arXiv:2601.09583v1 [cs.PL] 14 Jan 2026

MLIR-Forge：面向语言工匠的模块化框架

BERKE ATES，苏黎世联邦理工学院，瑞士PHILIPP SCHAAD，苏黎世联邦理工学院，瑞士TIMO SCHNEIDER，苏黎世联邦理工学院，瑞士ALEXANDRU CALOTOIU，苏黎世联邦理工学院，瑞士TORSTEN HOEFLER，苏黎世联邦理工学院，瑞士

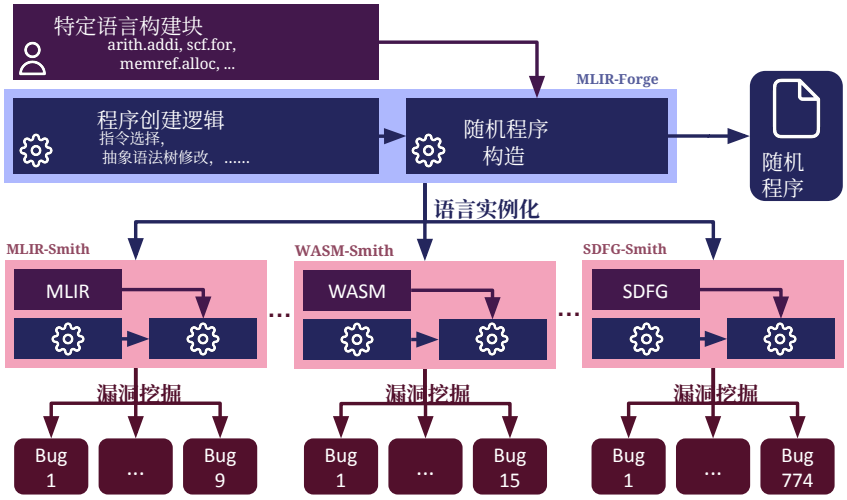


图 1. MLIR-Forge 概览。

优化编译器对于确保软件在各个科学领域中高效且正确地运行至关重要。领域专用语言（DSL）通常在其编译器流水线中使用更高级的中间表示（IR），以进行领域特定优化。随着这些中间表示增加了复杂性，对它们进行全面测试至关重要。随机程序生成器已被证明是通过差分测试和模糊测试来测试编译器的有效工具。然而，为编译器中间表示开发专用程序生成器并非易事，而且需要投入大量资源。我们推出了 MLIR-Forge，这是一种新颖的随机程序生成器框架，利用 MLIR 的灵活性，旨在简化专用程序生成器的创建。MLIR-Forge 将生成过程拆分为语言专属的基本构建模块，以及可复用的程序创建逻辑；后者利用这些构建模块构造随机程序，从而实现这一目标。这种方式隐藏了复杂性，而且即使是语言专属的组件，也可以使用一套通用工具来定义。我们通过生成采用内置方言的 MLIR、WebAssembly 以及数据中心程序表示 DaCe，展示了 MLIR-Forge 的能力——每种生成器的开发总耗时均不到一周。我们使用生成的程序开展差分测试，并利用相应的程序生成器发现了 9 组 MLIR 漏洞、15 组 WebAssembly 漏洞和 774 组 DaCe 漏洞；测试持续进行，直到新漏洞的发现速率趋于停滞。

作者联系方式：Berke Ates，berke.ates@inf.ethz.ch，苏黎世联邦理工学院，瑞士苏黎世；Philipp Schaad，philipp.schaad@inf.ethz.ch，苏黎世联邦理工学院，瑞士苏黎世；Timo Schneider，timo.schneider@inf.ethz.ch，苏黎世联邦理工学院，瑞士苏黎世；Alexandru Calotoiu，acalotoiu@inf.ethz.ch，苏黎世联邦理工学院，瑞士苏黎世；Torsten Hoefler，htor@inf.ethz.ch，苏黎世联邦理工学院，瑞士苏黎世。

1 引言

在当今的后摩尔时代，要实现出色的性能，硬件和软件都日益专门化，并针对特定任务进行优化 [Moreau 等人，2018年；Padmanabha 等人， 2015]。顺应这一专业化趋势，编程语言也不断演进，以满足特定领域的需求，由此出现了所谓的领域专用语言（DSL） [Jongmans 和 Arbab，2012年；Muller 等人， 2005]。领域专用语言针对特定任务或行业量身打造，让相关领域的专家更容易使用计算工具。绝大多数领域专用语言和通用语言都会在编译器流水线中使用中间表示（IR），以执行通用优化和特定用途的优化。通用编程语言通常会先使用多个较高层次的中间表示，再转向更通用的低层中间表示。这种方法使语言专属优化能够以模块化方式轻松实现，例如 Soot 框架就使用三种中间表示来有效优化 Java 字节码 [Vallée-Rai 等人， 2000]。

MLIR [Lattner 等人 2021]（多级中间表示）是该领域的一项关键进展。它通过提供一个能够处理多种中间表示的框架，并借助所提供的 Pass 基础设施、诊断工具、测试设施等促进复用，简化了创建新中间表示的过程。得益于 MLIR，创建和试验新的中间表示变得更加轻松快捷。这推动了人们通过使用专用中间表示，在机器学习、量子计算和代码优化领域开展更多研究 [Chelini 等人，2022年；Gysi 等人，2021年；Hu 等人，2023年；Le 等人，2020年；McCaskey 和 Nguyen 2021]。

尽早测试这类自定义中间表示至关重要，这有助于避免日后代价高昂的错误修复，并确保实验结果可信 [Rajamanickam 等人 2019]。随机程序生成器是发现编译器错误的宝贵工具。Csmith [Yang 等人，2011a] 和 DeepSmith [Cummins 等人2018] 展示了许多编译器即使接收到有效的输入程序，也可能崩溃或生成错误代码。手写测试通常遵循结构化模式，采用合理的输入，这是编写测试的人类开发者潜意识偏见所致。相比之下，随机程序生成器能够生成不同寻常、出乎意料的程序，其中包含的模式远超通常在合理代码中见到的范围。尽管两者都是有效程序，但随机生成的代码具有不可预测性，能够以独特的方式挑战编译器。因此，程序生成器能有效揭示标准测试可能遗漏的复杂漏洞。然而，为特定中间表示开发程序生成器可能既耗时又昂贵。这会抵消使用 MLIR 开发中间表示所带来的成本节约。

我们介绍了 MLIR-Forge，这是一种基于 MLIR 的程序生成器框架，将这项任务分为程序生成和语言规范两个部分，并自动化程序生成流程。这将开发者的职责限制在指定基本的语言构建模块上，例如定义该语言特有的 IR 操作。我们在图 1 中概述了我们的方法。MLIR-Forge 提供了一组通用工具，以最大限度地减少实现程序生成器所需的工作量，使测试更加经济且更加普及——这一目标与 MLIR 的理念一致。我们的方法促进了代码复用，使用户能够利用其他基于 MLIR-Forge 的程序生成器所完成的工作。

我们通过为 MLIR 的一部分内置方言创建程序生成器，以及为用于程序数据流优化的自定义方言 SDFG 方言 [Ben-Nun 等 2023]，展示了 MLIR-Forge 的使用方法。我们表明，即使是最初并非为 MLIR 设计的中间表示，也能通过将 MLIR 程序生成器与转换器结合起来，生成并测试 WebAssembly [Haas 等，2017a]代码而从 MLIR-Forge 中受益。

总而言之，我们的主要贡献如下：

- 通过在随机程序生成器开发中定义一个抽象层，将其划分为程序构造和语言规范两部分。

- 自动化程序构造过程，并将开发者职责缩减为仅负责语言规范。
- 通过共享语言规范来描述共通的语言特性，促进程序生成器之间的复用。
- 通过在 MLIR 中实现一个概念验证版本，展示了该框架的应用：不到一周便实现了三个程序生成器，并发现了 798组错误。

2 中间表示抽象

要构建能够生成多种中间表示的随机程序生成器框架，通用的中间表示抽象必不可少。该抽象为在框架中描述各种中间表示提供了统一方式。下文将介绍这一抽象，它为不同中间表示中的指令提供了标准化的表达方式。

2.1 操作模型

由于中间表示中的指令可以具有任意语义，固定的指令集不足以表示种类繁多的中间表示。我们通过采用一种对操作进行抽象的视角来应对这一挑战，如图2所示。

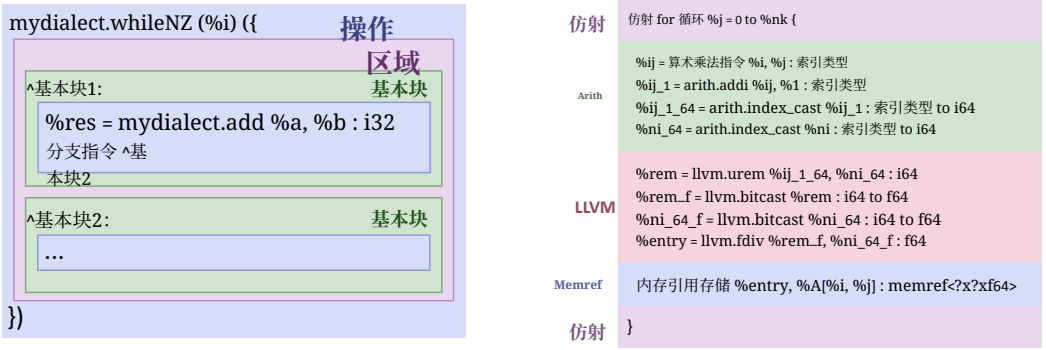


图 2. 操作概览。

这些操作可以接受任意数量的操作数或参数，并产生零个或多个结果。操作按方言分组，方言是操作和类型的命名空间。为标识操作所属的方言，会在操作前加上方言名称，这样不同方言就能为语义因应用而异的操作使用相同名称，例如算术和向量加法。

操作还可以包含表示静态信息的属性，这些信息能够改变操作的行为。例如，舍入操作可能带有一个用于设置舍入方法的属性。操作还可以使用其他操作直接产生的结果作为参数。虽然操作的类型通常能说明其结果的信息，但也可能提供有关其参数的信息。

操作遵循分层或嵌套设计，因此可以包含多个区域。每个区域由基本块组成，其中包含一系列操作。这种设计使我们能够将甚至（结构化的）控制流指令表示为操作，如图 3a 所示。通过将操作归入带命名空间的方言中，我们可以在同一程序中使用多个方言，如图 3b 所示。这种设计使中间表示开发者能够基于现有方言进行构建，专注于其中新增的内容。因此，专用方言可以更有效地针对硬件和软件中的特定领域。



(a) 嵌套操作结构示意图。 (b) 不同抽象层级的方言共存。

图 3. MLIR 概念示意图。

2.2 MLIR

这种抽象操作模型已在 MLIR 框架中得到实际实现，使其成为创建程序生成器框架的理想目标。它的一项突出特性是能够处理任意抽象层级的中间表示。如图 4 所示，MLIR 可用于串联多种此类中间表示，从而在编译过程的不同阶段利用不同抽象层级及其优势。MLIR 的设计兼具灵活性和模块化特征，因此可以便捷地创建不同的中间表示。它还提供了用于分析和调试这些自定义中间表示的工具，避免重复实现。借助 MLIR 的基础设施，用户可以实现将任意中间表示转换为 MLIR 所定义的中间表示，反之亦然。这引出了一个有趣的问题：我们能否将现有的随机程序生成器与转换器结合起来，为特定中间表示创建随机程序?

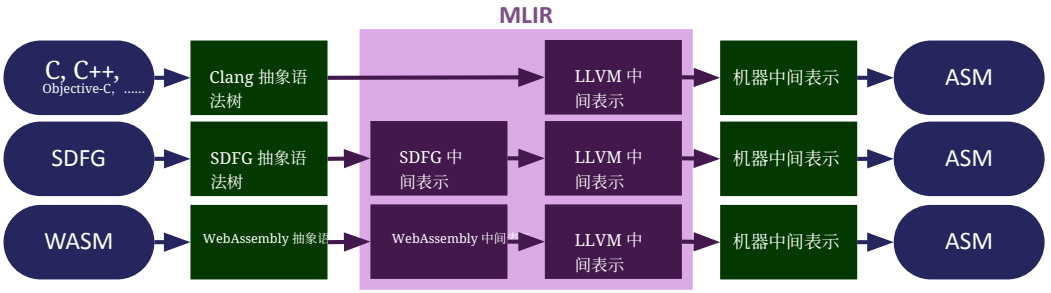


图 4. IR 专门化趋势及通过 MLIR 框架实现统一的示意图。

为 MLIR 定义的中间表示创建程序生成器，一种方法是使用现有生成器并搭配转换器。例如，我们可以使用 Csmith [Yang 等，2011b] 生成随机化 C 代码，然后使用 Polygeist 将其转换为 MLIR [Moses 等 2021], , 如图 5所示。依赖这种方法也有缺点。其中一项挑战是转换器的可靠性和准确性。加入转换器会增加一个可能产生漏洞的环节，使测试变得更加复杂。这并非凭空猜测。在我们使用 Csmith 和 Polygeist 进行的测试中，我们发现 Polygeist 会在处理 Csmith 生成的程序时崩溃。此外，由于目标是评估中间表示，转换器相关的问题会造成适得其反的干扰。

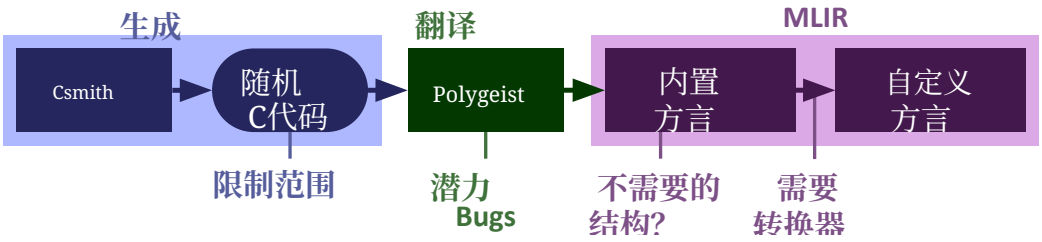


图 5. Csmith 和 Polygeist 流水线示例及其潜在缺陷示意图。

我们还受到源语言能力的限制。例如，C 不支持嵌套函数，也就是在函数内部声明函数，因此任何支持嵌套函数的中间表示都无法得到测试。此外，我们无法控制转换器如何处理输入代码，这可能导致出现不需要的代码结构。最后，转换器会生成 MLIR 内置方言。要使用自定义方言，我们还需要实现一个转换器，将内置方言转换为自定义方言。即使设计一个自定义转换器，直接生成所需的自定义中间表示方言，这仍需要大量工作，而且可能引入更多漏洞。鉴于这些挑战，直接在 MLIR 中生成随机程序更为有效。这种方法可以利用整个 MLIR 框架。具体而言，我们无需构建专用的中间表示生成器，而是可以开发一个框架，为中间表示设计者提供工具，使他们只需付出极少的额外工作，就能为自己的中间表示实现随机程序生成器。

3 构建程序生成器框架

我们构建程序生成器框架的方法，是将编程语言规范拆分为类似拼图块的基本构建模块。在此，我们详细介绍框架抽象，其示意图见图6。

3.1 定义拼图块

引入这些拼图块后，从用户或 IR 开发者的角度来看，生成过程转变为：生成器在适当的时机插入特定的程序元素，而开发者只需告知生成器程序元素可以如何插入。

每个拼图块都与编程语言的一项特定语言特性相关，例如一个操作。调用拼图块时，它会接收当前程序的现有结构以及下一次插入的位置。在此上下文中，拼图块必须确定是否能够在给定位置将该语言特性插入程序。如果条件不允许插入，拼图块便直接返回。否则，它会生成该语言特性的对象，填入所有必要的参数，例如参数和类型，并返回可供插入的对象。由于许多参数在不同拼图块之间具有共同的约束，我们还提供了用于辅助指定这些约束的工具。

为此，可以将语言规范视为一系列独立的生成器，简称 Gens。每个生成器都定义了语言规范中的一个特定组成部分。例如，语言支持的每种操作都有一个对应的操作生成器，简称 OpGen，用于描述如何将该操作插入现有程序。回到拼图的比喻，这描述了拼图片的形状，而拼图片代表一种操作。同样，类型由类型生成器（TypeGens）表示，它们发挥相同的作用。

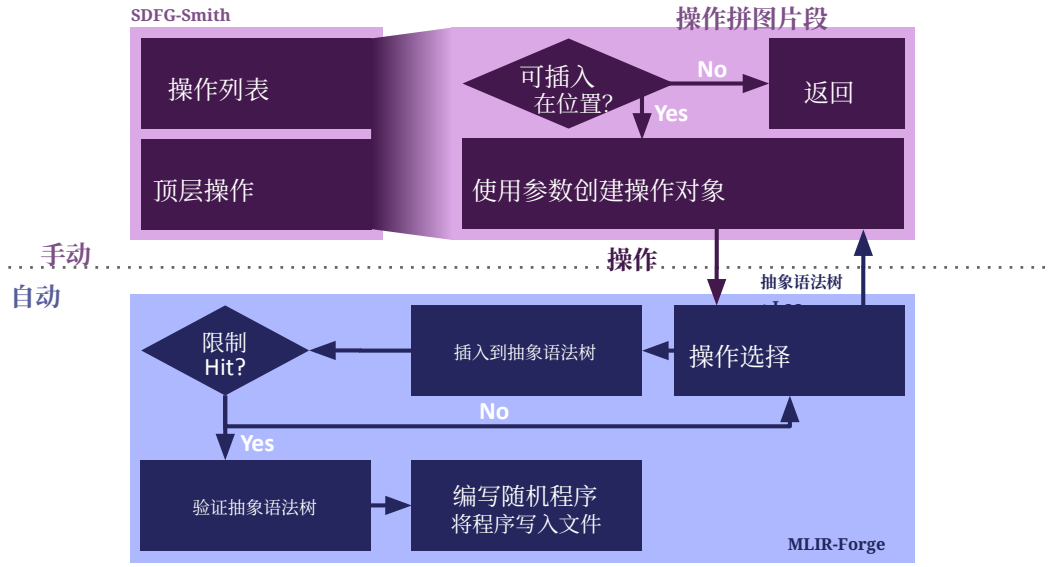


图 6. 随机程序生成器的各组件，以及通过 MLIR-Forge 实现抽象的示意图。我们以面向 SDFG 方言的随机程序生成器为例，但无论中间表示为何，这些组件都相同。

例如，我们考虑 SDFGIR [Ben-Nun 等人 2023], 提出的 SDFGIR，它包含多种操作，包括算术加法。采用我们的方法，我们会专门为这种算术加法实现一个 OpGen（即拼图片）。使用该 OpGen 时，它会检查当前位置是否有两个整数可用。如果有两个整数可用，OpGen 就会创建一个加法对象，并将这两个整数作为该加法的参数。

转向以插入为中心的视角，使得 Gens 能够在不同程序生成器之间复用。例如，许多语言都通用的算术表达式，可以只描述一次，然后在任何共享相同规范的其他表示中复用。这既提高了构建程序生成器的效率，也降低了出现漏洞的可能性。换言之，生成器现在能够仅根据拼图块的形状（即语言规范）和开发者的需求（即配置和标志），随机拼出一系列拼图块。无论用户交给它的是哪一堆拼图块，该框架都了解拼图游戏的规则。

3.2 抽象化并划分该过程

我们这种划分方法的第二部分涉及一个自动框架，该框架使用拼图块构建随机程序。框架首先随机挑选一个拼图块，并要求它提供相关的语言特性。如果该拼图块能够提供此特性，框架就会将其添加到抽象语法树（AST）中。每次插入后，框架都会检查是否已达到创建程序的限制。如果尚未达到限制，框架就会使用另一个拼图块继续这一过程。达到限制后，框架会确保 AST 遵循语言规范中的规则，并将随机程序保存到文件中。

再次以 SDFG 中间表示为例，框架最初可能会选择一个加法 OpGen。如果没有可用于加法操作的整数，OpGen 就会报告失败。框架可能会继续选择一个常量 OpGen。插入常量操作后，就

有了可用的整数，因此框架便可以成功插入加法操作。

将这些组件概括为可复用的模块后，程序生成器框架便能从语言规范中提取这些组件的工作方法，从而减轻程序生成器开发过程中的大部分负担。在这种抽象下，用户只需提供拼图块和一个顶层拼图块；至于如何选择拼图块以及如何修改抽象语法树等细节，则由框架隐藏。

4 MLIR-Forge

MLIR-Forge 是这种通用随机程序生成器框架的一个实践示例，它利用 MLIR 的某些特性，向中间表示开发者隐藏程序构造过程的复杂性。

在 MLIR-Forge 中，随机程序生成器从上到下，每次生成一个操作来构造程序。它首先从可用操作的选择池中随机选取一个操作，然后调用与之关联的 OpGen。该 OpGen 会尝试创建并放置此操作。如果放置成功，MLIR-Forge 就会将插入点（即插入新操作的位置）向下移动一步，并重复这一过程。如果无法创建某个操作，例如没有可用的所需类型的值，OpGen 就会将失败情况告知 MLIR-Forge。随后，MLIR-Forge 会将失败的 OpGen 从选择池中移除，再选取另一个操作。此选择过程会持续进行，直到选择池耗尽或找到一个成功的 OpGen。此时，选择池会重置，并重新加入此前失败的 OpGen。如果找不到任何成功的 OpGen，或达到程序长度限制，选择过程就会停止。控制流指令无需特殊处理，因为 MLIR 将它们表示为单个操作。

MLIR-Forge 的基础由三个主要组件构成：用于操作和类型的接口、命令行接口（CLI）框架，以及一个中间表示构建器。下文将介绍并进一步阐述这些组件。

4.1 接口与 CLI 框架

我们集成了用于操作和类型的接口，使 OpGen 和 TypeGens 能够添加到相应的操作和类型中。鉴于许多方言使用 LLVM 的 TableGen 及 MLIR 的操作定义规范（ODS）后端，我们还实现了操作和类型特征，使生成的操作和类型声明能够包含我们的接口。

这些接口不为 OpGen 或 TypeGens 提供默认实现，从而可检查是否遗漏了 Gen 实现。成功的 OpGen 会返回指向新添加操作的指针，便于追踪所做的更改。如果 OpGen 失败，则只会返回一个空指针。另一方面，TypeGens 会返回一个类型对象，供 OpGens 使用，而不会直接插入到中间表示中。

我们还引入了一个通用命令行框架，这是 MLIR-Forge 的主要入口。用户可以向命令行框架传入自己偏好的方言和一个顶层操作，从而构建自定义 CLI。该框架会解析用户标志，并据此修改配置。这些标志允许用户设置随机种子、加载和转储配置，以及指定输出文件。处理完标志后，命令行框架会创建一个模块操作（它是 MLIR 中隐式的顶层操作），并使用 IR 构建器的实用函数填充其基本块，从而启动程序生成。

4.2 IR 构建器

IR 构建器旨在为 OpGen 提供一种便捷的方式来操作中间表示。IR 构建器借鉴了 MLIR 的 OpBuilder，后者已提供用于遍历中间表示以及插入新操作、基本块和区域的工具。IR 构建器重写了 MLIR OpBuilder 中用于移动新操作插入点的函数，以防止插入点向上移动。它还增强了操作创建函数，使其在创建并插入所需操作之前先检查约束。该函数在成功时返回一个操作指针，否则返回空指针。这样，OpGen 就能确认操作创建是否成功，并将结果反馈给 IR 构建器。IR 构建器还提供了实用函数，可对值和类型进行采样，并收集当前中间表示插入点处所有可用的值和类型。具体做法是从插入点沿中间表示向上遍历，并逐步收集值和类型。遍历会在到达根操作或与外部作用域隔离的操作时停止。收集器工具会返回收集到的值和类型，而采样函数则会从收集到的选项中均匀地选取一个值。

IR 构建器还提供了随机创建操作以及用操作填充基本块的工具。这种设置支持递归方法，使 OpGen 能够在自身创建过程中生成操作。此外，接口设计允许 OpGen 在需要特定操作时直接调用另一个操作的 OpGen。IR 构建器包含一个嵌套配置类，用于设定各种配置，例如随机种子、嵌套区域的最大深度以及操作采样概率。该类具有良好的可扩展性，用户可以添加自己的配置，并在其 OpGen 中应用这些配置。在初始化时，IR 构建器会通过根据接口验证所有 OpGen 来收集它们。为提高效率，概率为零的操作会被省略。

4.3 挑战

生成完整的程序区域会带来一系列挑战。区域的结果类型取决于其中可用的值。因此，只有在区域完全生成后，才能确定这些结果类型；终结操作会标记代码块的结束，并据此追溯定义这些类型。例如，func 方言提供了 func.func 操作来定义函数。func.func 可以通过 func.return 返回其主体中定义的值。这些返回值的类型必须与主体区域的结果类型相匹配。因此，主体区域的结果类型由 func.return 的 OpGen 设置，而不是由 func.func 的 OpGen 设置。

虽然 MLIR-Forge 允许采样并创建任意终结符，但默认情况下，终结符不会被纳入操作选择池。我们假设父操作会插入必要的终结符。这一假设基于这样的认识：区域通常知道自身所需的终结符，而且这些终结符的种类往往不多。例如，func.func 需要 func.return，scf.while 需要 scf.condition 和 scf.yield，基本块则需要 cf.br、cf.cond_br 或 cf.switch。

某些操作可能对其区域的返回类型有特定要求。例如，scf.while 操作要求其条件区域中包含一个布尔值。确保存在这些类型颇具挑战，因为 MLIR-Forge 并不知道能够创建哪些类型。虽然可以考虑查询所有潜在类型以供决策，但方言可能定义无限集合的类型。例如，memref 类型可以具有任意数量的维度。此外，方言还可能定义由其他类型构成的复合类型。鉴于这些因素，MLIR-Forge 无法在区域中不断创建操作，寄望于最终出现所需类型，因为生成过程必须终止。因此，MLIR-Forge 不会假定自己了解能够创建哪些类型。

这一限制相对较小，主要有两个原因。首先，尽管从技术上来说操作可能拥有一个始终返回特定类型的区域，但这种情况并不常见。区域通常是为了灵活性而设计的，限制返回类型会削弱其适应性。其次，在研究 MLIR 的部分内置方言时，我们只发现了两个具有此类约束的操作：scf.while 和 omp.atomic.update。

4.4 操作频率

MLIR-Forge 的目标之一，是确保生成程序中操作的频率与配置中指定的频率一致。鉴于此，我们希望了解这些约束如何影响操作频率。具体而言，我们根据 §4.3 中提到的约束，确定生成一个 scf.while 操作的可能性。

假设 scf.while 操作之前有 N 个操作，且 scf.while 的条件区域包含 K 个操作。N 和 K 均服从概率为 p_g 的几何分布。某个操作生成所需布尔值的可能性表示为 p_{bool}。为简化计算，我们假设所有操作被选中的机会均等，且已存在的操作不会影响其他操作的概率。

总和 $S = N + K$ 的概率分布为：

$$\begin{aligned} P(S = s) &= \sum_{n=0}^{s} P(N = n) \times P(K = s - n) \\ &= \sum_{n=0}^{s} p_g \times (1 - p_g)^n \times p_g \times (1 - p_g)^{s-n} \\ &= \sum_{n=0}^{s} p_g^2 \times (1 - p_g)^s = (s + 1) \times p_g^2 \times (1 - p_g)^s \end{aligned}$$

对于固定的 N 和 K，至少有一个布尔值的概率为 $P(\#bool \geq 1) = 1 - (1 - p_{bool})^{N+K}$。据此，在已选中 scf.while 操作的条件下，生成 scf.while 操作的概率为

$$\begin{aligned} &P(\texttt{scf.while generated} \mid \texttt{scf.while chosen}) \\ &= \sum_{s=0}^{\infty} (1 - (1 - p_{bool})^s) \times P(S = s) \\ &= \sum_{s=0}^{\infty} (1 - (1 - p_{bool})^s) \times (s + 1) \times p_g^2 \times (1 - p_g)^s \end{aligned}$$

对于包含90个操作的原型，其中只有一个操作（arith.constant）能够无条件地产生布尔值，并使用 $p_{bool} = 1/90$ 和 $p_g = 0.2$，计算出的概率约为 P（生成了 | 个 scf.while, 选择了 scf.while） $\approx 0.0833 = 8.33\%$。为验证这一点，我们使用 MLIR-Smith 生成了 10, 000 个随机程序，并统计了 scf.while 的出现次数。观测频率为 8。05%，与理论预测一致。

5 锻造 smith

要使用 MLIR-Forge 构建 smith，用户需要使用配备 MLIR-Forge 的 MLIR 分支。还需要一个自定义方言，用来映射所讨论的特定编程语言或中间表示。该方言应涵盖该语言或中间表示中的操作。最高效的声明方式是

使用 MLIR 的 ODS 声明这些操作，因为它能够自动生成所需的样板 C 代码++ 。该方言中的操作必须实现 `GeneratableOp`接口，从而为其提供一个 `generate()` 函数。该函数接收一个中间表示构建器，并返回指向某个操作的指针；如果无法在当前位置插入操作，则返回空指针。如果可以插入，函数应收集所需的参数，构建一个操作对象，然后返回该对象。图7展示了一个实际示例，其中给出了 `arith.addi`的 OpGen。

```
操作            *arith::AddIOp::生成(GeneratorOpBuilder                    &构建器) {
  Type resultType   =构建器.getI32Type();
  llvm::Optional <Value > lhs   = 构建器.sampleValueOfType(resultType);
  llvm::Optional <值 >          右侧 =  构建器.sampleValueOfType(resultType);

  if  (!左侧.has_value()    ||  !右侧.has_value())
    返回      空指针;

  操作状态 state(builder.getUnknownLoc(),                  "arith.addi");
  arith::AddIOp::build(builder ,          state ,    lhs.value(),      rhs.value());
  操作          *op =  builder.create(state);

  返回     op;
}
```

图 7。算术方言中整数加法的简化版 OpGen。

实现所有 generate() 函数后，下一步是为 smith 建立一个入口点，通常通过将所有涉及的方言填充到注册表中并调用命令行框架来完成。通常，该框架默认将模块操作视为顶层操作，但也可以对其进行自定义。至此，一个能够生成该方言并通过标志提供命令行配置的 smith 便构建完成了。如果目标中间表示是 MLIR 方言，则无需进一步操作。否则，需要一个简单的转换器，将 MLIR 定义的语法转换为目标中间表示的语法。鉴于 MLIR 的适应性，可以重新配置它，使其在很大程度上与目标中间表示的语法相似。

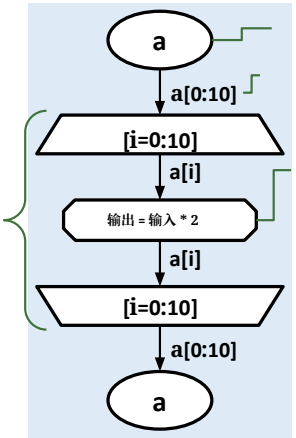
虽然我们的方法提供了灵活性，但也存在局限性。用户有责任实施相应的安全措施，例如防止除以零。由于用户定义的操作可能具有任意的语义含义，因此无法自动提供安全保障。出于同样的原因，确保操作的确定性也由用户负责。例如，用户必须在访问内存之前对其进行初始化，以避免非确定性行为。

6 评估

借助 MLIR-Forge，我们开发了三个随机程序生成器，分别针对在概念上截然不同的中间表示：MLIR、SDFG（状态化数据流多重图）中间表示 [Ben-Nun 等人 2019], 以及 WebAssembly [Haas 等人，2017b]（WebAssembly）。MLIR 是一种命令式中间表示，旨在通过单一框架表示广泛的抽象层级，因此既适用于高层领域专用操作，也适用于低层硬件专用指令。另一方面，SDFG 中间表示是一种数据中心型中间表示。它将程序表示为图，其中节点代表计算任务，边表示数据依赖。这种方法对于优化数据移动和内存使用尤为有效，而这在高性能计算和并行计算环境中至关重要。WebAssembly 是一种基于栈的中间表示，针对高效执行和紧凑表示进行了优化。其基于栈的特性使指令能够对栈中的数据值进行操作，因此非常适合 Web 应用，并能确保在不同平台间具备可移植性和紧凑性。图8 展示了这些中间表示之间的根本差异，并分别给出了用每种中间表示表达的同一个倍增函数。

```
func @double(%a: Memref <10xi32>)
{

  %2 = 算术.常量 2
  %0 = arith.常量 0
  %1 = 算术.常量 1
  %10 = 算术.constant 10

  scf.for %i =%0 至%10 步进 %1
  {

    %e = memref.load %a[%i]

    %d = 算术.乘法 %e, %2

    memref.store %d, %a[%i]

  }

  返回
}
```

(a) MLIR



(b) SDFG

```
(Func (参数 i32)
  (局部变量 i32i32)
  loop
    局部变量.get 0
    局部变量.get 1
    i32.const 2
    i32.shl
    i32.add
    本地.tee 2
    本地.get 2
    i32.load
    i32.const 1
    i32.shl
    i32.存储
    本地.get 1
    i32.const 1
    i32.add
    局部.tee 1
    i32.const 10
    i32.ne
    br_if 0
end)
```

(c) WebAssembly

图 8. 评估中使用的三种中间表示以截然不同的方式表达了同一个倍增函数。

为 SDFG 生成随机程序有其独特的挑战，主要在于准确建模和管理数据依赖及执行顺序。在 SDFG 中，数据流并非只是一连串操作，而是一个复杂的依赖网络；每个节点处数据的状态及其转换都可能对整个图产生连锁影响。要随机生成这类图，需要理解数据依赖，并能够创建连贯的数据流，同时遵循程序的执行顺序。这包括确保每个节点都能正确处理数据并将其传递给后续节点，同时保证整体图结构有效且可执行。

在随机生成 WebAssembly 程序的过程中，一项重大挑战是确保栈操作有效。作为一种基于栈的语言，WebAssembly 要求每项操作都与栈的当前状态相符，即从栈中取出值或向栈中添加值。因此，随机生成必须在整个程序中维持一致的栈状态。这需要准确预测每个位置的栈状态；而控制流变化和函数调用会使这项任务变得复杂。

通过使用概念上各不相同且具有独特约束的中间表示，我们展示了如何轻松地为成熟的和自定义的程序表示实现健壮的程序生成器。我们还展示了这如何助力程序测试：利用生成的程序进行差分模糊测试，从而发现中间表示的优化器和编译器处理流程中的漏洞。

6.1 SDFG-Smith

DaCe [Ben-Nun 等人 2019] 是一个以数据为中心的优化和代码生成框架，使用 SDFG（状态化数据流多重图）中间表示，从数据移动的角度表示程序。它通过将程序表示为有向无环图来实现这一点，这些图负责协调数据在一系列计算中的流动。DaCe 生态系统提供了 SDFG 方言 [Ben-Nun 等人 2023], ，该方言能够在 MLIR 中表示 SDFG 中间表示。第一个随机程序生成器会生成 SDFG 方言，我们通过转换器处理该方言以获得 SDFG 中间表示。我们使用随机生成的程序，对围绕 SDFG 中间表示的优化框架和编译器基础设施进行差分模糊测试。SDFG-Smith 生成的程序会接收一个或多个输入参数，随后将这些参数传递给模糊测试器以执行

差分测试。我们将程序交给两个模糊测试器：AFL++ [Fioraldi 等人 2020], , 其目标是通过最大化分支覆盖率来提高测试准确性；以及 FuzzyFlow [Schaad 等人 2023],, 一个针对 SDFG 中间表示的差分输入模糊测试器。当新发现的漏洞数量增长速度似乎停滞时，我们便停止测试。

由于多个输入可能触发同一个漏洞，我们通过提取错误消息并计算其 MD5 哈希来对它们进行分组。由于某些错误消息包含输入程序的行号，我们采取保守做法，将错误消息中的所有数字替换为一个固定字符。图 9 展示了在差分测试器的运行过程中，我们使用该方法识别出的错误组总数。我们在 14,681 小时内生成了 20 个程序，并识别出 774 个错误组。在1,154 个程序中，DaCe 的优化遍达到了我们设定的 10 秒时间限制。尽管对每个漏洞进行全面分析超出了本文的范围，但我们已与 DaCe 开发团队合作，识别出三个重要漏洞，并将在未来版本中修复。

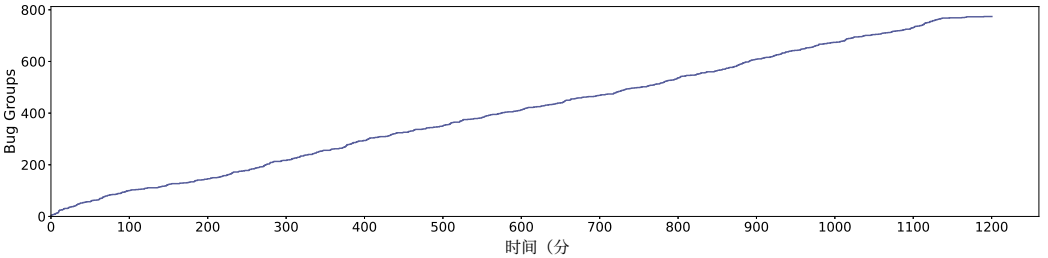


图 9. 使用我们的分组方法比较漏洞组与测试时间。

图 10 展示了其中一个漏洞。从以数据为中心的视角来看，简化后的输入 SDFG 只执行冗余拷贝，可以将其完全消除。输入 SDFG 还包含一个空的嵌套 SDFG，它是一个不执行任何操作的子程序，本质上相当于一个 空操作。使用 DaCe 的自动优化器进行优化后，我们得到一个有错误的 SDFG。自动优化器成功识别出 SDFG 底部的冗余拷贝，却未能将并行作用域识别为死数据流。这导致下方节点被错误移除，使并行作用域中留下了一项没有目标的悬空数据移动。

6.2 MLIR-Smith

MLIR-Smith 是我们的第二个随机程序生成器，它使用内置方言的一个子集生成 MLIR 代码，具体包括 SCF、Arith、Math、Memref 和 Func。流程首先生成这些方言，随后对生成的 MLIR 代码进行优化（`-cse -canonicalize -symbol-dce-loop-invariant-code-motion -inline`）。接下来，将未优化和优化后的 MLIR 降低为 LLVM 方言，再转换为 LLVM 中间表示 [Lat- tner 和 Adve 2004]，使用 MLIR 的 `mlir-opt` 和 `mlir-translate` 工具。我们使用 LLC（`-O0 -relocation-model=pic`）和 Clang（`-O0 -fPIC -march=native`）编译 LLVM 中间表示，并检查所有阶段是否出现错误或崩溃。我们在设定的时间限制内运行编译后的二进制文件，以识别运行时问题。如果两个二进制文件都达到时间限制，或以相同的代码退出，则测试被视为成功。如果只有一个二进制文件超出时间限制，则表明这两个二进制文件的终止行为可能存在差异。由于某个二进制文件可能只是运行得明显更慢，因此需要进一步分析，才能确认终止行为是否发生变化。两个二进制文件的退出代码不同，则明确表明行为发生了变化，也就意味着存在漏洞。

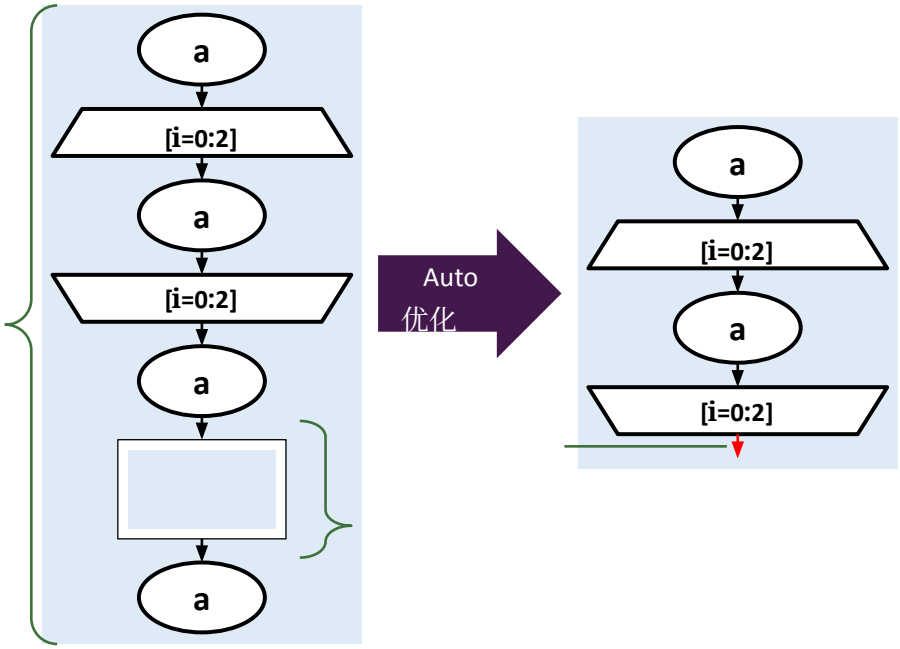


图 10. SDFG-Smith 生成的简化的 SDFG，揭示了 DaCe 的自动优化错误。

通过这一流程，我们发现了一个案例：未优化代码无法终止，而优化版本却能终止。图 11 展示了未优化代码的简化版本。终止行为的差异是由一个无限 while 循环导致的，该循环在优化过程中被移除。与 MLIR 开发者讨论后，我们了解到规范目前尚未涵盖这种情况。他们提出了两种选择：将此类行为定义为未定义行为，或者认为该循环没有副作用，从而允许在死代码消除过程中将其移除。我们还发现了涉及多次释放同一块内存以及释放栈分配内存的类似案例。在 23,065 个小时内，我们生成了 10 个程序，并发现了 9 个错误组。

```
func @main() {
  %true = arith.constant true

  scf.while () : () -> () {
    scf.condition(%true)
  } do {
    scf.yield
  }

  返回
}
```

图 11. MLIR-Smith 生成的简化 MLIR 代码，揭示了非终止 while 循环缺少规范。

```
func @主要内容()  -> i32 {

  %c0f   = 算术.常量 0.0:          f16
  %c0 = 算术.常量 0          : i32

  %1 = math.fpowi %c0f, %c0 : f16, i32
  %2 = math.absf  %2 : f16

  返回    %c0 : i32

}
```

图 12. WASM-Smith 生成的简化程序在使用 -O3 优化时导致 emcc 崩溃，该程序以 MLIR 编写。

6.3 WASM-Smith

基于 MLIR-Smith 的能力，我们的第三个随机程序生成器将生成过程扩展到 WebAssembly [Haas 等人，2017b]（WASM）代码。由其生成的输出

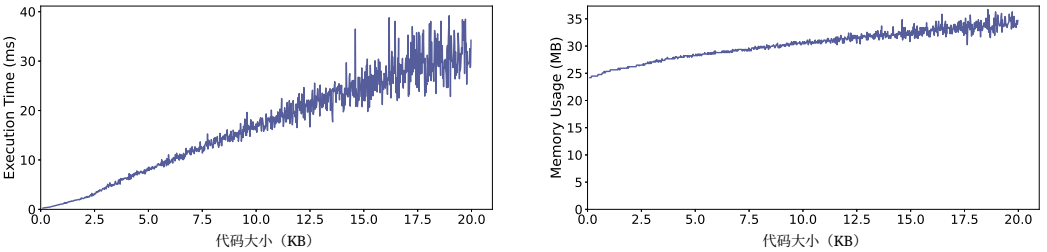
使用 MLIR 的 `mlir-opt` 和 `mlir-translate` 工具，可将 MLIR-Smith 降低为 LLVM 中间表示。我们使用 Emscripten [Zakai 2011] 编译器前端 `emcc` 来生成 WASM 代码。借助 WebAssembly 二进制工具包 WABT，我们可以将 WASM 代码转换为人类可读的 WebAssembly 文本格式（WAT）。这一设置突显了 MLIR-Forge 为中间表示生成复杂程序代码的能力，这些中间表示并未在 MLIR 中专门定义为方言。

对于 WASM-Smith，我们使用了两个转换器和一个编译器。MLIR 到 LLVM 转换器会重写语法。虽然 MLIR-Forge 可以使用 MLIR 中的 LLVM 方言直接生成 LLVM，但我们选择进行转换，以便利用现有的原型。对于 LLVM 到 WASM 的编译，我们采用了 WASM 生态系统中的编译器 emcc。WASM 到 WAT 的转换则只是将字节码转换为人类可读的文本。两个转换器和该编译器都是被测系统不可或缺的组成部分，因此在测试其正确性和可靠性方面尤其值得关注。

借助这一流水线，我们使用 `emcc`并结合 `-O0` 和 `-O3` 标志执行差分测试。图 12展示了 MLIR 中一个导致漏洞的输入，以便于阅读。 `emcc` 使用 `-O0` 可以成功编译此输入代码，但使用 `-O3` 时会崩溃。我们怀疑，常量传播遍在遇到 `math.fpowi` 时会出错，因为它会计算零的零次幂。有趣的是，如果移除 `math.absf`，`emcc` 便能使用 `-O3` 成功编译。我们在 12,014 个程序上运行了 10 小时的测试，并识别出 15 个错误组。

6.4 资源

为了展示此类测试对时间和资源的需求较低，我们测量了生成 100，000 个 SDFG-Smith 程序所耗费的时间和内存，以及生成程序的大小。实验中使用了一颗时钟频率为 3.00 GHz 的 Intel(R) Core(TM) i9-10980XE CPU。图 13 展示了按 1，000 个区间分组后的中位数结果。请注意，生成代码的大小以字节而非代码行数（LoC）表示，因为单行操作可能包含任意数量的输入，这会增加代码的字节数，却不会增加代码行数。



(a) 代码大小与执行时间的比较。 (b) 代码大小与内存使用的比较。

图 13. MLIR-Forge 性能测量。

如图 13 所示，对于小于 36 KB 的程序，MLIR-Forge 的内存使用低于 40 ms，且完成时间不到 20 MB，展现了它在消费级硬件上轻松运行的能力。这使其能够用于快速迭代开发工作流中的快速测试。时间和内存消耗似乎都与代码大小呈线性相关。值得注意的是，MLIR-Forge 尚未经过优化，经常执行多次抽象语法树遍历；通过缓存可以减少这些遍历，从而进一步降低相同工作负载下的时间和内存消耗。

为凸显 MLIR-Forge 的易用性，每个随机程序生成器都由一名开发者在不到一周的时间内完成。为了直观地衡量开发这些

程序生成器所需的工作量，我们在表 1中汇总了 OpGen 和 TypeGens 的代码行数。所有三个随机程序生成器都使用 arith 和 math 方言，其中包含许多结构相似的 OpGen，例如一元和二元运算。这也展现了 MLIR-Forge 的代码复用能力：arith 和 math 方言只需编写一次，就可供三个不同的程序生成器使用。

表1. Smith 中 OpGen 和 TypeGens 的代码行数（LoC）

方言	SDFG（类型）	SDFG	Arith	Math
LoC	24	423	1519	810
方言	Memref（类型）	Memref	SCF	Func
LoC	9	248	179	56

7 相关工作

近年来，编译器中间表示及其相关工具领域取得了显著进展。领域特定语言的开发、中间表示语义的形式化，以及机器学习在模糊测试中的应用，都是这一领域值得关注的趋势。

随机程序生成器。在软件测试这个不断发展的领域中，随机程序生成器已成为不可或缺的工具。这些生成器通过生成大量不可预测的程序，考验编译器、解释器和其他软件系统的稳健性。Csmith [Yang et al. 2011a] 已成为重要的随机程序生成器，在真实世界的编译器中发现了大量漏洞。其方法和成功为软件测试领域树立了标杆。Cummins et al. [Cummins et al. 2018] 更进一步，推出了用于编译器测试的机器学习工具 DeepSmith。DeepSmith利用开源代码中的数据，生成了大量逼真的程序用于测试。经过1,000小时的测试，他们在检查的每个编译器中都发现了问题，无论是商业编译器还是开源编译器。类似地，COM- FORT [Ye et al. 2021] 通过利用 GPT-2（一种基于深度学习的语言模型）生成有效的 JS 程序，为 JavaScript（JS）模糊测试带来了新思路。其主要目标是发现标准一致性错误，尤其是那些违反 ECMA-262（ECMAScript）标准的错误。EvoSuite [Fraser and Arcuri 2011] 和 Randoop [Pacheco and Ernst 2007] 在 Java程序生成和测试方面贡献突出。QuickCheck [Claessen and Hughes 2000],最初为 Haskell 开发，在为各种语言中的函数生成测试用例方面发挥了重要影响。DOMATO [Fratric 2023] 等工具则在生成随机 JavaScript 和网页内容以用于测试方面发挥了重要作用。Skyfire [Wang et al. 2017]专注于为网页浏览器生成输入，而 LangFuzz [Holler et al. 2012] 则致力于测试编程语言解析。PolyGlot [Chen et al. 2021] 以及其他基于语法的生成器能够生成独立于目标编程语言的程序。它们非常适合对编译器验证器进行模糊测试，因为验证器可以阻止编译器接受无效输入。然而，生成语义正确的代码并非易事，这限制了它们在差分测试编译器优化方面的效果。Xsmith [Hatch et al. 2023] 提出了一种用于创建程序生成器的框架，优先保证语义正确性，因此非常适合对优化进行差分测试，但不适用于对验证器进行模糊测试。相比之下，模块化的程序生成器框架（如 MLIR-Forge）兼顾这两种测试模式，弥合了两种技术之间的差距，从而为测试新型中间表示提供了全面的方法。此外，它还能够生成循环操作，即两个操作相互依赖对方的输出；这一能力

对于机器学习为核心的环境中普遍存在的基于图的中间表示尤为有利。这一能力使其有别于不具备此功能的其他抽象语法树构造算法。MLIR-Forge 具备这些工具的相同优势，但实现所需的资源显著更少。因此，它是测试实验性中间表示的可行方案。

输入模糊测试器。沿着利用随机性的相同趋势，输入模糊测试已成为识别软件系统中漏洞的强大技术。多年来，人们开发了大量模糊测试工具，每种工具都有其独特优势和专注领域。AFL [Zalewski 2023]和 AFL++ [Fioraldi 等人 2020]是最受欢迎的覆盖率引导模糊测试器之一，它们使用遗传算法，高效发现能够触发目标程序新内部状态的新输入。它们的方法奠定了模糊测试社区的基础，并启发了许多后续工具。LibFuzzer [Project 2023] 是另一种覆盖率引导模糊测试器，它在进程内运行，反复生成并执行测试输入。其进程内特性可实现快速测试周期，因此对某些应用尤其有效。Honggfuzz [Google 2023b] 是一种注重安全性的、基于进化的模糊测试器，着重发现安全漏洞。它的方法与 Spirv-fuzz [Donaldson 等人 2021], 形成对比；后者专注于生成语义有效但随机的着色器程序。其他值得一提的模糊测试器包括 Peach Fuzzer [Eddington 2023], , 它可以在协议层对应用程序进行模糊测试；以及 Boofuzz [Pereyda2023], , 它是用于网络协议模糊测试的 Sulley [OpenRCE 2023] 模糊测试框架的后继者。Radamsa [Helin 2023]是一种通用模糊测试器，可测试任何从标准输入读取数据的程序；而 ClusterFuzz [Google 2023a] 和 OSS-Fuzz [Serebryany 2017] 则代表了用于大规模模糊测试活动的可扩展模糊测试基础设施。MLIR-Forge 通过简化模糊测试用测试程序的创建，为模糊测试器提供了补充。通过提供丰富的测试程序集合，尽管 spirv-fuzz 等工具会修改现有测试以发现编译器问题，MLIR-Forge 却能提供丰富的初始测试程序，从而拓展测试过程的深度和广度。

中间表示设计。 从历史上看，中间表示的设计一直是一个手动过程，使该领域的创新成本高昂、耗时漫长。开发人员传统上使用通用编程语言来设计中间表示，导致实现冗长，也使修改成本高昂。为提高这一过程的效率，人们付出了大量努力。作为 LLVM [Lattner 和 Adve 2004] Project 的一部分，MLIR [Lattner 等人 2021]旨在通过提供可扩展且可复用的基础设施，降低构建领域专用编译器的成本。MLIR 还包含多种用于中间表示开发的常用工具，例如诊断、Pass 基础设施和测试工具。Fehr 等人 [Fehr 等人 2022] 通过引入中间表示定义语言（IRDL）进一步简化了中间表示开发。IRDL 是一种专为定义中间表示而设计的领域专用语言，有助于实现基于静态单赋值的中间表示。通过支持对中间表示进行简洁、明确的规范描述，IRDL 为开发有效工具、实现编译器构造过程自动化奠定了基础。MLIR-Forge 通过降低实现中间表示专用测试工具所需的开销，补充了这些工作。我们的测试框架有望直接集成到 IRDL 生态系统中。这样的集成将进一步减少编译器构造过程中所需的手动工作，确保中间表示不仅能够高效设计，还能得到严格测试。

本节重点介绍的工作凸显了编译器中间表示及其相关工具的多样化进展。我们的贡献 MLIR-Forge 借鉴了这些创新，为编译器研究领域提供了独特的视角。我们强调 MLIR-Forge 有望连接各种工具与方法，促进形成一个协调统一的编译器研究生态系统。

8 结论

鉴于中间表示测试对推动中间表示研究和实验的重要性，随机程序生成器发挥着重要作用。本文提出了一种随机程序生成器框架，可大幅缩短开发用于中间表示的随机程序生成器工具所需的耗时；这些工具可用作测试工具。该框架将随机程序生成器的构建过程分为两个不同的任务：程序构造和语言规范。

这种划分将开发者的责任简化为提供简洁的语言规范，具体做法是将各个拼图块定义为独立的操作和类型生成器。这些拼图块向程序生成器框架说明如何将各个程序元素串联起来，以及它们如何相互作用，使其能够在无需额外信息的情况下构造随机程序。操作和类型生成器可以在具有相同语言特性且表示形式相同的不同语言之间共享，从而提高不同程序生成器之间的复用性。

通过开发 MLIR-Forge 这一概念验证实现，我们展示了此类框架的实用性。我们将其应用于 DaCe 并行编程框架的差分测试，并通过使用它生成 WebAssembly 程序，展示了它不仅适用于基于 MLIR 的中间表示，也具有更广泛的适应性。所有这些都只需极少的开发工作和资源消耗即可实现，从而能够在消费级硬件上开展复杂的编译器和语言测试。MLIR-Forge 让研究人员能够减少测试所花费的时间，将更多精力投入实验。此外，我们希望借此增强人们对中间表示领域新研究成果的信心。

参考文献

Tal Ben-Nun、Berke Ates、Alexandru Calotoiu 和 Torsten Hoefler。2023年。弥合控制中心型与数据中心型优化。载于第 21 届 ACM/IEEE 代码生成与优化国际研讨会论文集（加拿大魁北克省蒙特利尔）（CGO 2023）。计算机协会，美国纽约州纽约，173–185。doi:10.1145/ 3579990.3580018 Tal Ben-Nun、Johannes de Fine Licht、Alexandros Nikolaos Ziogas、Timo Schneider 和 Torsten Hoefler。2019年。有状态数据流多重图：异构架构上实现性能可移植性的数据中心型模型。载于国际高性能计算、网络、存储与分析会议论文集（SC '19）。Lorenzo Chelini、Henrik Barthels、Paolo Bientinesi、Marcin Copik、Tobias Grosser 和 Daniele G. Spampinato。2022年。MOM：MLIR 中的矩阵运算。arXiv:2208.10391 [cs.PL] Yongheng Chen、Rui Zhong、Hong Hu、Hangfan Zhang、Yupeng Yang、Dinghao Wu 和 Wenke Lee。2021年。一个引擎模糊测试所有程序：通过语义验证进行通用语言处理器测试。载于 2021 年 IEEE 安全与隐私研讨会（SP）。642–658。doi:10.1109/SP40001.2021.00071 Koen Claessen 和 John Hughes。2000年。QuickCheck：用于 Haskell 程序随机测试的轻量级工具。SIGPLAN 通讯 35, 9（2000年9月），268–279。doi:10.1145/357766.351266 Chris Cummins、Pavlos Petoumenos、Alastair Murray 和 Hugh Leather。2018年。通过深度学习进行编译器模糊测试。载于第 27 届 ACM SIGSOFT 国际软件测试与分析研讨会论文集（荷兰阿姆斯特丹）（ISSTA 2018）。计算机协会，美国纽约州纽约，95–105。doi:10.1145/3213846.3213848 Alastair F. Donaldson、Paul Thomson、Vasyl Teliman、Stefano Milizia、André Perez Maselco 和 Antoni Karpiński。2021年。基于变换的编译器测试：几乎零成本实现测试用例缩减与去重。载于第 42 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集（线上，加拿大）（PLDI 2021）。计算机协会，美国纽约州纽约，1017–1032。doi:10.1145/3453483.3454092 Michael Eddington。2023年。Peach Fuzzer：协议级模糊测试。https://peachtech.gitlab.io/peach-fuzzer-community Mathieu Fehr、Jeff Niu、River Riddle、Mehdi Amini、Zhendong Su 和 Tobias Grosser。2022年。IRDL：面向 SSA 编译器的 IR 定义语言。载于第 43 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集（美国加利福尼亚州圣迭戈）（PLDI 2022）。计算机协会，美国纽约州纽约，199–212。doi:10.1145/3519939.3523700 Andrea Fioraldi、Dominik Maier、Heiko Eißfeldt 和 Marc Heuse。2020年。AFL++：融合模糊测试研究的渐进成果。载于第 14 届 USENIX 攻防技术研讨会（WOOT 20）。USENIX 协会。Gordon Fraser 和 Andrea Arcuri。2011年。EvoSuite：面向面向对象软件的自动测试套件生成。载于第 19 届 ACM SIGSOFT 研讨会和第 13 届欧洲软件工程基础会议论文集

（匈牙利塞格德）（ESEC/FSE '11）。计算机协会，美国纽约州纽约，416–419。doi:10.1145/ 2025113.2025179 Ivan Fratric。2023年。DOMATO：DOM 模糊测试器。https://github.com/googleprojectzero/domato Google。2023a。ClusterFuzz：可扩展模糊测试基础设施。https://github.com/google/clusterfuzz Google。2023b。Honggfuzz：基于进化的模糊测试器。https://github.com/google/honggfuzz Tobias Gysi、Christoph Müller、Oleksandr Zinenko、Stephan Herhut、Eddie Davis、Tobias Wicky、Oliver Fuhrer、Torsten Hoefler 和 Tobias Grosser。2021年。面向 GPU 的领域特定多级 IR 重写：用于 GPU 加速气候模拟的 Open Earth 编译器。ACM 架构与代码优化汇刊 18, 4，文章 51（2021年9月），23 页。doi:10.1145/3469030 Andreas Haas、Andreas Rossberg、Derek L. Schuff、Ben L. Titzer、Michael Holman、Dan Gohman、Luke Wagner、Alon Zakai 和 JF Bastien。2017a。让 WebAssembly 加速 Web。SIGPLAN 通讯 52, 6（2017年6月），185–200。doi:10.1145/3140587.3062363 Andreas Haas、Andreas Rossberg、Derek L. Schuff、Ben L. Titzer、Michael Holman、Dan Gohman、Luke Wagner、Alon Zakai 和 JF Bastien。2017b。让 WebAssembly 加速 Web。载于第 38 届 ACM SIGPLAN 编程语言设计与实现会议论文集（西班牙巴塞罗那）（ PLDI 2017 ）。计算机协会，美国纽约州纽约，185–200。doi:10.1145/3062341.3062363 William Hatch、Pierce Darragh、Sorawee Porncharoenwase、Guy Watson 和 Eric Eide。2023年。使用 Xsmith 生成符合规范的程序。载于第 22 届 ACM SIGPLAN 生成式编程国际会议：概念与经验论文集（葡萄牙卡斯凯什）（GPCE 2023）。计算机协会，美国纽约州纽约，86–99。doi:10.1145/3624007.3624056 Aki Helin。2023年。Radamsa：通用模糊测试器。https://gitlab.com/akihe/radamsa Christian Holler、Kim Herzig 和 Andreas Zeller。2012年。使用代码片段进行模糊测试。载于第 21 届 USENIX 安全研讨会（USENIX Security 12）。USENIX 协会，美国华盛顿州贝尔维尤，445–458。https://www.usenix.org/conference/usenixsecurity12/ technical-sessions/presentation/holler Pengchao Hu、Man Lu、Lei Wang 和 Guoyue Jiang。2023年。TPU-MLIR：使用 MLIR 的 TPU 编译器。arXiv:2210.15016 [cs.PL]Sung-Shik T. Q. Jongmans 和 Farhad Arbab。2012年。线程间协议的模块化与规范化。载于第五届面向并发与通信软件的编程语言方法研讨会论文集，PLACES 2012，爱沙尼亚塔林，2012年3月31日（EPTCS，第 109 卷），Simon J. Gay 和 Paul Kelly（编）。34–45。doi:10.4204/EPTCS.109.6 Chris Lattner 和 Vikram Adve。2004年。LLVM：用于持续程序分析与转换的编译框架。载于代码生成与优化国际研讨会：面向反馈与运行时的优化论文集（美国加利福尼亚州帕洛阿尔托）（CGO '04）。IEEE 计算机学会，美国，75。Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko。2021年。MLIR：面向领域特定计算的可扩展编译器基础设施。载于 2021 年 IEEE/ACM 代码生成与优化国际研讨会（CGO）。2–14。doi:10.1109/CGO51591. 2021.9370308 Tung D. Le、Gheorghe-Teodor Bercea、Tong Chen、Alexandre E. Eichenberger、Haruki Imai、Tian Jin、Kiyokuni Kawachiya、Yasushi Negishi 和 Kevin O'Brien。2020年。使用 MLIR 编译 ONNX 神经网络。CoRR abs/2008.08272（2020年）。arXiv:2008.08272 https://arxiv.org/abs/2008.08272 Alexander McCaskey 和 Thien Nguyen。2021年。量子汇编语言的 MLIR 方言。arXiv:2101.11365 [quant- ph] Thierry Moreau、Tianqi Chen、Ziheng Jiang、Luis Ceze、Carlos Guestrin 和 Arvind Krishnamurthy。2018年。VTA：面向深度学习的开放式软硬件栈。CoRR abs/1807.04188（2018年）。arXiv:1807.04188 http://arxiv.org/abs/1807. 04188 William S. Moses、Lorenzo Chelini、Ruizhe Zhao 和 Oleksandr Zinenko。2021年。Polygeist：将 C 提升为多面体 MLIR。载于 ACM 并行架构与编译技术国际会议论文集（线上）（PACT '21）。计算机协会，美国纽约州纽约，12 页。doi:10.1109/PACT52795.2021.00011 Gilles Muller、Julia L. Lawall 和 Hervé Duchesne。2005年。简化内核调度器开发的框架：设计与性能评估。载于 HASE 2005 - 高可靠系统工程会议。德国海德堡，56–65。OpenRCE。2023年。Sulley：无人值守模糊测试框架。https://github.com/OpenRCE/sulley Carlos Pacheco 和 Michael D. Ernst。2007年。Randoop：面向 Java 的反馈式随机测试。载于第 22 届 ACM SIGPLAN 面向对象编程系统与应用会议伴随会议论文集（加拿大魁北克省蒙特利尔）（OOPSLA '07）。计算机协会，美国纽约州纽约，815–816。doi:10.1145/1297846.1297902 Shruti Padmanabha、Andrew Lukefahr、Reetuparna Das 和 Scott Mahlke。2015年。DynaMOS：异构核心的动态调度迁移。载于第 48 届微体系结构国际研讨会论文集（夏威夷威基基）（ MICRO-48）。计算机协会，美国纽约州纽约，322–333。doi:10.1145/2830772.2830791 Joshua Pereyda。2023年。Boofuzz：网络协议模糊测试。https://github.com/jtpereyda/boofuzz

LLVM 项目。2023年。libFuzzer：进程内覆盖率引导模糊测试。 https://llvm.org/docs/LibFuzzer.html Leelavathi Rajamanickam、NABM Saat 和 SNB Daud。2019年。软件测试：生成工具。 国际计算机科学与工程高级趋势期刊 8, 2 (2019), 231–234。Philipp Schaad、Timo Schneider、Tal Ben-Nun、Alexandru Calotoiu、Alexandros Nikolaos Ziogas 和 Torsten Hoefler。2023年。FuzzyFlow：利用数据流发现并消除程序优化错误。载于 国际高性能计算、网络、存储与分析会议论文集 (美国科罗拉多州丹佛) *(SC '23)*。计算机协会，纽约州纽约，美国，第 88 号文章，共 15 页。 doi:10.1145/3581784.3613214 Kostya Serebryany。2017年。OSS-Fuzz——Google 面向开源软件的持续模糊测试服务。USENIX 协会，加拿大不列颠哥伦比亚省温哥华。Raja Vallée-Rai、Etienne Gagnon、Laurie Hendren、Patrick Lam、Patrice Pominville 和 Vijay Sundaresan。2000年。使用 Soot 框架优化 Java 字节码：是否可行？载于 编译器构造，David A. Watt （编）。Springer，德国柏林海德堡，18–34。Junjie Wang、Bihuan Chen、Lei Wei 和 Yang Liu。2017年。Skyfire：面向模糊测试的数据驱动种子生成。载于 *2017 IEEE 安全与隐私研讨会（SP）*。579–594。 doi:10.1109/SP.2017.23 Xuejun Yang、Yang Chen、Eric Eide 和 John Regehr。2011a。发现并理解 C 编译器中的漏洞。 *SIGPLAN 通讯* 46, 6（2011年6月），283–294。 doi:10.1145/1993316.1993532 Xuejun Yang、Yang Chen、Eric Eide 和 John Regehr。2011b。发现并理解 C 编译器中的漏洞。载于 第 *32* 届 *ACM SIGPLAN 编程语言设计与实现会议论文集* (美国加利福尼亚州圣何塞) *(PLDI '11)*。计算机协会，纽约州纽约，美国，283–294。 doi:10.1145/1993498.1993532Guixin Ye、Zhanyong Tang、Shin Hwei Tan、Songfang Huang、Dingyi Fang、Xiaoyang Sun、Lizhong Bian、Haibo Wang 和 Zheng Wang。2021年。通过深度编译器模糊测试对 JavaScript 引擎进行自动化一致性测试。载于 第 *42* 届 *ACM SIGPLAN 编程语言设计与实现国际会议论文集* (线上，加拿大) *(PLDI2021)*。计算机协会，纽约州纽约，美国，435–450。 doi:10.1145/3453483.3454054 Alon Zakai。2011年。Emscripten：LLVM 到 JavaScript 编译器。载于 *ACM 面向对象编程系统、语言与应用国际会议伴随会议论文集* (美国俄勒冈州波特兰) *(OOPSLA '11)*。计算机协会，纽约州纽约，美国，301–312。 doi:10.1145/2048147.2048224 Michal Zalewski。2023年。AFL：American Fuzzy Lop。 https://github.com/google/AFL