arXiv:2601.02060v1 [cs.PL] 5 Jan 2026

兴衰抉择？函数式编程中用于代码生成的大语言模型全面评估

Nguyet-Anh H. Lang[1], **EricLang**[2], **Thanh Le-Cong**[2*], **Bach Le**[2],**Quyet-Thang Huynh**[21] 河内科技大学[2]墨尔本大学

摘要

函数式编程为开发可靠、安全的软件系统奠定了坚实基础，但由于学习曲线陡峭，其应用仍不广泛。大语言模型（LLM）在代码生成方面的最新进展，为降低这些门槛带来了新的机遇。然而，对大语言模型的大规模评估主要聚焦于命令式编程语言，其在函数式编程语言（函数式编程）中的能力仍未得到充分探索。为弥补这一空白，我们推出了FPEval，这是一个基于 FPBench 构建的全面评估框架。FPBench 是一个新的基准测试，涵盖 3 种主流函数式编程语言——Haskell、OCaml 和 Scala 中的 721 个编程任务，分为 3 个难度级别。FPEval 提供了全面的评估基础设施，结合了配备全面测试套件的测试验证和静态分析工具，用于评估功能正确性、代码风格和可维护性。我们使用该框架评估了最先进的大语言模型，包括 GPT-3.5、GPT-4o 和 GPT-5 在函数式编程语言中的代码生成能力，并以 Java 作为命令式基线。结果表明，随着模型不断进步，大语言模型在函数式编程方面的性能显著提升；然而，纯函数式语言（Haskell 和 OCaml）中的错误率仍显著高于混合范式语言（Scala）或命令式语言（Java）。此外，大语言模型经常生成遵循命令式模式的非惯用函数式代码，这引发了人们对代码风格和长期可维护性的担忧。最后，我们发现，在提供静态分析反馈以及针对常见问题类型精心编写的指令后，大语言模型能够部分自我修复正确性和质量方面的问题。

1 引言

函数式编程是一种新兴的声明式编程范式，它将计算概念化为对数学函数的求值，而非广泛使用的命令式编程中按顺序执行会改变状态的命令。这种计算模型相较于命令式编程具有独特优势，使软件系统具备模块化、确定性，并易于进行形式化推理。例如，函数式编程提供了不可变性，确保变量的值在其生命周期内不会改变。不可变性使函数式编程能够缓解命令式编程中常见的软件缺陷来源，例如副作用和竞态条件。因此，函数式编程日益被视为实现可靠、安全软件开发的未来范式（Achten，2013；Hughes，1989；Ray 等人，2014年）。尽管具备这些优势，函数式编程的学习曲线仍然陡峭。这种范式要求人们从带有可变状态的命令式指令，根本性地转向对程序行为进行高层次抽象，例如递归、高阶函数和单子，而许多开发者认为这很难掌握。因此，在实际应用中，函数式编程仍远不如命令式编程普及。

近年来，由大语言模型（LLM）驱动的编码助手迅速兴起，例如 GitHub Copilot（Chen 等人，2021）、Cursor（Any-sphere，2023）和 Claude Code （ Anthropic，2025 ），为函数式编程带来了新的机遇。这些工具支持软件开发人员完成代码生成（Li 等人，2022；Chen 等人，2021）、缺陷修复（Xia 等人，2023；Le-Cong 等人，2025b）和技术问答（Xu 等人，2023）等任务，对编程初学者尤其有帮助，并有望降低编程整体的学习曲线（Prather 等人，2023；Kazemitabaar 等人，2023）。如果类似的益处也能延伸到函数式编程，这些基于大语言模型的编码助手就有望促进这一编程范式的普及，

*通讯作者：thanhlc@ieee.org

并推动其在实践中的应用。不幸的是，现有关于基于大语言模型的编程的研究（Fan等人，2023年）主要关注 Python 和 Java 等命令式编程语言。相比之下，大语言模型在 Haskell、OCaml 和 Scala 等函数式编程语言中的适用性，仍未得到充分探索。

在本研究中，我们开展了首个全面的实证研究，评估大语言模型在函数式编程中的代码生成能力。我们的目标包括三个方面：（1）评估大语言模型在函数式编程中生成正确且高质量代码的能力；（2）评估大语言模型生成的代码在功能正确性之外的质量，尤其关注代码风格和可维护性；（3）评估大语言模型自我修复错误的效果。为此，我们围绕以下研究问题展开研究：

- **研究问题1：** 大语言模型在函数式编程语言中的代码生成方面效果如何?
- **研究问题2：** 大语言模型在函数式编程语言中生成的代码，常见的代码风格和可维护性问题有哪些?
- **研究问题3：** 自我修复机制在提升大语言模型生成的函数式代码的正确性、代码风格和可维护性方面效果如何?

为回答这些问题，我们推出了 FPE-val，这是一个专为函数式编程代码生成设计的综合评估框架。FPE-val 由 FPBench 提供支持。FPBench 是一个新的多语言基准测试集，包含 721 项编程任务，分为三个难度级别（184 项简单、346 项中等和 191 项困难），涵盖 Scala、Haskell 和 OCaml 三种语言。为严格评估大语言模型生成的代码，我们为每项任务构建了全面的测试套件，在 LeetCode 的公开测试用例基础上补充了私有测试用例，以针对每项编程任务的特定边界条件和边缘情况。此外，FPEval 还集成了广为人知的静态分析工具，包括 HLint（Mitchell，2024年）、OCamlFormat（OCamlFormat Developers，2024年）和 Scalastyle（Farwell 和 Scalastyle 贡献者，2024年），从而能够同时评估生成代码的功能正确性、代码风格和可维护性。

我们使用 FPEval 评估了最先进的大语言模型（GPT-3.5、GPT-4o 和 GPT-5）在函数式编程任务上的表现，并将 Java 作为命令式基线纳入评估。评估结果表明，随着模型不断进步，大语言模型在函数式编程任务上的表现显著提升，GPT-5 生成的功能正确代码数量约为 GPT-3.5 的 3 倍。然而，纯函数式语言（Haskell 和 OCaml）与混合范式或命令式语言（Scala 和 Java）之间始终存在明显的性能差距。除正确性外，我们还发现，大语言模型生成的代码中有相当一部分风格欠佳、可维护性较差。尤其是，大语言模型经常生成非惯用函数式代码，采用命令式模式，而非遵循函数式编程最佳实践。值得注意的是，随着功能正确性提升，此类低质量代码的比例也随之上升，这表明存在一种奖励劫持现象：模型在优化正确性的同时忽视了非功能属性。这些发现凸显了未来的大语言模型训练和评估需要明确纳入函数式编程最佳实践。最后，我们展示了在提供静态分析反馈以及针对常见问题类型精心编写的指令后，大语言模型能够在一定程度上自我修复正确性和质量问题。

总而言之，我们的主要贡献包括：

- 我们推出了 FPEval，这是一个用于评估函数式编程范式下大语言模型代码生成的综合框架。FPEval 整合了 FPBench——一个精心整理的数据集，包含 721 个编程任务及其在 Scala、Haskell 和 OCaml 中可执行的测试套件——以及一个自动化评估流程，该流程利用软件测试和静态分析工具，严格评估生成代码的正确性、代码风格和可维护性。

- 我们首次对大语言模型的函数式编程代码生成能力进行了实证评估，系统地评估了生成代码的函数正确性、代码风格和可维护性，以及大语言模型的自我修复能力。

- 我们详细分析了大语言模型生成的函数式代码中的代码风格和可维护性问题，并展示了借助静态分析反馈和详细指导来解决常见问题的大语言模型自我修复潜力，从而改善代码风格和可维护性问题

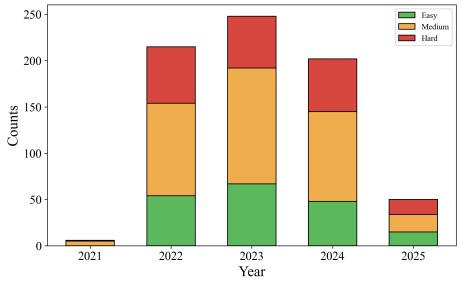

图1：不同时间段的任务分布

反馈以及针对常见问题的详细说明。

- 我们通过发布 FPE-val（采用 Apache 2.0 许可证），推动函数式编程代码生成研究的发展，降低学术研究的门槛，并为未来工作建立基础基准测试和指标。FPE-val 的早期访问版本可在此获取：`https://github.com/thanhlecongg/FPEval`。

2 基准测试构建与评估流程

2.1 基准测试构建

数据收集。为了对大语言模型在函数式编程中的代码生成开展严谨的实证研究，我们从 LeetCode（LeetCode, Inc.，2026年）整理出一个全面的基准测试数据集。LeetCode 是广泛用于竞赛编程和技术面试的平台。我们从全部题目中筛除仅限高级会员的内容，选出 721 道公开的算法任务。这些任务分为三个难度级别：184 道简单、346 道中等和 191 道困难的编程任务。对于每道任务，我们通过 LeetCode 公共 API 和自定义 HTML 解析器提取问题描述、输入输出约束及公开测试用例。收集到的公开测试用例被纳入提示构建，随后用于初步正确性检查。图1展示了数据集（即 FPBench）的多样性。我们在 2021 年至 2025 年期间收集数据，从而能够捕捉算法设计和问题类型不断演变的趋势。FPBench 还包含难度级别各异的编程任务，使我们能够系统地评估模型在不同问题复杂度下的表现。

语言模板生成。评估函数式编程在 LeetCode 任务中的表现面临一项关键挑战：缺乏对 Haskell 和 OCaml 等广泛使用的函数式编程语言的支持。具体而言，LeetCode 没有为这些函数式编程语言提供任何模板，导致大语言模型难以理解输入输出和类型要求。为弥补这一缺口，我们实现了一个转译器，可根据 LeetCode 提供的 Python 模板，为 Haskell 和 OCaml 构建语法和语义均有效的起始模板。由于编程范式和类型系统存在根本差异，这项转换颇具挑战性，例如动态类型与静态类型、命令式控制流与函数式控制流之间的差异，以及许多语言结构缺乏一一对应关系。我们的转译器通过以下方式解决这些差异：(1) 将 Python 函数定义进行结构化转换，在 Haskell 和 OCaml 中生成带类型签名的定义，包括参数绑定和返回类型；(2) 推断嵌套及复合数据结构的类型（例如，将 List[List[int]] 转换为 `[[Int]]`）；(3) 规范化输入输出语义，使其符合函数式模式，同时保留原始任务规范的语义。

2.2 评估流程

测试验证。FPEval 流程首先评估生成代码的功能正确性。虽然以往的研究（Liu 等人，2024年）通常使用 LeetCode 上的公开测试用例，但这些测试套件的覆盖范围往往有限，可能无法发现细微的逻辑错误。为解决这一问题，我们为每项任务补充了一套私有测试用例。遵循 Huang 等人（Jain 等人，2025年）的方法，我们使用 GPT-4o 直接根据任务描述合成这些严格的测试用例，以检验边界条件、边界情况和退化输入（具体提示词见附录A）。值得注意的是，在函数式编程范式中执行这些测试也面临一项关键挑战。与支持灵活、动态执行的 Python 不同，Haskell 和 OCaml 等静态类型语言需要严谨的类型框架、严格的编译以及运行时隔离。为此，我们针对函数式编程语言开发了专用测试基础设施，使用基于标准库的测试模板和基于 Docker 的隔离运行环境，详情见附录B。

代码风格与可维护性评估。除了函数式正确性之外，代码风格和

生成代码的可维护性也是我们评估中的一个关键维度。解法可能满足所有测试用例，但在遵循函数式编程最佳实践方面仍然质量低下。此类代码会积累技术债务，使系统难以维护（Kruchten 等人，2012），甚至可能引入潜在的安全问题（Perry 等人，2023年）。为了系统地量化这一维度，FPEval 集成了广泛使用的静态分析工具，用于检查代码片段是否严格遵循函数式编程最佳实践。具体而言，我们使用 `HLint`和 `GHC`处理 Haskell，使用 `Dune` 和 `OCamlFormat` 处理 OCaml，并使用 `Scalastyle` 处理 Scala。这些工具所应用规则的分类示例见附录D。

2.3 大语言模型的选择与配置

为确保在不同能力和成本水平下进行全面评估，我们评估了 OpenAI GPT 系列中的三种大语言模型，包括 GPT-3.5-turbo（OpenAI，2022年）、GPT-4o（OpenAI，2023年）和 GPT5（OpenAI，2025年）。我们采用零样本提示策略，仅向模型提供自然语言任务描述和特定语言的起始模板（示例见清单 G.1）。我们将 GPT-3.5-turbo 和 GPT-4o 的生成参数温度 `= 0.7`设为。对于 GPT5，我们采用其默认温度 `= 1.0`，因为该模型架构固定了温度，以优化其底层采样逻辑。为确保与早期模型进行公平比较，我们通过设置推理_力度 `= none`，明确禁用了 GPT5 的推理能力。所有模型的最大值_令牌 `= 2048`均受到限制。对于每项任务，我们生成一个候选解法，并对大语言模型的响应进行后处理，剔除对话文本以提取代码块。

3 实验

3.1 研究问题1：大语言模型在函数式编程语言中的代码生成效果如何?

为回答这一研究问题，参照（Jain 等人，2025年；Liu 等人，2024年），我们通过 `pass@1`评估性能，即模型首次生成的解法满足所有测试用例的任务比例，并将未成功的生成结果分为：（1）编译错误（语法/类型不匹配）；（2）测试失败（逻辑错误）；以及 (3) 超时（执行限制）。

主要结果。表 1展示了所研究的大语言模型在四种语言——Haskell、OCaml、Scala 和 Java——上的表现。总体而言，我们观察到，随着模型不断进步，大语言模型在函数式编程上的表现显著提升；与 GPT-3.5 相比，GPT-5 生成的功能正确代码数量约增加了 3 倍。然而，纯函数式语言（Haskell 和 OCaml）与混合型或命令式语言（Scala 和 Java）之间的性能差距依然显著。

具体而言，GPT-3.5 在函数式语言中的能力有限，在 Haskell 和 OCaml 中的通过率仅为 14.5% 和 9.43%。相比之下，该模型在 Scala（19.28%）和 Java（22.19%）中的表现更好。尽管 GPT-4o 和 GPT5 在 Haskell（27.18%/42.34%）和 Scala（36.2%/52.16%）中的通过率有了显著提升，但与 Java 相比，它们在各语言中的表现仍存在明显差距。具体而言，GPT-4o 在 Haskell 和 OCaml 中的通过率分别比 Java 低约 16.5 和 7.5 个百分点。这种差距在 GPT-5 中甚至进一步扩大，分别接近 18 和 9 个百分点。这些结果表明，尽管大语言模型不断扩展规模并持续改进，函数式代码生成与命令式代码生成之间的性能差距依然显著。

编译错误。 有趣的是，我们发现，与命令式代码生成不同，大语言模型在函数式编程语言中经常生成无法编译、存在语法或类型错误的代码。例如，GPT-3.5 在 OCaml 和 Haskell 中的编译错误率分别高达 43.14% 和 25.5%。尽管 GPT-4o 将这两项错误率降至约 21%，但仍不及其在 Java 中展现出的稳定性（6.24%）。值得注意的是，GPT-5 成功将 OCaml 中的编译错误率降至 13.73%，并在 Java 中实现了低至 4.62% 的错误率，但在 Haskell 中却未能取得类似改进，其错误率仍高达 24.28%。这些发现表明，即使是最先进的模型，也难以遵循某些函数式语言严格的语法规则；这很可能是因为与 Java 和其他命令式语言相比，训练数据中这类代码的数量有限。

模型	指标	Haskell	OCaml	Scala	Java
GPT-3.5	Pass	14.15%	9.43%	19.28%	22.19%
	测试失败	50.21%	43.13%	65.46%	57.28%
	编译错误	25.52%	43.14%	14.01%	10.40%
	超时	10.12%	4.30%	1.25%	10.12%
GPT-4o	Pass	27.18%	36.20%	38.83%	43.69%
	测试失败	45.08%	39.11%	51.60%	43.00%
	编译错误	21.36%	21.08%	9.57%	6.24%
	超时	6.38%	3.61%	0%	7.07%
GPT-5	Pass	42.34%	52.16%	58.36%	61.14%
	测试失败	32.63%	33.38%	32.37%	27.48%
	编译错误	24.28%	13.73%	9.27%	4.62%
	超时	0.76%	0.72%	0%	6.76%

表 1：所研究的大语言模型在 Haskell、Ocalm、Scala 和 Java 中进行代码生成的有效性。

3.2 研究问题2：大语言模型在函数式编程语言中生成的代码存在哪些常见的代码风格和可维护性问题?

为回答这一研究问题，我们采用了一套针对特定语言的静态分析工具，评估大语言模型生成的代码在函数式编程中对代码风格和可维护性最佳实践的遵循情况。具体而言，我们使用 HLint 和 GHC 警告检查 Haskell，使用 Dune 和 OCamlFormat 检查 OCaml，使用 Checkstyle 和 PMD 检查 Java，并使用 Scalastyle 检查 Scala。我们将代码片段定义为“干净”，仅当它通过所有适用的检查且不存在任何违规时，才符合这一标准。

主要结果。 表 2 展示了在所研究的大语言模型生成的、通过第一个研究问题中所有功能测试用例的函数正确代码样本中，“整洁”代码样本所占的比例。

对于 OCaml，尽管 GPT-3.5 和 GPT-4o 在这些正确解答中的整洁代码比例仍相对较高（分别为 63% 和 61%），但 GPT-5 的这一比例明显下降至 44%。这表明，较早期的模型虽然更常出错，但在成功时往往会遵循函数式编程惯用法和可维护性方面的最佳实践。相比之下，GPT-5 的正确率显著更高（如表 1所示），但它似乎优先考虑功能正确性，而忽视这些重要的非功能属性，可能会采用更具命令式或更复杂的结构来满足测试用例。

对于 Haskell，所有受研究的大语言模型始终难以生成整洁代码。具体而言，GPT-5 和 GPT-4o 的函数式正确解法中，整洁代码的比例仅约为 45%–46%。结合研究问题1中较高的编译率，大语言模型生成代码的质量始终不佳，这表明 Haskell 的严格约束仍是当前大语言模型面临的持续挑战。即使是最新的大语言模型 GPT-5，也未能改善 Haskell 代码的风格和可维护性。虽然考虑到 Haskell 的复杂性和严格性，这些结果可以理解，但也凸显了大语言模型在函数式编程方面的局限性

有趣的是，我们的整体分析揭示了 GPT-5 在代码正确性与代码质量之间存在意料之外的权衡；这种现象不仅出现在 Ocalm 和 Haskell 中，也出现在 Scala 和 Java 中。具体而言，在研究问题1中可以看到，GPT-5 在所研究的四种编程语言中始终提高了 pass@1。然而，在本实验中，我们发现，与 GPT-4o 相比，GPT-5 在所有这些语言中的代码风格和可维护性始终较差。具体而言，整洁代码率在 Scala 中从 77% $\rightarrow$ 57%、在 Java 中从 63% $\rightarrow$ 51%、在 Haskell 中从 46% $\rightarrow$ 45%、在 Ocaml 中从 61% $\rightarrow$ 44% 下降。这一趋势表明，模型在优化更高通过率时，可能会牺牲代码风格和可维护性。这些行为与大语言模型中通常观察到的“奖励劫持”现象一致。在实践中，这些行为可能会对软件系统造成危害，因为它们会引入不易察觉的技术债务；这类代码虽然能通过功能测试验证，却会增加维护成本。

常见问题。 为推动未来改进，我们针对低质量代码开展了半自动化分析，考察其在以下方面的表现：

模型	语言			
	Haskell	OCaml	Scala	Java
GPT-3.5	22%	63%	61%	58%
GPT-4o	46%	61%	77%	63%
GPT-5	45%	44%	57%	51%

表 2：所研究的大语言模型生成的功能正确输出中整洁代码的比例

静态分析工具检测出的函数式编程最佳实践。我们首先手动识别反复出现的代码风格和可维护性问题，并将其对应到静态分析工具的特定诊断信息；随后，我们编写了一个脚本，自动解析整个数据集中的工具日志，以量化这些问题的普遍程度。

在 Haskell 中，最常见的问题是不符合惯用风格的模式，例如不必要的 lambda 表达式（37×）、遗漏 η 归约（36×）以及多余的括号。GHC 还会针对未使用变量和非穷尽模式匹配发出警告。在 OCaml 中，大多数问题与代码可读性有关，由 ocamlformat 检出，例如多余的分号和括号。我们还观察到一些严重问题，包括类型错误，以及对不可变数据结构进行非预期修改，不过这些问题出现的频率较低。总体而言，尽管这些问题不会损害大语言模型生成代码的功能正确性，但它们违背了函数式编程中公认的最佳实践，并可能增加未来的维护风险。

最值得关注的问题出现在 Scala 中。具体而言，我们发现最常见的问题是使用 return（236 次）；这在符合 Scala 惯用风格的代码中并不提倡，因为它表明使用了命令式范式。其他违规情况还包括魔数、冗长方法以及不一致的过程定义。命令式结构持续出现，这表明大语言模型可能未能充分利用 Scala 的函数式特性，反而生成了类似 Java 风格的命令式逻辑；这种写法在 Scala 中虽被允许，却不受提倡。这些观察引出了一个更广泛的问题：大语言模型真的会生成函数式代码吗？还是只是在代码中嵌入命令式代码来“投机取巧”地获取奖励，从而提高功能正确性？

大语言模型生成的函数式代码中的命令式偏向。尽管我们明确要求大语言模型使用函数式编程语言生成代码，但观察到其输出往往体现出命令式编程偏向。这种现象在三种语言中均有出现，但在 Scala 中尤为明显，例如使用`return`、可变变量以及其他类似 Java 的模式。在 Haskell 和 OCaml 中，我们也观察到类似的偏向，例如在纯函数特性中使用可变引用、原地更新或 IO 效应，这与这些语言声明式且不可变的特性相冲突。这些观察表明，大语言模型并未完全遵循函数式编程的预期。相反，它们倾向于从训练语料库中占主导地位的命令式结构进行泛化，因此生成的输出虽然语法正确，却往往违背函数式编程的核心原则：纯洁性、不可变性和声明式控制流（代表性示例见附录E）。

为验证这些观察结果的普遍性，我们找出了函数式编程原则所不鼓励的常见命令式编程模式。具体而言，我们通过基于关键字的模式匹配，识别命令式和可变构造（例如，Scala 中的可变集合、变量声明、显式循环；Haskell 中的手动 `let` 绑定、过多使用 `if-then-else`；OCaml 中的可变引用和赋值）。随后，我们对 GPT-3.5、GPT-4o 和 GPT-5 在 Scala、Haskell 和 OCaml 中生成的代码进行了全面分析，以找出包含这些命令式模式的代码。表 3 展示了分析结果。可以看到，所有受研究的大语言模型在所有受研究的函数式编程语言中生成的代码都表现出显著的命令式偏向。值得注意的是，GPT-5 中命令式模式的出现比例最高：Scala、Haskell 和 OCaml 中分别有 94% 、 88% 和 80% 的生成代码包含此类构造。与前代模型 GPT-3.5 和 GPT-4o 相比，这些比例大幅上升。结果表明，这是一种“奖励破解”现象：尽管 GPT-5 似乎更擅长解决复杂问题（此前讨论的较高通过率便是佐证），但它越来越依赖命令式捷径来确保功能正确性，而非编写纯函数式编程代码。

3.3 研究问题3：自修复机制在提高大语言模型生成的函数式代码的正确性、代码风格和可维护性方面有多大效果？

已有研究（Liu 等人，2024年）表明，大语言模型可以通过反馈循环自我修复错误，

表 3：含命令式模式的大语言模型生成代码的比例

语言	GPT-3.5	GPT-4o	GPT-5
Scala	94%	88%	94%
Haskell	53%	63%	88%
OCaml	57%	42%	80%

本研究问题探讨大语言模型的自我修复能力能否提升其在函数式编程语言中生成内容的函数正确性、代码风格和可维护性。为此，我们考察 ChatGPT 在收到明确的修复提示后改进自身代码的能力。

沿用（Liu 等人，2024年）的做法，我们考虑以下两种修复方法：（1）简单修复。在这种方法中，我们向大语言模型提供一条高层系统指令，以纠正其先前生成的解法。该指令的表述见附录G.1。这一基线体现了最简修复设置：要求模型重新生成更好的解法，除了任务要求和函数式编程提示之外，不提供任何结构化指导。（2）指令引导修复。在这种方法中，我们采用了更具结构化的方式，结合静态分析反馈和针对常见问题精心编写的指令。具体而言，我们使用静态分析和测试验证来识别函数正确性、代码风格和可维护性方面的潜在问题。随后，我们将生成代码中观察到的常见质量问题分类，例如编译错误、类型错误、命令式编程风格，并针对每类常见问题向模型提供明确的修复指令。这些指令针对各函数式编程语言（Haskell、OCaml 和 Scala）量身定制，并将常见错误类别直接对应到可操作的修复指令。这些指令见附录7、8、9中，附录G。

结果。 表 4 详细列出了三种设置下功能正确的解法数量：初始生成、简单修复和指令引导修复。总体而言，我们可以看到，在所有模型和语言中，具备自我修复能力的大语言模型均能通过这两种研究过的修复方法，持续提升其解法的功能正确性。进一步观察可以发现，修复方法与解法质量之间存在微妙的相互作用。尽管指令引导修复在某些情境（例如，GPT-3.5 在 Scala 中： $118 \rightarrow 139$）显著提升了性能，但就原始通过率而言，它并非总是优于 简单修复 。例如，在 Haskell 和 OCaml 中使用 GPT-5 时，简单修复 的正确性略高（例如，Haskell：301 对 295）。分析生成代码的代码风格和可维护性，可以解释这一现象。如表 5所示，在大多数情况下，指令引导修复产生整洁代码的比例最高，也就是说，这些代码最符合函数式编程实践，其表现显著优于简单修复。这表明，大语言模型需要在代码风格、可维护性与功能正确性之间权衡。简单修复 允许模型不择手段地优先通过测试用例，甚至保留或引入命令式捷径来修复错误。与此同时，指令引导修复通过施加严格的函数式编程约束，增加了生成任务的难度。模型试图找到一种既正确，又在代码风格和可维护性方面具有高质量的解法。如果模型无法合成高质量解法，就可能无法生成有效代码；因此，尽管成功解法的代码风格和可维护性更好，通过率仍可能略有下降。

值得庆幸的是，指令引导修复与简单修复之间的性能差距微乎其微。例如，在 Haskell 中，使用 GPT-5 的简单修复比指令引导修复仅多生成了 6 个正确解法。这些结果表明，指令引导修复能够以极小的 pass@1 比率代价提供更高质量的代码。它们还表明，如果明确向大语言模型提供函数式编程的最佳实践，大语言模型就越来越能够在实现功能正确性的同时，兼顾代码风格和可维护性。这些发现表明，应将这些实践更明确地纳入大语言模型训练和推理中，以便生成既具备功能正确性，又在代码风格和可维护性方面质量上乘的代码。

4 相关工作。

代码生成基准测试。 人们提出了许多数据集，用于评估大语言模型（LLM）的代码生成能力。早期基准测试主要聚焦于 Python 代码生成，例如 Hu-

表 4：采用不同修复策略后，各语言和模式下正确解法的数量 ls.

设置	GPT-3.5			GPT-4o			GPT-5		
	Haskell	OCaml	Scala	Haskell	OCaml	Scala	Haskell	OCaml	Scala
代码生成	93	68	118	196	261	280	287	376	420
简单修复	**102**	78	117	204	**272**	**297**	**301**	**396**	431
指令修复	98	**85**	**139**	**209**	266	288	295	391	**433**

表 5：所研究的大语言模型生成的正确整洁代码输出所占比例。来源、相似度报告和实例表示分别表示原始生成代码、经简单修复修复的代码，以及经指令引导修复修复的代码所获得的结果。

模型	方法	Haskell	OCaml	Scala
GPT-3.5	来源	22%	63%	61%
	相似度报告	46%	78%	75%
	实例表示	**61%**	**87%**	**79%**
GPT-4o	来源	46%	61%	**77%**
	相似度报告	48%	**92%**	61%
	实例表示	**56%**	78%	66%
GPT-5	来源	45%	44%	57%
	相似度报告	58%	87%	**62%**
	实例表示	**63%**	**91%**	60%

manEval（Chen 等人，2021）和 MBPP（Austin 等人，2021）。近年来，研究社区转向通过 MultiPL-E（Cassano 等人，2023年）和 HumanEval-XL（Peng 等人，2024年）等数据集开展多语言评估，尽管这些工作仍主要侧重于命令式编程范式（例如 C++、Java 和 Go）。其他研究则考察了面向规范的语言，例如 F*（Lahiri，2024年）、Dafny（Loughridge 等人，2024年）和 JML（Le-Cong 等人，2025a）。不同于这些已有研究，FPEval 专门针对函数式编程语言；这一范式在当前基准测试文献中仍明显缺乏代表性。此外，我们还超越了传统的函数正确性评估，对代码风格和可维护性进行了全面评估。这种整体性方法使我们能够深入揭示函数式编程中大语言模型生成代码的整体质量与技术债务。

函数式编程中的自动化。 文献中也有多项工作研究了函数式编程的自动化。值得注意的是，Gissurarson 等人提出了 PropR（Gissur- arson 等人，2022年），这是一种基于属性的自动程序修复（APR）框架，旨在利用基于属性的测试和类型驱动合成来修复 Haskell 程序中的错误。最近，Van Dam 等人（Van Dam 等人，2024年）针对 Haskell 代码补全对 UniXCoder 和 CodeGPT 进行了微调，证明面向特定语言的模型在这一领域能够显著优于通用大语言模型。不同于这些主要局限于 Haskell 生态系统的研究，我们将研究范围扩展到多种函数式编程语言，包括 Haskell、Scala 和 OCaml。此外，我们并未专注于对较小架构进行微调，而是对最新的前沿大语言模型开展了全面评估，包括 GPT-3.5、GPT-4o 和 GPT-5。这样，我们便能够评估这些最先进模型在多种函数式语言上的能力，从而更广泛地审视当前人工智能驱动的软件工程在函数式编程领域的边界。

5 结论

本文介绍了 FPEval，这是首个全面评估框架及实证研究，旨在评估大语言模型在函数式编程范式下的代码生成能力。FPEval 采用了双重评估方法，结合了严格的测试套件和静态分析工具，使我们不仅能够评估大语言模型生成代码的功能正确性，还能评估其代码风格和可维护性。

我们的研究发现，当前大语言模型的表现存在显著的“范式差距”：与混合式或命令式基线相比，这些模型在 Haskell 和 OCaml 等纯函数式编程语言中的错误率要高得多。此外，我们的分析表明，即使大语言模型实现了功能正确性，也经常会生成违反函数式编程最佳实践的低质量代码，从而削弱函数式编程固有的安全性和可维护性优势。

尽管存在这些局限性，我们还发现，

随着模型不断发展，大语言模型在函数式编程中的代码生成能力也会提升。此外，我们的结果表明，在静态分析提供的详细反馈和定制指令的指导下，大语言模型有望自我修复其错误。总体而言，我们希望 FPEval 能够成为社区的可靠基准测试，并提供实证见解，以指导大语言模型在函数式编程领域的发展。

6 局限性

本研究的局限性如下：

首先，我们的评估受研究模型范围有限的制约。受计算资源和经费限制，我们的评估仅关注三个大语言模型：GPT-3.5、GPT-4o 和 GPT-5。这一狭窄的选择可能会影响研究发现对更广泛的开源模型或其他专有模型的普适性。为尽量降低这一风险，我们采取了以下具体措施：（1）选择来自当前行业领先者 OpenAI 的模型，以确保研究结果反映该领域的最先进能力。（2）选择 OpenAI GPT 系列三个不同世代的模型，以便纵向分析大语言模型的演进和性能变化轨迹。（3）纳入实验时可用的最先进模型 GPT-5，确保我们的基准测试能够体现当前大语言模型能力的上限。

其次，本研究主要关注大语言模型在函数式编程中的代码生成能力。虽然生成函数正确且高质量的代码是实际软件工程的基本要求，但它仅代表开发者工作流程的一部分。我们的评估未涵盖其他必要的软件开发任务，例如缺陷修复或代码优化。因此，我们的研究结果反映了模型在初始合成方面的有效性，但未必能推广到其在迭代式软件演化或调试中的表现。不过，我们希望本研究能够成为大语言模型时代函数式编程的基础性基准测试，为未来开展更广泛的软件工程自动化研究提供必要的基线。

最后，我们的基准测试主要由来自教育和竞赛编程平台的问题构成。尽管这些任务为评估算法正确性提供了严格的约束，但它们可能无法充分代表工业规模的函数式编程。现实世界的软件涉及复杂的模块间依赖关系、遗留架构约束，以及特定的领域驱动设计模式，而这些通常不会出现在独立的编程挑战中。在未来的工作中，我们将考虑扩展数据集，纳入大型开源函数式编程代码库。

参考文献

Peter Achten。2013。为什么函数式编程对我很重要。载于《函数式代码之美》。

Anthropic。2025年。Claude Code：智能体式编程工具。技术发布。

Anysphere。2023年。Cursor：AI 优先的代码编辑器。访问日期：2025-11-19。Jacob Austin、Augustus Odena、Maxwell Nye、Maarten Bosma、Henryk Michalewski、David Dohan、Ellen Jiang、Carrie Cai、Michael Terry、Quoc Le 等 1 人。2021年。使用大语言模型进行程序合成。*arXiv* 预印本 *arXiv:2108.07732*。Federico Cassano、John Gouwar、Daniel Nguyen、Syd- ney Nguyen、Luna Phipps-Costin、Donald Pinckney、Ming-Ho Yee、Yangtian Zi、Carolyn Jane Anderson、Molly Q Feldman 等 1 人。2023年。MultiPL-E：一种用于神经代码生成基准测试的可扩展多语言方法。*IEEE* 软件工程汇刊，49(7):3675–3691。Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman、Alex Ray、Raul Puri、Gretchen Krueger、Michael Petrov、Heidy Khlaaf、Girish Sastry、Pamela Mishkin、Brooke Chan、Scott Gray 等 39 人。2021年。评估在代码上训练的大语言模型。Angela Fan、Beliz Gokkaya、Mark Harman、Mitya Lyubarskiy、Shubho Sengupta、Shin Yoo 和 Jie M. Zhang。2023年。用于软件工程的大语言模型：综述与开放问题。载于*2023*年 *IEEE/ACM* 国际软件工程大会：软件工程的未来（*ICSE-FoSE*），第 31–53 页。Matthew Farwell 和 Scalastyle Contributors。2024年。Scalastyle：Scala 风格检查器。`http://www.scalastyle.org`。访问日期：2024年5月20日。Matthías Páll Gissurarson、Leonhard Applis、Annibale Panichella、Arie Van Deursen 和 David Sands。2022 年。PropR：基于属性的自动程序修复。载于第 *44* 届国际软件工程大会论文集，第 1768–1780 页。

John Hughes。1989年。为什么函数式编程很重要。计算机杂志，32:98–107。

Naman Jain、King Han、Alex Gu、Wen-Ding Li、Fanjia Yan、Tianjun Zhang、Sida Wang、Armando Solar- Lezama、Koushik Sen 和 Ion Stoica。2025年。Live- codebench：面向代码的大语言模型的整体性、无污染评估。载于第十三届国际学习表征会议。

Majeed Kazemitabaar、Justin T.H. Chow、Carl Ka To Ma、Barb Ericson、David Weintrop 和 Tovi Gross- man。2023年。研究人工智能代码生成器对支持初学者学习入门编程的影响。*2023*年计算机系统中的人为因素会议论文集。

Philippe Kruchten、Robert L. Nord 和 Ipek Ozkaya。2012年。技术债务：从隐喻到理论与实践。*IEEESoftware*，29(6):18–21。

Shuvendu K. Lahiri。2024年。评估大语言模型驱动的用户意图形式化在验证感知语言中的应用。*2024*年 计算机辅助设计中的形式化方法 *(FMCAD)*，第 142–147 页。

Thanh Le-Cong、Bach Le 和 Toby Murray。2025a。大语言模型能否推理程序语义？对大语言模型进行形式规范推断的全面评估。收录于 第 *63* 届计算语言学协会年会论文集（第 *1* 卷：长篇论文），第 21991–22014 页，奥地利维也纳。计算语言学协会。

Thanh Le-Cong、Bach Le 和 Toby Murray。2025b。采用语义引导补丁生成的内存高效大语言模型程序修复。预印本，arXiv:2410.16655。

LeetCode, Inc. 2026年。LeetCode：全球领先的在线编程学习平台。访问日期：2026-01-05。

Yujia Li、David Choi、Junyoung Chung、Nate Kushman、Julian Schrittwieser、Rémi Leblond、Tom Eccles、James Keeling、Felix Gimeno、Agustin Dal Lago、Thomas Hubert、Peter Choy、Cyprien de Mas- son d'Autume、Igor Babuschkin、Xinyun Chen、Po- Sen Huang、Johannes Welbl、Sven Gowal、Alexey Cherepanov等另外7人。2022年 达到竞赛级别的AlphaCode代码生成。 *Science*，378(6624):1092–1097。

Yue Liu、Thanh Le-Cong、Ratnadira Widyasari、Chakkrit Tantithamthavorn、Li Li、Xuan-Bach D. Le 和 David Lo。2024年 优化 ChatGPT生成的代码：代码质量问题的特征分析与缓解。 *ACM*软件工程与方法学汇刊，33(5)。

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin 和 Max Tegmark。2024年。Dafnybench：一个用于形式化软件验证的基准测试。机器学习研究汇刊，2025年。

Neil Mitchell。2024年。HLint：Haskell 源代码建议工具。https://github.com/ndmitchell/hlint。访问日期：2024年5月20日。OCamlFormat Developers。2024年。OCamlFormat：OCaml 代码自动格式化工具。

https://github.com/ocaml-ppx/ocamlformat。访问日期：2024年5月20日。OpenAI。2022年。介绍 ChatGPT。访问日期：2025- 12-29。OpenAI。2023年。GPT-4 技术报告。预印本，arXiv:2303.08774。OpenAI。2025年。介绍 GPT-5。访问日期：2025-12- 29。

Qiwei Peng、Yekun Chai 和 Xuhong Li。2024年。Humaneval-xl：用于跨语言自然语言泛化的多语言代码生成基准测试。载于*2024*年计算语言学、语言资源与评估联合国际会议论文集（*LREC-COLING 2024*），第8383–8394页。

Neil Perry、Megha Srivastava、Deepak Kumar 和 Dan Boneh。2023年。用户使用人工智能助手时会编写更多不安全代码吗？载于*2023*年 *ACM SIGSAC* 计算机与通信安全会议论文集，CCS '23，第2785–2799页。ACM。

James Prather、Brent N. Reeves、Paul Denny、Brett A. Becker、Juho Leinonen、Andrew Luxton-Reilly、Garrett B. Powell、James Finnie-Ansley 和 Eddie Antonio Santos。2023年。"它居然知道我想要什么，这感觉真奇怪"：编程初学者使用 Copilot 的可用性与交互。*ACM*计算机与人机交互汇刊，31:1–31。

Baishakhi Ray、Daryl Posnett、Vladimir Filkov 和 Premkumar T. Devanbu。2014年。GitHub 中编程语言与代码质量的大规模研究。第 *22*届 *ACM SIGSOFT* 软件工程基础国际研讨会论文集。

Tim Van Dam、Frank Van der Heijden、Philippe De Bekker、Berend Nieuwschepen、Marc Otten 和 Maliheh Izadi。2024 年。研究语言模型在函数式编程语言中完成代码的性能：Haskell案例研究。载于 *2024*年 *IEEE/ACM*首届人工智能基础模型与软件工程国际会议论文集，第91–102页。

Chunqiu Steven Xia、Yuxiang Wei 和 Li ngming Zhang。2023年。 大型预训练语言模型时代的自动化程序修复。载于 *2023*年 *IEEE/ACM* 第 *45*届软件工程国际会议（*ICSE*），第1482–1494页。

Bowen Xu、Thanh-Dat Nguyen、Thanh Le-Cong、Thong Hoang、Jiakun Liu、Kisub Kim、ChenGong、Changan Niu、Chenyu Wang、Bach Le 和 David Lo。2023年。我们是否已准备好拥抱用于软件问答的生成式人工智能？*2023*年*IEEE/ACM*第*38*届自动化软件工程国际会议（*ASE*），第1713–1717页。

A 用于合成测试用例生成器的提示

为自动为每项编程任务构造高质量且多样化的测试用例，我们使用 GPT-4o 直接根据自然语言任务描述合成 Python 输入生成代码。提示模型扮演专家竞赛程序员，并在构造测试数据时严格遵循所有题目约束。以下是用于生成测试用例合成代码的确切提示模板。助手会收到 (i) 任务描述和 (ii) Python模板框架，并被要求输出一个名为 construct_inputs() 的 Python 函数，该函数返回 10 个多样化输入，但不打印它们。

输入生成提示（" system " , " 你是一位 Python 竞赛程序员，你的目标是为编程竞赛问题构造输入生成器，以便进行测试。你将编写相关的生成器，最后构造一个 `construct_inputs `函数，该函数会返回从生成器中采样得到的多样化输入列表。请务必严格遵守问题描述中的说明和约束。\ n" ） , (" human " , " ### 说明：\ n" " - 从问题描述中确定约束。\ n" " - 构造一个 Python 函数 `construct_inputs ()` ，用于生成多样化的测试用例。\ n " " - 确保所有生成的值都严格遵守约束。\ n\n" " ### 示例参考：\n" " import numpy as np \n" " def random_input_generator (weight_min , weight_max , size_min , size_max) :\ n" " weights_size =np . random . randint (size_min , size_max +1) \n " " weights =np . random . randint (weight_min , weight_max , size= weights_size). tolist () \n" " k =np . random . randint (1 , len (weights) +1) \n" " return weights , k\n\n" " def construct_inputs () :\ n" " inputs_list = [ ]\ n" " ## 随机输入 \n" " for i in range (8) :\ n" " inputs_list . append (随机_输入_生成器 (1 , 50 , 1, 10)) # 根据约束调整 \n" " return inputs_列表 \n\n" " ### 任务：\n" "{ task_ 描述 }\n" "{ python_ 模板 } \n" " 构造一个随机输入生成器。按照上述示例所用的格式，返回一个构建多样化输入的单一函数，命名为 `构造_输入 `。不要打印输入_列表。\n"），

B 测试执行。

为确保能够执行公开和私有测试用例，我们为函数式语言开发了一套专用测试基础设施，分为四个阶段。首先，FPEval 使用标准库（例如，OCaml 的 OUnit2、Haskell 的 HUnit）为每个问题生成单元测试。其次，它构建可编译测试模板，并将大语言模型生成的解答注入其中。第三，该基础设施通过在 Docker 隔离环境中编译并执行生成的文件，验证公开和私有测试用例。最后，它会捕获执行日志，严格区分语法错误、静态类型不匹配、运行时故障和功能错误。该基础设施还会自动处理数据格式（例如，布尔值、列表、字符串）的差异，支持基于文档字符串的类型提取，并根据类型对测试输入和预期输出进行格式化。这种自动化对于将评估扩展到数百个问题，并确保对函数式编程语言中大语言模型效能进行高质量评估至关重要。

C 数据污染。

我们还研究了数据污染对所研究大语言模型的潜在影响。我们特别选择 GPT-4o 进行这项分析，因为其知识截止日期使评估日期前后的任务分布较为均衡。为量化数据污染的影响，我们根据发布时间将编程任务划分为六个不同的时间段：(1) 2022年9月之前，(2) 2022年10月—2023年3月，(3) 2023年4月—9月，(4) 2023年10月—2024年3月，(5) 2024年4月—9月，以及 (6) 2024年10月之后。图 2 展示了以下模型的通过率：

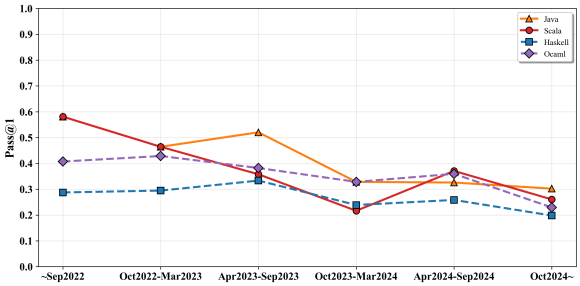

图2：GPT-4o 的 pass@1 表现

GPT-4o 在四种所研究的编程语言中的表现。可以看出，GPT-4o 在知识截止日期之后，Java 和 Scala 的性能显著下降：Java 从 52.0% 降至 32.8%，Scala 则从 35.8% 降至 21.6%。这一结果表明预训练数据中可能存在数据污染。相比之下，该模型在 OCaml 和 Haskell 中始终保持较低且稳定的准确率，这与它们在预训练语料库中的占比较低相一致。

D 静态分析规则分类

为确保代码整洁度指标的可复现性，我们对所采用的静态分析规则进行了分类。FPEval。表6详细说明了各语言所使用的具体配置。这些规则旨在检测两类主要违规行为：

- **命令式偏向：**语法上有效、但违反函数式编程原则的结构（例如使用可变变量、显式循环或过程式控制流）。
- **非惯用用法：**未能充分利用该语言表达能力的代码。

对于**Haskell**，我们使用 HLint 检测冗余结构并建议采用高阶替代写法。对于**OCaml**，我们直接调用编译器，并使用 -warn-error + a 标志强制执行严格的静态语义。对于**Scala**，我们采用**默认** Scalastyle 配置，该配置体现了社区对惯用代码的标准。不过，为优先保证语义正确性而非纠结于格式细节，我们明确禁用了所有与空白相关的规则（例如制表符检查、间距对齐），严格专注于逻辑和范式违规。

E 命令式代码模式的代表性示例

1. 可变变量

- **命令式风格（不正确）：**

```
变量声明 计数 = 0
计数 = 计数 + 1


// 函数式风格（推荐）

val 计数 = 0
val 新计数 = 计数 + 1
```

在函数式编程中，变量理想情况下应是不可变的（Scala 中的 val，OCaml 中不重新赋值的 let）。然而，GPT 经常生成使用可变绑定（变量声明）或重新赋值操作符的代码。

2. 命令式循环

- **命令式风格（错误）：**

```
循环 (i <-0 到 n) {
  打印 (i)
}
```

- **函数式风格（推荐）：**

```
(0 到 n)。foreach( 打印 )
```

- **或者递归（在 OCaml 中）：**

```
递归定义 打印_数字 i n =如果 i <
n 则 ( 打印_整数 i; 打印_换行 ();
打印_数字 (i + 1) n)
```

GPT 经常默认使用 for、while 或 do while 循环，这些循环在命令式语言中很常见，但在函数式编程中不太符合惯用写法。函数式解法会使用高阶函数或递归。

3. 数据结构的原地修改

- **命令式风格（不正确）：**

```
val arr =数组 (1, 2, 3) arr (0)
 = 42
```

- **函数式风格（推荐）：**

```
val arr =向量 (1, 2, 3)val newArr =arr .
updated (0, 42)
```

GPT 往往不会将数据结构视为不可变，而是直接修改数组或列表，尤其是在性能相关场景中。这违背了函数式原则，即应构造新值，而非修改现有值。

表6：FPEval 中采用的静态分析规则分类

语言	Tool	规则 / ID	描述与依据
Haskell	HLint	使用 *map / fmap*	建议将显式递归或列表推导式替换为高阶函- 数（例如 map），从而- 强化符合惯用法的函数式编程风格。
		冗余 *do*	标记包含单个表达式的 do 代码块，从而- 鼓励使用纯函数表达式，而非单子式
		使用 *let*	顺序执行。 建议使用 let 绑定，而非单子生成器
		η 归约	(<-) 表示纯值。 通过移除冗余的 函数参数。
	GHC	- *Wall*	启用所有警告，严格标记被遮蔽的变- 量以及命令式逻辑移植中常见的未使用 模式匹配。
Scala	Scalastyle (默认配置文件)	变量检查器	*(*标准*)*检测可变变量（变量声明），并推动使 用不可变值（val）。
		返回检查器	（标准）检测显式的 return 关键字，确保 表达式导向的控制流。
		空值检查器	（标准）禁止使用 null，确保类型安全的错误 处理通过 Option 实现。
		圈复杂度	*(*标准*)* 惩罚复杂度较高的方法 （> 10），通常由嵌套式命令式 if-else
		PublicMethodsHaveType	区块 · *(*标准*)* 要求公共方法具有显式返回类型 方法，确保函数式 API 中的类型清晰明确。
OCaml	编译器 (ocaml -c)	*-warn-error +a*	**严格模式：**将所有警告视为致命错误。这 确保生成代码严格遵循 OCaml 静态语义。
		警告 *8*	部分匹配：通过严格模式强制执行。拒绝 未涵盖所有情况的模式匹配。
		警告 *26*	未使用变量： 通过严格模式强制执行。拒绝 过程式逻辑生成经常遗留下来的未使用的绑定 。
	OCamlFormat	格式化	强制采用社区标准缩进，阻止—— 加剧命令式编程中常见的深层嵌套结构 code.

4. 使用 null 而非 Option 类型

- **命令式风格（不正确）：**

```
def findUser(name: 字符串): 用户 ={if(db.
contains(name))db(name)else null}
```

- **函数式风格（推荐）：**

```
def findUser(name: 字符串): Option[用户] =
db.get(name)
```

函数式编程不鼓励使用 null，而是倾向于采用 Option、Maybe 或模式匹配等更安全的替代方案。

F 大语言模型生成的函数式代码中的命令式偏向

尽管提示大语言模型使用 Haskell、OCaml 和 Scala 等函数式编程语言生成代码，但其输出往往体现出命令式编程偏见。本节将通过典型示例和定量证据说明这一现象。

示例 1：Scala

- **命令式风格（错误）：**

```
导入 scala.collection.mutable
```

```
def 交替子数组计数(数字:
    列表[整数]): Int = {
    val n = 数字.长度
    // 可变 变量 'n'
    变量声明 计数 = 0
    // 可变变量（变量声明）
    变量声明 左侧 = 0
    // 可变变量（变量声明）
    变量声明 右侧 = 0
    // 可变变量（变量声明）
    val 集合 =可变.哈希集合[整数]()
    // 使用可变集合(
        哈希集合)
    循环 (右侧 < n){
    // 命令式循环
        集合.add(数字(right))
        // 副作用: 可变集合
        当 (集合.size > 2) {
            集合.remove(数字(left))
            // 副作用: 变异中
                set
            左侧 += 1
            // 变异
        }
        计数 += 右侧 - 左侧 + 1
        // 变异
        右侧 += 1
        // 变异
    }

    计数
}
```

问题：计算交替子数组

- **函数式风格（首选）：**

```
def 交替子数组计数(数字:
    列表[整数]): Int = {
  数字.索引.左折叠((0, 0, 映射.
      空[整数, 整数], 0)){
    case ((左, 右, 频率映射,
        计数), r) =>
      val num = 数字(r)
      val 更新后的频率 = 频率映射。
          更新（数字, 频率映射。
          获取或默认值（数字, 0) + 1)

      // 滑动 left 指针，同时
          超过 2 个不同的
          元素
      val (新左值, 新频率) =
          迭代器.iterate((左值,
          更新后的频率)){
        case (l, 频率映射) =>
          val 左侧数字 = 数字(l)
          val 新计数 = 频率映射(
              左侧数字) - 1
          val 新频率映射 = if (新计数
              == 0) fmap - 左侧数字
            else 频率映射.更新(
              左侧数字, 新计数)
          (l + 1, 新频率映射)
      }.丢弃直到{匹配(_, 频率映射) =
          >频率映射.size > 2 }.next()

      val 新计数 = 计数 +(r -
          newLeft + 1)
      (newLeft, r + 1, newFreq,
          新计数)
  }._4
}
```

函数式版本使用不可变的‘映射’代替可变的‘哈希集合’，消除了所有‘变量声明’，并通过‘左折叠’以状态传递取代重新赋值。它还采用‘迭代器迭代’，以受递归启发的风格表达滑动窗口逻辑。

示例 2：OCaml

- **命令式风格（不正确）：**

```
let 最短时间 (技能: int 列表) (法力:
    int 列表) : 整数 =
  let n =列表.length 技能 in
  let m =列表.长度 mana in
  令 动态规划 =数组.创建_矩阵 n m 0
    in
  (*使用可变数组 *)
  动态规划.(0).(0) <- 技能.(0) * 法力.(0)
    ;
  (* 使用变异 "<-" *)
  对于 j = 1 to m - 1 do
  (* 命令式 loop *)
    动态规划.(0).(j) <- max 动态规划.(0).(j-1) (
      技能.(0) * 法力.(j));
    (* 变异 内部 循环 *)
  完成;
  for i = 1 to n - 1 do
  (* 命令式循环*)
    动态规划.(i).(0) <- 动态规划.(i-1).(0) +
      技能.(i) * 法力.(0);
    (* 变异 *)
    循环 j = 1 to m - 1 do
    (* 嵌套 循环 *)
      动态规划.(i).(j) <- max 动态规划.(i).(j-1)
        动态规划.(i-1).(j) + 技能.(i)
        * 法力.(j);
      (* 变异 *)
    完成;
  完成;
  动态规划.(n-1).(m-1)
```

问题：求出酿造药水所需的最短时间

- **函数式风格：**

```
let 最短时间 (技能: int 列表) (法力:
    int 列表) : 整数 =
  let rec 构建_动态规划 prev_行 i =
    匹配 列表.第 n 个_选项 技能 i with
    | 无 -> 上一行
    | 某个 s ->
      令 行 =
        列表.映射索引 (函数 j m ->
          let 来自_左侧 = 如果 j = 0
            then 无 否则 某个(
            列表.第 n 个 row (j - 1))
            in
          let 来自_顶部 =列表.第 n 个
              上一行 j in
          令 单元格_值 =
```

```
                匹配 自_左开始
                | 无 -> 自_顶端 +s *
                      m
                | 有 l ->最大值 l
                      自_顶端 +s * m
            in
              单元格_值
        ) 法力
    in
    构建_动态规划 行 (i + 1)
in
匹配 技能 with
| [] -> 0
|s0 :: 剩余 ->
  设 第一个 _ 行 =
    snd ( 列表 . 折叠_left (fun (
        最大值_val, acc ) m ->
      let curr = 最大值 最大值_val (s0 *
          m) 于
      ( 当前, acc @ [当前 ])
    ) ( 最小_整数, []) 法力 )
  in
  令 最终_动态规划 =构建_动态规划
      第一_行 1 in
  List . hd ( List . rev final_dp )
```

在这个版本中：

- 动态规划矩阵通过不可变的方式，使用列表.折叠_left 和纯函数重建。
- 没有命令式循环或变异（<- 已不复存在）。
- 递归和列表变换保持了纯洁性和不可变性。

示例 3：Haskell

- **命令式风格（不正确）：**

```
导入 数据 . 列表
最大操作 :: [整数 ] → Int
最大值操作 数字 = 辅助函数 数字 0
  其中
    辅助函数 数字 计数
      | 长度 数字 < 2 = 计数
      | 否则 =令 得分 = 总和
          $ 取 2 个数字
            in 辅助函数 (删除 (head
                数字) $ 删除 (head
                  $ reverse 数字)
                数字) (计数 + 1)
```

问题：相同得分的最大操作次数 ii

- **函数式风格（首选）：**

```
最大操作次数 =执行 0其中

    go count (x: xs@(_: _)) =令 mid
     = init xs于 go (count + 1)
    midgo count _ = count
```

G 提示设计与特定语言的自我修复指令

本附录详细介绍了我们实验中使用的提示策略和特定语言的修复映射。内容分为两部分：(1) 用于代码生成和自我修复的提示模板；(2) 针对各函数式语言（Haskell、OCaml 和 Scala），将常见的编译器警告或错误映射到相应自修复指令。

G.1 提示模板

第一部分介绍了用于基线代码生成和自我修复设置的确切提示。

用于生成代码的提示

```
("系统",
 "你是一位{lang}程序员。 你将会 获得一个 问题（问题
 规范）并 will 生成一个正确的{lang}程序，使其符合
 规范并 通过所有 测试。 你绝不会 return 任何内容，除了程序
  以及必要的导入。",
),
("人类" ,
 "### 问题：{description}"
 "### 格式：你将 使用 以下起始代码编写
  问题的解法 问题 并将你的代码放在以下
 分隔符之间。\n{template}\n"
 "### 答案：（使用提供的 格式，并用反引号括起来）\n "
 )
```

自修复提示基线

```
("system","你是一名乐于助人的编程助手，也是精通{lang}的程序员。之前生成的代码质量不佳，未通过测试，也不符合惯用的函数式编程风格。请根据任务描述和函数的要求，采用惯用的函数式编程风格，提供更好的代码实现。你必须仅使用一次代码分隔符，将完整修正后的程序括在其中。",),("human","### 代码：\n{pre_code}\n")
```

G.2 特定语言的自我修复映射

第二部分总结了编译器反馈与相应自修复指令之间的详细映射。这些映射可作为为模型提供结构化指导，使其在自修复过程中纠正错误，并改善代码风格和可维护性。

表 7：Haskell 自我修复的指令映射

警告 / 错误	自我修复指令
避免使用 λ	已有函数或运算符时，避免使用 λ 表达式存在 ·
η 归约	通过 η 归约移除不必要的参数。
冗余括号	消除冗余括号以简化表达式。
使用中缀运算符，避免使用 λ 表达式	尽可能用中缀运算符替换 λ 表达式。
使用 zipWith	对并行列表操作使用 zipWith，而不是组合 map 和 zip。
避免使用 reverse	避免在 head、last 或索引操作之前使用 reverse，因为这样效率低下——效 .
移动括号以避开 $	为提高可读性，优先使用括号，避免过度使用 $ 运算符。
移动 filter	将 filter 尽可能靠近数据源应用，以减少不必要的计-算。
使用 uncurry	将函数应用于元组时使用 uncurry。
使用中缀	使用中缀表示法，以提高使用运算符时的可读性。

表 8：OCaml 自我修复的指令映射

警告 / 错误	自我修复指令
类型错误	检查类型注解，并确保表达式与预期类型匹配。例如，如果函数需要一个整数，却接收到一个字符串，请转换或调整类型。
未绑定标识符	在使用标识符之前声明它，或修正拼写。确保已打开/导入正确的模块。
解析错误	修复 OCaml 语法：检查是否缺少 in、->、;;，或括号位置是否错误-圆括号/方括号。
未使用	移除未使用变量，或在它们前面加上 _ if（如果是有意-未使用。
穷尽性错误	在模式匹配中添加缺失的情况以涵盖所有构造器，或 use _ 作为兜底情况 .
元数错误	调用函数时，请检查参数的数量。提供缺少参数或移除多余参数。
缺少标签参数	调用函数时添加必需的标签参数（例如，f ~x:value（如果函数需要 ~x）。
对不可变值进行变异	OCaml 值默认不可变。改写代码以避免变异，并使用新的绑定（let x = ...），而不是试图更新旧的绑定。

表 9：Scala 自我修复的指令映射

警告 / 错误	**自我修复指令**
If 块需要花括号	始终用花括号 {} 括起 if 块，以提高清晰度并避免歧义。
避免使用 return	不要使用 return；应改为依赖表达式值。
魔数	将魔数替换为命名常量或枚举，以提高可读性。
圈复杂度为	将函数重构为更小的函数，或使用函数式 control 结构（映射、折叠、递归）以降低复杂度。
在其前面应有一个空格加号（+）	在运算符周围添加适当的空格（例如，a + b，而不是 a+b）。
文件行长度超过	将过长的行拆分为多行较短的内容，以提高可读性。
文件必须以换行符结尾	确保文件以换行符结尾。