编译器引导的推理时适应：提升 GPT-5 在 Idris 中的编程表现

Minda Li 和 Bhaskar Krishnamachari*Ming*
Hsieh 电气与计算机工程系，维特比工程学院，南加州大学
美国加利福尼亚州洛杉矶 {mcli, bkrishna}@usc.edu

arXiv:2602.11481v1 [cs.PL] 12 Feb 2026

摘要——**OpenAI 推出的先进大语言模型 GPT-5，在 Python、C++ 和 Java 等广泛使用的编程语言中表现出色；然而，它在低资源或较少使用的语言中的能力仍未得到充分探索。本研究探讨 GPT-5 能否通过迭代式、基于反馈的提示，有效掌握一门陌生的函数式编程语言 Idris。我们首先建立基线：在零样本提示下，该模型使用 Exercism 平台仅能解答 56 道 Idris 练习中的 22 道，与资源更丰富的语言相比表现明显逊色（Python 为 50 道中解答 45 道，Erlang 为 47 道中解答 35 道）。随后，我们评估了多种改进策略，包括依据平台反馈进行迭代式提示、在提示中加入文档和错误分类指南，以及利用本地编译错误和失败测试用例进行迭代式提示。在这些方法中，纳入本地编译错误带来的提升最大。通过这一结构化的错误引导改进循环，GPT-5 的表现显著提升，成功解答了 56 道题目中的 54 道。这些结果表明，尽管大语言模型起初可能难以应对低资源环境，但结构化的编译器级反馈能够在释放其能力方面发挥关键作用。**

一、引言

OpenAI 的 GPT-5 等大语言模型在 Python、C++ 和 Java 等广泛使用的编程语言中表现出色。这种高水平的熟练度得益于网上大量可用的训练数据和文档。然而，对于这些模型处理低资源或较少见语言的能力，人们所知甚少。评估模型在这类语言中的表现，有助于了解大语言模型能否在高频训练样例之外实现泛化，以及能否通过迭代式提示学会新的抽象概念。

Idris 是研究这一问题的理想案例。作为一种依赖类型函数式编程语言，它在公开代码仓库中的占比远低于主流语言。GitHub 仓库数量充分体现了这种差距：Idris 有 2,275 个仓库，与 Forth（2,255 个）和 Agda（3,049 个）等小众语言的规模相近。相比之下，更高级的函数式编程语言（如 Racket，22,655 个）、工业级语言（如 Erlang，33,058 个；Elixir，121,841 个），尤其是通用语言（如 Python，24,286,135 个；C++ ，6,182,683 个）的代表性要高得多。这种不平衡表明，大语言模型在训练期间接触 Idris 的机会可能要少得多。因此，Idris 提供了一个可控的测试环境，用以检验模型能否适应未曾深入学习过的语言。

为了将模型在 Idris 上的表现置于背景中考察，我们还测定了模型在更常见语言中的基线表现。由于 Exercism 的各语言练习路径规模不同，跨语言基线是在规模最大的重叠题目子集上计算的，这些子集包含相同的练习：Idris 中全部 56 道可用题目、 Python 与 Idris 练习路径共有的 50 道题目，以及 Erlang 与 Idris 练习路径共有的 47 道题目。在 Python 中，GPT-5 解出了 90% 的题目（45/50），所有解答均成功编译，但有少数未能通过测试用例。在 Erlang 中，这是一种较不常见、但相较于 Idris 仍有显著更多资源的语言，模型解出了 74% 的题目（35/47）。这些基线为评估模型面对资源日益匮乏的语言时的表现提供了有用的比较参照。

由于其相对稀缺、严格类型系统以及陌生的函数式模式，Idris 提供了一个独特的环境，用以探索一个关键研究问题：GPT-5 能否通过迭代式、反馈驱动的改进来学习一门新的编程语言?

本研究探讨以下研究问题：

- **RQ1:** 与资源更丰富的编程语言相比，GPT-5 在低资源、依赖类型的编程语言中表现如何?
- **RQ2:** 迭代式、错误引导的提示，尤其是在纳入编译器级反馈时，能在多大程度上提升 GPT-5 在低资源编程语言中的表现?

总而言之，本文做出了三项贡献。首先，据我们所知，我们首次在明确规定的评估协议下，对 GPT-5 在 Idris Exercism 练习轨道上的表现进行了测量研究。

© 2026 IEEE。允许个人使用本材料。对于所有其他用途，必须获得 IEEE 的许可，包括在任何当前或未来媒介中为广告或推广目的重印/再版本材料、创作新的汇编作品、用于转售或再分发至服务器或列表，或在其他作品中重用本作品的任何受版权保护的组成部分。

其次，我们比较了四种推理时优化策略，从零样本提示到检索增强提示和迭代式编译/测试反馈循环，并量化了它们对求解率的影响。第三，我们的结果表明，编译器诊断信息是推动性能提升的主要学习信号，这一点也得到了尝试情况细分和观察到的错误模式的印证。

II. 相关工作

我们的研究处于两个领域的交汇处：评测大语言模型（LLM）在低资源编程语言上的表现，以及通过迭代式、编译器引导的改进来提升其性能。

A. 多语言评测与低资源适配

早期的代码生成基准（如 HumanEval）主要聚焦于 Python [1], ，而近期研究则凸显了不同编程语言之间的性能差距。*MultiPL-E* 将基于单元测试的评测扩展至多种语言，证实模型性能与训练语料库中的语言表示程度高度相关[2]。为应对数据稀缺问题，*MultiPL-T* 证明，利用合成且经测试验证的数据进行高资源语言的训练时迁移，可以提升模型在低资源环境下的能力 [3]。不同于这些以训练为中心的方法，我们研究的是推理时适配，探讨通用模型能否仅凭迭代反馈和文档，掌握 Idris 这类小众语言。

B. 迭代优化与反馈循环

近期研究强调，应从单次生成转向迭代反馈循环。*Self-Refine*证明，大语言模型无需额外训练，也能通过迭代式批评与修订来改进自身输出 [4]。针对代码，*SELF-DEBUGGING* 使模型能够执行代码、分析执行轨迹并生成反馈以修复错误 [5]。*LLMLOOP*等框架则通过打通生成、编译和测试之间的闭环，进一步实现了这一过程的自动化 [6]。我们的研究借鉴了这些迭代原则，但重点放在编译器层面的反馈上；我们发现，与基于执行的反馈相比，这才是依赖类型语言中的关键瓶颈。

C. 编译器诊断信息与形式化约束

使用编译器诊断信息进行程序修复已是成熟做法。在大语言模型出现之前，像*DrRepair* 这样的系统利用基于图的建模，将源代码符号与编译器错误消息关联起来，以实现自动修复 [7]。在大语言模型时代，这种方法已扩展到形式化环境。*Baldur*利用证明助手提供的错误上下文生成证明并修复错误，这与利用编译器反馈进行修复的做法如出一辙 [8]，；而*LeanDojo* 则为定理证明提供了检索增强环境 [9]。此外，Blinn 等人表明，公开静态语义（例如带类型空洞），能够有效约束并引导大语言模型生成内容 [10]。我们的结果印证了这些发现，表明在严格的函数式环境中，结构化的编译器反馈比单独使用自然语言提示能提供更强的学习信号。

III. 方法论

以下各方法论的全部代码可在[录用后提供的网址获取，因双盲评审而省略]。本项目最初启动时，我们希望寻找很少使用且不熟悉的编程语言。在这一搜索过程中，我们发现了平台 Exercism。Exercism 拥有从 Python 到 Cobol 的多种语言，共计 78 种不同语言。我们最终聚焦于 Idris，一种依赖类型程序。Idris 有 56 道题目（不包括起始的 Hello, World! 题目）。Exercism 提供将每道题目下载到本地计算机的选项，因此我们使用了这一功能。

在全部 56 道题目下载到本地计算机后，我们使用两个不同的 Python 脚本来建立基线。第一个 Python 脚本是查询编写器。为自动为所有题目编写查询，该 Python 脚本首先提取包含题目描述的 README.md 内容，并抓取 Idris 文件中提供的起始代码行。查询编写器的格式随后写成如下：

下面是生成的示例查询。

```
编写一个 Idris 程序来完成以下任务，并将
      该程序放入一个 JSON 对象中，作为
      对应于键 'code' 的值。
# Accumulate

欢迎来到 Exercism 的 Idris 练习路径中的 Accumulate。
如果你需要帮助来运行测试或提交
      你的代码，请查看 `HELP.md`。

## 说明

实现 `Accumulate` 操作，它接收一个
       集合，以及一个要应用于每个
      集合元素的操作                    ，返回一个新的
      集合，其中包含对每个输入
      集合元素应用该操作后的结果
      。

给定数字集合：

- 1, 2, 3, 4, 5

以及操作：

- 将数字平方（`x => x * x`）

你的代码应能够生成平方数集合：


- 1, 4, 9, 16, 25

查看测试套件，了解预期的
      函数签名。

## 限制
```

```
别动那个 collect/map/fmap/
        什么玩意儿功能，它是由你的
        标准库提供的!
用其他基础工具自己解决这个问题
        吧。

## 来源

### 贡献者

- @keiravillekode
- @ktosiek
- @kytrinyx
- @NobbZ
- @yurrriq

### 基于

与 James Edward Gray II 的对话 - http://
        gravsoftinc.com/
编写符合以下模板的代码。程序
        应在以下几行之后继续:
        行:
module Accumulate


export
accumulate : (a -> b) -> List a -> List b
accumulate f [] = []
accumulate f (x :: xs) = f x :: accumulate f xs
```

生成所有查询后，我们编写了第二个 Python 脚本，通过 OpenAI API 密钥将每个查询发送给 GPT-5，并自动保存模型生成的代码。该脚本逐一读取查询文件，将文件内容作为请求发送给 GPT-5 模型，然后获取模型的响应。由于每个查询都要求 GPT-5 以 JSON 格式返回答案，并将 Idris 代码存放在名为 "code" 的键下，因此脚本只需从响应中提取该字段，并将其保存到提供初始代码的文件中。通过这一流程，我们得以一致地生成并存储所有模型输出。所有答案都提交回 Exercism，我们也记录了题目是否通过。

确定 GPT-5 能够解决的题目数量的基线后，我们开始探索通过几种不同的方法提升其表现。首先，我们使用 Exercism 平台返回的失败测试用例评估了迭代式提示，将单次迭代和五次反馈迭代分别作为一种迭代优化方法。接着，对于最初未能解决的题目，我们通过提供额外的外部信息来探索提示增强策略。这些信息包括：(i) 根据常见基线错误编写的错误文档手册，以及 (ii) 官方 Idris 参考文档，两者均与题目查询一同提供。最后，我们评估了一种全面的迭代优化策略，最多进行 20 轮本地编译并分析失败测试用例，将编译器诊断信息和测试失败直接纳入后续提示中。

A. 方法*1*：将 *Exercism* 错误消息纳入查询

将每个解答提交到 Exercism 平台后，未解决的题目会返回错误消息。我们将这些错误消息纳入后续查询中，指示 GPT-5 修正未通过测试所指出的具体问题。这种方法成功解决了少数题目，因此我们进一步扩展了这一流程：针对每道未解决的题目，最多进行五轮错误引导的改进。

B. 方法*2*：错误规避手册

在审查初始错误消息时，我们发现了几种反复出现的模式。我们记录了第一轮测试中的所有错误，并将它们汇编成一个数据集。随后，我们要求 GPT-5 将该数据集载入上下文，并将错误归类为有意义的类别，例如语法错误、缺失的库、未填补的空洞和逻辑错误。对于每个类别，GPT-5 都生成了描述、指出其出现频率，并解释了这些错误在 Idris 中产生的原因。基于这套分类，我们提示 GPT-5 创建一份参考手册，概述 Idris 中常见的陷阱、如何避免这些问题，以及编写有效 Idris 代码的最佳实践。我们的目标是确定，向 GPT-5 提供这份由其自行生成的手册，是否能够减少重复错误，并提升其生成正确 Idris 解决方案的能力。

为了测试参考手册能否提升 GPT-5 的 Idris 性能，我们将手册存储在向量存储中，以便模型在每次查询时检索相关章节。脚本首先将 PDF 上传到向量存储，在其中对其进行嵌入、分块和索引，以供语义搜索使用。解决新的 Idris 问题时，模型会从该存储中检索最相关的段落，并在生成代码前将其作为上下文。这使 GPT-5 能够持续复用相同的知识，而无需重新上传文档，并减少了不同任务中的重复错误。

C. 方法 *3*：*Idris* 参考手册

除了自行生成的错误规避手册外，我们还评估了向 GPT-5 提供官方语言文档是否能进一步提升其完成 Idris 练习的表现。我们找到了一个 PDF 格式的外部 Idris 参考手册，其中全面介绍了该语言的语法、类型系统、标准库用法以及常见编程模式。我们将此文档上传到向量存储并进行嵌入，以支持语义检索。

与方法 2 类似，我们将参考手册分块并建立索引，以便在解题过程中动态检索相关章节。对于每道尚未解决的 Idris 练习，模型首先查询向量存储，检索与当前问题最相关的文档段落，然后在生成解答之前将检索到的上下文纳入提示词。

通过将此方法与错误规避手册进行比较，我们旨在评估：仅凭官方语言文档是否足以提升性能，还是模型生成的、针对特定错误的指导对解决低资源语言任务更有帮助。

D. 方法*4*：本地编译与失败测试

在最后一种也是最全面的改进策略中，我们评估了一种迭代式、编译器驱动的方法，将本地编译错误和失败测试用例都纳入提示循环。与主要依赖平台报告的（Exercism）测试失败或静态参考资料的方法1–3不同，该方法利用了在本地构建每个 Idris 解决方案时生成的详细编译器诊断信息。

对于每个尚未解决的练习，GPT-5 首先生成一个初始解，然后使用Idris编译器在本地进行编译。如果编译失败，则捕获相应的编译器错误消息（包括类型错误、缺失定义、模式匹配失败或完全性违规），并将其直接纳入后续提示。GPT-5 会收到指示，针对所报告的编译问题修改代码。这个过程反复进行，直到程序成功编译，或达到预设的 20 次迭代上限。对于每个问题，一旦程序成功编译并通过所有本地测试，优化循环便立即终止，并将解提交至 Exercism；否则，循环会在最多迭代 20 次后停止。

解成功编译后，接着会使用完整的本地测试集运行该解。任何失败的测试用例都会产生额外的诊断输出，这些输出同样会附加到下一次查询中，用于指导进一步修改。只有当解成功编译并通过所有本地测试用例后，才会将其提交至 Exercism 平台。这种方法确保提交内容在接受平台评估之前，语法有效、类型正确且功能正常。

此方法在方法1中探索的错误引导提示策略基础上进行了扩展，纳入了更底层的编译器反馈，而非仅依赖高层次的测试失败。通过向 GPT-5 提供精确的编译器诊断信息，并在开发流程的每个阶段强制确保正确性，这种方法为纠正错误提供了显著有力的指导。

IV. 结果与分析

我们首先展示一个定性示例，说明编译器引导的逐步优化在实践中的应用，随后给出量化结果，概述在所有 Idris 练习中的性能提升。

为说明编译器引导的推理时适应，我们以 Bob练习为例进行简要分析。GPT-5 最初生成的零样本解在语义上看似合理，但由于对 Idris语法和可用标准库函数的理解有误，未能通过编译。初始零样本输出（节选）：

```
模块 Bob 导出 response : 字符串 ->字符串 response 备注| 是沉默
| 否则 = “随便。”
```

初始零样本解因语法错误、缺失导入和未定义辅助函数而失败——这些都是表II中的常见失败模式。经过迭代优化，GPT-5 重构了控制流，移除了不受支持的导入，并显式定义了字符级谓词。最终收敛的解（节选）：

```
module Bob import Data.String import Data.List export
```

经过五轮迭代，解决方案成功编译，并通过了所有 Exercism 测试用例。此示例展示了编译器诊断信息如何提供精确的实例特定反馈，使大语言模型能够调整输出，以适应低资源静态类型语言的严格语义。

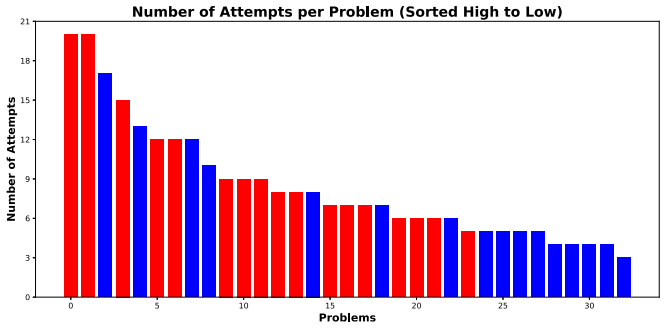
表 I 迭代精炼方法带来的性能提升（N = 56）。

方法	已解决	+已解决	% 已解决
基线（GPT-5.0）	22	–	39
包含失败测试用例的 1 次迭代	27	+4	48
包含失败测试用例的 5 次迭代	27	+4	48
错误文档手册	30	+7	54
Idris 参考手册	34	+11	61
编译并进行失败测试的 20 次迭代	54	+31	96

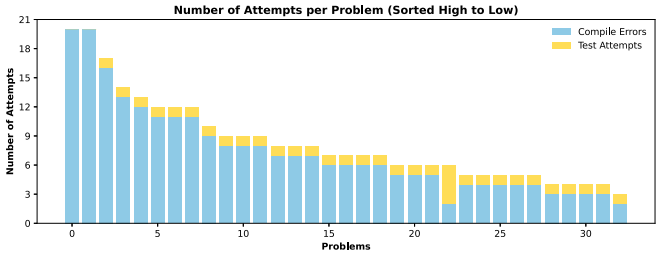
表 I 总结了各迭代优化方法在 Exercism 平台上的 56 道 Idris 练习中取得的性能提升。基线 GPT-5 模型解出了[1]22 道题（39%），仍有 33 道题未解；后续优化方法均针对这些未解题目应用（彼此独立，并非依次进行）。利用来自 Exercism 平台的失败测试用例反馈，效果有所提升：经过一轮和五轮错误引导提示后，解出的题目数都增至 27 道（+4），成功率达到 48%；超过一轮后则未观察到额外提升。提供补充参考材料带来了更明显的提升。提供人工编写的错误文档手册后，解题数增至 30 道（54%）；在提示中加入官方 Idris 参考手册后，解题数进一步增至 34 道（61%）。最大的性能提升来自本地编译与失败测试优化策略。通过迭代纳入编译器诊断信息以及

[1]这里的“解决”是指通过 Exercism 上的所有测试用例。

将本地测试失败情况纳入提示循环后，GPT-5 解决了 56 道题目中的 54 道（96%），显著优于所有其他方法。达到 20 次迭代的上限后，仍有两道题目未能解决。在这两种情况下，反复进行的自动修复在辅助定义、类型和测试框架中引发了连锁的编译问题，尽管多次尝试，仍无法稳定收敛。

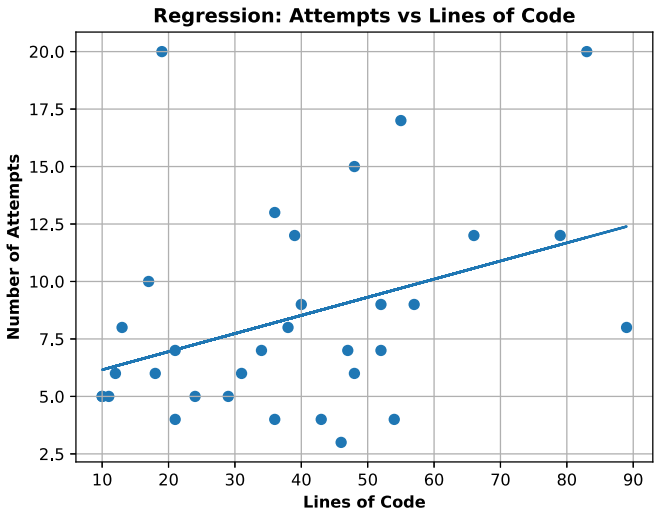


上方柱状图展示了基线评估后仍未解决的 33 道题目，并按 GPT-5 使用方法4解决题目所需的尝试次数从高到低排序。蓝色柱表示使用方法1–3（基于平台的测试反馈、错误文档手册或 Idris 参考手册）中的至少一种方法成功解决的题目。红色柱表示仅使用方法4解决、而未能通过方法1–3解决的题目。该图表凸显出，所需尝试次数最多的题目大多只能通过编译器驱动的改进来解决，体现了方法4应对更具挑战性案例的有效性。



上方柱状图展示了方法4解决最初未能解决的 33 道题目时所需的尝试次数，并将尝试分解为编译器驱动的修复和测试驱动的修复。结果表明，对于困难题目，尝试次数主要由编译器纠错而非测试失败所构成。许多题目需要经过大量迭代才能解决编译器错误。相比之下，编译成功后，通常只需较少的额外迭代即可处理未通过的测试用例。这一模式表明，在 Idris 环境中，大语言模型的主要瓶颈在于满足该语言严格的类型系统和编译要求，而非修正测试所暴露的逻辑错误。总尝试次数最多的题目往往需要先经历一连串编译器修复，随后

通过一个简短的测试精炼阶段，这表明编译器层面的反馈为引导模型修正提供了最具信息量的信号。总体而言，这一分解说明了方法4为何显著优于其他精炼策略：通过显式地迭代编译器诊断信息，模型能够解决仅凭测试失败无法捕获的低层语义错误和类型错误，从而即使在最具挑战性的题目上也能成功收敛。



我们进行了线性回归分析，以考察解的规模（以代码行数（LOC）衡量）与成功完成每道练习所需尝试次数之间的关系。使用合并后的数据集，估计的回归系数为斜率 $\beta_1 \approx 0.079$、截距 $\beta_0 \approx 5.37$ 。皮尔逊相关系数为 $r \approx 0.37$，决定系数为 $R^2 \approx 0.14$。

正斜率表明，需要更多代码行数的练习往往需要更多次尝试。从数量上看，代码行数每增加约10行，平均尝试次数约增加0.8次。

然而，这种相关性属于中等，而非强相关。相对较低的 R^2 值表明，仅解答长度只能解释尝试次数方差的约14%，这意味着其他因素——例如问题结构、语言特定语义以及与类型系统的交互——在决定模型性能方面也发挥着重要作用。表II总结了在对 Idris 练习进行 GPT-5 基线评估期间观察到的编译错误分布。最常见的错误类别是未定义名称（123 次）和有歧义的 elaboration（111 次），这表明名称解析和作用域消歧是模型在依赖类型语言中面临的主要挑战。这些错误通常源于缺少辅助函数、导入不正确，或局部定义的标识符与标准库定义之间发生冲突，反映出模型在管理 Idris 命名空间和 elaboration机制方面存在困难。另有大量错误被归类为非编译器错误（114 次），其中包括平台级消息和执行产物

表II 基线评估中观察到的编译错误分类

错误类型	出现次数
未定义名称	123
有歧义的 elaboration	111
其他错误／非编译器错误	114
解析／语法错误（一般）	33
隐私／可见性（未导出／私有）	36
缺少预期的 ‘else’ 解析错误	22
缺少模块／未找到导入项	15
类型不匹配／合一错误	11
未知运算符	2
缺少入口点（未找到 Main）	1
全性／终止错误	1

而非真正的编译器诊断信息。在真正的编译器错误中，解析错误和语法错误相对常见（33 次一般解析错误，以及 22 次缺少 else 分支），这表明模型经常在 Idris 的 do 记法中生成结构不完整的控制流结构。隐私和可见性错误（36 次）进一步表明，模型在正确导出定义和浏览模块边界方面面临挑战，而 Idris 对这些方面的要求尤为严格。

总体而言，错误分布表明，GPT-5 在 Idris 中面临的主要障碍并非深层类型理论推理能力不足，而是与作用域、命名和编译规范相关的表层语义及结构问题。这一发现有助于解释为何方法4——由编译器诊断信息驱动的迭代优化——尤其有效：编译器级反馈直接针对最常见的错误类别，从而能够系统地纠正那些最常导致编译失败的问题。

五、讨论与结论

本研究证实，尽管 GPT-5 在 Python（成功率 90%）和 Erlang（74%）等高资源语言中表现出色，但在依赖类型语言 Idris 中却面临显著困难，基准成功率仅为 39%。然而，我们证明，无需额外训练也能弥合这一差距。通过实现由本地编译器诊断信息驱动的迭代优化循环（方法4），模型的表现大幅提升至 96%（54/56 道题）。这些结果直接回应了我们的研究问题，表明通用大语言模型虽然对小众语法缺乏先天熟悉度，但在精确的错误引导反馈下具备很强的适应能力。尽管我们只评估了 Idris，方法4的核心机制——利用精确的编译器诊断信息进行迭代优化——应可推广到其他低资源、强类型或形式化语言（如 Agda、Coq、Lean）。

编译器驱动的改进优于文档增强提示，这凸显了模型所面临困难的本质。我们的错误分析发现，主要障碍并非深层逻辑缺陷，而是未定义名称和含糊不清的 elaboration 等表层问题。通用文档虽提供了背景信息，却未能纠正这些特定实例中的错误。相比之下，编译器充当了严格的验证器，准确指出作用域和类型定义中的具体错误。这表明，对于严格函数式语言，编译器提供的学习信号远比自然语言提示或高层级测试失败更有效。

一项效度威胁在于，部分 Exercism Idris 解答或密切相关的代码模式可能已出现在模型的预训练数据中，从而可能夸大表面上的“新语言”性能。未来工作应在新编写的或按时间留出的 Idris 任务（和/或通过重命名与改写题述得到的扰动变体）上进行测试，以便更好地区分语义问题求解与记忆召回。未来工作还应探索将大语言模型与形式化验证工具相结合，把这种编译器在环方法扩展到 Coq 或 Agda 等其他形式化环境。

参考文献

[1] M. Chen、J. Tworek、H. Jun、Q. Yuan、H. P. d. O. Pinto 等，“评估在代码上训练的大语言模型”，*arXiv* 预印本 *arXiv:2107.03374*，2021。 [在线]。可获取：https://arxiv.org/abs/2107.03374 。F. Cassano、A. Jangda、A. Guha 等，“MultiPL-E：一种可伸缩且可扩展的神经代码生成基准评测方法”，*arXiv* 预印本 *arXiv:2208.08227*，2022。 [在线]。可获取：https://arxiv.org/abs/2208.08227 。F. Cassano、J. Gouwar、F. Lucchetti、C. Schlesinger、A. Freeman、C. J. Anderson、M. Q. Feldman、M. Greenberg、A. Jangda 和 A. Guha，“面向代码大语言模型的从高资源到低资源编程语言的知识迁移”，*arXiv* 预印本 *arXiv:2308.09895*，2023。 [在线]。可获取：https://arxiv.org/abs/2308.09895 。A. Madaan、N. Tandon、P. Gupta、S. Hallinan、L. Gao、S. Wiegreffe、U. Alon、N. Dziri、S. Prabhumoye、Y. Yang、S. Gupta、B. P. Majumder、K. Hermann、S. Welleck、A. Yazdanbakhsh 和 P. Clark，“Self-Refine：基于自我反馈的迭代优化”，*arXiv* 预印本 *arXiv:2303.17651*，2023。[在线]。可获取：https://arxiv.org/abs/2303.17651 。X. Chen、M. Lin、N. Schärli 和 D. Zhou，“教大语言模型进行自我调试”，载于国际学习表征会议（*ICLR*），2024。[6] R. Ravi、D. Bradshaw、S. Ruberto、G. Jahangirova 和 V. Terragni，“LLMLOOP：通过自动化迭代反馈循环改进大语言模型生成的代码和测试”，载于软件维护与演化国际会议（*ICSME*），工具轨道，2025。[在线]。可获取：https://valerio-terragni.github.io/assets/pdf/ravi- icsme-2025.pdf [7] M. Yasunaga 和 P. Liang，“基于图的、利用诊断反馈的自监督程序修复”，载于国际机器学习会议（*ICML*），2020。[在线]。可获取：https://arxiv.org/abs/2005 10636 。E. First、M. N. Rabe、T. Ringer 和 Y. Brun，“Baldur：利用大语言模型进行完整证明生成与修复”，载于*ACM* 欧洲软件工程联合会议暨软件工程基础研讨会（*ESEC/FSE*），2023。 [在线]。可获取：https://arxiv.org/abs/2303 04910 。K. Yang、A. M. Swope、A. Gu、R. Chalamala、P. Song、S. Yu、S. Godil、R. Prenger 和 A. Anandkumar，“LeanDojo：基于检索增强语言模型的定理证明”，载于 *NeurIPS* 数据集与基准轨道，2023。 [在线]。可获取：https://arxiv.org/abs/2306 15626 。A. Blinn、X. Li、J. H. Kim 和 C. Omar，“利用带类型空洞对大语言模型进行静态情境化”，*arXiv* 预印本 *arXiv: 2409 00921*，。在线。可获取：2024 [ ]。https://arxiv.org/abs/2409 00921 。