软件学报 ISSN 1000-9825, CODEN RUXUEW
2026,37(9):3598−3614 [doi: 10.13328/j.cnki.jos.007606] [CSTR: 32375.14.jos.007606]

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

基于大语言模型的 Python 到 Dafny 代码翻译*

杜奕成[1,2], 卢奕函[1,2], 朱雪阳[2], 张文辉[2]

[1](国科大杭州高等研究院 智能科学与技术学院, 浙江 杭州 310024)

[2](基础软件与系统重点实验室 (中国科学院 软件研究所), 北京 100190)

通信作者: 朱雪阳, E-mail: zxy@ios.ac.cn

摘　要: 近年来, 基于大模型的代码生成被广泛应用于软件开发领域. 然而, 由于大模型生成结果具有一定的随机性, 如何保证生成代码的正确性成为重中之重. 形式化验证是保障软件正确性的一种有力手段, 但验证前需要将代码及相应需求形式化为相关工具的规约语言. Dafny 是一种可验证的编程语言, 并有相应的自动化验证工具支持. 旨在探索大模型在将 Python 代码翻译为 Dafny 语言方面的能力. 为此, 提出了基于提示词的静态翻译和动态修复的翻译生成方法, 以及基于程序测试的翻译结果评估方法. 静态翻译方法考虑 3 种渐进的提示词模板 (基础模板、Dafny 示例模板、对照示例模板), 使用大模型一次性生成翻译; 动态修复方法多次调用大模型对翻译进行修复, 每次将上一轮代码测试的错误信息加入提示词. 评估时, 在没有现成测试用例集的情况下, 利用大模型生成测试用例集; 对于翻译结果, 应用 Dafny 测试工具结合测试用例集进行测试, 以评估其正确性. 在 GPT-4o 和 DeepSeek-V3 等大模型上使用 HumanEval、MBPP 和 LeetCode 等数据集进行实验. 结果表明, 使用对照示例模板和动态修复方法能够有效提升大模型在 Python 到 Dafny 翻译任务上的表现. 此外, 还基于实验结果分析影响翻译质量的因素, 并对评估方法与标准进行讨论.

关键词: 代码翻译; 提示词工程; 形式验证; Dafny

中图法分类号: TP311

中文引用格式: 杜奕成, 卢奕函, 朱雪阳, 张文辉. 基于大语言模型的Python到Dafny 代码翻译. 软件学报, 2026, 37(9): 3598–3614. http://www.jos.org.cn/1000-9825/7606.htm

英文引用格式: Du YC, Lu YH, Zhu XY, Zhang WH. Translation from Python into Dafny Based on Large Language Models. Ruan Jian Xue Bao/Journal of Software, 2026, 37(9): 3598–3614 (in Chinese). http://www.jos.org.cn/1000-9825/7606.htm

Translation from Python into Dafny Based on Large Language Models

DU Yi-Cheng[1,2], LU Yi-Han[1,2], ZHU Xue-Yang[2], ZHANG Wen-Hui[2]

[1](Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou 310024, China)

[2](Key Laboratory of Systems Software (Institute of Software, Chinese Academy of Sciences), Beijing 100190, China)

Abstract: In recent years, code generation based on large language models has been widely applied in software development. However, due to the inherent stochasticity of model outputs, ensuring the correctness of generated code remains a critical challenge. Formal verification provides a rigorous means to guarantee software correctness, but it requires both source code and corresponding requirements to be formalized into the specification language of verification tools. Dafny is a verification-oriented programming language equipped with automated verification support. From this perspective, this study explores the ability of large language models (LLMs) to translate Python code into Dafny. To this end, a prompt-based translation generation method is proposed, which consists of static translation and dynamic repair, together with a program-testing-based evaluation method. The static translation method employs three progressively refined prompting templates-namely, a basic template, a Dafny example-based template, and a Python-to-Dafny contrastive example-based template—to generate translations in a single step. The dynamic repair method iteratively invokes LLMs to refine translations by

* 本文由“形式化方法与应用”专题特约编辑安杰副研究员、顾斌研究员、李建文教授推荐.
收稿时间: 2025-09-08; 修改时间: 2025-10-28, 2025-12-08; 采用时间: 2025-12-16; jos 在线出版时间: 2025-12-24
CNKI 网络首发时间: 2026-03-12

incorporating error messages produced during the previous round of code testing into the prompt. For evaluation, when no existing test suites are available, test cases are automatically generated using LLMs. The translated Dafny programs are then verified using Dafny testing tools in conjunction with the generated test cases to assess translation correctness. Experiments are conducted on HumanEval, MBPP, and LeetCode datasets using representative large language models, including GPT-4o and DeepSeek-V3. Experimental results demonstrate that the contrastive example-based prompting template and the dynamic repair method effectively improve the performance of LLMs on the Python-to-Dafny code translation task. Furthermore, factors influencing translation quality are analyzed based on experimental observations, and the evaluation methodology and criteria are discussed.

Key words: code translation; prompt engineering; formal verification; Dafny

近年来, 大语言模型 (以下简称“大模型”)[1]的兴起引发全社会的广泛关注, 大模型也在社会各行业得到了广泛应用[2,3]. 基于大模型的软件开发[4–7]是大模型研发投入的重点领域之一, 其反映出大模型在逻辑推理能力和代码生成能力[8–10]上的潜力. 然而, 由于大模型生成结果具有一定的随机性[11,12], 如何保障大模型生成代码的正确性成为自动化生成代码流程中的难点. 形式化验证是保障软件正确性的一种有力手段. 验证前需要将代码与相应需求形式化为相关工具的语言, 本文专注于代码的形式化工作.

Dafny[13,14]是一种可验证的编程语言. 一个 Dafny 程序可包含方法实现也可描述需求规约, 使用 Dafny 配套的验证工具可以对程序进行自动验证. 要利用 Dafny 验证代码的正确性, 一方面需要将源代码翻译成 Dafny 代码, 另一方面需要将相应代码的需求形式化为 Dafny 的规约语言, 形成完整的可验证 Dafny 程序. 这两方面的工作都很耗时且需要较深的专业知识. 本文研究如何高效地将主流编程语言 Python 翻译成 Dafny 代码, 以节省人工工作量, 为后续的验证提供支持.

传统的代码翻译工作通过编写语法规则实现两种语言间的转译. 这种方法在可靠性和转译效率上表现较好, 但需要精通两种语言的专家进行大量的前期投入. 此外, 在两种语言间存在某些用法差异导致需要进行代码改写时, 简单的规则式转译方法难以完成.

最近, 借助大模型进行代码翻译成为代码翻译领域新的突破口. 第 1 种利用大模型进行翻译的方法是根据大量同时包含源语言和目标语言的语料数据对基础模型进行训练或微调, 得到针对特定语言进行翻译的目标模型, 如 MIRACLE[15]、TransCoder[16]、TransCoder-ST[17]、TransCoder-IR[18]等. 相比直接使用通用大模型, 这种方法使用模型参数量较小的预训练模型进行微调就能获得较高的准确率, 单次翻译的花费也较小, 但在数据准备和训练工作上的开销相对高昂, 并且在改变翻译的语言种类时需要重新准备数据和训练. 第 2 种方法直接通过提示词工程引导现有的通用大模型对源代码进行翻译得到目标代码, 代表方法有 UniTrans[19]、TRANSAGENT[20]和 LAC2R[21]等. 考虑到 Dafny 语言相关的语料数据体量较小, 本文采用第 2 种方法.

本文旨在探索大模型在将 Python 代码准确翻译成相应的 Dafny 语言代码方面的能力, 并分析 Dafny 的语言特性对转译的影响, 为进一步的验证工作提供支持. 本文主要贡献如下.

(1) 提出基于提示词的静态翻译和动态修复方法. 静态翻译方法考虑 3 种渐进的翻译提示词模板 (基础模板、Dafny 示例模板、对照示例模板), 使用大模型一次性生成翻译; 动态修复方法设计了一类修复提示词模板, 多次调用大模型对翻译进行修复, 每次将上一轮翻译结果的错误信息加入提示词.

(2) 提出基于程序测试的翻译结果评估方法. 为了对翻译结果做出评估, 使用已有的测试用例集或借助大模型生成测试用例集, 在测试用例集上使用 Dafny 测试工具进行测试.

(3) 在 GPT-4o 和 DeepSeek-V3 大模型上使用 HumanEval、MBPP 和基于已有数据集构建的 LeetCode 数据集对所提方法进行实验, 相关实验代码和数据集已发布于 GitHub (https://github.com/Electroplating/Python2Dafny). 实验结果表明, 使用引入对照的 Python 和 Dafny 示例的提示词模板以及使用迭代修复方法能够有效提升大模型在 Python 到 Dafny 翻译任务上的表现. 此外, 我们还分别分析了可能进一步提高翻译生成正确率和评估方法可靠性的方法.

本文第 1 节介绍程序翻译的相关方法和研究现状. 第 2 节介绍所需的 Dafny 相关基础知识. 第 3 节介绍总体研究设计. 第 4 节和第 5 节分别介绍代码翻译和翻译结果评估部分的具体研究设计. 第 6 节通过实验与分析回答

提出的研究问题. 最后总结全文.

1 相关工作

目前对 Dafny 的翻译研究主要集中在将 Dafny 语言程序翻译为其他语言, 如 Dafny 的官方翻译工具支持将 Dafny 翻译为 C#、Java、Python 等语言[14]. 这是由于 Dafny 的语法直接支持的功能较少, 只能对应主流编程语言的一个子集, 使得以 Dafny 为源语言的翻译较为简单而逆向翻译存在困难. 本文工作将为解决那些无法直接对应翻译的问题提供参考.

考虑到语法结构主体的相似性, 主流编程语言之间的互译工作仍然对 Dafny 的翻译研究具有一定的参考价值. 这些工作主要分为两类, 即基于语法规则的转译和基于大模型的翻译.

传统的程序翻译工作使用手动编写的语法规则和静态分析方法进行转译, 代表性工作有 C2Rust[22]、CxGo[23]、DBridge[24]、JavaToCSharp[25]、Sharpen[26]等. 这类工作具有翻译成本低、准确率相对较高的优势, 如在 CodeNet 数据集[27]的测试中, C2Rust 在 C 到 Rust 的翻译中能达到 95.0% 的正确率[28]; 但投入语法规则编写的开发成本极大, 且需要调用在两种语言间存在差异的库函数和 API 时, 转译正确率会显著下降[28]. 考虑到 Dafny 在库函数方面支持远少于 Python, 使用传统方法进行 Python 到 Dafny 的翻译的正确率也会受到库函数调用的制约. 此外, 传统方法翻译得到的代码在可读性上也往往较差, 不利于后续不变式生成工作的进行.

为克服传统方法存在的缺陷, 大模型被广泛投入程序翻译应用当中. Pan 等人[28]在 CodeNet[27]、AVATAR[29]和 EvalPlus[12]等数据集上进行了程序翻译的对比研究工作, 系统比较了在 C、C++、Go、Java、Python 这 5 种主流编程语言间进行互相翻译时, 传统翻译工具、专用大模型和通用大模型各自的表现, 并总结归纳了使用大模型翻译时产生语法和语义错误的几类原因, 揭示了大模型在代码翻译工作上的局限性与改进空间.

根据 Pan 等人[28]的研究, 将大模型应用于程序翻译工作主要有以下两类方法.

一类是使用语料对基础模型进行训练或微调使模型获得在特定语言间翻译的能力, 以 MIRACLE[15]、TransCoder[16]、TransCoder-ST[17]、TransCoder-IR[18]、CMTrans[30]等工作为代表. 相比直接使用参数量大的通用大模型, 这种方法用模型参数量较小的预训练模型进行微调就能获得较高的准确率, 但为了更好的训练效果, 需要在语料的两种语言代码间构建高度的对应关系. 然而, 在实际应用中, 高度对应的代码极为稀缺. 为解决这一问题, MIRACLE[15]结合静态分析和编译手段生成对应的代码; TransCoder-ST[17]结合测试手段形成对应关系; TransCoder-IR[18]借助编译中间表示构建源语言与目标语言代码的对应关系. 对于本文的任务而言, 一方面, 由于 Dafny 相关的语料较为稀少, 使用这种方法的数据准备成本较高; 另一方面, Tao 等人[31]的研究表明, 使用传统的预训练和微调方法在以 Python 为源语言的翻译任务上效果不佳.

另一类方法是通过提示词工程直接引导现有的通用大模型进行翻译工作, 如 UniTrans[19]、TRANSAGENT[20]、TransGraph[32]、InterTrans[33]和 LAC2R[21]等. 这类工作主要受到大模型在代码生成任务[8–10]和代码归纳任务[34,35]上的优秀表现的启发. Macedo 等人[36]的研究进一步展示了选择合适的提示词样式并控制输出格式对大模型生成的翻译结果的正确性的重要作用. 在明确指定翻译任务、不含额外信息的恰当的提示词引导下, 大模型在程序翻译任务上具有可观的基础表现; 在提示词中引入更多有效信息则能够进一步提升大模型的表现. UniTrans[19]在进行翻译前先借助大模型生成一些测试样例, 再通过在提示词中加入预先生成的测试样例以及使用修复提示词这 2 种手段提升大模型的翻译准确率. 本文的工作证明了使用修复提示词在 Python 到 Dafny 的翻译中同样能够在一定程度上提高翻译准确率, 但从模型询问开销的角度考虑, 修复的效率略低. TRANSAGENT[20]通过引入多智能体, 将翻译修复工作细化, 进一步提升了翻译表现. 由于该工作在一定程度上依赖目标语言成熟的语法分析工具, 其在 Dafny 上的表现仍然有待观察.

在对翻译工作的效果进行评估时, 不同语言间固有的语法差异使得翻译任务的难度不尽相同. Tao 等人[31]使用多种大模型在 14 种编程语言上展开研究, 结果表明在各种主流编程语言的互相转译中, 以 Python 为目标语言的翻译任务较容易被大模型完成, 而以 Python 为源语言的翻译结果准确率往往较低. Tao 等人认为这是由于 Python

具有较为宽松的语法要求, 使得 Python 代码能够较好地被大模型生成, 而相应地难以被大模型理解和翻译. 考虑到 Dafny 的语法要求较为严格, 在 Python 到 Dafny 的翻译中, 两种语言间的语法差异将成为重大的挑战.

2 准备知识

本文涉及 Python 与 Dafny 两种编程语言, 由于 Python 的语言特性已被大家所熟知, 我们仅对 Dafny 进行介绍.

Dafny[13,14]是一种具有自动验证能力的命令式编程语言. 在 Dafny 中, 开发者可以在程序中编写定义函数和方法的前置条件和后置条件、循环中的循环不变式、递归和循环的终止性参数等内置规约. 图 1(a) 展示了在一个整数序列中找到最小值的 Dafny 方法的例子. 在这个方法中, 要求执行前成立的条件 (前置条件) 是序列 a 不为空; 在前置条件满足的情况下, 方法执行后须确保成立的条件 (后置条件) 是序列 a 中任意一个值不小于返回值 res. 对于方法中的循环, 在循环执行过程中始终成立的条件 (循环不变式) 是循环变量 i 的取值范围在序列 a 下标范围内和返回值 res 在已遍历的序列部分中最小; 随着循环执行单调不增并且最终能够到达极小值的量 (终止性参数) 是序列 a 长度与循环变量 i 的差值.

Dafny

```
method find_min(a:seq<int>) returns (res:int)
  requires |a|>0 // pre-condition
  ensures forall i | 0<=i<|a| :: res<=a[i] // post-condition
{
  var i:=0;
  res:= a[0];
  while (i<|a|)
    invariant 0<=i<=|a| // loop invariant
    invariant forall j | 0<=j<i :: res<=a[j] // loop invariant
    decreases |a|-i // termination metric
  {
    if (a[i]<res)
    {
      res:=a[i];
    }
    i:=i+1;
  }
}
```

(a) 含规约的Dafny方法示例

Dafny

```
method find_min(a:seq<int>) returns (res:int)


{
  var i:=0;
  res:= a[0];
  while (i<|a|)



  {
    if (a[i]<res)
    {
      res:=a[i];
    }
    i:=i+1;
  }
}
```

(b) 不含规约的Dafny方法示例

图 1 Dafny 示例

在编写完成后, Dafny 验证器可对这些规约进行自动验证. 如果验证未能通过, Dafny 验证器还会列出那些无法被证明的规约. 凭借其自动化的验证机制与有效的反馈信息, Dafny 是一种高效的、对开发者友好的形式化验证工具语言.

Dafny 语言中的内置规约仅用于验证, 在编译生成可执行文件时, 这一部分会被忽略. 因此, 图 1(a) 展示的 Dafny 方法在编译中与图 1(b) 等价. 默认情况下, 为了保证生成程序的正确性, Dafny 编译器会先调用验证器进行验证, 但这一行为可通过编译参数进行修改. 本文实验将使用该方法略过验证相关部分, 只考虑可执行部分的正确性, 以便后续规约补充编写等验证相关工作的开展.

除验证工具外, Dafny 还有配套的测试工具. 其通过生成一个调用所有带有测试标记 ({:test}) 的方法的主方法, 在运行时检测其中所有测试 (expect 语句) 是否通过. 若所有测试均通过, 测试工具会返回测试通过的信息; 否则, 会返回测试不通过的信息和第 1 个未通过测试的位置. 图 2(a) 展示了一个带有测试标记的方法对待测函数 f(·) 进行测试的示例. 图 2(b) 展示了测试全部通过时及有测试未通过时的反馈信息. 本文实验的评估部分即基于此进行.

Dafny

```
method{:test} valid_test(){
    expect f(0)==0;
    expect f(1)==1;
    expect f(2)==2;
    expect f(3)==3;
    expect f(4)==4;
}
```

(a) 测试方法示例

Command line

```
> dafny test pass.dfy --no-verify
Dafny program verifier did not attempt verification
valid_test: PASSED

> dafny test fail.dfy --no-verify
Dafny program verifier did not attempt verification
valid_test: FAILED
        fail.dfy(5,4): expectation violation
[Program halted] fail.dfy(1,0): Test failures occurred: see above.
```

(b) 测试反馈信息示例

图 2 Dafny 测试示例

在不同的版本中, Dafny 的语法发生了较大的变化. 例如, 在 Dafny3 中, 定义只在验证中可见的函数的关键词是 function, 定义在编译中也可见的函数的关键词是 function method; 然而, 在 Dafny4 中, 定义只在验证中可见的函数的关键词改为 ghost function, 使用 function 定义的函数在编译中是可见的[37]. 本文实验中涉及的 Dafny 语法均基于 Dafny4.

3 总体研究框架

本文旨在探究如何引导大模型高效地从 Python 代码生成 Dafny 翻译以及如何有效评估生成的 Dafny 翻译的质量, 总体研究框架如图 3 所示.

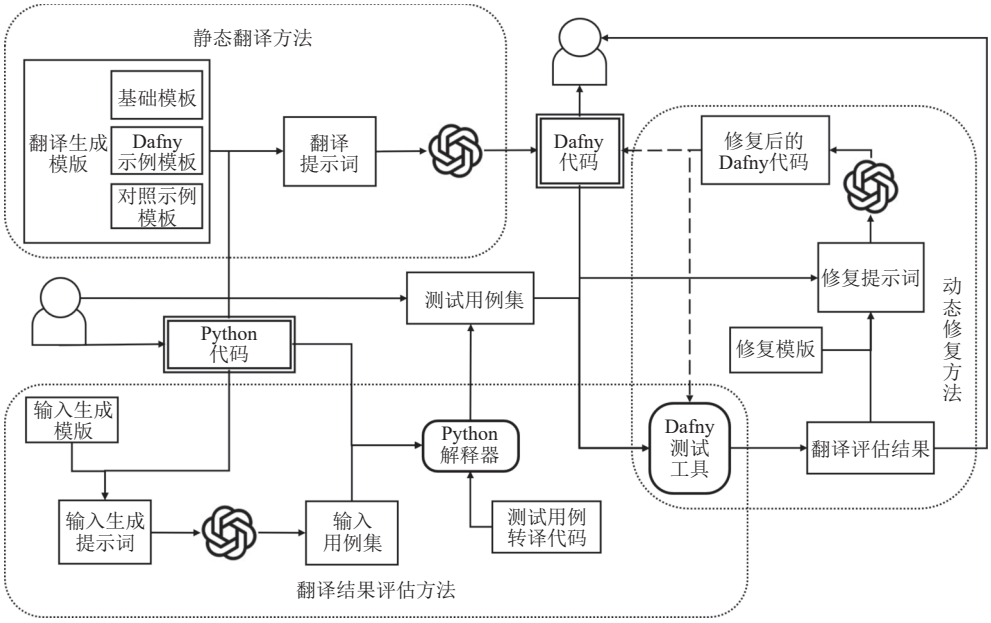


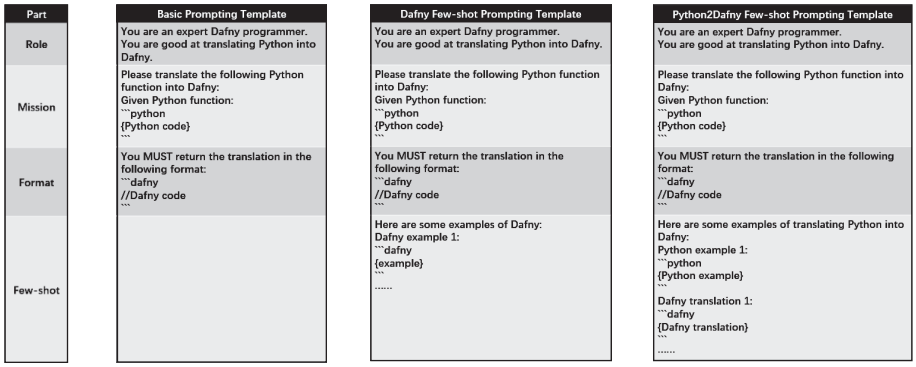
图 3 总体研究框架示意图

本文工作包括代码翻译和翻译结果评估两个部分. 其中, 代码翻译部分有两种方法, 分别是静态翻译方法和动态修复方法. 在静态翻译方法中, 对待翻译的 Python 程序, 可选一种翻译生成模板生成翻译提示词, 再使用大模型一次性生成相应的 Dafny 代码. 翻译得到的 Dafny 代码正确与否通过测试来评估. 若用户提供了测试用例集, 则直接使用 Dafny 测试工具进行测试, 根据测试结果对翻译效果进行评估. 若没有现成的测试用例集, 本文提出一个测试用例生成方法: 使用输入生成模板结合待翻译的 Python 程序引导大模型生成相应的 Python 输入用例集, 再运行 Python 程序生成对应的输出, 进而生成 Dafny 测试用例集. 由静态翻译方法生成的 Dafny 代码若不正确, 还可进一步使用动态修复方法, 迭代地利用测试反馈的错误信息, 使用修复模板, 生成修复提示词, 引导大模型对 Dafny 代码进行修复, 直到生成正确的代码或者达到设定的迭代次数.

4 代码翻译

4.1 静态翻译

为了以简洁的方法和流程取得更好的成果, 我们采用上下文学习方法进行提示词设计, 通过单轮询问大模型获得翻译结果. 上下文学习方法是提示词工程中的一个基础而关键的方法, 它涉及在提示词中为大模型提供上下文信息或具体示例, 使其能够学习现在的任务[38]. 为了研究提示词中不同类型上下文信息的影响, 我们设计了 3 种逐步增强的提示词策略, 分别使用基础模板、Dafny 示例模板和对照示例模板, 如图 4 所示.



(a) 提示词组件 (b) 基础模板 (c) Dafny示例模板 (d) 对照示例模板

图 4 3 种提示词模板示意图

(1) 基础模板

为了评估大模型生成 Dafny 翻译的基础能力, 并识别固有问题, 我们设计了一个基础模板, 如图 4(b) 所示. 该模板包含了角色 (role)、任务 (mission) 和返回格式 (format) 的指令, 没有附加其他信息.

(2) Dafny 示例模板

根据少样本提示的思想[39], 我们猜想在提示词上下文中引入少量 Dafny 示例能够缓解 Dafny 语言代码在大模型语料库中占比较少的问题, 引导大模型理解 Dafny 的语法规则, 从而提升翻译表现. 对应的模板称为 Dafny 示例模板, 如图 4(c) 所示.

(3) 对照示例模板

研究中我们发现大模型容易出现字面翻译现象, 即将 Python 代码直接语法转写为 Dafny 代码而不考虑两种语言间某些用法的差异. 图 5 展示了其中一种情况: 在翻译时, 由于 Dafny 中不存在 Python 自带的 sum 函数, 直接使用缺少定义的函数会产生语法错误, 需要进行如图虚线框内的补充定义. 为了解决这一问题, 我们转而在提示词上下文中引入构成对照的 Python 和 Dafny 示例, 以引导大模型构建两种语言间的逻辑对应关系, 避免字面翻译现象. 为控制变量, 两种含示例的提示词模板中, Dafny 示例是相同的. 对应的模板称为对照示例模板, 如图 4(d) 所示.

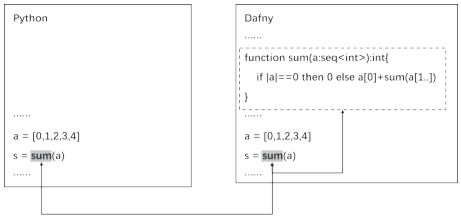


图 5 Python 到 Dafny 翻译的“字面翻译”现象示例

4.2 动态修复

由静态翻译方法生成的 Dafny 代码若不正确, 还可进一步使用动态修复方法, 迭代地利用测试反馈的错误信息, 使用修复模板根据最后一次测试的错误信息生成修复提示词, 引导大模型对 Dafny 代码进行修复, 直到生成正确的代码或者达到设定的迭代次数. 考虑到在语义错误和代码块级语法错误中, 错误信息位置和需要修复的位置不一致, 我们允许大模型进行任意规模的改写修复, 只收集修复后生成的完整代码. 为了对修复进行剪枝, 由无语法错误的代码修复得到的有语法错误的代码将被丢弃. 在本研究中, 错误类型共有语法错误、终止性错误 (超时)、语义错误 (未通过 expect 测试) 这 3 类, 不同的错误类型对应不同的任务描述, 因此需要不同的修复模板. 以语法错误为例的模板如图 6(a) 所示, 其中可填入语法错误的位置、类型等具体信息. 对于终止性错误, 由于无法收集详

细错误信息, 修复模板中的错误信息部分转为提示大模型注意循环和递归是否正确终止; 对于语义错误, 修复模板中的错误信息部分为未通过的测试内容.

Syntax Error Fixing Template

```
You are an expert Dafny programmer.You are good at fixing syntax errors in Dafny.

Please Fix the syntax errors in the following Dafny code.
Given Dafny code:
```dafny
{Error Dafny code}
```

You MUST return the fixed Dafny code in the following format:
```dafny
// Dafny code
```

Here are some error messages from Dafny resolver:
Error lineno 1: {lineno of Error 1}
Error pos 1: {position of Error 1}
Error content 1: {content of Error 1}
Error type 1: {type of Error 1}
...
```

(a) 修复语法错误的修复模板

Syntax Error Fixing Prompt

```
...
Here are some error messages from Dafny resolver:
Error lineno 1: 8
Error pos 1: 8
Error content 1:          b := a % b;
Error type 1: Error: LHS of assignment must denote a mutable variable
Error lineno 2: 8
Error pos 2: 8
Error content 2:          a := temp;
Error type 2: Error: LHS of assignment must denote a mutable variable
```

(b) 图6(c)对应填入的错误信息

Dafny

```
method greatest_common_divisor(a: int, b: int)
returns (res: int) {
    while b != 0 {
        var temp := b;
        b := a % b;  //形参不可变（non-mutable）
        a := temp;
    }
    return a;
}
```

(c) 语法错误的Dafny代码片段示例

Dafny

```
method greatest_common_divisor(a: int, b: int)
returns (res: int) {
    var a := a;
    var b := b; //得到可变（mutable）的变量
    while b != 0 {
        var temp := b;
        b := a % b;
        a := temp;
    }
    res := a;
}
```

(d) 图6(c)修复后通过测试的Dafny代码片段

图 6　修复语法错误的示例

图 6 展示了一个动态修复的例子. 图 6(c) 是待修复的 Dafny 代码, 其中存在语法错误. 对测试反馈的错误信息进行提取填入修复模板, 生成的修复提示词的错误信息部分如图 6(b) 所示. 使用修复提示词引导大模型进行修复后, 大模型生成了如图 6(d) 所示的代码. 该 Dafny 代码可以通过测试, 因此不再进行迭代, 动态修复过程结束. 该次修复中, 修复位置与错误信息位置不一致, 印证了我们不限制大模型的改写修复方式、只收集修复后完整代码的必要性.

5　翻译结果评估

本文工作不仅提出从 Python 代码到 Dafny 代码的翻译方法, 还提出翻译结果评估方法, 以检查生成的 Dafny 代码是否保持源 Python 代码的功能. 理想的评估方法是为双方建立严格的形式语义并在语义层面进行数学验证, 但该方法难度太大, 目前代码释译工作也无法做到. 因此我们采用较为可行的方案, 即测试的方法. 在测试时, 我们以函数功能的一致性作为正确性评估的标准, 即只要翻译前后的函数在同一组输入参数上产生相同的输出, 就可以认定翻译是正确的. 我们将测试用的输入用例集和在 Python 中运行产生的相应输出用例集合称为测试用例集.

对于已有测试用例集的情况, 运行 Dafny 测试工具即可对生成的 Dafny 翻译代码进行测试, 测试将检测出翻译中的语法错误 (表现为解释器报错) 或语义错误 (表现为测试不通过) 或终止性错误 (表现为调用测试工具的进程超时). 如果出现前两类问题, 对于测试结果中的报错信息, 我们使用正则表达式提取其中的错误位置与错误类型, 再通过错误位置信息提取错误的代码部分, 将这些信息记录或用于动态修复方法. 终止性问题没有显式的具体

错误位置信息, 因此我们转而提示检查循环或递归边界是否存在问题.

对于需要生成测试用例集的情况, 我们使用提示词引导大模型生成符合原 Python 函数的输入参数集[19], 在 Python 中运行得到相应的输出参数集, 构成测试用例集, 再通过语法规则转写为相应的 Dafny 形式. 由于大模型不能对函数行为进行详细的分析, 在生成输入参数时可能出现输入不合法的情况, 因此需要多次迭代, 尽可能地让大模型生成合理的输入参数. 相应的输入生成模板如图 7 所示.

Inputs Generating Template

```
You are an expert Python programmer.You are good at generating test inputs for Python functions.

Please generate ten groups of differentiated valid inputs for the following Python function.
Given Python function:
```python
{Python code}
```

You MUST return the inputs in the following format without any comments:
```python
input1=(parameter1,parameter2,...)
input2=(parameter1,parameter2,...)
...
input10=(parameter1,parameter2,...)
```
```

图 7 输入生成模板示意图

6 实验及结果分析

本节我们将详细说明包括大模型种类选择、温度选择、实验数据集构建在内的实验设置, 随后对实验结果进行分析. 我们希望通过实验回答以下问题.

问题 1: 大模型在 Python 到 Dafny 代码的翻译任务上的基准表现 (使用基础提示词的表现) 如何?

问题 2: 在提示词上下文中引入 Dafny 示例与引入对照示例分别能对大模型的翻译表现带来怎样的影响?

问题 3: 在设计提示词上下文中的示例时, 如何避免幻觉效应的影响?

问题 4: 动态修复方法会如何提升大模型的翻译表现? 我们应如何选择是否迭代进行动态修复和迭代轮数?

问题 5: Dafny 的语言特性对翻译通过率的影响程度如何? 大模型在 Python 到 Dafny 的翻译任务上还有怎样的提升空间?

问题 6: 如何保证当前的翻译正确性评估方法的可靠性? 在当前的翻译任务中, 还有哪些合适的评估标准? 采用这些评估标准又需要怎样的评估方法?

6.1 实验设置

6.1.1 大模型与温度选择

本实验使用的通用大模型是 GPT-4o 和 DeepSeek-V3. GPT-4o 和 DeepSeek-V3 是当前较为典型、使用广泛的通用大模型, 能够反映通用大模型在翻译任务上的普遍表现. 为了更加客观地评估大模型的翻译能力, 实验中直接调用官方大模型而不是本地部署蒸馏模型. 在调用大模型时, 温度参数对实验结果影响显著. 现有的相关工作往往在特定的温度参数下进行实验, 如 UniTrans[19]将实验温度设置为 0.8. 本实验对 GPT-4o 在温度设置为 0.3、0.5、0.7 时分别进行实验以探究较为合适的实验温度并获得可靠的实验结果, DeepSeek-V3 使用官方文档推荐的 0.0 温度设置进行实验[40].

6.1.2 实验数据集构建

(1) HumanEval 数据集

HumanEval 数据集[10]是一个用于测试根据需求生成的 Python 代码是否正确的数据集. 本实验使用 Human-

Eval 数据集的手写 ground-truth 部分作为 Python 源代码数据集, 并将其自带的 Python 形式测试用例自动转写为对应的 Dafny 测试用例进行测试. HumanEval 数据集中共有 163 个问题, 其中 4 个问题使用 Python 随机函数进行实时测试, 无法转写为 Dafny. 我们将其余的 159 个问题作为全体问题集.

(2) MBPP 数据集

MBPP 数据集 (mostly basic Python problems dataset)[41]包含了近 1 000 个常见的 Python 基础编程问题, 同样广泛应用于代码生成测试中. 本实验选用其中经过人工验证的 sanitized-mbpp 部分, 对其进行与 HumanEval 数据集相同的处理. 原数据集中共有 427 个问题, 其中 14 个问题在测试时需要调用 Python 特有的库函数, 无法转写为 Dafny. 我们将其余的 413 个问题作为全体问题集.

(3) LeetCode 数据集

为了验证本实验的方法在复杂算法问题上的表现, 减少 Python 和 Dafny 间语句级语法差异的影响, 评估翻译结果和源代码在程序结构上的一致性, 本文基于 LeetCode 构建了一个数据集. 我们选取了 20 个在 LeetCode 上分类为 Hard 的问题, 以保证问题的复杂性; 在选取它们对应的通过 LeetCode 测试的 Python 代码[42]时, 尽可能地选择含 Python 语法特性较少的代码或在不影响语义的情况下进行改写. 由于 LeetCode 不支持直接评测 Dafny 代码, 我们获取了每个问题的至少 50 组测试用例 (若 LeetCode 上不足 50 组则全部使用), 并转写为 Dafny 形式在本地进行评测. 由于 HumanEval 数据集和 MBPP 数据集中的 Python 代码主要是较为基础的函数, 该数据集可以在复杂算法问题的翻译结果评估上进行有效的补充.

6.2 实验结果与分析

首先, 我们在 GPT-4o 和 DeepSeek-V3 两个大模型上使用 HumanEval 数据集进行多次实验, 得到了基础提示词、Dafny 示例提示词、对照示例提示词这 3 种提示词各自的翻译表现 (表 1). 其中, syntax@k (semantic@k) 表示实验 k 次时至少一次编译通过 (测试通过) 的任务通过率, 即编译通过率 (测试通过率).

表 1　3 种提示词在 HumanEval 数据集上采用不同实验温度生成翻译的结果 (%)

提示词种类	k	GPT-4o						DeepSeek-V3	
		0.3		0.5		0.7		0.0	
		syntax@k	semantic@k	syntax@k	semantic@k	syntax@k	semantic@k	syntax@k	semantic@k
基础提示词	1	11.32	8.18	8.81	7.55	11.95	10.69	17.61	16.35
	2	14.47	10.06	16.98	15.72	16.98	15.09	20.75	19.50
	3	16.98	13.21	23.27	22.01	20.13	19.50	25.16	22.01
	4	22.01	18.24	24.53	23.27	23.27	23.27	27.04	23.27
	5	<u>25.79</u>	21.38	25.16	23.90	<u>25.79</u>	**25.16**	**27.67**	<u>23.90</u>
Dafny示例提示词	1	42.77	38.99	42.14	36.48	38.36	33.96	60.38	55.97
	2	57.23	52.83	52.20	45.91	49.69	44.03	64.78	60.38
	3	60.38	55.35	56.60	50.94	55.35	52.83	66.67	61.64
	4	63.52	59.12	61.01	57.23	57.23	54.09	67.92	62.89
	5	<u>65.41</u>	<u>61.01</u>	<u>65.41</u>	60.38	59.12	56.60	**69.18**	**64.15**
对照示例提示词	1	47.17	41.51	50.31	45.28	49.06	45.28	65.41	59.75
	2	59.75	53.46	57.86	52.20	57.23	53.46	72.33	65.41
	3	61.01	54.72	62.26	57.86	60.38	55.97	74.21	66.67
	4	63.52	57.23	66.04	61.01	64.15	59.12	74.84	67.92
	5	65.41	58.49	<u>68.55</u>	<u>63.52</u>	66.67	61.64	**76.73**	**69.18**

注: 加粗表示同类数据中的最高者, 下划线表示次高者

我们可以根据这些结果定量分析回答问题 1 和问题 2.

问题 1: 大模型在 Python 到 Dafny 的翻译任务上的基准表现 (使用基础提示词的表现) 如何?

根据表 1 数据, 使用基础提示词时, 大模型生成翻译的编译通过率 (syntax@k) 最高为 27.67%, 测试通过率 (semantic@k) 最高为 25.16%. 其中, DeepSeek-V3 模型拥有最高的编译通过率, 且编译通过率和测试通过率均随 k

上升较快. 对 GPT-4o 模型而言, 在温度为 0.7 时拥有最高的测试通过率, 且该模型的表现总体上随着温度的上升略有提高, 反映出在不确定性提升时其更有可能进行正确的翻译.

问题 2: 在提示词上下文中引入 Dafny 示例与引入对照示例分别能对大模型的翻译表现带来怎样的影响?

根据表 1 数据, 使用 Dafny 示例提示词后, 大模型生成翻译的编译通过率最高为 69.18%, 测试通过率最高为 64.15%. 控制模型和温度不变的情况下进行对比, Dafny 示例提示词相比基础提示词的编译通过率和测试通过率提升在 23.27%–44.03% 不等. 这说明在提示词上下文中引入代码示例是提升大模型在代码翻译任务中表现的有效手段.

类似的, 使用对照示例提示词时, 与使用 Dafny 示例提示词相比, 除了 GPT-4o 模型在温度为 0.3 的情况下, 编译通过率和测试通过率有 3.14%–11.32% 的提升, 且在 k 较小时提升幅度较为明显, 说明引入对照的双语示例能够进一步引导大模型进行正确的代码翻译, 在提升翻译效率 (减少翻译所需轮次) 上作用较为显著. 参考 MIRACLE[15] 等使用对照示例进行训练或微调的工作, 我们推断在提示词中使用对照示例也是通过引导大模型构建两种语言间的对应关系来获取翻译效果提升的. GPT-4o 模型在温度为 0.3 时出现了负提升情况, 推测是因此大模型未能充分利用提示词上下文中的代码示例, 增加的代码示例内容反而对大模型产生了干扰.

在根据大模型的翻译结果不断修改代码示例设计的过程中, 我们注意到大模型的幻觉效应带来的危害, 接下来我们将用一个实例来说明其对设计代码示例的影响, 即回答问题 3.

问题 3: 在设计提示词上下文中的示例时, 如何避免幻觉效应的影响?

图 8 展示了某版对照的双语代码示例的其中一部分. 这组示例的目的是向大模型展示 Dafny 中向下取整函数 Floor 的正确用法. 然而, 大模型可能会错误地将其中的 Dafny 示例部分视为在翻译中可直接使用的上下文, 在其生成的翻译中直接对这些函数进行调用, 产生的错误翻译结果如图 8 所示. 在使用不含 Python 代码示例、仅含 Dafny 代码示例的提示词时, 产生的翻译中出现了同样现象, 排除了 Python 示例的干扰, 印证了 Dafny 示例对大模型的影响.

Python示例	Dafny示例	错误的Dafny翻译结果

`def Floor(s:float)->float:`
`  return floor(s)`
	`function Floor(s:real):real{`
	`  s.Floor as real`
	`}`
		`function truncate_number(number: real): real{`
		`  number - Floor(number)`
		`}`

图 8　产生幻觉效应的对照代码示例及错误翻译结果

为解决这一问题, 容易想到的方法是通过恰当设计代码示例避免其被大模型直接调用. 图 9 展示了一种经测试证实有效的修改方法及其得到的正确翻译结果. 原示例函数 (Floor 函数) 是对两种语言中取整功能实现方式的直接封装. 我们将其修改为 floating_division 函数, 其功能为计算两个浮点数的商的取整, 并在函数实现上体现了两种语言间取整功能实现方式的对应. 因此, 可以得出结论: 在设计提示词上下文中的代码示例时, 将示例函数设计为某种特殊用法的直接封装是应当避免的; 相反, 将函数功能设计得较为复杂或少见, 使其无法被直接调用, 并在函数实现中体现需要在两种语言间对照的语法特性, 能够正确地引导大模型理解其中的对应关系, 同时避免幻觉效应影响翻译结果.

Python示例	Dafny示例	正确的Dafny翻译结果

`def floating_division`
`(a:float,b:float)->float:`
`  return floor(a / b)`
	`function floating_division`
	`(a:real,b:real):real{`
	`  (a/b).Floor as real`
	`}`
		`function truncate_number(number: real): real{`
		`  number - (number.Floor as real)`
		`}`

图 9　修改后的对照代码示例及正确翻译结果

问题 4: 动态修复方法会如何提升大模型的翻译表现? 我们应如何选择是否迭代进行动态修复和迭代轮数?

为探究问题 4, 我们在表 1 的基础上继续进行实验. 通过表 1 选取表现最佳的两种情况: 使用对照示例提示词, 分别在温度为 0.5 的 GPT-4o 模型上与在温度为 0.0 的 DeepSeek-V3 模型上进一步迭代使用动态修复方法进行测试, 得到的对比结果如表 2 所示.

表 2 在 HumanEval 数据集上使用迭代修复方法的正确性与资源消耗结果

迭代修复种类	k	GPT-4o@0.5			DeepSeek-V3@0.0		
		syntax@k (%)	semantic@k (%)	token 平均使用量 (k)	syntax@k (%)	semantic@k (%)	token 平均使用量 (k)
不进行迭代修复 (静态翻译)	1	50.31	45.28	533.0	65.41	59.75	571.7
	2	57.86	52.20		72.33	65.41	
	3	62.26	57.86		74.21	66.67	
	4	66.04	61.01		74.84	67.92	
	5	68.55	63.52		76.73	69.18	
3轮迭代修复 (静态翻译+2轮动态修复)	1	52.20	46.54	1 033.6	70.44	62.26	962.6
	2	63.52	55.35		74.84	66.67	
	3	69.18	61.64		76.10	69.81	
	4	71.70	64.78		77.36	71.07	
	5	72.96	66.67		77.36	71.70	
5轮迭代修复 (静态翻译+4轮动态修复)	1	66.04	57.86	1 412.8	72.33	63.52	1 290.9
	2	72.33	64.15		76.73	69.81	
	3	72.96	65.41		76.73	71.07	
	4	72.96	66.67		78.62	71.07	
	5	74.21	68.55		79.25	71.70	

从表 2 中可以看出, 动态修复方法能够同时有效提升翻译结果的编译通过率和测试通过率, 这说明确实存在大模型首次翻译错误的代码在迭代过程中被修复. 进一步分析, 5 轮迭代修复相比 3 轮迭代修复的提升更快, syntax@5 和 semantic@5 也有约 2% 的提升. 其中, DeepSeek-V3 的 semantic@5 在 3 轮迭代修复和 5 轮迭代修复中相同. 我们检查并对比了修复后仍未通过测试的代码, 发现大模型在多轮修复中始终没有找到正确的修复方法, 因此可以认为已经达到了当前修复方法的通过率上限.

我们再以 token 使用量为指标分析迭代使用动态修复方法的资源消耗. 表 2 数据说明, 3 轮迭代修复的 token 平均使用量近似为不使用迭代修复的 1.6–1.9 倍, 5 轮迭代修复的 token 平均使用量则约为 2.2–2.6 倍. 其中, 由于动态修复仅需对未通过正确性测试的代码进行修复, 初次生成通过率较高的 DeepSeek-V3 的 token 使用量显著少于 GPT-4o. 在控制 token 使用量近似相同时, 使用迭代修复方法的通过率普遍低于多次使用静态翻译. 因此, 在大模型资源紧张时, 使用静态翻译方法多次生成能够以较小的开销获得可观的编译通过率和测试通过率, 避免在成功率过低的修复中消耗过多资源; 而在以通过率为导向时, 迭代使用动态修复方法能够解决部分大模型无法一次生成正确翻译的复杂翻译任务, 从而有效提升翻译的编译通过率和测试通过率的上限.

为了对动态修复方法进行更为客观的评价, 我们对 5 轮迭代修复得到的所有代码进行检查. 图 10 展示了一个经迭代修复后通过的例子, 其中对错误修复的注释由大模型自发生成. 我们发现, 多轮迭代使用动态修复方法对首次翻译的错误是简单语法错误的情况有显著修复效果, 但对语义错误和整体结构错误的修复效果较差. 对于这些问题, 直接引导大模型重新进行静态翻译可能得到易于修复或可通过测试的代码. 因此, 迭代轮数不宜过多; 当迭代轮数达到一个阈值时, 可以认为这份代码不能被修复转而重新进行静态翻译. 结合本文的测试结果, 5 轮是一个合适的迭代轮数上限.

为了验证该方法在不同数据集上的泛化性, 我们在 MBPP 数据集上使用同样的实验设置进行实验, 实验结果如表 3 所示. 将表 3 中的数据与表 2 中的对应数据进行对比, 我们发现, 本实验所用方法在两个数据集上的表现接近, 相同情况下的编译通过率和测试通过率差距均在 4% 以内. 这说明该方法具有较好的泛化性, 在函数级 Python 到 Dafny 代码翻译上具有稳定的表现.

问题 5: Dafny 的语言特性对翻译通过率的影响程度如何? 大模型在 Python 到 Dafny 的翻译任务上还有怎样的提升空间?

在本实验中, 由表 2 和表 3 可知, 在 HumanEval 和 MBPP 数据集上, Dafny 翻译结果的 syntax@5 和 semantic@5 最高分别为 79.25% 与 74.09%, syntax@1 和 semantic@1 最高分别为 72.33% 与 64.16%. 在不限定任务相关语言的

参考中, UniTrans 在不同语言翻译任务上的平均 semantic@1 为 87.92%[19]; TRANSAGENT 在不同语言翻译任务上的平均 semantic@1 为 91.6%[20]; 而 LAC2R 在 C 到 Rust 翻译任务上仅使用 GPT-4o 时的正确率 (对应 semantic@1) 为 53.43%[21], 略低于本文使用 GPT-4o 时的 57.86% 和 57.63%. Tao 等人[31]使用多种大模型在 14 种编程语言上展开的研究表明, 在借助大模型进行的当前主流编程语言的互相翻译中, 同样条件下, 以 Python 作为目标语言的翻译正确率是所有目标语言中最高的, 而以 Python 作为源语言的翻译正确率是所有源语言中最低的. 基于此结论, 我们仅将以 Python 作为源语言的翻译正确率作为参考, 选取该研究中综合表现最好的 CodeLlama-13B 模型在基于 HumanEval 数据集的 Python 翻译任务上的表现与本实验的结果作为比较, 结果如图 11 所示. 从图中可以看出, 以 Dafny 为目的语言的翻译正确率略低于平均水平, 接近 Rust 等具有特殊限制与语法要求的语言. 这与 Dafny 语法限制较多、语法要求较为严格的特点一致.

Dafny

```
method factorize(n: int) returns (fact: seq<int>) {
    fact := [];
    var i := 2;
    var num := n; // Create a new variable because method parameters are immutable
    while i <= num {
        if num % i == 0 {
            fact := fact + [i];
            num := num / i;
        } else {
            i := i + 1;
        }
    }
    if num > 1 {
        fact := fact + [num];
    }
}
```

图 10　经迭代修复后测试通过的代码翻译示例

表 3　在 MBPP 数据集上迭代修复方法对正确性的影响 (%)

迭代修复种类	k	GPT-4o@0.5		DeepSeek-V3@0.0	
		syntax@k	semantic@k	syntax@k	semantic@k
不进行迭代修复	1	50.61	44.31	68.04	60.77
	2	64.65	57.38	74.09	67.55
	3	67.07	59.81	76.27	70.46
	4	67.80	61.50	77.48	71.91
	5	70.70	64.41	77.48	72.88
5轮迭代修复 (静态翻译+4轮动态修复)	1	67.31	57.63	70.22	64.16
	2	70.70	62.95	76.03	69.98
	3	71.43	63.44	77.97	72.15
	4	71.91	64.65	78.69	73.37
	5	72.15	65.62	78.93	74.09

为分析在 Python 到 Dafny 翻译任务上测试通过率较低的具体原因并回答问题 5, 我们对在各组测试中均没能通过的翻译任务进行了分析与归类. 这些翻译任务的失败源于 Python 和 Dafny 的语法间存在差异, 主要可以归为以下几类.

(1) Dafny 的严格类型要求

在 Dafny 中, 一个变量需要在定义时指定类型或者通过赋值推断类型, 此后不能进行更改. 因此, 所有在 Python 源代码中可能对应多个类型的变量在 Dafny 目标代码中需要通过复杂的归纳类型指明其所有可能对应的类型再进行定义与调用, 否则无法通过编译. 具体的问题包含以下几类.

1) 函数参数或返回值具有多种类型的函数, 如 HumanEval/12[10]. 需要注意的是, Dafny 不支持函数重载, 函数间通过函数名唯一区分, 因此不能使用为每个类型分别编写一个同名函数实现的方法来解决这一问题.

2) 以 None 为初始值的过程变量, 如 HumanEval/20[10]. 由于 Dafny 中没有 None 类型, 初始值需要另外指定, 这需要大模型具有阅读代码并考虑合法的初始值取值的能力.

3) 内含多种类型的列表, 如 HumanEval/151[10]. Dafny 不支持这类列表, 需要将多种类型转为统一的归纳类型再定义该归纳类型的列表.

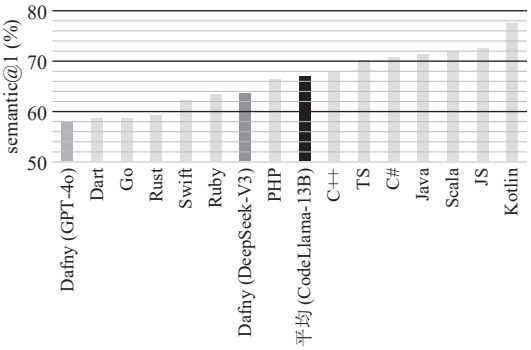


图 11 以 Python 为源语言的翻译测试通过率 (semantic@1) 比较

在实际工业生产中, Python 变量往往具有复杂的类型. 因此, 如果能够引导大模型掌握归纳类型的使用, 将为解决这一系列复杂任务带来希望, 进而提高翻译结果在实际场景中的可靠性.

(2) Dafny 中缺少 Python 内置行为和内建操作

许多 Python 常见的内置行为和内建操作在 Dafny 中并不存在. 例如, 在 Dafny 中, 字符串的本质是字符序列, 而序列间<和≤运算符仅用于判断是否是严格前缀和前缀, 因此字符串间根据字典序比较大小的方法在 Dafny 中需要补充定义. 另一个例子是 Python 中常用的序列切片支持设置步长, 而 Dafny 中序列切片的步长只能为 1.

(3) Dafny 中缺少 Python 库函数所对应的函数

Python 丰富的库函数在 Dafny 中同样缺少对应. Dafny 语言几乎不含内置函数, 在标准库中也只包含了相当有限的常用函数, 如 Join、Split 等. 绝大部分 Python 中的库函数在 Dafny 中使用前需要补充定义. 考虑到 Python 还具有大量的第三方库函数, 且这些库函数的功能和具体实现未必能由大模型推理得到, 该类问题将成为翻译问题中的重大挑战. 本文使用提示词上下文中对照的双语示例成功引导大模型在使用前补充定义了较为简单的常用函数, 如求和函数 Sum 等; 然而, 当大模型需要补充定义较为复杂或不太常见的函数时, 由于缺少相关语料和理解, 难以对函数实现进行完整补充. 因此, 可能需要训练专家模型辅助解决这一问题.

为了尽可能排除上述语法差异的干扰, 我们构建了 LeetCode 数据集 (构建方法见第 6.1.2 节), 并在该数据集上使用迭代修复方法进行实验. LeetCode 数据集基于复杂算法问题的解决代码构建, 可在一定程度上评估翻译方法在复杂情况下的表现, 能够有效反映翻译结果的程序结构正确性, 实验结果如表 4 所示.

表 4 在 LeetCode 数据集上使用迭代修复方法进行实验的结果 (%)

k	GPT-4o@0.5		DeepSeek-V3@0.0	
	syntax@k	semantic@k	syntax@k	semantic@k
1	70	50	95	60
2	80	50	100	60
3	90	55	100	65
4	90	55	100	65
5	90	55	100	65

从表 4 的结果中可以看出, 在排除两种语言间语法差异的干扰后, syntax@k 保持在 70% 以上, 说明本文方法能够有效保证翻译结果的程序结构正确性, 也进一步验证了两种语言间的语法差异对翻译通过率的不利影响. 对于语义错误的情况, 我们分析错误信息, 发现错误集中在大模型对循环边界的判断和理解存在问题.

问题 6: 如何保证当前的翻译正确性评估方法的可靠性? 在当前的翻译任务中, 还有哪些合适的评估标准? 采用这些评估标准又需要怎样的评估方法?

当前的翻译正确性评估方法基于程序测试. 保证程序测试方法的可靠性需要对测试数据集进行增强. 一方面, 需要增大测试数据量. 这可以通过结合大模型生成大量输入数据实现, 但由于原 Python 代码未必有对输入合法性的检查, 在生成输出数据的过程中可能出现由不合法的输入数据导致的未定义行为. 另一方面, 需要增强测试数据的多样性, 使其覆盖尽可能多的等价类. 其中, 为了评估利用大模型生成测试用例集的效果, 我们使用原测试用例集得到的测试结果与其进行对比. 在 HumanEval 数据集的 159 个翻译任务中: 有 28 个翻译任务存在大模型生成空测试用例集的问题, 即大模型生成的输入数据不合法导致无法生成输出数据; 有 3 个翻译任务在原测试用例集上通过, 但在大模型生成的空测试用例集上出现了类型检查问题, 经检查发现是大模型生成的测试用例与 Dafny 翻译的类型规定不符; 有 1 个翻译任务在原测试用例集上出现了类型检查问题, 但在大模型生成的测试用例集上通过, 说明大模型没有考虑测试用例可能的不同类型; 在其他 127 个翻译任务上, 测试结果类型相同, 可以认为这些测试用例是可靠的. 根据大模型生成测试用例集的上述表现, 我们认为, 对大模型生成输入数据的类型检查与合法性检查是利用大模型生成测试用例集需要解决的关键问题.

在翻译正确性评估中, 除本文使用的函数功能一致性外, 还有 EmAcc[19]、CodeBLEU[43]等关注函数内部结构相似度的标准. 这是因为在函数功能一致时, 函数内部结构和语义可能并不相同. 例如, 图 10 中通过测试的 Dafny 代码和对应的 Python 源代码 (图 12) 具有不同的循环边界条件, 两个函数存在内部语义的差别, 但同样实现了分解质因数的函数功能. 而在当前的翻译任务中, 对函数内部结构的相似度进行评估时, 考虑到两种语言间的语法差异可能带来不可避免的补写或改写, 翻译前后的逻辑结构相似度比语法结构相似度更适合作为评估标准.

Python

```
from typing import List

def factorize(n: int) -> List[int]:
    import math
    fact = []
    i = 2
    while i <= int(math.sqrt(n) + 1)://在图10中,这行在语义上被修改成了while i<=n,但由于质因数分解的性质,至多会有一个n的因子i满足int(math.sqrt(n)+1)<i<=n,这个因子会在if n > 1处加入列表,函数行为没有变化
        if n % i == 0:
            fact.append(i)
            n //= i
        else:
            i += 1
    if n > 1:
        fact.append(n)
    return fact
```

图 12　图 10 中的 Dafny 代码对应的 Python 源代码

为评估逻辑结构的相似度, 可结合程序分析手段对关键变量设置中间检查点, 通过测试手段判断各检查点的变量行为在翻译前后是否相同. 例如, 在图 10 和图 12 对应的代码中, 当 n 有一个较大的质因数时, 图 10 的 Dafny 翻译会在循环内将该质因数加入 fact 列表, 而图 12 的 Python 源代码则在循环外将该质因数加入 fact 列表, 因此可以通过在循环结束时对 fact 列表进行中间检查发现函数行为的变化.

7 总　结

本文提出了借助提示词工程将 Python 语言代码翻译为 Dafny 语言的方法, 及使用测试用例集对得到的翻译结果进行评估的方法. 实验表明, 在 HumanEval 和 MBPP 数据集上, 使用引入对照的 Python 和 Dafny 示例的提示词和多轮动态修复方法生成的 Dafny 程序的表现最好, 最高可达 79.25% 的编译通过率和 74.09% 的测试通过率. 在示例设计方面, 本文发现了对翻译结果产生不利影响的大模型幻觉现象并提出避免这一现象的示例设计方式.

对于那些未能通过的翻译任务, 本文根据翻译结果整理出未能正确翻译的原因, 分析了 Python 代码到 Dafny 代码翻译中的困难与可能进一步提升翻译效果的方法, 并通过构建 LeetCode 数据集并在该数据集上进行实验印证了语法差异对翻译工作造成的不利影响. 为进一步保证当前正确性评估方法的可靠性, 本文提出了借助大模型生成测试用例集的方法, 并分析了借助大模型生成的测试用例集和已有的测试用例集间的差异. 本文还指出逻辑结构相似度是该翻译任务的另一种合适的评估标准, 并给出了该评估标准对应的评估方法.

本文对大模型将 Python 代码翻译为 Dafny 代码方面的能力进行了初步探索, 还有诸多工作亟待研究. 例如, 在代码翻译方面, 对于以 Python 为源语言进行翻译时存在的共性问题, 如多类型变量、复杂数据结构、功能和实现不明确的第三方库函数等在工业级复杂场景中可能出现的问题, 本文所提出的方法目前对复杂任务的翻译确实还难以达到较高的正确率, 我们将在进一步工作中尝试解决这些问题并扩展该方法在实际工业生产中的适用范围; 在翻译结果评估方面, 本文仅考虑了测试方法, 利用程序分析、验证等其他手段的方法还有待探索. 本文工作只是形成完整的可验证 Dafny 程序的一部分, 由于篇幅限制, 我们在文献 [44] 中介绍了将自然语言描述的需求形式化为 Dafny 规约语言的工作.

References

[1] Raiaan MAK, Mukta MSH, Fatema K, Fahad NM, Sakib S, Mim MMJ, Ahmad J, Ali ME, Azam S. A review on large language models: Architectures, applications, taxonomies, open issues and challenges. IEEE Access, 2024, 12: 26839–26874. [doi: 10.1109/ACCESS.2024.3365742]

[2] Chang YP, Wang X, Wang JD, Wu Y, Yang LY, Zhu KJ, Chen H, Yi XY, Wang CX, Wang YD, Ye W, Zhang Y, Chang Y, Yu PS, Yang Q, Xie X. A survey on evaluation of large language models. ACM Trans. on Intelligent Systems and Technology, 2024, 15(3): 39. [doi: 10.1145/3641289]

[3] Myers D, Mohawesh R, Chellaboina VI, Sathvik AL, Venkatesh P, Ho YH, Henshaw H, Alhawawreh M, Berdik D, Jararweh Y. Foundation and large language models: Fundamentals, challenges, opportunities, and social impacts. Cluster Computing, 2024, 27(1): 1–26. [doi: 10.1007/s10586-023-04203-7]

[4] Sporsem T, Ulfsnes R. Towards requirements engineering for RAG systems. arXiv:2505.07553, 2025.

[5] Ogenrwot D, Businge J. PatchTrack: Analyzing ChatGPT's impact on software patch decision-making in pull requests. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: ACM, 2024. 2480–2481. [doi: 10.1145/3691620.3695338]

[6] Hao HZ, Tian Y. Engaging with AI: An exploratory study on developers' sharing and reactions to ChatGPT in GitHub pull requests. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering Workshops. Sacramento: ACM, 2024. 156–160. [doi: 10.1145/3691621.3694946]

[7] Fan A, Gokkaya B, Harman M, Lyubarskiy M, Sengupta S, Yoo S, Zhang JM. Large language models for software engineering: Survey and open problems. In: Proc. of the 2023 IEEE/ACM Int'l Conf. on Software Engineering: Future of Software Engineering (ICSE-FoSE). Melbourne: IEEE, 2023. 31–53.

[8] Dong YH, Jiang X, Jin Z, Li G. Self-collaboration code generation via ChatGPT. ACM Trans. on Software Engineering and Methodology, 2024, 33(7): 189. [doi: 10.1145/3672459]

[9] Yuan ZQ, Liu MW, Ding SJ, Wang KX, Chen YX, Peng X, Lou YL. Evaluating and improving ChatGPT for unit test generation. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 76. [doi: 10.1145/3660783]

[10] Chen M, Tworek J, Jun H, *et al*. Evaluating large language models trained on code. arXiv:2107.03374, 2021.

[11] Ouyang SY, Zhang JM, Harman M, Wang M. An empirical study of the non-determinism of ChatGPT in code generation. ACM Trans. on Software Engineering and Methodology, 2025, 34(2): 42. [doi: 10.1145/3697010]

[12] Liu JW, Xia CS, Wang YY, Zhang LM. Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation. In: Proc. of the 37th Int'l Conf. on Neural Information Processing Systems. New Orleans: Curran Associates Inc., 2023. 943.

[13] Leino KRM. Dafny: An automatic program verifier for functional correctness. In: Proc. of the 16th Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning. Dakar: Springer, 2010. 348–370. [doi: 10.1007/978-3-642-17511-4_20]

[14] Dafny Reference Manual. 2025. https://dafny.org/dafny/DafnyRef/DafnyRef

[15] Zhu M, Karim M, Lourentzou I, Yao D. Semi-supervised code translation overcoming the scarcity of parallel code data. In: Proc. of the

39th IEEE/ACM Int'l Conf. on Automated Software Engineering. Sacramento: ACM, 2024. 1545–1556. [doi: 10.1145/3691620.3695524]

[16] 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.

[17] Roziere B, Zhang JM, Charton F, Harman M, Synnaeve G, Lample G. Leveraging automated unit tests for unsupervised code translation. In: Proc. of the 10th Int'l Conf. on Learning Representations. OpenReview.net, 2022. 1–20.

[18] Szafraniec M, Roziere B, Leather H, Charton F, Labatut P, Synnaeve G. Code translation with compiler representations. arXiv: 2207.03578, 2022.

[19] Yang Z, Liu F, Yu ZX, Keung JW, Li J, Liu S, Hong YF, Ma XX, Jin Z, Li G. Exploring and unleashing the power of large language models in automated code translation. Proc. of the ACM on Software Engineering, 2024, 1(FSE): 71. [doi: 10.1145/3660778]

[20] Yuan ZQ, Chen WT, Wang HL, Yu K, Peng X, Lou YL. TRANSAGENT: An LLM-based multi-agent system for code translation. arXiv:2409.19894, 2024.

[21] Sim H, Cho H, Go Y, Fu ZL, Shokri A, Ravindran B. Large language model-powered agent for C to Rust code translation. arXiv:2505.15858, 2025.

[22] C2Rust. 2025. https://github.com/immunant/c2rust

[23] CxGo. 2025. https://github.com/gotranspile/cxgo

[24] Emani KV, Deshpande T, Ramachandra K, Sudarshan S. DBridge: Translating imperative code to SQL. In: Proc. of the 2017 ACM Int'l Conf. on Management of Data. Chicago: ACM, 2017. 1663–1666. [doi: 10.1145/3035918.3058747]

[25] JavaToCSharp: Java to C# converter. 2025. https://github.com/paulirwin/JavaToCSharp

[26] Sharpen. 2021. https://github.com/mono/sharpen

[27] Puri R, Kung DS, Janssen G, Zhang W, Domeniconi G, Zolotov V, Dolby J, Chen J, Choudhury M, Decker L, Thost V, Buratti L, Pujar S, Ramji S, Finkler U, Malaika S, Reiss F. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. In: Proc. of the 35th Conf. on Neural Information Processing Systems. NeurIPS, 2021. 1–13.

[28] 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 IEEE/ACM Int'l Conf. on Software Engineering. Lisbon: ACM, 2024. 82. [doi: 10.1145/3597503.3639226]

[29] Ahmad WU, Tushar MGR, Chakraborty S, Chang KW. AVATAR: A parallel corpus for Java-Python program translation. In: Proc. of the 2023 Findings of the Association for Computational Linguistics. Toronto: ACL, 2023. 2268–2281. [doi: 10.18653/v1/2023.findings-acl.143]

[30] Xie YQ, Naik A, Fried D, Rose C. Data augmentation for code translation with comparable corpora and multiple references. In: Proc. of the 2023 Empirical Methods in Natural Language Processing (EMNLP 2023). Singapore: ACL, 2023. 13725–13739. [doi: 10.18653/v1/2023.findings-emnlp.917]

[31] Tao QX, Yu TR, Gu XD, Shen BJ. Unraveling the potential of large language models in code translation: How far are we? In: Proc. of the 31st Asia-Pacific Software Engineering Conf. (APSEC). Chongqing: IEEE, 2024. 353–362. [doi: 10.1109/APSEC65559.2024.00046]

[32] Luo Y, Yu R, Zhang FJ, Liang L, Xiong YQ. Bridging gaps in LLM code translation: Reducing errors with call graphs and bridged debuggers. In: Proc. of the 39th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Sacramento: IEEE, 2024. 2448–2449. [doi: 10.1145/3691620.3695322]

[33] Macedo M, Tian Y, Nie PY, Cogo FR, Adams B. InterTrans: Leveraging transitive intermediate translations to enhance LLM-based code translation. In: Proc. of the 47th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Ottawa: IEEE, 2025. 1153–1164. [doi: 10.1109/ICSE55347.2025.00236]

[34] Ahmed T, Devanbu P. Few-shot training LLMs for project-specific code-summarization. In: Proc. of the 37th IEEE/ACM Int'l Conf. on Automated Software Engineering. Rochester: ACM, 2022. 177. [doi: 10.1145/3551349.3559555]

[35] Geng MY, Wang SW, Dong DZ, WangHT, Li G, Jin Z, Mao XG, Liao XK. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In: Proc. of the 46th IEEE/ACM Int'l Conf. on Software Engineering (ICSE). Lisbon: IEEE, 2024. 453–465. [doi: 10.1145/3597503.3608134]

[36] Macedo M, Tian Y, Cogo FR, Adams B. Exploring the impact of the output format on the evaluation of large language models for code translation. In: Proc. of the 1st IEEE/ACM Int'l Conf. on AI Foundation Models and Software Engineering. Lisbon: IEEE, 2024. 57–68. [doi: 10.1145/3650105.3652301]

[37] Quick migration guide from Dafny 3.X to Dafny 4.0. 2023. https://github.com/dafny-lang/ide-vscode/wiki/Quick-migration-guide-from-Dafny-3.X-to-Dafny-4.0

[38] Dong QX, Li L, Dai DM, Zheng C, Ma J, Li R, Xia HM, Xu JJ, Wu ZY, Chang BB, Sun X, Li L, Sui Z. A survey on in-context learning. In: Proc. of the 2024 Conf. on Empirical Methods in Natural Language Processing. Miami: ACL, 2024. 1107–1128.

[39] Wang YQ, Yao QM, Kwok JT, Ni LM. Generalizing from a few examples: A survey on few-shot learning. ACM Computing Surveys (CSUR), 2021, 53(3): 63. [doi: 10.1145/3386252]

[40] DeepSeek API Docs. 2025. https://api-docs.deepseek.com/quick_start/parameter_settings

[41] Austin J, Odena A, Nye M, Bosma M, Michalewski H, Dohan D, Jiang E, Cai C, Terry M, Le Q, Sutton C. Program synthesis with large language models. arXiv:2108.07732, 2021.

[42] LeetCode. 2025. https://github.com/walkccc/LeetCode

[43] Ren S, Guo DY, Lu S, Zhou L, Liu SJ, Tang DY, Sundaresan N, Zhou M, Blanco A, Ma S. CodeBLEU: A method for automatic evaluation of code synthesis. arXiv:2009.10297, 2020.

[44] Lu YH, Zhu XY, Zhang WH, Yan RJ. Formalizing requirements into Dafny specifications with LLMs. In: Proc. of the 26th Int'l Conf. on Formal Engineering Methods. Hangzhou: Springer, 2025. 97–116. [doi: 10.1007/978-981-95-4213-0_6]

作者简介

杜奕成, 硕士生, CCF 学生会员, 主要研究领域为形式化方法, 约束求解.

卢奕函, 硕士生, CCF 学生会员, 主要研究领域为形式化方法.

朱雪阳, 博士, 副研究员, CCF 高级会员, 主要研究领域为形式化方法, 嵌入式系统设计.

张文辉, 博士, 研究员, CCF 高级会员, 主要研究领域为形式化方法, 演绎推理, 模型检测.