代码片段, 极大地推动了软件开发和维护的自动化进程. 在这样的背景下, 本节将介绍一种 LLM 赋能的 Datalog 代码翻译技术, 该技术旨在解决当前 Datalog 代码翻译技术中的一个关键问题——在多 Datalog 方言背景下, 开发人员人工设计翻译规则的方式灵活性不足, 存在大量重复工作.

鉴于 LLM 强大的代码理解和生成能力, 目前已经有一些工作研究如何利用 LLM 来进行代码翻译[12]. 基于 LLM 的 Datalog 代码翻译有望解决现有翻译技术的灵活性问题, 减轻开发人员重复开发翻译规则的工作量. 然而, 使用 LLM 执行 Datalog 代码翻译任务仍面临来自以下 3 方面的挑战.

(1) LLM 上下文长度限制. 现实世界的 Datalog 应用程序规则复杂, 以 Doop 框架中 1-callsite-sensitive 精度指针分析为例, 其在 open program 选项下的代码 Token 数达到了 57k. 这远远超出了目前主流 LLM 的上下文长度限制, 导致 LLM 无法完成代码翻译任务.

(2) Datalog 代码预训练语料极少. Datalog 作为一门小众的逻辑式编程语言, 其代码语料在 GPT、LLaMA、

© 中国科学院软件研究所 http://www.jos.org.cn

Claude 等通用 LLM 和 Code LLaMA 等代码 LLM 的预训练语料库中所占比例微乎其微. 因此, 直接让 LLM 执行 Datalog 代码翻译任务的效果非常不理想.

(3) LLM 幻觉问题. LLM 可能会为 Datalog 代码翻译任务生成看似合理但却虚假或有误导性的输出. 程序分析是 Datalog 的一个重要应用领域, 当将程序分析作为错误查找工具时, 若翻译生成错误分析规则, 会导致在使用其对代码进行分析时产生错误报告, 这对开发人员而言是不可接受的.

针对上述 3 个挑战, 我们设计并提出了一种 LLM 赋能的 Datalog 代码翻译技术, 旨在为不同 Datalog 方言对实现高精度的代码翻译. 下面, 我们将分别阐述此 Datalog 代码翻译技术中用于解决上述挑战的解决方案.

2.1 整体架构

图 1 为本文所提出的 Datalog 代码翻译技术的整体架构. 我们的 Datalog 代码翻译技术的工作流程如下. 首先, 对于输入的 Datalog 源方言待翻译代码, 我们采用分治策略将其按规则进行拆分 (第 2.2 节), 得到一系列 Datalog 源方言单规则. 随后, 通过结合少样本提示和思维链提示的提示工程 (第 2.3 节), 源方言单规则将在 LLM 中被翻译为 Datalog 目标方言的单规则 (候选). 接下来, 我们将使用基于检查-反馈-修复的迭代纠错机制 (第 2.4 节), 分别对翻译结果进行语法检查和语义检查. 如果检查发现存在语法/语义错误, 我们通过提供反馈来辅助 LLM 修复错误. 在语法和语义检查通过之后, 最后一步是将第 1 步中拆分出的一系列 Datalog 源方言单规则对应的一系列 Datalog 目标方言单规则进行合并 (第 2.5 节), 并最终生成完整的 Datalog 目标方言翻译结果.

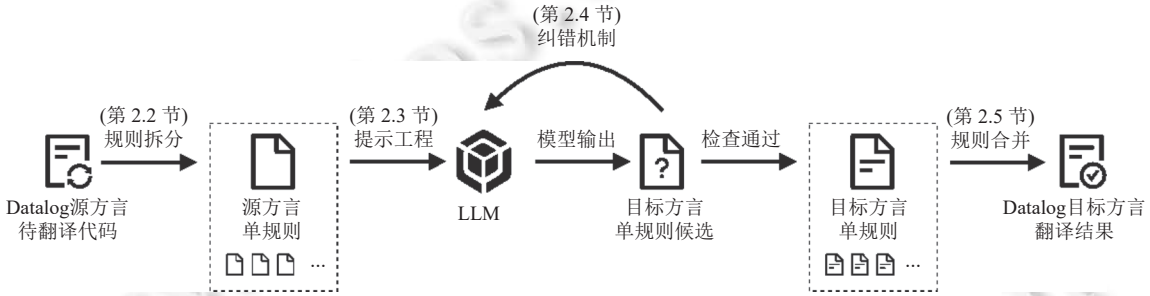


图 1 LLM 赋能的 Datalog 代码翻译技术整体架构图

2.2 分治策略

本节介绍我们为应对 LLM 上下文长度限制问题 (挑战 1) 所提出的分治翻译策略.

LLM 上下文长度是指模型在处理输入文本时能考虑的最大文本量, 通常以 LLM 一次能处理的最大 Token 数来表示, 当输入文本长度超过上下文长度限制时, 模型可能无法考虑到超出部分的信息, 其性能将急剧下降. 提供超长上下文的 LLM 可以处理更复杂的查询和更长的文档, 具有更强的理解能力, 但是超长上下文也会带来诸如推理时间显著变长、所需显存空间显著变大等问题.

就目前主流 LLM 所基于的 Transformer 架构而言, 任何一个 LLM 的上下文长度总是有限的. 因此, 目前主流 LLM 所支持的最大上下文长度均有一定限制, 例如, LLaMA 3 具有 8k Token 的上下文长度[23], 而 GPT-4 最大为 32k Token[13]. 对于本文所关注的 Datalog 代码翻译任务, 从理论上来说, 任何一种 Datalog 方言都可以有任意长的代码. 在现实的 Datalog 应用中, 如 Doop 指针分析框架, 其中 1-callsite-sensitive 精度指针分析在 open program 选项下的 Datalog 代码对应的 Token 长度达到了 57k. 因此, 一次性输入完整的大型 Datalog 代码对于 LLM 而言是一件极具挑战甚至不可能完成的任务. 因此, 我们必须设计一个机制来处理这个问题. 此外, 我们还必须要考虑到大型 Datalog 代码输入对反馈和修复 (第 2.3 节) 的影响, 其中反馈机制期望能够快速定位到 LLM 翻译异常的规则, 而大型 Datalog 代码无疑为异常规则定位提出了更高的挑战.

为此, 针对 Datalog 代码翻译任务, 基于 Datalog 规则之间的独立性, 我们设计了一种基于规则拆分的分治策略. 得益于 Datalog 语言的声明式特性, 每条 Datalog 规则都可以被视为一个完全独立的逻辑断言或命题, 其定义

© 中国科学院软件研究所 http://www.jos.org.cn

了事实的推导规则, 而无需指定具体的计算过程, 其中不包含任何“如何求解”的控制流信息, 因此 Datalog 规则之间不存在控制流依赖或先后关系, 它们共同构成了一个逻辑知识库. 基于 Datalog 规则之间的独立性, 我们在翻译设计时就可专注于仅针对单条规则而非完整的 Datalog 代码. 因此, 我们的 Datalog 代码翻译技术的第 1 步就是将完整 Datalog 代码拆分为一组 Datalog 单规则. 这不仅方便提示词的编写 (第 2.3 节), 同时还便于反馈机制更加精确地定位翻译异常的规则 (第 2.4 节), 因为拆分后的单规则已经是 Datalog 的最小语义单元.

具体而言, 我们的分治策略包含两部分: 一是预处理阶段的规则拆分, 二是后处理阶段的规则合并 (第 2.5 节), 本节主要关注预处理阶段的规则拆分. 在利用 LLM 进行代码翻译之前, 我们会先对输入的 Datalog 代码进行规则拆分. 拆分器的输入是一段完整的 Datalog 代码, 包含了一系列 Datalog 规则; 而它的输出则是一组 Datalog 单规则. Datalog 语言的声明式特性保证了其规则之间并不存在语义层面的依赖关系. 因此我们可利用 Datalog 引擎提供的语法解析器对输入的 Datalog 代码进行一次遍历, 将其拆分为一组 Datalog 单规则. 对于某些方言, 其代码中可能还包含声明部分, 这部分我们会将它们独立提取, 以在后续的流程中使用.

受益于 Datalog 规则之间的独立性, 预处理阶段的拆分和后处理阶段的合并从技术上来说并不复杂, 但是这是下文中提示词设计 (第 2.3 节) 与翻译纠错机制 (第 2.4 节) 的基础. 接下来, 对于拆分后的每一条 Datalog 单规则, 我们会为其构造相应的提示词, 然后交由 LLM 来完成后续的翻译任务.

2.3 提示工程

本节介绍我们为应对 Datalog 代码预训练语料极少问题 (挑战 2) 所设计的提示工程.

基于 LLM 执行 Datalog 代码翻译是一项复杂的任务, 这需要 LLM 能够同时理解 Datalog 代码的语法 (以生成语法正确的代码) 和语义 (以在翻译过程中保留原有代码的功能). 此外, LLM 的能力主要来源于其预训练语料, 而 Datalog 作为一门小众的逻辑编程语言, 其代码语料在现有主流 LLM (包括针对代码生成和理解任务所训练的各种代码 LLM) 的预训练语料库中所占比例微乎其微; 而且 Datalog 本身仅作为一种语言规范, 其分散的各种方言使得语料稀缺问题更为突出. 以 Stack v2[24]为例, 这是用于代码 LLM 的最大预训练语料库, 其包含超过 30 亿个文件, 涵盖了 658 种编程语言. 然而, 其中仅包含一种 Datalog 方言 CodeQL, 其文件数占比不超过 0.001%; 另一种相关语言 Prolog 的占比也不超过 0.01% (https://huggingface.co/datasets/bigcode/the-stack-v2/blob/main/language_stats.csv). 因此, 直接让 LLM 执行 Datalog 代码翻译任务的效果非常不理想.

在本文所关注的 Datalog 代码翻译任务中, 针对 Datalog 在 LLM 训练语料上显著稀缺的问题, 最直接的方式是构造一组不同 Datalog 方言之间的等价代码对数据集, 对现有 LLM 进行微调, 以使 LLM 获得 Datalog 的代码等价翻译能力. 然而, 等价代码对数据集的自动化构造是一项极具挑战的工作. 对于一部分 Datalog 方言, 存在一些对其 Datalog 引擎进行模糊测试的工作[25,26], 这些现有的 Fuzzing 工具可以自动化生成大量语法正确的 Datalog 代码. 然而对于生成等价的 Datalog 方言代码对问题, 目前还缺少成熟的工具和方案.

鉴于训练/微调数据难以获得的问题, 本文采用了提示工程的方案, 结合少样本提示 (few-shot prompting)[16]和思维链 (chain-of-thought, CoT) 提示[17]技术, 通过在提示词中提供任务背景和相关示例, 我们期望模型能在上下文中学习并结合其自身能力, 完成对不同 Datalog 方言的代码进行等价翻译的任务.

一般而言, 一个基于少样本提示的提示词由以下 5 部分构成: 系统引导语、示例、指令、输入数据和输出指示. 我们可以很容易地设计出一个用于 Datalog 代码翻译任务的少样本提示词: 只需给出一些示例和待翻译的 Datalog 代码, 然后让 LLM 给出响应即可. 然而, 想要获得一个高效率的提示词, 仍然存在一些挑战.

(1) Datalog 语法: Datalog 语言的语法本身并不复杂, 但其作为一种逻辑编程语言, 其语法规则和结构与自然语言以及常见的命令式编程语言依旧有较大差异. 而 LLM 主要针对自然语言和主流的编程语言的语料进行预训练, 对 Datalog 语法的理解和处理能力可能有限. 因此我们需要仔细设计提示词, 提供相应的示例和说明, 帮助 LLM 正确理解 Datalog 规则的语法和语义.

(2) 翻译的准确性和一致性: 由于 Datalog 规则通常具有严格的逻辑语义, 翻译结果的准确性和一致性非常重要. 然而, LLM 生成的结果可能存在一定的随机性和不确定性. 虽然提示工程不能完全解决此问题, 但我们仍应在

© 中国科学院软件研究所 http://www.jos.org.cn

LLM 翻译早期在此方面做出相应努力.

(3) 少样本的局限性: 尽管少样本提示对样本数量要求不高, 但对于复杂的 Datalog 翻译任务, 仅提供少量样本可能不足以让模型完全掌握翻译规则和策略. 需要权衡样本数量和模型性能之间的平衡, 并考虑在提示词中加入更多指导性的说明和规则.

下面具体介绍我们在 Datalog 代码翻译任务中的提示词设计.

2.3.1 少样本提示

少样本提示中最核心的部分是小规模示例样本的构造和选取. 在 Datalog 代码翻译任务中, 示例样本为 Datalog 源方言和目标方言的等价代码对, 即一份 Datalog 源方言代码和一份语义等价的 Datalog 目标方言代码. 前文已提及现有技术难以自动化构造等价代码对, 而鉴于前文中已提到的 Datalog 预训练语料显著稀缺问题, 基于 LLM 来自动化构造等价代码对的技术方案同样不可行. 而 Datalog 语言的一大特点是其核心语言规范非常简洁 (见第 1.1 节), 仅包含一组谓词声明和一组规则定义, 二者都是易于理解和构造的. 因此, 对于开发人员而言, 构造小规模的 Datalog 方言等价代码对的难度和人力成本都很小.

开发人员初始时需要手工构造一个小规模的等价代码对数据集. 对于输入的待翻译的源方言代码, 我们采用检索增强生成的方式来选取一组与待翻译代码最相关的等价代码对. 当 LLM 完成翻译任务后, 对于通过检查 (见第 2.3 节) 的输出目标方言代码, 开发人员可将其进一步添加到等价代码对数据集. 采用检索增强生成来选取等价代码对既可以减少示例所占用的上下文空间, 从而为提示词其余部分留下更大空间, 也可以通过选取最相关的等价代码对以提高 LLM 对输入待翻译代码的翻译结果的准确性和一致性. 在本文中, 初始手工构造的等价代码对数据集的规模因方言对而异, 一般不超过 20; 检索增强生成选取的少样本数量为 5.

2.3.2 思维链提示

对于复杂的 Datalog 翻译任务, 仅通过提供少量样本可能不足以让模型完全掌握翻译规则和策略. Wei 等人[17]提出的思维链 (CoT) 提示通过中间推理步骤实现了复杂的推理能力. 通过将少样本提示与思维链提示相结合, 可以让 LLM 在回答之前对于翻译任务以及对应的翻译规则和策略有更充分的理解. 因此, 我们为 Datalog 代码翻译任务设计了相应的思维链提示. 对于一个待翻译的 Datalog 谓词声明 $P(e_1,\ldots,e_k)$, 我们首先让 LLM 翻译谓词名, 然后对谓词定义中的变量 (变量名与变量类型) 进行逐个翻译, 最后将翻译后的谓词名和变量声明进行组合, 得到翻译后的完整谓词定义 (部分 Datalog 方言对于变量名和谓词名有特殊要求, 因此需对二者进行翻译以保证符合命名要求. 对于命名无特殊要求的方言, 直接使用源方言代码中的变量名和谓词名即可). 对于一条待翻译的 Datalog 规则 $A \leftarrow B_1, B_2, \ldots, B_n$, 我们首先让 LLM 分别对规则头和规则体中的谓词进行翻译 (类似谓词声明的翻译), 接下来翻译规则连接符 (不同的方言使用的连接符不尽相同), 最后将规则头、连接符和规则体进行组合, 得到翻译后的完整规则定义. 思维链提示的链式思考流程是人工设计的, 但对于给定的翻译示例, 其构造是自动化完成的. 通过为 LLM 提供包含具体翻译步骤的思维链提示, 可以提高 LLM 在翻译任务上的准确性.

在本节最后, 我们展示一个为 LLM 赋能的 Datalog 代码翻译技术设计的提示词示例 (图 2). 此示例是我们 Datalog 代码翻译技术的 Ascent-DDlog 实例中实际使用的一个提示词, 其目的是将 Ascent[27]代码翻译为等价的 DDlog[11]代码, 以使用 DDlog 所提供的增量评估能力.

如图 2 所示, 我们在提示词的开头为 LLM 提供基础背景. 随后给出 Ascent 和 DDlog 的规则语法对比, 指出 DDlog 需要在关系名称前加“R”前缀 (这是因为 DDlog 在关系名词法规则上的约束). 之后则是将 Datalog 代码翻译时需要遵循的最基本规则给出, 如保持变量和关系的顺序等 (此处由于篇幅限制没有给出全部内容). 然后是一个和补充说明相关的内容: 方括号 [] 内的信息对翻译很重要, 但不应出现在 DDlog 输出中. 以上是我们设计的提示词在具体给出少样本提示之前的内容的结构. 我们通过其中针对 Datalog 方言结构的额外信息, 最基本的翻译原则以及对输出格式的规范来增强翻译的准确性和一致性. 在示例样本中, 我们通过 CoT 提示为 LLM 提供翻译指导. 对于具体示例构造, 我们尝试让示例能尽量覆盖所有可能的语法情况, 并且在必要的时候通过 [] 来加入更多额外的信息. 例如一部分 Datalog 方言中提供了一些类似 tonumber()、max()、contains() 这样的内置函数, 我们会提供相应的示例, 并使用 [] 来给出更多的补充信息.

© 中国科学院软件研究所 http://www.jos.org.cn

```
You are a Datalog expert and you are working on translating Ascent programs into equivalent DDlog ones.
In Ascent, rules are defined using the syntax:
```ascent
relation(variables) <-- body;
```
In DDlog, the equivalent rules are written as:
```ddlog
Rrelation(variables) :- Rbody.
```
Notice that in DDlog, a relation name is prefixed with 'R'.

Please follow these principles when translating:
- Maintain the order of variables and relations.
- ...

During the translation process, you might encounter additional information in brackets [ ].
This information is crucial for the translation but should not appear in the output.

Let's start with some simple examples before moving on to more complex rules.
Here are a couple of examples to illustrate the process:

Example 1:
Q: Translate the following Ascent rule into an equivalent DDlog counterpart.
```ascent
path(x, y) <-- edge(x, y);
```
A: Let's think step by step.
First, translate the rule head into 'Rpath(_x, _y)'.
Second, translate the rule body into 'Redge(_x, _y)'.
Third, translate the symbol into ':-'.
Finally, the equivalent DDlog rule is
```ddlog
Rpath(_x, _y) :- Redge(_x, _y).
```
...

Q: Translate the following Ascent rule into an equivalent DDlog counterpart.
```ascent
...
```
A: Let's think step by step.
```

图 2 Datalog 代码翻译提示词示例

通过本节所介绍的少样本提示和思维链提示, 我们尝试通过这种示例和说明的结合方式来帮助 LLM 正确理解 Datalog 规则的语法和语义, 并增强在少样本提示下 LLM 在 Datalog 代码翻译任务上的能力.

2.4 纠错机制

本节介绍我们为应对 LLM 幻觉问题 (挑战 3) 所设计的迭代纠错机制.

通过在超大规模的语料库上进行训练, LLM 获得了可以生成连贯、有意义的语言输出的能力. 然而就其数学本质而言, LLM 与此前的深度学习模型一样, 仍是一个概率模型. LLM 通常还具有特意设计的机制 (例如温度、Top-p 等[28]) 以控制输出的随机性. 因此, LLM 输出带有原生的不确定性. 然而, 代码翻译任务对于 LLM 输出具有严格的要求, 既要求符合目标语言语法, 也要求与源语言代码输入保证语义等价性. 即使我们已经在提示工程上为提高代码翻译的正确性提出了各种设计, LLM 输出的这种不确定性仍意味着仅依赖于 LLM 赋能的 Datalog 代码翻译技术从原理上难以保证其翻译结果的可重复性和准确性.

代码翻译结果的正确性包含语法和语义两个方面[12]. 为此, 我们设计了一种基于检查-反馈-修复的迭代纠错机制 (算法 1), 以确保 LLM 在 Datalog 代码翻译过程的准确性和可靠性. 我们的纠错机制包含错误检查、信息反馈和错误修复 3 个关键组件, 形成一个迭代工作流. 当 LLM 生成输出后, 错误检查组件会对输出分别进行语法和语义检查; 如果检查未通过, 信息反馈组件会将相应报错信息提供给 LLM 以供其理解错误; 最后错误修复组件通过特定修复策略利用 LLM 重新生成翻译输出; 每次修复后, 系统会重新进行错误检查, 这个流程不断迭代, 直到翻译结果通过所有检查, 或者达到预设的修复尝试次数上限. 下面具体介绍纠错机制中 3 个关键组件的设计.

© 中国科学院软件研究所 http://www.jos.org.cn

算法 1. 迭代纠错算法.

输入: Datalog 源方言单规则 SourceRule;
输出: Datalog 目标方言单规则 TargetRule.

```
1. Feedback ← [ ];
2. while upper limit of attempts is not reached do
3.    Candidate ← LLMTrans(SourceRule, Feedback); ▷ 翻译生成/修复生成
4.    if not SyntaxCheck(Candidate); ▷ 语法检查
5.    then
6.       ErrReport ← SyntaxError(Candidate);
7.       ErrExamples ← ErrRAG(ErrReport); ▷ 错误数据库 RAG
8.       Feedback ← ErrReport ++ ErrExamples; ▷ 语法错误反馈
9.       continue;
10.   end
11.   if not SemanticsCheck(Candidate); ▷ 语义检查
12.   then
13.      ErrReport ← SemanticsError(Candidate);
14.      Feedback ← ErrReport; ▷ 语义错误反馈
15.      continue;
16.   end
17.   TargetRule ← Candidate;
18.   return TargetRule;
19. end
```

2.4.1 错误检查

错误检查组件包含语法检查器和语义检查器这两类检查器, 分别用于检查 LLM 翻译后的代码中可能存在的语法错误和语义错误.

语法检查器 (第 4 行): 对翻译后的代码进行语法检查, 确保其符合目标 Datalog 方言的语法规范. 我们设计了两种方案. 第 1 种方案是直接使用 Datalog 引擎中的语法检查组件. 每一种 Datalog 方言都有其对应的 Datalog 引擎, 引擎中包含了对 Datalog 方言的源代码进行语法检查的组件 (编译器、解释器等). 因此, 纠错机制中的语法检查器可以直接委托给对应方言的 Datalog 引擎中的语法检查组件, 这样既可以通过使用官方引擎以保证语法检查的有效性, 也可以消除开发人员为不同方言开发语法检查器的开销. 然而, 部分 Datalog 引擎 (如 DDlog) 的语法检查组件的报错信息较为简略, 对后续的信息反馈组件不够友好. 第 2 种方案是基于 Datalog 方言的文法定义, 利用标准的语法解析器生成器 (如 ANTLR[29]等), 自动化生成一个对信息反馈组件友好的语法解析器, 在处理语法错误时能给 LLM 生成更具指向性的反馈. 这种方式标准化程度更高, 且可以自动生成语法解析器, 无需开发人员手动开发.

语义检查器 (第 11 行): 通过语义检查来保证翻译前后的代码具有相同的语义, 否则就是一次失败的翻译. 与语法检查器不同的是, 不存在现成的语义检查器. 一个最理想的语义检查器是通过形式化验证的方式来严格保证以两种 Datalog 方言编写的代码具有相同的语义. 然而, 这种方式需要为每一种方言的语义都进行形式化建模, 这种方式开发成本高且不具备可拓展性. 因此, 我们采用了代码翻译以及程序合成领域中最常用的基于测试的方案. 每种 Datalog 方言都具有对应的可执行 Datalog 引擎, 可用于作为执行语义检查的基础设施. 单规则的输入为规则体谓词的事实, 输出为规则头谓词的事实. 如果翻译前后的规则对于相同的输入事实能产生相同的输出事实, 我们

© 中国科学院软件研究所 http://www.jos.org.cn

即认为测试通过. 我们从完整 Datalog 程序的事实数据库中抽取相应谓词的事实作为测试用例, 在为单规则补齐执行必要的上下文环境之后, 分别借助现有基础设施进行执行, 再比对输出事实是否相同, 以检查是否存在代码语义不等价问题.

2.4.2 信息反馈

当语法检查器或语义检查器检查到错误后, 信息反馈组件会将相应的错误报告反馈给 LLM, 以使 LLM 对其所犯的错误进行充分的理解, 便于后续的错误修复.

语法错误 (第 6–8 行): 当语法检查器检查到语法错误后, 信息反馈组件会将源语言代码、错误翻译的目标语言代码以及语法检查器生成的错误信息组织为一份错误报告, 反馈给 LLM.

语义错误 (第 13–14 行): 当语义检查器检查到语义错误后, 信息反馈组件会将源语言代码、错误翻译的目标语言代码以及测试用例通过/失败的具体信息组织为一份错误报告, 反馈给 LLM.

2.4.3 错误修复

在得到错误报告之后, 我们分别为语法错误和语义错误设计了对应的修复策略, 并基于错误报告和修复策略, 增强 LLM 的错误修复能力.

语法修复: 鉴于 LLM 强大的代码理解和生成能力, 在我们精心设计的提示词引导下, LLM 一般不会在目标 Datalog 方言的语法上出现显著错误. 此外, 语法错误一般具有特定的分类与特征. 因此, 我们为语法错误设计了一种基于错误数据库的检索增强生成的修复策略 (第 7 行). 我们通过对语法正确的 Datalog 规则进行随机变异来生成错误数据库. 具体而言, 给定一对语义等价的规则对 (如开发人员为少样本提示构造的等价规则对), 我们对正确的目标方言规则进行随机变异, 生成大量的候选语法错误翻译. 接下来, 我们同样通过语法检查器来检查候选语法错误翻译是否为真的语法错误, 若为真语法错误, 我们将源方言规则、正确的目标方言规则、包含语法错误的目标方言规则以及相应的错误信息作为错误数据库中的一个四元组. 此外, 为了提高错误数据库的效率, 我们也会对生成的四元组进行聚类分组和约减. 在错误数据库构造完成后, 我们通过检索增强生成的方式, 将错误数据库中与当前错误报告最相关的几条语法错误四元组作为提示词中的上下文, 以指导 LLM 对当前的语法错误进行修复.

语义修复: 与语法错误不同, 语义错误没有显式的报错信息, 只有测试用例的通过/失败的报告, 因此对语义错误的修复需要对代码语义有较深的理解, 其修复没有一种固定的模板. 为此, 我们尝试使用能力更强的 LLM 来修复语义错误. 我们的核心见解是能力更强的 LLM 对代码语义的理解会比能力稍弱的 LLM 更深刻. 加之我们提供的错误报告, 能力更强的 LLM 可能有较大的概率能够修复语义错误.

通过引入基于检查-反馈-修复的迭代纠错机制, 本文提出的 LLM 赋能的 Datalog 代码翻译技术能够有效地发现并纠正 LLM 生成的翻译结果中的错误, 提高翻译的准确性.

2.5 合并翻译

在 LLM 完成翻译任务, 并且翻译结果通过检查 (第 2.4 节) 之后, 我们得到了一组成功翻译的 Datalog 目标方言单规则. 接下来, 我们将对这些单规则进行后处理, 使用合并器将其进行合并. 得到一段完整的 Datalog 目标方言代码. 鉴于 Datalog 语言的声明式特性, 与规则拆分器类似, 合并器的主要工作就是对所有 Datalog 目标方言单规则进行遍历并组合, 再加上谓词声明的翻译, 最后合并为一段完整目标代码. 然而, 与拆分器不同之处在于, 由于 LLM 的内在不确定性, 待合并的单规则之间可能存在不一致性. 具体而言, 合并器有以下两个难点需要解决.

一是同一个谓词在不同的规则中可能会被翻译为不同的谓词名. 仅翻译单规则虽然有利于提示工程和纠错机制, 但不同规则分别翻译则提高了谓词名翻译不一致的风险. 我们设计了谓词名一致性检查算法 (算法 2) 来检查不同规则之间谓词的一致性. 我们以翻译前后的谓词定义作为谓词名的标准, 分别检查规则头 (第 4–6 行) 和规则体 (第 8–12 行) 中的谓词名是否一致. 当出现不一致时, 会对翻译后单规则中的谓词名进行修复. 此外, 我们也通过在提示词设计中明确指出谓词名的翻译要求以及利用 Self-Consistency 解码策略[30]来缓解这个问题.

二是在代码翻译过程中, 我们可能会引入一些辅助谓词, 对不同单规则分别翻译可能会导致在合并时发生辅助谓词名重复的情况. 针对这个问题, 我们在规则拆分时为 Datalog 源方言的单规则引入唯一序号, 在引入辅助谓

© 中国科学院软件研究所 http://www.jos.org.cn

词时, 使用序号来进行区分, 以避免辅助谓词名重复.

算法 2. 谓词名一致性检查.

输入: 翻译前后谓词名映射 PredMap; 翻译前后规则映射 RuleMap;
输出: 一组通过谓词名一致性检查的 Datalog 目标方言单规则 TargetRules.

```
1. TargetRules ← [ ];
2. foreach SourceRule, TargetRule in RuleMap do
3.    // 规则头谓词名一致性检查
4.    if TargetRule.head.name != PredMap[SourceRule.head.name] then
5.       TargetRule.head.name ← PredMap[SourceRule.head.name];
6.    end
7.    // 规则体谓词名一致性检查
8.    for i in range(SourceRule.tails.length) do
9.       if TargetRule.tails[i].name != PredMap[SourceRule.tails[i].name] then
10.          TargetRule.tails[i].name ← PredMap[SourceRule.tails[i].name];
11.      end
12.   end
13.   TargetRules.add(TargetRule);
14. end
15. return TargetRules;
```

基于我们设计的“拆分-翻译-合并”的分治翻译策略, 本文所提出的 LLM 赋能的 Datalog 代码翻译技术能够有效地解决 LLM 上下文长度受限的问题, 为大型 Datalog 代码的翻译提供了坚实基础.

3 应用: 增量程序分析框架

本节将基于其设计一个通用的声明式增量程序分析框架, 以验证 LLM 赋能的 Datalog 代码翻译技术的有效性.

基于 Datalog 的声明式程序分析框架在程序分析领域受到了广泛关注, 其基本架构如图 3(a) 所示. 基于 Datalog 的程序分析框架其声明式表达的简洁性以及与分析算法描述的高度契合, 使得其可以自然地表达分析算法. 然而, 增量程序分析算法[31,32]很少基于 Datalog 的声明式程序分析框架来开发, 因为增量程序分析算法中包含了大量用于处理变更追踪、失效移除和重新计算的细节, 这些算法细节并不适合用 Datalog 来表达.

图 3 基于 Datalog 的声明式程序分析框架

© 中国科学院软件研究所 http://www.jos.org.cn

已有研究人员对声明式增量程序分析框架进行了初步研究[9,10], 如图 3(b) 所示. 这些工作通过将原本基于不支持增量评估的 Datalog 方言实现的分析算法翻译为支持增量评估的 Datalog 方言上的等价代码, 从而借助增量 Datalog 引擎自动获得增量分析的能力. 增量 Datalog 引擎在基础设施层面支持了变更追踪、失效移除和重新计算等能力, 上层的分析算法只需用 Datalog 表达全量分析算法逻辑, 增量 Datalog 引擎可以为分析算法自动提供增量化支持. 然而, 现有的声明式增量程序分析框架仅针对特定的 Datalog 源方言和目标方言. 例如, 沈天琪等人[9]的工作仅支持从 Soufflé方言翻译为 DDlog 方言, Szabó[10]对 CodeQL 的增量化工作也仅支持从 QL 方言翻译到 VQ 方言. 这些工作都需要研究人员人工设计 Datalog 方言间的翻译规则, 面对 Datalog 方言的多样性以及方言自身演化带来的翻译规则失效问题, 现有工作在灵活性和可拓展性上都存在很大的限制.

为此, 我们基于 LLM 赋能的 Datalog 代码翻译技术, 设计了如图 3(c) 所示的通用声明式增量程序分析框架. 我们的增量分析框架与现有的声明式分析框架在整体架构上基本相同, 主要的区别在于将现有技术中基于人工设计规则的 Datalog 翻译技术替换为了 LLM 赋能的 Datalog 代码翻译技术. 因此, 我们的声明式增量程序分析框架能够灵活地为以不同 Datalog 方言编写的程序分析规则自动提供增量化能力.

我们的通用声明式增量程序分析框架的基本运行流程如下: 对于非增量 Datalog 方言 A 和增量 Datalog 方言 B, 首先, 框架利用 LLM 将基于方言 A 实现的静态分析规则 (实现 A) 翻译为方言 B 上的实现 (实现 B), 其中会对代码翻译结果进行检查和修复; 其次, 将不同版本的代码库对应的输入事实作为实现 B 的输入; 最后, 实现 B 根据代码变更进行增量分析后报告最新的分析结果.

4 实验评估

为了展示本文提出的 LLM 赋能的 Datalog 代码翻译技术的能力, 以及基于其实现的增量程序分析框架的效率, 本文挑选了一组 Datalog 方言及其上面的应用进行实验评估. 实验评估主要尝试回答以下 3 个研究问题.

- RQ1: LLM 赋能的 Datalog 代码翻译技术准确率如何?
- RQ2: LLM 赋能的 Datalog 代码翻译技术在不同 Datalog 方言上的可拓展性如何?
- RQ3: 在此代码翻译技术的基础上实现的增量程序分析框架对分析效率的提升效果如何?

4.1 实验设置

实验评估中的通用实验设置如下.

(1) 实验在一台配备 Intel® Xeon® Gold 6240 @2.60 GHz 处理器 (75 线程), 285 GB 内存, 1 TB 磁盘的 Ubuntu 20.04.6 LTS 服务器上运行.

(2) 我们的实现并不绑定特定的 LLM. 在实验过程中, 使用来自 Azure 的 OpenAI 云服务, 通过这种方式使用了 GPT-3.5-Turbo 和 GPT-4 模型. 其中 GPT-3.5-Turbo 起最主要的作用, 包括全部代码翻译以及绝大多数错误反馈后的规则修复工作, GPT-4 仅在修复环节被少量使用.

(3) 在我们的纠错机制中, 会迭代对翻译出错规则进行修复. 为了在翻译效率和正确率之间取得平衡, 我们设置语法错误最大修复次数为 4, 语义错误最大修复次数为 2. 考虑到模型能力和性价比, 我们在语义修复前 3 次使用 GPT-3.5-Turbo, 最后 1 次使用 GPT-4; 语义修复均使用 GPT-4.

(4) 在实验评估中, 涉及以下 4 种 Datalog 方言: LogiQL 是一种商业化方言, 曾用于 Doop 框架早期版本实现; Soufflé是一种专为静态分析设计的方言, 也是当前 Doop 框架所用方言; DDlog 是一种支持高效增量评估的 Datalog 方言; Ascent 是一种通过宏机制嵌入 Rust 的 Datalog 方言, 支持与 Rust 互操作.

4.2 RQ1: 翻译准确率

本节实验对本文提出的 LLM 赋能的 Datalog 代码翻译技术的翻译性能进行评估. 我们通过翻译结果的语法检查通过率和语义检查通过率来衡量翻译性能. 考虑到 LLM 赋能的 Datalog 翻译流程上处理的是单条规则 (见第 2.2 节), 我们在实验中也以单条规则作为翻译正确/有误的基本单位.

本节实验评估的源方言为 Soufflé, 目标方言为 DDlog. 我们使用的 Datalog 代码数据集有 3 个来源: 1) 由

© 中国科学院软件研究所 http://www.jos.org.cn

queryFuzz[26]随机生成的 63 条规则, 使用随机生成的规则是为了防止 LLM 出现数据泄漏, 使用其训练集中可能已经记录下来的信息, 以确保我们的方法拥有足够的泛化能力; 2) 一个公开的增量 Datalog 评估测试数据集 (https://github.com/frankmcsherry/dynamic-datalog), 包括 doop-micro 和 galen; 3) 来自真实 Datalog 项目 Doop 指针分析框架中的指针分析算法, 包括 doop-ci.

表 1 展示了本文提出的代码翻译技术的整体翻译性能. 从表中可以看出, 在语法检查方面, 本文技术在 3 个数据集上达到了 100.00% 语法检查通过率, 在最大数据集 doop-ci 上的通过率 (99.63%) 也接近 100.00%. 从这个数据可以看出, LLM 具备强大的代码生成能力, 在 Datalog 代码语料显著稀缺的情况下, 通过我们的提示词设计和纠错机制, LLM 即可生成几乎没有语法错误的目标 Datalog 方言代码. 在语义检查方面, 本文技术在最小的 galen 数据集上可以达到 100.00% 通过率, 在最大的 doop-ci 数据集上达到了 81.00% 通过率. 从本文技术的语义检查通过率比语法检查通过率略低可以看出, LLM 在代码语义理解能力上还略显不足, LLM 设计与训练的工作还应重点在此方向进行努力. 但同时也可以看到, 本文技术在最复杂的数据集上仍取得了 81.00% 的语义通过率, 证明了 LLM 在 Datalog 代码翻译任务上取得了较好的效果并具有可观的前景.

表 1　整体翻译性能

数据集	#规则	语法检查		语义检查	
		#规则	通过率 (%)	#规则	通过率 (%)
doop-ci	821	818	99.63	665	81.00
doop-micro	124	124	100.00	118	95.16
queryFuzz	63	63	100.00	55	87.30
galen	6	6	100.00	6	100.00

接下来, 我们设计了一组消融实验, 针对本文所提出的 LLM 赋能的 Datalog 代码翻译技术中提示工程、纠错机制以及错误数据库等核心组件, 分别验证其对翻译性能的影响与贡献.

对于提示工程消融实验, 我们为翻译技术设置了两个对照版本 (pv1 和 pv2), 为了排除纠错机制对翻译结果的影响, 这两个版本都不使用纠错机制, 仅在提示词设计上有所区别. pv1 使用仅包含简单的翻译指令和输入代码作为提示词; pv2 使用完整提示工程. 我们通过比较翻译结果的语法/语义检查通过率来衡量翻译性能.

表 2 展示了针对提示工程的消融实验结果. 从表中可以看出, 在语法检查方面, 当不使用我们设计的提示工程时 (pv1), 通过率最高 (doop-micro) 仅有 35.48%, 在最大数据集 (doop-ci) 上的通过率为 17.30%, 而在最小数据集 (galen) 上的通过率甚至下降到了 0.00%; 而当使用我们设计的提示词时 (pv2), 通过率最低 (doop-ci) 为 89.40%, 最高 (galen) 可达 100.00%. 对比可看出我们设计的提示工程可以显著提高翻译结果的语法检查通过率. 在语义检查方面, pv1 在所有数据集上的通过率均为 0.00%; 而 pv2 的通过率最低 (doop-ci) 为 56.15%, 最高 (doop-micro) 可达 85.48%. 因此我们设计的提示工程可以显著提高翻译结果的语义检查通过率. 综合两组数据, 我们可以得出结论, 我们设计的提示工程对于 LLM 在 Datalog 代码翻译任务上的性能至关重要, 可以显著提高 LLM 对 Datalog 代码的生成能力 (语法检查通过率) 和理解能力 (语义检查通过率).

表 2　提示工程消融实验结果

版本	数据集	#规则	语法检查		语义检查	
			#规则	通过率 (%)	#规则	通过率 (%)
pv1	doop-ci	821	142	17.30	0	0.00
	doop-micro	124	44	35.48	0	0.00
	queryFuzz	63	2	3.17	0	0.00
	galen	6	0	0.00	0	0.00
pv2	doop-ci	821	734	89.40	461	56.15
	doop-micro	124	116	93.55	106	85.48
	queryFuzz	63	60	95.24	36	57.14
	galen	6	6	100.00	5	83.33

© 中国科学院软件研究所 http://www.jos.org.cn

对于纠错机制的消融实验, 我们同样为翻译技术设置了两个对照版本 (fv1 和 fv2), 这两个版本都使用完整的提示工程, 仅在纠错机制上有所区别. fv1 不使用纠错机制, 直接对 LLM 的输出进行语法/语义检查; fv2 使用完整的纠错机制. 我们通过比较两个版本的翻译结果的语法检查通过率和语义检查通过率来衡量翻译性能.

表 3 展示了针对纠错机制的消融实验结果. 可以看出, 使用我们设计的纠错机制 (fv2) 比不使用纠错机制 (fv1) 时, 在语法检查通过率和语义检查通过率上都获得了提升. 特别是在语义检查通过率上, 分别将 doop-ci 从 56.15% 提升到了 81.00%, queryFuzz 从 57.14% 提升到了 87.30%. 因此, 综合两组数据, 我们可以得出结论, 我们设计的纠错机制可以增强 LLM 对翻译错误的理解, 可以显著增强 LLM 对 Datalog 代码的语义理解能力, 从而帮助 LLM 修复其在之前迭代轮次中所犯的错误, 这对于 LLM 在 Datalog 代码翻译任务上的性能至关重要.

表 3　纠错机制消融实验结果

版本	数据集	#规则	语法检查		语义检查	
			#规则	通过率 (%)	#规则	通过率 (%)
fv1	doop-ci	821	734	89.40	461	56.15
	doop-micro	124	116	93.55	106	85.48
	queryFuzz	63	60	95.24	36	57.14
	galen	6	6	100.00	5	83.33
fv2	doop-ci	821	818	99.63	665	81.00
	doop-micro	124	124	100.00	118	95.16
	queryFuzz	63	63	100.00	55	87.30
	galen	6	6	100.00	6	100.00

最后, 我们还设计了消融实验来评估纠错机制中的错误数据库对翻译性能和修复效果的影响. 我们为翻译技术设置了两个对照版本 (dv1 和 dv2), 这两个版本都使用完整的提示工程, 仅在纠错机制上有所区别. dv1 使用不包含错误数据库的纠错机制, 仅利用反馈信息, 完全依赖 LLM 自身进行修复, 重新产生输出; dv2 使用完整的纠错机制. 我们通过比较两个版本的翻译结果的语法检查通过率和语义检查通过率来衡量翻译性能, 并通过对比平均修复次数来评估错误数据库对修复效果的影响.

表 4 展示了针对错误数据库的消融实验结果. 就语法检查和语义检查的结果而言, 不使用错误数据库而仅依赖于 LLM 来进行修复时 (dv1), 已经能够取得较好的翻译性能. 在 4 个数据集上, 语法通过率最低 (doop-ci) 为 97.81%, 最高 (doop-micro, galen) 可达 100.00%; 语义通过率最低 (queryFuzz) 也达到了 77.78%, 最高 (galen) 可达 100.00%. 而使用错误数据库可以进一步提高翻译性能. 在 4 个数据集上, 语法通过率最低 (doop-ci) 为 99.63%, 最高 (doop-micro, queryFuzz, galen) 可达 100.00%; 语义通过率最低 (doop-ci) 也达到了 81.00%, 最高 (galen) 可达 100.00%. 就每条发生语法错误的规则的平均修复次数而言, 此数据在使用错误数据库时在 4 个数据集上都得到了有效地降低. 因此, 综合两组数据, 我们可以得出结论, 使用错误数据库可以进一步提高 LLM 赋能的 Datalog 代码翻译技术的翻译性能, 并且能有效针对减低发生语法错误的规则的平均修复次数.

表 4　错误数据库消融实验结果

版本	数据集	#规则	语法检查		语义检查		平均修复次数
			#规则	通过率 (%)	#规则	通过率 (%)	
dv1	doop-ci	821	803	97.81	654	79.66	2.78
	doop-micro	124	124	100.00	109	87.90	2.14
	queryFuzz	63	62	98.41	49	77.78	2.50
	galen	6	6	100.00	6	100.00	1
dv2	doop-ci	821	818	99.63	665	81.00	1.49
	doop-micro	124	124	100.00	118	95.16	1.13
	queryFuzz	63	63	100.00	55	87.30	1.67
	galen	6	6	100.00	6	100.00	0

© 中国科学院软件研究所 http://www.jos.org.cn

本节实验评估结果表明, 我们设计的 LLM 赋能的 Datalog 代码翻译技术整体上展现出了优异的翻译性能, 其中提示工程、纠错机制、错误数据库等核心组件均为充分利用和增强 LLM 对 Datalog 代码的理解和生成能力做出了突出贡献, 对 LLM 在 Datalog 代码翻译任务上的性能至关重要.

4.3 RQ2: 可拓展性

为了评估本文提出的 LLM 赋能的 Datalog 代码翻译技术对更多 Datalog 方言的可拓展性, 我们在另外两种具有代表性的 Datalog 方言 (Ascent[27]和 LogiQL[33]) 上评估了其代码翻译能力. 我们选择 Ascent 是因为它作为一种新方言 (于 2022 年发布), 其代码在 LLM 训练语料中更为稀少, 在 Ascent 这样的新方言上的性能评估可进一步展现我们的翻译技术的能力; 我们选择 LogiQL 是因为它是 Doop 框架早期使用的 Datalog 方言, 但 Doop 在后续发展中因 LogiQL 存在的性能和学术使用限制等原因而迁移到了 Soufflé上[8], 在 LogiQL 上的实验评估可以展示我们的翻译技术在过时遗留代码现代化任务上的能力.

具体而言, 我们通过将提示工程和相关组件中特定方言相关的部分针对 Ascent-DDlog 和 LogiQL-DDlog 进行了实例化, 这使得我们可以基于 LLM 赋能的 Datalog 代码翻译技术执行将这两种方言翻译为 DDlog 方言的任务. 我们用以验证对应翻译性能的数据集分别为旧版本 Doop 框架 (基于 LogiQL 实现) 中的指针分析规则 doop-micro, 以及使用 Ascent 实现的一个支持 Doop 框架前端的指针分析规则 doop-thin. 其余实验设置与 RQ1 中相同. 此外, 我们已经在 RQ1 中对翻译技术的各核心组件做了完善的消融实验, 因此, 在本节的实验评估中, 我们仅展示在完整翻译配置下的最终语法/语义检查通过率.

表 5 展示了 LLM 赋能的 Datalog 代码翻译技术在不同方言对上的翻译性能. 可以看出, 我们提出的 Datalog 代码翻译技术在拓展到其他 Datalog 方言时仍表现出了良好性能. 两个数据集上的语法检查通过率分别为 98.04% 和 97.96%, 均接近 100%; 就语义检查通过率而言, 两个数据集上的表现也均在 90% 左右.

表 5　不同方言上的翻译性能

数据集	#规则	源方言	目标方言	语法检查		语义检查	
				#规则	通过率 (%)	#规则	通过率 (%)
doop-micro	204	LogiQL	DDlog	200	98.04	189	92.65
doop-thin	98	Ascent	DDlog	96	97.96	87	88.78

实验结果表明, 尽管不同的 Datalog 方言在语法和特性上存在一定差异, 我们提出的 LLM 赋能的 Datalog 代码翻译技术仍然能很好地处理不同方言之间的翻译任务. 同时, 我们的方法在拓展性上的优势也得到了体现. 传统的翻译器开发通常需要针对每一种新的语言对从词法分析到代码生成的整个流程进行大量适配, 工作量巨大. 而我们的方法只需要对提示词进行适当修改并开发少量新的后处理组件, 就能以较低的成本支持新的 Datalog 方言, 大大降低了 Datalog 方言支持拓展的难度.

4.4 RQ3: 增量分析效果

在 RQ1 (第 4.2 节) 和 RQ2 (第 4.3 节) 中, LLM 赋能的 Datalog 代码翻译技术已经在 Soufflé、LogiQL 和 Ascent 这 3 种 Datalog 方言到 DDlog 的翻译任务上展现了优异的翻译性能. 接下来, 我们就在提出的通用声明式增量程序分析框架 (第 3 节) 中评估基于这 3 种方言的分析规则在翻译到 DDlog 上后的增量分析效果.

我们在评估中使用的分析应用包括以下 3 组: 1) 基于 Soufflé的 doop-ci 分析; 2) 基于 LogiQL 的 doop-micro 分析; 以及 3) 基于 Ascent 的 doop-thin 分析. 对于 RQ1 和 RQ2 中存在的少量未能正确翻译的单规则, 我们通过结合人工以及与 LLM 交互的方式进行手动修复, 并对修复后的完整 DDlog 代码进行了测试. 我们的分析项目数据集中包括 pmd、ZooKeeper、errorprone 这 3 个项目, 采用连续 20 次提交对应的 JAR 包, 按序依次进行增量分析. DDlog 引擎评估所需的增量事实复用了 DDoop 的前端来进行生成. 在实验中, 我们关注的是方言翻译后使用 DDlog 引擎增量评估带来的收益, 因此我们仅关注引擎切换的性能对比, 即各个 Datalog 引擎进行评估的耗时.

图 4 展示了在我们的通用声明式增量程序分析框架下, 原本基于 LogiQL 的 doop-micro 分析规则和基于

© 中国科学院软件研究所 http://www.jos.org.cn

Ascent 的 doop-thin 分析规则在 DDlog 的自动增量化支持下的增量效果. 为方便起见, 图 4 的纵坐标采用了对数轴. 可以看出, 基于 LLM 赋能的 Datalog 代码翻译技术的增量化机制在实际项目中取得了显著的加速效果. 对于 LogiQL 语言, 其 Doop-pmd、Doop-errorprone 和 Doop-ZooKeeper 这 3 个数据集在切换到 DDlog 引擎后, 首轮分析耗时比原本的 LogicBlox 引擎分别减少了 48.20%、50.70% 和 27.50%, 而增量评估的平均耗时更是大幅降低到了原来的 0.28%、0.30% 和 0.48%, 提升效果非常明显. 这主要是由于开放提供给学术研究受限使用的 LogicBlox 引擎的版本非常老旧, 性能不足. 对于 Ascent 语言, 其也有类似的加速效果: 在切换到 DDlog 引擎后, 3 个数据集 Doop-pmd、Doop-errorprone 和 Doop-ZooKeeper 的首轮分析耗时比原本的 Ascent 引擎分别减少了 93.60%、84.50% 和 82.60%, 而增量评估的平均耗时更是大幅降低到了原来的 0.05%、0.16% 和 0.13%. DDlog 相比 Ascent 具有巨大性能优势的原因可能是程序分析算法中存在大量的复杂递归规则, 而 Ascent 引擎应对复杂递归规则的能力目前尚显不足. 对于 Soufflé语言, 切换到 DDlog 之后的首轮分析耗时比原来的 Soufflé更长, 这是因为 Soufflé 引擎是当前最先进的非增量 Datalog 引擎之一, 其中实现了大量的优化技术. 后续增量评估的平均耗时分别降低到了原来的 75.34%、22.26% 和 37.84%, 仍然取得了较好的加速效果.

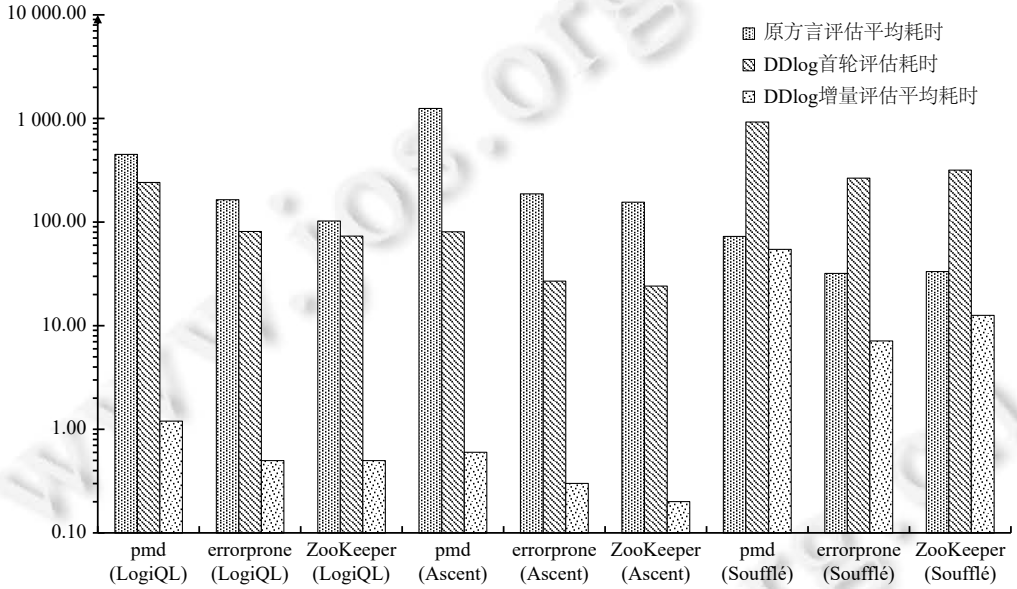


图 4 基于 LLM 赋能的 Datalog 代码翻译的增量化机制评估效果

实验结果表明, 基于 LLM 赋能的 Datalog 代码翻译技术的增量化机制能够有效地为基于不同 Datalog 方言的应用引入增量评估能力, 其在实际项目中取得了可观的加速效果. 这说明了我们的 Datalog 代码翻译技术能够为现有的 Datalog 应用提供一种低人力成本获取高效增量能力的新思路.

5 讨 论

5.1 人工成本

本文所提的 LLM 赋能的 Datalog 代码翻译技术的主要目标是为了减少开发人员在处理多种不同 Datalog 方言之间的代码翻译任务时的工作量. 人工直接重写代码需要耗费开发人员数月时间[8], 人工设计翻译规则需要耗费数周甚至上月的时间[9], 而使用本文所提的 LLM 赋能 Datalog 代码翻译技术, 开发人员耗时可以缩短到数天之内. 然而, 本文技术并不能完全消除开发人员的人工工作量, 仍需要开发人员提供小规模的等价代码对示例样本以辅助 LLM 的翻译. 这同时也意味着仍要求开发人员具有 Datalog 相关的专业知识, 但相比现有的人工重写和人工设计翻译规则这两类技术, 专业要求有明显降低.

© 中国科学院软件研究所 http://www.jos.org.cn

5.2 Datalog 方言特性

面向不同需求和应用所设计的不同 Datalog 方言, 其所支持的语言特性也有所不同. 语言特性可分为语法特性 (语法糖, 方便开发人员使用) 和语义特性 (增强语言表达能力). 对于使用了源方言语法特性的代码, 可以通过解语法糖的方式转换为不使用相应特性的等价版本, 解糖之后的代码可以交给代码翻译技术进行正常翻译. 对于语义特性, 所有 Datalog 代码翻译技术都只能在仅使用了方言对的语义特性交集的待翻译代码上实现等价翻译. 如果待翻译的源方言代码使用了目标方言不支持的语义特性, 则待翻译代码不能等价翻译到目标方言, 只能实现部分代码片段的等价翻译. 而另一方面, 如果开发人员想要使用目标方言中的新语义特性, 则其可以通过代码翻译的方式, 在等价翻译的目标代码基础之上, 使用目标方言新特性来继续开发.

5.3 局限性

本文工作虽然在 Datalog 代码自动化翻译任务上表现优异, 但仍存在以下局限性. (1) 本文工作仍需用户手动设计小规模等价代码对, 并需要对特定源方言和目标方言所使用的 LLM 提示词进行适配, 后续工作可以探索进一步提高自动化水平. (2) 本文工作仅在 GPT 系列 LLM 上进行了评估, 后续工作可以评估其他商用和开源 LLM 在 Datalog 代码自动化翻译任务上的表现. (3) 鉴于 LLM 对翻译结果无任何保证, 本文工作使用测试来对结果进行语义检查, 但这不能保证语义完全等价, 后续工作可进一步探索更完备的语义检查技术.

6 相关工作

6.1 代码翻译

代码翻译将一种编程语言的源代码等价翻译为另一种语言的源代码. 目前主要存在两大类代码翻译方法. 第 1 类包括转译器或源到源编译器, 它们利用程序分析技术将代码从一种语言翻译为另一种语言. 例如, C2Rust[34]和 CxGo[35]分别是将 C 翻译为 Rust 和 Go 的转译器, 而 Sharpen[36]和 Java2CSharp[37]将 Java 翻译为 C#. 第 2 类包括基于学习的技术, 包括基于统计机器翻译[38]和基于树的神经网络[39]的工作, 以及利用深度学习和无监督学习[22,40]来翻译 C++、Java 和 Python 代码的工作. 最近的基于学习的技术利用了大语言模型 (例如, StarCoder[41]、Code LLaMA[15]、CodeT5[42]和 CodeGeeX[43]等) 来执行代码生成和翻译任务. 基于 LLM 的代码翻译相关工作更多关注的是通用编程语言之间的代码翻译, 而本文关注 LLM 赋能的 Datalog 代码翻译任务.

6.2 LLM 赋能的软件工程 (LLM4SE)

LLM 赋能的软件工程将传统软件工程中的许多挑战重新表述为数据、代码或文本分析任务. LLM 为软件工程提供了新的视角, 这些模型处理复杂任务的潜力正在从根本上重塑许多软件工程实践和工具. LLM 已经被广泛应用于各种软件工程任务中, 如代码摘要[44]、代码补全[45]、代码翻译[12]、软件测试[46]、漏洞检测[47]、程序分析[48]、错误定位[49]和程序修复[50]等, 取得了瞩目的效果. 目前已经有许多专门为代码任务和软件工程任务开发的代码 LLM. 然而, 目前的 LLM 赋能的软件工程仍存在很多挑战, 最核心的问题在于 LLM 的决策和输出往往是黑箱的, 不具备可解释性. 在本文工作中, LLM 同样存在这些问题. 提高 LLM 的可解释性和透明度对于 LLM 赋能的软件工程的可靠性至关重要.

6.3 增量程序分析框架

增量程序分析框架通过在底层提供依赖追踪等能力, 为分析开发者提供了开发增量程序分析的基础设施. 分析开发者只需要按照增量程序分析框架提供的接口实现相应的分析算法, 增量程序分析框架会自动提供增量分析的能力, 大大降低了增量程序分析算法的开发难度. 基于 Datalog 的声明式增量程序分析框架是其中一类代表性技术[9,10], 这类技术基于支持增量评估的 Datalog 引擎, 利用代码翻译技术, 将现有的基于非增量 Datalog 方言开发的程序分析规则翻译到增量引擎上执行, 从而自动获得增量化能力. 本文工作进一步推动了这类技术的发展, 可以支持将以不同 Datalog 方言开发的程序分析规则翻译到增量引擎上执行. 此外, Reviser[51]提供了 IFDS/IDE 可分配程序分析算法[52,53]的增量框架, 可自动化增量执行可分配分析; Infer[54]为组合式程序分析算法提供了增量框架, 可以自动化执行组合式分析.

© 中国科学院软件研究所 http://www.jos.org.cn

7 总结与展望

本文提出了一种 LLM 赋能的 Datalog 代码自动化翻译技术, 旨在解决现有手工设计翻译规则的 Datalog 代码翻译方案的灵活性不足和高人力成本的问题, 并在此基础上设计了一种通用的基于 Datalog 的声明式增量程序分析框架. 我们的 Datalog 代码翻译技术通过“拆分-翻译-合并”的分治翻译策略应对 LLM 的上下文长度受限问题, 通过精心设计的提示工程来激发 LLM 对 Datalog 代码的理解和生成能力, 并通过检查-反馈-修复的迭代纠错机制来缓解 LLM 的幻觉问题, 增强其对 Datalog 代码的理解和生成能力. 实验评估结果表明, 我们的 Datalog 代码翻译技术的翻译性能表现优异, 并具有很好的可拓展性, 可轻松拓展到不同 Datalog 方言; 基于该翻译技术实现的增量化机制, 能够为不同 Datalog 上的实际应用带来显著的加速效果和效率提升. 在下一步工作中, 我们计划通过自动化选择具有代表性的等价 Datalog 代码对来增强少样本提示以及评估 Datalog 代码翻译技术能力, 使用 SMT 求解等形式化验证技术来保障 Datalog 代码翻译结果的语义正确性, 评估不同 LLM 在 Datalog 代码翻译任务上的表现, 并探索 Datalog 代码翻译技术的更多应用场景.

References:

[1] Loo BT, Condie T, Garofalakis M, Gay DE, Hellerstein JM, Maniatis P, Ramakrishnan R, Roscoe T, Stoica I. Declarative networking: Language, execution and optimization. In: Proc. of the 2006 ACM SIGMOD Int’l Conf. on Management of Data. Chicago: ACM, 2006. 97–108. [doi: 10.1145/1142473.1142485]

[2] Seo J, Guo S, Lam MS. SociaLite: An efficient graph query language based on Datalog. IEEE Trans. on Knowledge and Data Engineering, 2015, 27(7): 1824–1837. [doi: 10.1109/TKDE.2015.2405562]

[3] Shkapsky A, Yang MH, Interlandi M, Chiu H, Condie T, Zaniolo C. Big data analytics with Datalog queries on Spark. In: Proc. of the 2016 Int’l Conf. on Management of Data. San Francisco: ACM, 2016. 1135–1149. [doi: 10.1145/2882903.2915229]

[4] Bravenboer M, Smaragdakis Y. Strictly declarative specification of sophisticated points-to analyses. In: Proc. of the 24th ACM SIGPLAN Conf. on Object Oriented Programming Systems Languages and Applications. Orlando: ACM, 2009. 243–262. [doi: 10.1145/1640089.1640108]

[5] Jordan H, Scholz B, Subotić P. Soufflé: On synthesis of program analyzers. In: Proc. of the 28th Int’l Conf. on Computer Aided Verification. Toronto: Springer, 2016. 422–430. [doi: 10.1007/978-3-319-41540-6_23]

[6] Marczak WR, Huang SS, Bravenboer M, Sherr M, Loo BT, Aref M. SecureBlox: Customizable secure distributed data processing. In: Proc. of the 2010 ACM SIGMOD Int’l Conf. on Management of Data. Indianapolis: ACM, 2010. 723–734. [doi: 10.1145/1807167.1807246]

[7] Aref M, ten Cate B, Green TJ, Kimelfeld B, Olteanu D, Pasalic E, Veldhuizen TL, Washburn G. Design and implementation of the LogicBlox system. In: Proc. of the 2015 ACM SIGMOD Int’l Conf. on Management of Data. Melbourne: ACM, 2015. 1371–1382. [doi: 10.1145/2723372.2742796]

[8] Antoniadis T, Triantafyllou K, Smaragdakis Y. Porting doop to Soufflé: A tale of inter-engine portability for Datalog-based analyses. In: Proc. of the 6th ACM SIGPLAN Int’l Workshop on State of the Art in Program Analysis. Barcelona: ACM, 2017. 25–30. [doi: 10.1145/3088515.3088522]

[9] Shen TQ, Wang XZ, Bin XR, Bu L. DDoop: Incremental pointer analysis framework based on differential Datalog evaluation. Ruan Jian Xue Bao/Journal of Software, 2024, 35(6): 2608–2630 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/7100.htm [doi: 10.13328/j.cnki.jos.007100]

[10] Szabó T. Incrementalizing production CodeQL analyses. In: Proc. of the 31st ACM Joint European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. San Francisco: ACM, 2023. 1716–1726. [doi: 10.1145/3611643.3613860]

[11] Ryzhyk L, Budiu M. Differential Datalog. In: Proc. of the 3rd Int’l Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0 2019). Philadelphia: CEUR, 2019. 56–67.

[12] Pan R, Ibrahimzada AR, Krishna R, Sankar D, Wassi LP, Merler M, Sobolev B, Pavuluri R, Sinha S, Jabbarvand R. Lost in translation: A study of bugs introduced by large language models while translating code. In: Proc. of the 46th Int’l Conf. on Software Engineering. Lisbon: ACM, 2024. 82. [doi: 10.1145/3597503.3639226]

[13] GPT-4. 2023. https://openai.com/index/gpt-4-research/

[14] Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, Rozière B, Goyal N, Hambro E, Azhar F, Rodriguez A, Joulin A, Grave E, Lample G. LLaMA: Open and efficient foundation language models. arXiv:2302.13971, 2023.

© 中国科学院软件研究所 http://www.jos.org.cn

[15] Roziere B, Gehring J, Gloeckle F, *et al*. Code Llama: Open foundation models for code. arXiv:2308.12950, 2023.

[16] Ma H, Zhang CQ, Bian Y T, Liu LM, Zhang ZR, Zhao PL, Zhang S, Fu HZ, Hu QH, Wu BZ. Fairness-guided few-shot prompting for large language models. In: Proc. of the 37th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2023. 1870.

[17] Wei J, Wang XZ, Schuurmans D, Bosma M, Ichter B, Xia F, Chi EH, Le QV, Zhou D. Chain-of-Thought prompting elicits reasoning in large language models. In: Proc. of the 36th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2022. 1800.

[18] Besson F, Jensen T. Modular class analysis with Datalog. In: Proc. of the 10th Int'l Symp. on Static Analysis. San Diego: Springer, 2003. 19–36. [doi: 10.1007/3-540-44898-5_2]

[19] Hinchey M, Jackson M, Cousot P, Cook B, Bowen JP, Margaria T. Software engineering and formal methods. Communications of the ACM, 2008, 51(9): 54–59. [doi: 10.1145/1378727.1378742]

[20] Albrecht PF, Garrison PE, Graham SL, Hyerle RH, Ip P, Krieg-Brückner B. Source-to-source translation: Ada to Pascal and Pascal to Ada. In: Proc. of the 1980 ACM-SIGPLAN Symp. on Ada Programming Language. Boston: ACM, 1980. 183–193. [doi: 10.1145/948632.948658]

[21] Balogh GD, Mudalige GR, Reguly IZ, Antao SF, Bertolli C. OP2-Clang: A source-to-source translator using clang/LLVM LibTooling. In: Proc. of the 5th IEEE/ACM Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). Dallas: IEEE, 2018. 59–70. [doi: 10.1109/LLVM-HPC.2018.8639205]

[22] Roziere B, Lachaux MA, Chanussot L, Lample G. Unsupervised translation of programming languages. In: Proc. of the 34th Int'l Conf. on Neural Information Processing Systems. Vancouver: Curran Associates Inc., 2020. 1730.

[23] Welcome Llama 3—Meta's new open LLM. 2024. https://huggingface.co/blog/llama3

[24] Lozhkov A, Li R, Allal LB, *et al*. StarCoder 2 and the stack v2: The next generation. arXiv:2402.19173, 2024.

[25] Mansur MN, Wüstholz V, Christakis M. Dependency-aware metamorphic testing of Datalog engines. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis. Seattle: ACM, 2023. 236–247. [doi: 10.1145/3597926.3598052]

[26] Mansur MN, Christakis M, Wüstholz V. Metamorphic testing of Datalog engines. In: Proc. of the 29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Athens: ACM, 2021. 639–650. [doi: 10.1145/3468264.3468573]

[27] Sahebolamri A, Gilray T, Micinski K. Seamless deductive inference via macros. In: Proc. of the 31st ACM SIGPLAN Int'l Conf. on Compiler Construction. Seoul: ACM, 2022. 77–88. [doi: 10.1145/3497776.3517779]

[28] Chung J, Kamar E, Amershi S. Increasing diversity while maintaining accuracy: Text data generation with large language models and human interventions. In: Proc. of the 61st Annual Meeting of the Association for Computational Linguistics (Vol.1: Long Papers). Toronto: Association for Computational Linguistics, 2023. 575–593.

[29] Parr TJ, Quong RW. ANTLR: A predicated-*LL* (*k*) parser generator. Software: Practice and Experience, 1995, 25(7): 789–810. [doi: 10.1002/spe.4380250705]

[30] Wang XZ, Wei J, Schuurmans D, Le QV, Chi EH, Narang S, Chowdhery A, Zhou D. Self-consistency improves chain of thought reasoning in language models. In: Proc. of the 11th Int'l Conf. on Learning Representations. Kigali: ICLR, 2022. 1–24.

[31] Liu BZ, Huang J. SHARP: Fast incremental context-sensitive pointer analysis for Java. Proc. of the ACM on Programming Languages, 2022, 6(OOPSLA1): 88: 1–28. [doi: 10.1145/3527332]

[32] Liu BZ, Huang J, Rauchwerger L. Rethinking incremental and parallel pointer analysis. ACM Trans. on Programming Languages and Systems, 2019, 41(1): 6. [doi: 10.1145/3293606]

[33] Green TJ. LogiQL: A declarative language for enterprise applications. In: Proc. of the 34th ACM SIGMOD-SIGACT-SIGAI Symp. on Principles of Database Systems. Melbourne: ACM, 2015. 59–64. [doi: 10.1145/2745754.2745780]

[34] C2Rust. 2024. https://github.com/immunant/c2rust

[35] CxGo. 2024. https://github.com/gotranspile/cxgo

[36] Sharpen-Automated Java->C# coversion. 2024. https://github.com/mono/sharpen

[37] Java 2 CSharp Translator for Eclipse. 2013. https://sourceforge.net/projects/j2cstranslator/

[38] Nguyen AT, Nguyen TT, Nguyen TN. Migrating code with statistical machine translation. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 544–547. [doi: 10.1145/2591062.2591072]

[39] Chen XY, Liu C, Song D. Tree-to-tree neural networks for program translation. In: Proc. of the 32nd Int'l Conf. on Neural Information Processing Systems. Montréal: Curran Associates Inc., 2018. 2552–2562.

[40] Lachaux MA, Roziere B, Szafraniec M, Lample G. DOBF: A deobfuscation pre-training objective for programming languages. In: Proc. of the 35th Int'l Conf. on Neural Information Processing Systems. Virtual: Curran Associates Inc., 2021. 1147.

© 中国科学院软件研究所 http://www.jos.org.cn

[41] Li R, Allal LB, Zi YT, *et al*. StarCoder: May the source be with you! arXiv:2305.06161, 2023.

[42] Wang Y, Wang WS, Joty S, Hoi SCH. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Proc. of the 2021 Conf. on Empirical Methods in Natural Language Processing. Punta Cana: ACL, 2021. 8696–8708. [doi: 10.18653/v1/2021.emnlp-main.685]

[43] Zheng QK, Xia X, Zou X, Dong YX, Wang S, Xue YF, Shen L, Wang ZH, Wang AD, Li Y, Su T, Yang ZL, Tang J. CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X. In: Proc. of the 29th ACM SIGKDD Conf, on Knowledge Discovery and Data Mining. Long Beach: ACM, 2023. 5673–5684. [doi: 10.1145/3580305.3599790]

[44] Sun WS, Fang CR, You YD, Miao Y, Liu Y, Li YK, Deng GL, Huang SH, Chen YC, Zhang QJ, Qian HW, LiuY, Chen ZY. Automatic code summarization via ChatGPT: How far are we? arXiv:2305.12865, 2023.

[45] Tang Z, Ge JD, Liu SQ, Zhu TW, Xu TT, Huang LG, Luo B. Domain adaptive code completion via language models and decoupled domain databases. In: Proc. of the 38th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Luxembourg: IEEE, 2023. 421–433. [doi: 10.1109/ASE56229.2023.00076]

[46] Deng YL, Xia CS, Peng HR, Yang CY, Zhang LM. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In: Proc. of the 32nd ACM SIGSOFT Int'l Symp. on Software Testing and Analysis (ISSTA 2023). Seattle: ACM, 2023. 423–435. [doi: 10.1145/3597926.3598067]

[47] Sun YQ, Wu DY, Xue Y, Liu H, Wang HJ, Xu ZZ, Xie XF, Liu Y. GPTScan: Detecting logic vulnerabilities in smart contracts by combining GPT with program analysis. In: Proc. of the 46th Int'l Conf. on Software Engineering (ICSE). Lisbon: ACM, 2024. 166. [doi: 10.1145/3597503.3639117]

[48] Li HN, Hao Y, Zhai YZ, Qian ZY. Enhancing static analysis for practical bug detection: An LLM-integrated approach. Proc. of the ACM on Programming Languages, 2024, 8(OOPSLA1): 474–499. [doi: 10.1145/3649828]

[49] Wu YH, Li Z, Zhang JM, Papadakis M, Harman M, Liu Y. Large language models in fault localisation. arXiv:2308.15276, 2023.

[50] Jiang N, Liu K, Lutellier T, Tan L. Impact of code language models on automated program repair. In: Proc. of the 45th Int'l Conf. on Software Engineering (ICSE). Melbourne: IEEE, 2023. 1430–1442. [doi: 10.1109/ICSE48619.2023.00125]

[51] Arzt S, Bodden E. Reviser: Efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In: Proc. of the 36th Int'l Conf. on Software Engineering. Hyderabad: ACM, 2014. 288–298. [doi: 10.1145/2568225.2568243]

[52] Reps T. Program analysis via graph reachability. Information and Software Technology, 1998, 40(11–12): 701–726. [doi: 10.1016/S0950–5849(98)00093–7]

[53] Reps T, Horwitz S, Sagiv M. Precise interprocedural dataflow analysis via graph reachability. In: Proc. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. San Francisco: ACM, 1995. 49–61. [doi: 10.1145/199448.199462]

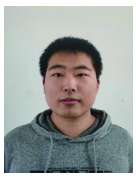
[54] Distefano D, Fähndrich M, Logozzo F, O'Hearn PW. Scaling static analyses at Facebook. Communications of the ACM, 2019, 62(8): 62–70. [doi: 10.1145/3338112]

附中文参考文献:

[9] 沈天琪, 王熙灶, 宾向荣, 卜磊. DDoop: 基于差分式 Datalog 求解的增量指针分析框架. 软件学报, 2024, 35(6): 2608–2630. http://www.jos.org.cn/1000-9825/7100.htm [doi: 10.13328/j.cnki.jos.007100]

王熙灶(1995－), 男, 博士生, CCF 学生会员, 主要研究领域为程序分析与验证, 程序设计语言.

宾向荣(2000－), 男, 博士生, CCF 学生会员, 主要研究领域为程序分析.



沈天琪(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为程序分析.

卜磊(1983－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为模型检验, 形式化方法, 信息物理系统, 复杂软件分析与验证.

© 中国科学院软件研究所 http://www.jos.org.cn