Vx：一种面向可扩展、可维护和安全系统的编程语言

ADITYA KUMAR，独立研究者，美国

面向现代机器的程序会运行在多种芯片上。如今的编译器看不到这些机器的内存空间、连接和容量。因此，主机读取设备内存、工作集无法容纳，或发生数据竞争，只有在程序运行时才会显现。Vx 是一种系统编程语言，它将这些事实写入程序文本。机器是一份声明源文件。值的类型表明它存放在哪个内存空间中。除一个明确说明的例外外，函数的签名就是调用方所能看到的完整契约。标志列出了程序中的代码和事实，这些内容无需证明即可得到编译器的信任。本报告将概览该语言在某次提交时的状态，介绍六项设计原则、核心语言，以及服务于三种特性的功能。Vx 具有可扩展性，因为同一份程序文本可以跨机器、核心和 GPU 线程运行。通过一个 Option，编译器还可以并行检查函数体。它具有可维护性，因为签名就是函数的契约，并且大多数诊断信息都带有代码。并行编译器的输出不取决于线程数量。它具有安全性，因为安全代码具备所有权、借用、竞态和放置检查。在安全代码中，标准库会要求获得打开文件或套接字的权限，但列出的少数例外除外。对于每项功能，本报告都会给出其状态及相关证据，同时列出实现尚未达到设计要求的部分。

附加关键词与短语：系统编程语言、异构计算、所有权、数据竞争自由、编译时求值、契约、并行编译、机器模型

arXiv:2611.32217v1 [cs.PL] 24 Nov 2026

1 引言

如今的大型程序并非运行在单个处理器上。模型服务器将权重保存在图形处理器内存中，通过主机内存分阶段处理请求，并将工作发送到其他机器。模拟程序将循环分配到 CPU 核心和图形处理器线程上。此类程序的每个部分都有自己的内存空间、彼此之间的连接方式，以及各自的限制。

这些程序所使用的编程语言并未意识到这些事实。在 C++ 和 CUDA 中，指向图形处理器内存的指针与指向主机内存的指针具有相同的类型。容量要么是配置文件中的一个数字，要么只存在于程序员的脑海中。因此，错误往往很晚才会显现。

读取设备指针的主机操作会导致崩溃。工作集超出容量时，往往要等到有人租用硬件后才会失败。数据竞争会在上千次运行中偶尔破坏一次结果。Rust 消除了数据竞争
安全代码 [Jung 等人 2018]，但它也没有表示“此值位于图形处理器内存中”的类型。

Vx [Kumar，2026 年i]是一种系统编程语言，它将这些事实写入程序文本中。其核心理念是机器是程序的一部分。机器是一个源文件，其中声明了内存空间、容量、连接和执行层级。值的类型会指出它存放的位置，因此张量`<f32, [4, 4], Memory::GPU_`高带宽内存`>`和张量`<f32, [4, 4]>`是不同的类型。设备无法寻址的代码无法通过针对该设备的编译，而错误信息会指出可修复问题的传输操作。

三种特质。标题列出了三种特质。本报告以狭义解释每个词，并为每项主张提供证据。

- **可扩展**。同一份程序文本可在不同机器、CPU 核心和 GPU 线程上运行。编译器会检查每个值的放置位置是否符合各个已声明的机器。编译器也能随程序规模扩大而扩展。它可以并行检查函数体，因为任何函数体都不会添加其他函数体所需的事实。

- **可维护性**。 读者可以根据函数的签名和函数体理解函数。没有重载、没有隐式转换，也没有隐藏的数据移动。大多数诊断信息

作者联系方式：Aditya Kumar，hiraditya@msn.com，独立研究者，美国。

都有对应的代码，每段代码都有一个页面。编译器的输出不取决于线程数量，而且有一种工具可以报告库接口在两个版本之间发生了哪些变化。

- **安全。** 安全代码具有所有权和借用检查机制，并会检查任务是否发生竞态。其他检查确保设备只读取其地址范围内的内存。整数溢出具有明确的结果，并且契约会通过了解这一点的编码进行检查。在安全代码中，`File::open` 以及标准库的套接字调用都需要权限，而函数会从其调用者处获得该权限。第6.5 节列出了例外情况。编译器会在没有证明的情况下信任某些代码和某些事实，并通过一个标志列出程序中的这些部分。

年轻语言的状态。 Vx 是一种早期研究型语言。它的首次提交来自2026年5月，本报告描述的是2026年11月23日提交719726333（版本0.0.5）时的主分支。介绍一种年轻语言时，很容易声称其能力超过编译器实际支持的范围。因此，本报告中的每个特征都带有一个状态词：

- [已测试]：代码已存在，并由持续集成（CI）进行测试；
- [手动测试]：它曾在真实硬件上运行，例如图形处理器，但没有持续集成任务运行它；
- [部分实现]：描述中的某些行为可以正常运行；
- [已设计]：文档对其进行了描述，但编译器尚未实现。

第 6.8 节列出了安全代码目前尚不安全的各个位置，第 8 节则在一个表格中汇总了每项特征的状态。

贡献。 本报告有四项贡献。

(1) 本报告在同一个提交版本中描述了整门编程语言：核心语言，以及用于 scale、可维护性和安全性的特征（第 3–6 节）。报告还描述了实现（第7 节）。(2) 报告阐述了六项设计原则，以及使每项原则得以成立的编程语言规则。对于每项原则，报告都指出一个实现尚未遵循该原则的位置（第 2 节）。(3) 报告给出了每项特征的状态及其证据：测试、硬件和形式化模型（第 8 节）。(4) 报告说明了哪些方面尚不安全或尚未实现，并列出跟踪每项差距的问题单（第 6.8 和10 节）。

配套论文。 Vx 的若干部分各自都有专门的论文，本报告会引用这些论文以获取详细信息。舰队级报告描述了机器、放置以及 跨 循环的多节点工作 [Kumar, 2026 年]。Must-Fold 和编时报告描述了编译时求值及其可靠性 [Kumar 2026c, f]。结构化任务和足迹描述了并发模型及其 Lean 证明 [Kumar 2026g]。Erasable Parallelism 描述了图形处理器循环嵌套 [Kumar 2026d]。

Agree Without Locking 描述了并行编译器前端 [Kumar 2026a]。一项对机器模型的研究使用租用的硬件测试机器文件 [Kumar 2026e]。一项对大语言模型服务的研究使用真实GPU测试容量判定 [Kumar2026b]。

第一个程序。 代码清单 1 展示了 Vx 程序的形状。它在主机内存中创建两个矩阵，将它们移动到加速器的内存中，在那里将它们相乘，然后把 Result 移回。

如果程序在块内使用 a 并删除使 a_npu 的那一行，编译器会拒绝它：

```
错误[E6003]，位于 first.vx:7:9：'a' 存放在 CPU_动态随机存取存储器中，但 NPU[0]只能访问 [NPU_高
带宽内存]；请插入一次显式传输，将数据传输到 NPU_高带宽内存（在声明的路径上成本为 50）
```

编译器知道设备无法访问主机内存，因为内置的 NPU 拓扑表明，设备只能看到 NPU_高带宽内存。它还会查找已声明的成本最低的路径及其成本。在我们运行此程序的 Apple M4 上，运行时未能为神经引擎找到预构建的模型。它

```
fn main() -> i32{ let a = 张量<f32, [4, 4]>::填充(2.0); let b =
张量<f32, [4, 4]>::填充(3.0); let mut c = 张量<f32, [4, 4],
内存::NPU_高带宽内存>::新建(); let a_npu = 传输(a, 内存::
NPU_高带宽内存); let b_npu = 传输(b, 内存::NPU_高带宽内存);
spawn on( 拓扑 ::NPU[0]) { c = a_npu @ b_npu; } let c_ 主机
 = 传输(c, 内存::CPU_动态随机存取存储器); 打印(c_主机[0][0]); //
24 return 0; }
```

清单 1。第一个程序。内存空间之间的每次移动都是显式传输，而 spawn on 则指定运行该块的设备。

打印结果表明该块运行在 CPU 上，结果为 24。随附的 M4 机器文件拒绝此程序，因为其神经引擎未声明 `f32` 类型（E6026）。第 4.11 节说明了每个目标上运行的内容。

2 设计原则

六项原则塑造了 Vx。本节针对每项原则，给出使其成立的编程语言规则，同时指出 719726333 中的实现尚未遵循该原则的一处位置。这些反例并非旁枝末节。只有当读者能够依据某项原则检验代码时，该原则才具有实际意义。

2.1 P1：机器是程序的一部分

Vx 程序会说明每个值存放在哪里，以及每个块由哪个设备运行。项目宣言用一句话概括了这一点：“异构性属于类型系统，而非运行时。”三条规则确保这一点得以实现。

- 机器是一份由声明组成的源文件（第 4.1 节）。编译器通过 `-machine` 读取它，而程序不会指定该文件的名称。
- 值的类型包含其内存空间信息。张量类型具有元素类型、形状和空间；原始指针也可以指明其空间（Memory::GPU_HBM 中的 *mut f32）。
- 数据只能通过显式传输来移动。即使是在统一内存芯片上，这条规则也成立。在这种芯片上，模型中复制不产生任何开销；如果每次使用都只是读取，编译器便会跳过复制。移动操作仍会写明，因此读者可以看出程序在何处移动数据。

代码尚未遵循这一规则之处：编译器每次编译只读取一个机器文件，而且机器文件不能导入另一个文件。因此，节点文件会复制一块 GPU 的声明。八块 GPU 的文件将其余七块 GPU 放入同一个 560 GiB 空间，因此允许的放置方案可能需要的容量超过任何一块 GPU 所能提供的容量。

2.2 P2：签名就是完整契约

Vx 函数的调用方只需要函数的签名。这一原则同时服务于规模扩展和可维护性。它使编译器能够并行检查函数体，也使读者无需查看被调用函数的函数体，就能理解一次调用。项目将其概括为八条规则，每条规则都是 C++ 或 Rust

表1. 这八条规则让编译器在检查任何函数体之前冻结每个全局事实，摘自 docs/language_rules_for_a_parallel_compiler.md，并附有 C++ 或 Rust 中会破坏各条规则的特征。

Vx 规则	一个破坏它的特征
签名就是完整的契约：每个 写明参数和结果类型。	C++auto f(); Rust -> impl Trait 和 async fn，其中 a 结果的属性来自函数体。
冻结时已知每种布局。	Rust [u8; n()]：大小来自运行一个 const fn。
泛型在其定义处闭合。	C++ 使用后的特化和实参依赖查找； Rust 特化。
一个名称，一个函数：没有重载—— ing.	C++ 开放的重载集合。
没有隐式转换。	C++转换构造函数和 operator T()。
Copy 和 Drop 属于类型。	C++在首次使用时生成的特殊成员函数。
宏在各自的阶段展开。	C 预处理器；Rust 宏展开与名称 解析交错进行。
函数体不会声明任何全局内容。	Rust 在函数体内为 S 实现 impl Trait。

中断（表1）。编译器检查任何函数体之前，都会收集所有全局事实（每个签名、布局和实现），并将其冻结。函数体只读取冻结后的表。

一个全局事实仍可能出现在函数体内：泛型的新实例化。编译器为它赋予一个身份，不受哪个线程先 find 到它的影响（第4.9节）。

代码尚未遵循这一点的地方：Vx 签名没有生命周期。因此，编译器会从被调用函数的函数体中读取关于结果的一条事实。这条事实指出返回的引用可能指向何处（第3.5节）。编译器会在检查任何函数体之前，从每个函数体的语法中读取这条事实，因此检查可以保持并行。但函数体的更改可能改变哪些调用方能够通过编译，文档对此有明确说明。

2.3 P3：宁可拒绝，也不猜测

当编　　译器无法判定某条事实时，就会拒绝该程序。它不会选择一个看似合理的答案 .

- 一个编译时块若无法由编译器完成，就会报错（E3033）。不会静默回退到运行时（第3.7节）。
- 契约需要 Z3 求解器。缺少 Z3 时，每个契约都会失败，并显示说明原因的警告（第 3.8 节）。
- 已编译库接口的读取器会安全失败。对于无法编码的签名，它会跳过；同时，它会检查每次读取的边界。
- 无法寻址某个空间的设备不会进行静默 Copy。程序会收到 E6003，消息中会指出此次传输。

代码尚未遵循这一点的地方：运行时边界检查默认处于关闭状态。因此，当编译器无法在编译时检查索引时，程序不会在运行时检查该索引（第 6.8 节）。此外，程序放置在 Apple 神经引擎上的大多数区域都会在 CPU 上运行，且不会显示任何消息（#3266）。有一条路径仅根据缓冲区的形状选择模型，并会产生错误结果（#3793，第 4.11 节）。

2.4 P4：每个数字都有源

容量判定的可靠程度取决于所使用的容量。因此，机器文件中的每个数值都附有一条注释，注明其来源。来源可以是供应商文档（spec:）或测量结果（measured:）。没有来源的数值应标记为 UNVERIFIED。随编译器附带的 16 个机器文件中，

共有 126 条 spec: 注释、6 条 measured: 注释和 42 个 UNVERIFIED 标记。单位是精确的：GB 等于 10^9 字节，而吉比字节等于 2^{30}，因此，从供应商表格中复制的数值与表格中的含义一致。

一项机器模型研究租用了硬件、测量了数据，并记录了声明中的错误之处 [Kumar 2026e]。该项目的基准测试政策对性能声明也采用同一规则：要求提供绝对数字、最佳基线和至少三次运行的数据，并将性能损失与性能提升放在同一张表中。

代码尚未遵循这一规范的部分：来源注释目前只是一种惯例。没有任何工具会读取这些注释，持续集成也不会检查每个数据是否都有来源注释。H100 文件中，一些片上数据既没有来源，也没有标记。英特尔 Arc 文件声明一个块中有 1,024 个线程束。注释称这一数据未经测量，但该数据没有标注“未经验证”。

2.5 P5：在每个目标上含义一致

Vx 程序在每个目标上的含义都相同，项目通过测试同一程序的第二个版本来验证这一点。

- 一个 跨步 循环会在图形处理器上并行执行各遍。删除每个 跨步，循环便会按顺序运行并得到相同结果，除非内核调用 `lanes::` 例程（第 4.7节）。编译器还会为每个内核生成一份 CPU 副本，测试可以比较两者。
- 编译时块给出的值与同一段代码在运行时给出的值相同。测试套件会按原样运行每个程序，再将编译时替换为普通块后重新运行，两个输出必须一致。
- 编译器有两个代码生成器，差分模糊测试器会将每个生成的程序分别交由这两个代码生成器处理，并通过等效的 Rust 程序运行。
- 使用 `-j`时，编译器在 1、4 和 8 个线程下生成的输出字节完全相同（第 5.4节）。

代码目前尚未遵循之处： 两个代码生成器在某些程序上的结果仍不一致。比较它们在整个测试套件上的结果的任务于2026年11月10日启动，其四次运行均在11月23日前失败。

2.6 P6：说明哪些内容受信，以及每项承诺在哪里止步

每种编程语言都会信任某些代码。Vx 试图说明哪些代码受信。一个 不安全代码 块、一个 不安全函数、一个不安全实现 以及一个 外部实现 都会标记出编译器无法检查其声明的代码。外部函数是一项未经检查的承诺。位于 `mlir!` 块之外的 不安全代码 必须是编译器能够验证的代码。标志 `-audit` 会列出程序中的受信代码和受信事实，并统计 `core` 和 `std`中的相关内容。标志 `-no-trusted-code` 会拒绝软件包中的受信代码（第 6.6节）。

本书关于并发的页面会陈述每项承诺，然后说明它在哪里止步。它还会说明每项承诺如何得到检查：通过关于模型的证明、由模型检查器检查编译代码，或仅通过测试。宣言将这条规则表述为一种拒绝：“Vx 不会声称任何未经其检查的内容”。

代码尚未遵循这一点的地方： 已声明为安全 的外部函数在其调用处无需 不安全代码 标记。因此，文件可以将 C 函数 `system` 声明为 安全 ，并在安全代码中运行 Shell 命令。 `-audit` 会列出该声明，但每次调用看起来都像调用安全的 Vx 函数（第 6.5节）。

3 核心语言

本节将介绍供单个 CPU 使用的 Vx 部分。Rust 与 Vx 最为接近，因此本节常会说明 Vx 与 Rust 的差异。每个小节末尾都会列出其中各项特征的状态。

3.1 条目、类型和数字

Vx 文件是一个模块。它声明函数、结构体、枚举、特征、实现、常量和宏。函数会写明每个参数及其结果的类型：没有结果时为 -> void，永不返回时为 -> !。常量需要指定类型。Vx 没有 static、union、内联汇编或属性语法。

整数类型分为有符号和无符号类型，位宽范围为 4 到 64 位（i4 . . . i64、u4 . . . u64），以及 usize。浮点类型包括 f16、bf16、f32 和 f64，此外还包括用于机器学习的八位和四位格式。向量类型写作 <N x T>。不存在隐式转换。所有转换都必须使用显式 as，数字字面量则从程序使用它的位置获取其类型。不适合其类型的字面量会导致错误（E3045）。

整数算术对每个输入都有明确的结果：

- +、- 和 * 会按照类型的位宽回绕，就像二进制补码硬件一样；
- / 和 % 遇到零会停止程序，最小值除以 −1 也会停止程序；
- 对 usize 进行算术运算时，如果发生溢出，程序会停止运行；usize 的取值范围是从 0 到 i64 的最大值。
- as 转换会回绕，但超出范围的值转换为 usize 会导致程序停止运行。编译器知道该值时，也会发出警告 W1014。

移位量达到或超过类型的位宽，是唯一的未定义情况，文档对此也有说明。契约检查器使用相同的回绕规则（第 3.8 节），因此溢出不会让错误的后置条件看起来成立。

状态： [已测试]。128 位整数属于 [部分支持]：算术运算可用，但代码生成器无法处理打印。

3.2 数据：结构体、枚举、元组和张量

结构体包含命名字段，结构体字面量按声明顺序书写这些字段（E3062）。枚举由标签和负载组成，变体始终会写出其类型，例如 Option<i64>::Some(3)。元组 (A, B) 是库结构体 Tuple2<A, B>，提供包含两个到六个元素的版本；解析器会将元组模式重写为该结构体。

Vx 没有单独的数组类型。定长数组就是张量：Tensor<i64, [4]> 包含四个整数，而 Tensor<f32, [64, 300]> 是一个矩阵。维度可以是数字、常量泛型，或表示只有在运行时才知道大小的 ?。张量的行、行范围、重塑结果和字段都是视图，借用检查器会将每个视图视为一次借用。不存在切片类型 &[T]。递归数据类型需要通过 Box，例如 Option<Box<Node»。

match 接受枚举变体（带有名称或 _ 负载）、字面量、元组和 _。编译器会检查对枚举或布尔类型执行的 match 是否覆盖每种情况。Vx 目前还没有守卫、或模式、范围模式、结构体模式、嵌套载荷模式或 if let。

状态： 结构体、枚举、元组和张量 [已测试]；模式匹配 [部分支持]。这里有两个缺陷。一个结构体在没有 Box 的情况下包含自身时，编译器会导致其堆栈溢出，而不是报错（#3786）。此外，对数字执行 match 时，覆盖检查只有在 match 返回值时才会运行。因此，不带 _ arm 的 match 语句可能会让函数执行到末尾（#3785，第6.8节）。

3.3 泛型和特征

泛型通过单态化进行编译：编译器会为每组类型参数生成一个泛型函数的副本。类型参数可以具有特征约束，写作 <T : Area>。Aconst 泛型参数可以表示一个维度，因此函数只能接受内维度一致的矩阵（否则会出现 E3003）。列表 2 展示了一个特征、两个实现以及一个带约束的泛型函数。

特征 Area **{函数** area(self : &Self) -> f64; **}结构体** Rect { w : f64, h : f64 **}结构体** Square { s : f64 **}实现**Area**针对**Rect **{函数** area(self : &Rect) -> f64 **{ 返回** self.w * self.h; **} }实现** Area **针对** Square **{函数** area(self : &Square) -> f64 **{ 返回** self.s * self.s; **} }函数** twice<T : Area>(x : &T) -> f64 **{返回** 2.0 * x.area(); **}** 列表 2。一个特征、两个实现，以及一个带约束的泛型函数。它打印出 12 32。

特征具有默认方法和关联类型。该库定义了运算符特征（加法、乘法、取负等）和比较特征。它还定义了克隆、Copy、默认、From、哈希和Drop。标记特征包括Send、Sync、Owned 和HostOnly。分派是静态的。没有动态特征对象，没有impl Trait ，也没有where 子句，placement boundReachable除外（第4.2节）。

有两种选择与 Rust 不同。首先，约束会限制泛型的调用方。但编译器会针对每次实例化检查泛型的函数体，就像 C++ 检查模板一样。其次，当两个实现都匹配同一个张量接收者时，更具体的形状模式优先。对于泛型结构体，目前以文件中第一个实现为准（#3787）。

状态： [已测试]。只有声明结构体或枚举的文件才能为其编写普通实现（E3066、#3774）。宏所编写的实现视为调用它的文件中的代码。另一个文件可以编写外部实现，这属于可信代码。一致性仍有一些缺陷。如果某个字段的类型是参数类型，泛型结构体就可以实现 Copy，而不必对该参数施加约束。这样的结构体会复制其所持有的内容，而复制后的 Vec 会读取已释放的内存（#3861）。某个类型可以同时实现 Drop 和 Copy，此时其析构函数永远不会运行（#3742）。</output>

3.4 闭包和函数值

闭包会捕获其函数体使用的内容。捕获模式由函数体决定。函数体只读取的数字会被复制，而函数体修改的局部变量会被可变借用。函数体读取的结构体会被借用，而函数体移动的值会被移入。没有 move 关键字。

闭包类型写作 `|i64| -> i64`，函数指针类型写作 `fn(i64) -> i64`。不安全函数 不安全函数有自己的类型，因此不能用于需要安全函数的地方。清单 3展示了三种捕获模式。

复制值的闭包会保持其副本。如果程序稍后更改变量并仍然调用闭包，警告 W1034 表示该闭包看不到更改。

状态： [已测试]。函数体移动其捕获的值的闭包只能运行一次。第二次调用会产生 E4001。复制、返回或传递此类闭包会产生 E4017 (#3789)。

```
let double = |x : i64| x * 2; // 不捕获任何内容 let k : i64 = 10; let 加法_k = |
x : i64| -> i64 { x + k }; // 复制 k let mut total : i64 = 0; let 加法 = |x : i64| -
>void { total +=x; }; // 可变借用 total 加法(5i64); 加法(7i64); let c =
Counter { count : 3 }; let peek = || ->i64 { c.count }; // 借用 c
```

清单 3。捕获模式，来自持续集成运行的书中程序。它打印 double 8 加法_k 14，然后是 total 12，最后是 peek 3 count 3。

3.5 所有权、借用和析构

Vx 的所有权模型来自 Rust。值会被移动，除非其类型是 Copy。编译器会跟踪每个分支以及循环每次迭代中的移动。读取可析构字段会将其从结构体中移出。

引用可以是共享的（`&T`）或独占的（`&mut T`）。值不能同时拥有独占引用和任何其他引用。清单 4 中的调用会被拒绝：

```
函数 二者(x : &可变i64, y : &可变 i64) -> 空 { *x = *x + 1; *y = *y + 1; }
...
both(&mut n, &mut n); // 错误[E4004]: 无法借用 'n' ，因为它已经被独占借用。
```

清单 4。一次调用中对一个变量的两个独占借用。

借用在其最后一次使用时结束，这与 Rust 的非词法生命周期相同。结构体的字段可以分别借用，重新借用也可以正常工作。张量视图是对其张量的借用。借用检查器逐个函数地处理每个函数的语法树，一次处理一个函数，因此编译器可以并行运行它（P2）。编译器中用于借用、移动和逃逸检查的 Rust 代码约有 7,800 行。

无需生命周期语法的区域。 Vx 没有生命周期参数。在函数内部，区域就是一个块。

跨调用时，编译器需要知道返回的引用可能指向何处。它从被调用函数的函数体中读取这一信息：对于每个函数，返回来源摘要 记录结果可能指向哪些参数。当分析无法跟踪某种情况时，摘要会说明结果可能指向每个引用参数。于是，调用方会保持其所有参数处于借用状态。程序员可以在 借用子句 中通过 确保子句声明一个更精确的契约。清单 5展示了一个示例。

如果没有该子句，对 key 的赋值会因 E4009 而被拒绝。find 的摘要无法排除结果指向 key 的可能性。这些子句是 points_into(return, *p)、reaches(return, *p) 和 keeps(p)，通常使用 ! 书写。编译器会针对函数体检查每个子句（E4016），而 –print-borrow-clauses 会打印它能够证明的子句。

析构。 当值的所有者离开作用域时，编译器会释放该值，并采用两种策略。只拥有内存的值（例如张量或数字向量）会被提前释放。编译器会在它最后一次使用之后，或对它的引用最后一次使用之后释放它。具有自身 Drop 的类型会一直等到其块结束，而值会按照其声明的逆序进行析构。

丢弃标记会处理仅在一个分支上移动的值。core::mem 提供 drop 和 forget。恐慌不会展开，因此恐慌后不会运行析构。

```
fn find(v : &Vec<i64>, key : &i64) -> &i64 确保子句 !可达(return,
*key) { for i in 0..v.len() { if *v.get_ref(i) == *key { return v.get_
ref(i); } } return v.get_ref(0); } fn main() -> i32 { let mut v = Vec<i
64>::new(); v.push(3i64); v.push(5i64); let mut key = 5i64; let r =
find(&v, &key); key = 7i64; // 允许：r 不指向 key 内部 println!("{}␣
{}", *r, key); return 0; }
```

清单 5。借用子句。find 承诺其 Result 不会指向 *key。编译器会根据函数体检查这一承诺。因此，Caller 可以在仍持有 r 的同时修改 key。程序打印 5 7。

文档将那些不得被静默析构的张量称为“线性”张量。实际实现的规则是仿射的：未使用的设备缓冲区只会触发警告 W1001，forget 可用，而 transfer 会复制其输入。唯一的必须使用检查是 E6036，针对程序从未等待的异步传输令牌。

状态： 移动、借用、借用子句和析构 [已测试]；线性值 [已设计]。问题 #1501 跟踪借用检查器的可靠性。其中列出了约 33 个已修复的缺陷。其中 14 个源自 Rust 编译器的 78 份缺陷报告，项目将其转换为 Vx。一个 drop 缺陷尚未解决，其中有两个探测程序：持有引用的 struct，其 `Drop` 可以读取已释放的内存（#3784，第 6.8节）。

3.6 错误与恐慌

Vx 没有异常。`Option` 和`Result` 是库中的枚举。运算符 ? 适用于`Result`，它会原样返回函数中的错误。它不会转换错误，也不适用于`Option` (E3063)。清单6 使用了? 和错误枚举。

panic(msg) 会打印消息和调用位置，然后停止程序。Caller 类型的参数会获取调用位置，因此库函数可以报告其调用者所在的行。库中还有 abort、unreachable、todo 和 unimplemented。函数体可能正常结束的 -> ! 函数会产生错误（E3028）。

状态： [已测试]。不会进行栈展开。

3.7 编译时求值

一个编译时块会在编译器内部运行，编译器会用其值替换它。清单7在编译时计算阶乘。`if comptime` 会在编译器检查未执行的分支之前将其移除。函数体是编译时块的闭包必须在每次调用时都能折叠求值。

该规则称为 *Must-Fold*：编译器无法完成求值的块会报错（E3033），编译器绝不会将其移至运行时。对会在自身作用域之外写入变量的块，也适用同样的规则。求值器设有预算：循环在迭代 1,000,000 次后停止（E8005），调用则会在深度达到

```
枚举 加法错误 { 不是数字(i64), 过大(i64), }函数 加法(a : Result<i64, 加法错误>, b : Result<i64, 加法错误>) -> Result<i64, 加法错误> {变量 sum = a? + b?;如果 sum > 99 {返回 Result<i64, 加法错误>::Err(加法错误::过大(sum)); }返回 Result<i64, 加法错误>::Ok(sum); }函数 report_error(e : 加法错误) ->空 {match e { 加法错误::不是数字(c) => { println!("error: the character {} is not a digit", c); } 加法错误::过大(n) => { println!("error: the sum {} is too large", n); } } }
```

列表 6。错误作为值，摘自一本书中的程序，持续集成会运行该程序。调用中 sum 的值超过 99 时会返回 TooLarge，而 report 会打印错误：sum 108 过大。

```
fn fact(n : i64) -> i64 {if n <= 1 { return 1; }return n * fact(n - 1); }fn main() -> i32 {let table = comptime { fact(10) }; println!("{}", table);return 0; }
```

列表 7。编译时求值。这个块在编译器中运行，程序会打印 3628800。

256 (E8004)。块中的失败断言会中止编译（E8002）。任何其他会导致程序停止的检查，例如除零，都会产生 E3033。块中的整数溢出没有对应的值，因此该块会被拒绝。

Vx 比 C 更进一步++ `constexpr` 和 Zig `comptime`。块可以引用一个仅在运行时才确定的变量，前提是结果不依赖于它。随后，编译器会删除这个块并保留它的值。Must-Fold 论文阐述了这条规则，并在 Lean 中证明，对于核心演算而言，这种删除是可靠的 [Kumar2026f]。

每个块的运行时孪生体都是测试预言机。测试程序先运行包含该块的程序，再运行将该块作为普通代码的程序。两次输出必须一致。一个生成器生成了 57,950 个这样的程序，没有任何程序折叠为错误的值。稍后的一次提交重新运行了前 33,493 个程序，也没有发现错误 [Kumar, 2026 年, f]。

宏是macro_规则定义，包含 expr、ident 和 ty 片段。它们在自己的阶段展开，在任何名称解析之前（P2），嵌套展开上限为 64 次。文件可以调用自身的宏以及所导入文件的宏。一个调用如果可能表示两个宏，则会报错（#3768）。如果某个文件的宏所生成的代码中出现错误，错误位置会指向该调用（#3771）。

状态：[已测试]。求值器使用 64 位整数进行计算，持续集成中的测试套件通过运行时双重检查运行了 13 个程序。

3.8 契约

函数可以声明前置条件（requires）和后置条件（确保子句，其中 return 表示结果）。循环可以声明不变式，结构体可以声明不变式，而assert 可以声明一个事实。编译器会将每个契约转换为一个查询，交由 Z3 求解器 [de Moura 和 Bjørner 2008]处理。它必须在每个返回处证明每个确保子句（E8001），并在每次调用处证明每个requires （E8006）。调用时，它会使用被调用函数的确保子句。编码对路径敏感，并对 +、-、* 和取负进行建模，并在类型位宽范围内回绕。

回绕很重要。清单8 中的契约看似为真，但对于最大的i32却是假的，因为 x + 1 会回绕。Z3 找到了这种情况，编译器因此拒绝该函数。加入前置条件 requires x < 2147483647后，编译器就接受它了。

```
fnadd_one(x:i32) ->i32确
保子句 return >x{return
x + 1;}
```

清单 8。整数溢出会破坏的后置条件。没有前置条件，编译器会因 E8001 而拒绝它。

求解器作为独立进程运行，并具有固定的资源限制。因此，某个版本的 Z3 每次都会给出相同的答案，但 Vx 不会固定该版本。缺少 Z3 时，每个契约都会失败（P3）。

状态： 需要子句、确保子句、循环不变式和断言 [已测试]。共有 137 个使用需要子句或确保子句的测试程序。其中，47 个位于契约测试目录中。结构体不变量是 [部分]的。目前，它们会将类型的字段设为其模块私有，但编译器不会证明每个值都满足该不变量。包装类型 Verified<T> 标记其断言已得到证明的结果，但任何代码都可以构建这种类型（[部分]）。除法尚未建模。

3.9 不安全代码、外部函数和内联 MLIR

不安全代码块允许执行编译器无法检查的操作。在其中，程序可以通过原始指针读取或写入（在不安全代码块之外为 E5002）。它可以调用不安全函数或外部函数（E5001），也可以创建未初始化值（::uninit()）。不安全函数内部仍然需要使用不安全代码块。不安全代码中的放置检查仍会启用：在那里，主机读取设备内存同样会触发 E6003。

原始指针是 *const T或 *mut T，并且可以标明其内存空间。指针可以在契约中携带 指针子句： valid(p, n)、initialized(p, n)、!overlaps(p, q, n) 和!frees(p)。安全调用必须证明其被调用函数的子句，而不安全代码 调用则信任这些子句。这些子句还会告知 LLVM，指针不为空且不与其他指针别名。假设 在不安全代码 中表示在没有证明的情况下陈述一个事实。代码清单 9 展示了一个带有两个指针子句的函数。

外部函数在 extern "C" 块中声明。除非声明将其标记为 safe fn（这是一项未经检查的承诺），否则对它的调用是不安全的。安全外部函数不能返回带有不变式的类型的值（E8015）。它也不能接受可变引用或指向此类类型的指针，因为 C 代码不会维护该不变式。外部函数的确保子句会受到信任。pub fn 会使用 C 调用约定导出，因此 C 代码可以调用 Vx。

mlir! 块嵌入 MLIR 操作。它声明其输入、Result、所使用的方言以及会破坏的内容。编译器会检查方言列表（E8010），并解析操作。在 unsafe 之外，块必须是编译器能够验证为安全的块（E8016）。这样的块只使用 arith 和 math 方言。它只接收和返回数字或数字向量，并且不会写入程序中的任何变量。它的每个操作都位于一个操作列表中，该列表提供一个

```
函数 sum(p: *常量i64, n: i64) -> i64要
求 valid(p, n)要求 initialized(p, n) {令
mut total = 0i64; 对于 i 在 0i64..n{
total = total + 不安全代码 { p[i] }; }
返回 total; }
```

列表 9。指针子句。sum 的安全调用者必须证明 p 对 n 个元素而言有效且已初始化；不安全调用者则信任它，而 –audit 会列出这种信任。

为每个输入定义结果，例如不带溢出标志的 arith.addi。从地址加载、整数除法以及带溢出标志的加法不在列表中，因此需要不安全代码。只有包含不变式的类型的文件，才能在 mlir! 块中生成该类型的值（E8014）。此规则在不安全代码中同样适用。

状态： 不安全代码、原始指针、指针子句、`extern` 和 `mlir!` [已测试]。通过原始指针进行方法调用现在也需要 不安全代码 （E5002，#3798）。编译器不会检查 `mlir!` 块在 不安全代码 中的操作。E8016 的检查存在缺陷：它每行只读取第一个操作。因此，一行包含两个操作的块可以在没有 不安全代码 的情况下从地址加载（#3864）。`-audit` 仍会列出这样的块。第 6.6 节将回到可信代码。

3.10 能力

能力是一种赋予执行某项操作权限的值。不持有该值的代码无法执行这项操作。Vx 根据其对具有不变式的结构体所施加的规则来创建能力（第 3.8 节）。此类类型遵循三条规则：

- 不可伪造。 只有声明该类型的文件才能构建它的值（E8014）。其他文件中的 mlir! 块无法创建此类值（E8014），安全外部函数也无法返回此类值（E8015）。
- 不可复制。 结构体会移动（E4001）。只有声明该类型的文件才能为其编写普通的 实现（E3066），因此其他文件无法让它实现 `Copy`。其他文件可以编写 外部实现，这属于可信代码（第 6.6 节）。
- 所有者销毁后不可再引用。 对能力的引用不能比它所借用的值存活更久（E4005），但 #3784 的间隙除外。所有者持有该值期间，代码可以保留借用引用。

不变式可以是 invariant true（不变式为真），这不陈述任何阶乘。它只会使类型成为仅能由其自身文件构建的类型。

标准库会对操作系统使用这些规则。模块 `std::root` 声明 `Root`，其中包含两项权限。 文件是打开文件的权限，而 网络 是建立连接和绑定地址的权限。程序会将 `Root` 作为 `main` 的参数，并将每项权限交给需要它的代码（清单 10）。`File::open`、`TcpStream::connect`、`TcpListener::bind` 和 `UdpSocket::bind` 都会将权限的引用作为第一个参数。因此，没有获得 `&Files` 的库无法在安全代码中调用 `File::open` 。不带参数的 `main` 仍然是正确的，而不需要任何权限的程序会保持这种形式。

状态： 这三条规则以及 Files 和 Network 权限 [均已测试]。对于每项权限，测试都会尝试复制它、构建它，以及在没有该权限的情况下打开文件或套接字。每次尝试都失败。单文件夹权限是

```
导入 std::fs; 导入 std::string; // 解析器没有权限，因此无法调用 File::open。 fn
parse_name(request : &字符串)-> 字符串 { return String::new(); // 从请求中读取
名称 } // 缓存仅在调用期间拥有文件权限。 fn open_cached(files : &Files, name : &
字符串) -> Result<File, 文件错误> { return File::open(files, name,
OpenMode::Read()); } fn main(root : Root) -> i32 { let request =
String::new(); let name = parse_name(&request); match open_cached(&
root.files, &name) { Result<File, 文件错误>::Ok(_file) => { return0; } Result<
File, 文件错误>::Err(_e) => { return 1; } } }
```

列表 10。能力。解析器没有权限，因此无法调用 File::open。缓存仅在一次调用期间获得文件权限。

[设计](#3750)。进程、内存映射和本机代码的权限尚无设计。可信代码仍然可以在没有权限的情况下访问操作系统：一个不安全代码块可以调用`File::open_c_str`或 C 函数。漏洞使安全代码能够复制权限，或从原始指针构建一个文件或套接字（表7）。第6.5 节说明这些权限赋予什么，以及它们的边界。

4 可扩展性

程序能够使用更多机器、更多核心和更多图形处理器线程，而无需更改程序文本，就称得上可扩展。随着程序规模增大，编译器检查程序所需的时间不会大幅增加，编译器就称得上可扩展。本节涵盖这两方面。首先介绍机器的声明，然后跟踪一个值从主机传到设备再返回的过程。接着介绍并行代码的三个层级：CPU 核心上的任务、图形处理器线程上的跨循环并行，以及在其他机器上执行的工作。

4.1 机器作为声明

机器文件是一个仅包含 内存 和 拓扑 声明的 Vx 文件。它描述一个部件：其内存空间、它们的容量和带宽、它们之间的连接，以及运行代码的线程层级。清单 11 展示了 NVIDIA H100 的部分文件内容。

内存声明给出容量、带宽和包含它的空间（within）。它说明该空间由硬件管理（缓存），还是由程序管理（显式）。它还给出该空间的作用域：设备、多处理器、块或线程。最后，它给出副本数量、分配单元（粒度）和对齐方式。

拓扑定义了架构、默认空间，以及其代码可见的空间。它还定义了所支持的元素类型、线程层级和传输边。每条边最多有一个成本来源（若有第二个，则触发 E6013）。源可以是链路速率、无量纲权重，或编译器根据内部信息推导出的带宽。

```
// 规格：NVIDIA H100 产品页面，SXM 列（已于 2026-08-03 验证）内存 HBM { 容量：80 吉比字节，带宽：3.35 太字节/秒，管理方式：显式，对齐：16 B，作用域：设备 } // 规格：NVIDIA Hopper 架构白皮书——50 MB 二级缓存。内存 L2 { 位于：Memory::HBM 内，容量：50 兆二进制字节，带宽：12 太字节/秒，管理方式：缓存，作用域：设备 } // 副本：实测：探针_报告在两台租用的 H100 SXM 设备上有=132 个流式多处理器 内存 SMEM { 位于：Memory::L2 内，容量：228 千二进制字节，带宽：128 B/周期，时钟频率：1.98 GHz，粒度：1 KiB，管理方式：显式，对齐：16 B，作用域：SM，副本：132 } // 规格：PCIe Gen5 每个方向 x16 = 63.0 GB/s。此 SKU 的数据未经验证。拓扑 Device { 架构：nvptx64，芯片：sm_90a，层级：［个线程块（最大_个线程束：1024，内存：cta），子组（线程束：32），线程束（内存：线程）］,内存：Memory::HBM，可见：［Memory::HBM、Memory::L2、Memory::SMEM、Memory::REG］,传输 Memory::CPU_DRAM -> Memory::HBM：63 GB/s，传输 Memory::HBM -> Memory::CPU_DRAM：63 GB/s，传输 Memory::HBM -> Memory::L2，传输 Memory::L2 -> Memory::SMEM copy_引擎 }
```

清单 11。fleet/h100-sxm.vx 的部分内容，注释已缩短。大多数数值都有源注释。一些芯片上的数值没有注释；文件头称它们未经验证。

编译器会检查文件本身。连接中存在环（E6006）、空间大于其父级（E6007），以及数值不是正数（E6008），这些都属于错误。

编译器附带 16 个机器文件。其中 10 个描述加速器。6 个是 NVIDIA GPU：A100 40 GB 和 80 GB、H100 SXM 和 PCIe、H200 以及 B200。其余 4 个分别是 AMD MI300X、Intel Arc A770、配备统一内存和神经引擎的 Apple M4，以及 Arm Cortex-M7 微控制器。其他机器文件描述了配备两块和八块图形处理器的节点，以及三台 x86 主机。

使用 –machine 可针对其中一个机器文件编译程序，而 –host 用于指定主机。程序文本无需更改。在持续集成中，有 249 个测试文件针对某个机器文件进行编译。

状态： [已测试]，且无需硬件：所有检查都在编译机器上运行。来源注释只是一种约定，没有任何工具会检查它（P4）。机器文件无法导入其他文件，因此各节点文件会重复声明其图形处理器（#3350）。

4.2 类型中的放置

张量的类型会标明其内存空间：Tensor<f32，[4，4]，Memory::HBM>。未指定空间的张量位于主机内存中。原始指针也可以标明其空间，指向一个空间的指针不能用于要求指向另一个空间的指针的位置（E6029）。函数可以针对空间进行泛型化。约束 Reachable< A, B> 表示，声明的路径会将数据从拓扑 A 传送到拓扑 B。

在（设备拓扑）上 spawn { . . . } 在设备上运行一个块。编译器会将该块提取为内核，并检查设备是否能够寻址该块读取的每个值（E6003，详见第 1 节）。普通函数属于主机，因此从设备块中调用它会产生错误（E6001）。在设备块中使用通道端点或锁也会产生错误（E6038），因为设备无法

等待主机内存。其含义是顺序执行的：spawn on 之后的语句会将该块视为已完成。

状态： [已测试]。目前，声明的 GPU 中只有设备 0 会运行代码（其他设备会触发 E6042）。有两个尚未解决的问题属于检查机制的漏洞。现在，在以数字结尾的区域上执行 spawn 会获得一个 GPU 内核（#3715）。在此类区域末尾的其他值仍会保留在主机上（#1809）。内核可以通过方法调用 CPU 专用函数（#3638）。

4.3 传输和路径

transfer(x, Memory::M) 给出一个 x 的 Copy，其类型名为 M。编译器会在已声明的边上查找路径，缺少路径会导致错误（E6002）。它会将包含多个跳转的路径拆分为每个跳转一个经过检查的传输。

为了选择一条路径，编译器首先会查找所有边都具有每字节价格的路径，该价格根据带宽或链路速率确定。它会选择其中成本最低的一条。如果没有任何路径的所有边都具有价格，则使用无量纲权重。价格按字节计算，因此路径不取决于张量的大小。一项测试会将每条选定的路径与针对所有机器文件执行的 Bellman–Ford 算法搜索结果进行比较。

transfer_async 会返回一个令牌，在令牌上调用 .wait() 会得到放置好的张量。程序丢弃令牌而不调用 wait，会导致错误（E6036）。编译器不会选择放置位置：程序员编写每次传输，而成本模型只解释其成本。

机器也可以在 Vx 中说明如何跨越一条边移动字节。它为 impl Transfer<Memory::A, Memory::B> for Topology::X 编写一个函数体。函数体只能使用底层原语（raw::load、raw::store、raw::barrier、raw::async_copy）。编译器使用 Z3 证明每个原语的边界，并拒绝在条件分支下使用屏障。

状态： 传输与路由 [已测试]；用户传输主体 [部分完成]，，因为编译器仅会为指向多处理器内存的边生成这些主体。模型没有针对每次传输的延迟，因此对小型复制的时间预测严重偏低。在 A100 主机链路上，4 KiB 时的预测值约低了 99% [Kumar 2026e]。模型也没有考虑数据流之间的争用。

4.4 容量

编译器会累加程序在每个具有容量的空间中放置的字节数，并拒绝无法容纳的程序。E6009 是一个大于其所在空间的单个张量。针对 H100 文件，下面的准入程序给出：

```
错误[E6009]位于 admit.vx:87:3：传输的张量需要 141104775168 字节，但内存空间 'HBM' 的容量为
85899345920 字节
```

E6010 是一组同时存活且合计无法容纳的块。该数量是跨作用域的峰值，而不是整个程序的总和。块会一直存活到其最后一个读取者；if 分支或循环函数体结束时，会释放其中放置的内容。粒度单元会将每个块向上取整；对于带有副本的空间，则按一份副本进行检查。

在跨调用的情况下，每个函数都会导出其在每个空间中的峰值摘要。对调用图进行折叠时，会将被调用函数的峰值加到调用时仍存活的内容上（E6027）。同时运行的任务会累加。在具有容量的空间中放置块的递归会被拒绝（E6028）。

对于相同的程序和机器，三个块的工作集为：

```
错误[E6010]：放置在“高带宽内存”内存空间中的工作集（三个块）总计 142514061312 字节，超过其
85899345920 字节的容量；请放置更少或更小的块，或声明它`超额提交`
```

一个程序，多台机器。 编译器随附一个准入程序 fleet/admit.vx。它以 16 位浮点数运行 Llama-3.1-70B，上下文为 4,096 个词元，批次为一。它根据模型形状计算权重、KV 缓存和激活值的大小

表 2。一个未更改程序（fleet/admit.vx：Llama-3.1-70B、16位、4k上下文、批次1）针对 15 个机器文件在 719726333 处的判定。容量是该程序所占用空间的容量。

机器文件	容量	判定	代码
b200	183,359 MiB	准入	—
h200	141 GiB	准入	—
MI300X	192 吉比字节	准入	—
H100-SXM，H100-PCIe，A100-80	80 吉比字节	拒绝	E6009，E6010
node-2gpu-a100，node-2gpu-h100，node-8gpu	80 吉比字节（一块 GPU）	拒绝	E6009, E6010
a100-40	40 GiB	拒绝	E6009 ，E6010
m4-uma	32 GiB	拒绝	E6009 ，E6010
Arc A770	16 GiB	拒绝	E6009 ，E6010
Xeon 8275CL ，Xeon E5-2666 v3	96, 30 GiB	拒绝	E6009 ，E6010
Cortex-M7	512 千二进制字节	拒绝	3 × E6009 ，E6010

参数，并使用 const泛型算术。该文件在不同机器之间始终不变。表 2展示了它针对每个机器文件的判定；这些机器文件描述了某个设备，来自主分支的一次构建。仅权重就需要 141,104,775,168 字节（131.4 吉比字节），因此只有内存更多的三台机器会准入该程序。

节点文件拒绝该程序，因为它没有将模型拆分到多个图形处理器上：它使用了该节点的一块 GPU。采用张量并行性的版本会将每个大小除以图形处理器的数量，而大语言模型服务的研究使用的是此类版本 [Kumar2026b]。

-diagnostics-json 会将判定写为记录，供其他工具使用。每个容量错误都会包含空间、所需和可用字节数，以及余量。每条路径都会包含其边、字节数、推导出的时间，以及各空间中的流量。一项关于大语言模型服务的研究使用了这些记录。在租用任何图形处理器之前，它就预测了哪些配置适用于哪些图形处理器。在 H100 和 A100 上进行的全部 24 次启动结果都与预测一致 [Kumar2026b]。

状态： [已测试]。通过函数值进行的调用不会计数（#3632），而动态形状会跳过检查，并发出警告 W1029。容量指的是设备的容量，而不是服务引擎预留的预算。

4.5 接缝检查

接缝是传输中的一次跳转。使用 –verify-seams 时，编译器会为每次跳转构建一个位向量查询，并运行 Z3。同步的传输会保持缓冲区的值。程序或机器文件标记为宽松的传输会使该值变为未知。跳转后读取该值会报错（E6004），而 Z3 模型就是反例。如果缺少 Z3，检查就会失败（E6024）；除非 VX_ALLOW_UNVERIFIED=1 将错误转为警告 W1031。

状态： [已测试]，，默认关闭。只有三个测试文件会启用它。答案仅取决于跳转是否采用宽松模式，因此查询决定语法检查将决定什么。契约的值部分将一个值与其自身进行比较（#662）。目前还没有关于接缝、传输或容量的形式化模型（#1209）。

4.6 在 CPU 核心上的任务

Vx 具有结构化并发。一个任务块会通过 spawn 启动任务，并且只有在其中每个任务都结束后，该块才会结束。因此，任务可以借用其周围代码中的局部变量，如清单 12所示。

h.join() 会给出任务的值。在任务块外使用 spawn 会导致错误（E3056），并且句柄不能离开其块。函数体给出 Result 的任务可能会失败。其失败会使

```
fn sum_range(v : &Vec<i64>, from: i32, to : i32) -> i64 {let
mut sum : i64 = 0;for i in from..to{ sum = sum + *v.get_
ref(i); }return sum; }fn main() -> i32 {let mut v = Vec<i64>::
new();for i in 1..101 {v.push(i as i64); }let total = 任务
{let low = spawn { sum_range(&v, 0, 50) };let high =
spawn { sum_range(&v,50, 100) }; low.join() + high.join() };
println!("total␣{}", total);return 0; }
```

Listing 12. 两个任务各自将向量的一半加起来，该向量由 y 从 main 借用。该 程序打印 总计为 5050。

块，其他任务则收到取消信号。当没有任何东西使用可能失败的任务块的值时，编译器会发出警告 W1038，并且该块会析构错误。

没有异步，也没有等待：等待是一次普通调用，同一个程序可以在任一运行时上运行。默认运行时为每个任务分配一个操作系统线程。第二种运行时将任务作为纤程运行在工作线程池上，并从同一个先进先出队列中取出任务。用于测试的第三种运行时会在单个线程上运行所有任务，运行顺序由一个种子决定。

借用检查器将每个任务视为一次借用，其持续时间从 spawn 开始，到 join 为止。因此，两个任务若要写入同一个变量，就会被拒绝：

```
错误[E4002]位于 race.vx:8:7：无法访问“x”，因为它正被可变借用。第 4 行 spawn 的任务会在第 5 行
使用它，而且该任务尚未 join。
```

移动或写入值的任务需要 Send，而在原位置读取共享值的任务需要 Sync。编译器会根据类型的字段推导出这两者。

该库提供有界通道，以及单次发送通道、广播通道、监视通道和工作队列。select2 到 select6 函数会等待不同类型的源。源可以是各种通道的接收端、套接字、信号量、区间或截止时间。每次调用都会给出第一个就绪源的索引。Caller 随后取出值，因此未获选的源会保留其值。互斥锁按照请求的顺序授予锁。该库还提供信号量、Arc，以及采用 C++ 内存顺序的原子操作。Vx 会将每个 Relaxed 读取编译为 Acquire 读取，以排除凭空产生的值；标志则会让宽松读取保持宽松。

当所有任务都在等待，且没有任何事情能够唤醒它们时，监视线程会终止程序。它会打印每次等待、导致各个通道或锁进入等待状态的代码行，以及等待环路。对于 select，它会指出该调用正在等待的每个通道和信号量（#3816）。在下面这次运行中，两个任务都在发送之前等待接收对方发送的内容：

```
vx：死锁：程序中的每个任务都在等待，没有任何东西能唤醒它们。程序停止运行。.....

  这些等待构成一个循环：来自 `spawn` 于 dl.vx:26 的任务在 `recv`（dl.vx:10）处等待通道 1，该通
  道创建于 dl.vx:21，只有来自 `spawn` 于 dl.vx:29 的任务才能在其上 Send；来自 `spawn` 于
  dl.vx:29 的任务在 `recv`（dl.vx:10）处等待通道 2，该通道创建于 dl.vx:22，只有来自 `spawn` 于
  dl.vx:26 的任务才能在其上 Send。这些等待所涉及的通道：通道 1，创建于 dl.vx:21： `recv` 中有
  1 个等待， `send` 中有 0 个等待。 1 `发送端` 尚未被 Drop。
```

```
函数 main() -> i32 {令 可变 grid = 传输(张量<f32, [64, 300]>::fill(1.0), 内存::
HBM);在...上启动(拓扑::设备) {对于 y 在 0..64 跨越 线程块 {对于 x 在 0..300 跨越
线程束(256){ grid[y][x] = grid[y][x] * 2.0 + (y 作为 f32); } } }令 back = 传输
(grid, 内存::CPU_动态随机存取存储器); 打印并换行!("角点␣{},␣最后一行␣{}",
back[0][0], back[63][299]);返回 0; }
```

清单 13。作为循环嵌套的 GPU 内核。外层循环的每次遍历都在各自的块中运行，内层循环则运行在 256 个线程束上。使用 H100 文件时，在没有 NVIDIA GPU 的 Mac 上，CPU 副本会打印 corner 2、last row 65；不使用 across words 时，程序打印的结果相同。

```
通道 2，在 dl.vx:22 创建： `recv` 中有 1 个等待，而 `send` 中有 0 个等待。 1 `发送端` 未被丢弃。
```

状态： 任务、通道、互斥锁、信号量、`Arc` 和原子操作 [已在] x86-64 上通过持续集成测试；`select` 跨不同类型的源进行选择 [已在]（#3807、#3814、#3815）上测试。死锁监视器是 [不完整的]。它仅在每个线程都处于等待状态时报告。在 0.0.4 构建版本中，如果某个任务在自旋，它旁边的死锁并未被报告。任务的借用检查会锁定整个变量，而不是字段：两个任务分别写入同一结构体的两个不同字段时，会被拒绝。第6.2 节描述了该设计的证明和模型检查。

4.7 GPU 线程上的循环

Vx 中的 GPU 内核是一个循环嵌套。紧跟在 `for`循环范围之后的单词，指定 GPU 运行该循环各遍所处的层级。可用的层级有 跨线程块、跨子组、跨线程束或跨线程束(n)。这些层级来自机器文件。清单 13对 GPU 内存中矩阵的每个元素加倍，并加上该元素所在的行号。

程序的含义不取决于 across：删除每个 across，循环仍会按顺序运行，结果也相同。这适用于不调用 lanes:: 例程的内核，因为该例程需要其跨循环设置。要保证这一点，编译器必须证明，没有两遍会访问同一个元素且其中一遍会写入该元素。对于循环嵌套写入的每个张量，每次访问都必须使用能够确定循环变量的索引。

编译器使用混合进制分离规则检查仿射索引形式。它接受常量间隔（`2*j` 和 `2*j+1`），也接受除以循环终值因子的除法和取余。它还接受嵌套数位，例如 `(l % 32) / 4`。它会拒绝可能被另一遍同时写入的写操作，也会拒绝读取另一遍写入的元素。它还会拒绝向循环外部的变量写入。如果清单 13 中的内核写入 `grid[0][x]`，而不是 `grid[y][x],`，那么每个线程块都会写入第 0 行，编译器会拒绝这种写法：

```
错误[E6032]位于 race.vx:6:9：此块循环的两次遍历可能会写入`grid`中的同一元素：写入的索引必须固
定 `y`
```

两种包装器类型各增加一个阶乘。类型 Distinct 表示索引张量中的元素各不相同。类型 DisjointRanges 表示范围表之间没有重叠。

每个车道循环的结尾都是块内线程束的屏障，因此每个线程束都必须到达这里。检查器要求，车道循环周围的每个条件对于块内的所有线程束都相同。车道循环之间用于写入内存的代码只在车道 0 上运行，其后会有一个屏障。

编译器会将嵌套结构降级为 MLIR GPU 方言中的网格步幅循环。随后，它会将这些循环降级为 NVIDIA 的 PTX、Apple GPU 的 Metal 着色语言，或 Intel GPU 的 SPIR-V。每个内核也有一个 CPU 副本，其中车道索引的值为 0，块计数为 1。因此，测试可以比较图形处理器结果与 CPU 结果。

《Erasable Parallelism》论文给出了竞争规则及其证明概要。论文将该规则应用于 llm.c、PolyBench/GPU 和 Rodinia 中的 38 个内核。检查器接受了 27 个，拒绝了 11 个，其中 3 个拒绝的确实存在数据竞争 [Kumar2026d]。

状态： 这些检查 [经过测试]，无需硬件，以静态检查的方式运行。NVIDIA 上的内核 [经手动测试]，使用 A100、A6000、H100 和 H200 运行，相关记录见提交和问题。持续集成会检查 93 个程序的 PTX 或 LLVM 文本，但没有图形处理器。Apple GPU 上的内核 [经手动测试]：有 29 个程序需要 macOS，而持续集成没有 macOS 测试任务。该数据竞争检查没有机械化证明。较旧的两个代码生成器之一会丢弃一些屏障（#3062），默认的代码生成器也可能退回使用它。</output>

4.8 张量与机器学习

张量类型包含元素类型、形状和空间。元素类型包括 bf16、八位浮点格式和四位整数。维度可以是 ?，表示其大小只有在运行时才知道。该库提供逐元素运算、归约、用于矩阵乘积的 @，以及视图。

自动微分是内置形式： `grad(loss, 3.0)` 给出导数，而 `vjp`和 `jvp` 给出向量–雅可比积和雅可比–向量积。编译器将相关工作交给 Enzyme [Moses 和 Churavy 2020]，，后者作为 LLVM 插件运行。模块 `std::nn` 包含 Transformer 解码器的 49 个函数，使用 `bf16`、八位和四位权重。模块`std::collective` 调用 NCCL 进行全归约和全收集。

状态： 张量和 `@` [已测试]；自动微分 [已测试]，但仅针对标量函数。检查器目前还不会拒绝张量参数，而这类参数随后会在 Enzyme 中失败。模块 `std::nn`已 [手动测试]，测试在 Apple GPU 上进行，并在持续集成中进行 CPU 测试。集合通信已 [部分], 在 CPU 上使用一个秩进行测试。对于形状为 `@` 的两个张量 `[?, ?]`，其内维度不会被检查。因此，这样的乘积可能会读到张量末尾之外，即使使用 `-fbounds-checks` （#3792，表 7）。

4.9 编译大型程序

P2 的设计规则允许编译器并行检查函数体。每个符号和类型都有一个 256 位的全局标识符（GID）。编译器根据模块路径和名称计算该标识符，因此它不取决于哪个线程先找到该符号。泛型实例化通过一个串行步骤按固定顺序获得编号。使用 -j N 时，编译器会分阶段运行其前端。大多数阶段都是针对模块或函数进行并行映射，并读取一张已冻结的表。注册表冻结、声明检查以及合并新增泛型实例化都是串行步骤，且合并按固定顺序进行。

输出不受线程数影响。持续集成会分别使用 1 个和 8 个线程以及顺序调度来编译程序，并要求生成的 MLIR 字节完全相同。另一个门禁会在包含 1,000 个模块和 16,000 个函数的程序上比较 1 个和 4 个线程。八月构建的前端在 48 个至强核心上编译为 MLIR，速度提升了 5.9 倍，并且在所有线程数下输出都相同 [Kumar 2026a]。九月在另一台 48 核机器上的测试，速度提升了 3.9 至 4.5 倍。对于一次完整编译，串行 MLIR 后端耗时最长。

状态： [已测试], 但需要选择加入。普通的 vxc file.vx 会运行顺序驱动程序，其输出在函数顺序上有所不同，并且只降低 main 可达的代码。并行路径不会在 Vx 代码中获取锁。但每个阶段都是线程池的分叉与汇合，而阶段之间的合并是串行的。目前有六个标志不能与 -j 一起使用，其中包括 –verify-seams、–diagnostics-json 和 –print-borrow-clauses。

表 3. 719726333 中的后端，以及各自的 测试方式。

目标	Path	状态
CPU（x86-64、AArch64）	MLIR → LLVM IR → 目标文件，通过子进程	[已测试]（持续集成中的 x86-64）
NVIDIA GPU	MLIR 图形处理器 → NVVM → PTX；CUDA 驱动程序；cuBLAS 适用于带标签产品	[经人工测试]
Apple GPU	MLIR 图形处理器 →Metal 着色语言	[经人工测试] (macOS)
Apple 神经引擎	四种预构建的 CoreML 模型，按模式选择；否则 CPU	[部分]
Intel GPU	SPIR-V、Level Zero、oneMKL；跨循环、通道例程、共享瓦片和原子操作；无持续集成	[部分]（单卡运行）
AMD GPU、Cortex-M7	仅机器文件；无代码生成器	[设计]
远程工作进程	基于 TCP 的线路协议	[已测试] (环回)

4.10 在其他机器上运行

带有 offload: true 的拓扑可能会在另一台机器上的工作进程中运行。程序启动时，清单会将拓扑映射到工作进程，线路协议则传输缓冲区。当主机失去工作进程时，会因错误而停止：主机会检查链接，并在 30 秒内未收到响应后停止。带有 node: N 的内存是主机的 NUMA 域。运行时会将放置绑定到其节点，并固定读取它的线程。在双插槽 c5.metal 上，远程成本与本地成本的预测比率为 2.26，实测比率为 2.35。

状态： [已测试] 在一台机器上运行，工作进程通过环回接口连接；记录了一次在两台机器上的运行。该协议使用未经身份验证的纯 TCP（第 6.7 节）。NUMA 绑定是[手动测试的]。

4.11 后端

表 3 列出了每个目标上运行的内容。有两点需要注意。首先，CPU 后端不使用进程内 JIT。编译器将 `mlir-translate`、`opt`、`llc` 和 `clang` 作为子进程运行，然后运行程序。

其次，编译器不会为 Apple 神经引擎生成代码。运行时有四个预构建的 CoreML 模型，并根据区域的三种模式选择其中一个。神经引擎上的其他所有区域都在 CPU 上运行。编译器标记为 4×4 矩阵乘积的区域会使用 CoreML 在 CPU 上规划的模型。大小为 512 和 1024 的方形 16 位乘积会使用 CoreML 在神经引擎上规划的模型。

有一种路径仅根据缓冲区的形状选择模型。它会导致所有匹配到的内核产生错误结果，仿射内核也不例外，而且还会将结果写入输入中（#3793）。

声明的机器所对应的设备不一定实际存在。在没有图形处理器的计算机上，运行时会运行每个内核的 CPU 副本，因此结果相同。使用 CUDA 构建时，会打印提示，说明未找到设备。有些 CPU 副本是会终止程序的桩函数，例如线程组的 mlir! 块副本。编译器可以在笔记本电脑上为声明的 H100 生成 PTX，但主机对象始终会采用编译机器的三元组。

5 可维护性

当一个人无需先通读整个程序，就能修改其中的一部分时，这个程序便具备可维护性。工具还必须指出某项更改破坏了什么。本节描述 Vx 中服务于这一

目标的各个部分：局部推理、模块与接口、诊断信息、确定性以及工具的规则。本节最后介绍项目用于确保自身论断成立的实践。

5.1 局部推理

P2 的规则也有助于读者理解程序。一次调用只会指向一个函数，因为没有重载。值只有在程序写明类型转换时才会改变类型，因为不存在隐式转换。函数签名给出了每个参数和返回结果的类型，而函数契约则说明了它的前置要求和确保的结果。

内存空间之间的传输在源代码中表现为一次调用，而不是运行时执行的复制。编译时块要么被折叠，要么报错。因此，读者无需判断某个块是在编译器中运行还是在运行时运行。

Vx 并非在所有方面都能实现这一目标。返回来源摘要来自被调用函数的函数体（第 3.5 节），因此函数体的更改可能改变哪些调用方能够通过编译。在签名中添加借用子句即可固定契约。标志`-print-borrow-clauses` 会显示编译器能够证明的子句，因此库作者可以将它们明确写出。

5.2 模块、接口和版本

文件就是一个模块，而导入会引入另一个模块所声明的内容。编译器首先在标准库中查找导入项，然后在它正在编译的文件所在目录中查找。它从不查找当前目录，因此构建结果不会取决于程序在哪个位置运行。–package 名称=目录名称一个库目录。使用 -j 时，模块会分波次加载，编译器会并行解析每一波。

可以使用 vxc –emit-interface lib.vx -o lib.vxlib 将库编译为接口文件。该文件保存冻结的签名、内存布局和实现表。它还保存非泛型函数的扁平化 HIR 函数体，并省略泛型函数体。随后，程序无需源代码即可连接该接口。格式读取器会检查每次读取的边界，并在失败时安全终止。

vx-api diff OLD NEW 会比较两个接口，并根据差异所需的最小版本变更，以语义化版本的术语打印结果。移除一个函数和新增一种参数类型都需要主版本变更。一个现在从另一个参数借用的 Result，以及一个变为不安全代码的函数，也需要主版本变更。新增函数或结构体需要次版本变更。对于结构体和枚举，新增字段或新增变体同样需要主版本变更。原因在于客户端可能会构造该值或对其执行 match。

十七项测试检验了这个工具。在下面的示例中，一个小型库的版本 2 更改了 scale 的参数类型，并新增了一个名为 origin 的函数：

```
$ vx-api diffv1/shapes.vxlib v2/shapes.vxlib 重大变化 函数 `scale`已更改，从 `fn(&Point, i64) ->Point`变为 `fn(&Point, i32) ->Point`次要变化 函数 `origin`已添加 未比较：特征、带数据的枚举、常量和泛型参数界限 版本升级：重大
```

该工具会说明它未比较的内容，让读者了解其结论的局限性。

状态： 模块、导入项和接口 [已测试]；vx-api [已测试]。该工具指出了四个盲点：特征、带数据的枚举、常量和泛型界限。它还有其他未说明的盲点。它不比较契约或可见性，也会跳过接口格式所跳过的签名。因此，它报告的变化可能过少。

目前的可见性非常精简：pub fn 会从目标文件中导出一个符号，而每个顶层项对所有导入方都可见。一个包含四个层级（私有、软件包、工作区、公开）并为每个层级规定版本规则的设计正在 [设计], ，另有一篇配套论文对此进行了阐述。包管理器尚不存在（#1275）。

表4. vxc 按组划分的诊断代码，摘自生成的诊断索引。

组	范围	代码	示例
警告	W1001–W1038	29	W1022：传输到值已经所在的空间中
解析器	E1001–E1013	13	已定义，尚未发出（#3227）
名称	E2001–E2007	7	E2001：未定义变量
类型	E3001–E3066	65	E3033：无法折叠的编译时块
所有权与借用	E4001–E4017	17	E4004：第二次独占借用
安全性	E5001–E5003	3	E5002：不安全代码之外的原始指针访问
放置与机器	E6001–E6042	42	E6003：设备无法寻址的值
张量与数学	E7001–E7004	4	E7001：@ 两侧的形状不一致
契约	E8001–E8016	16	E8001：求解器无法证明的后置条件

5.3 诊断信息

编译器定义了 196 个诊断代码：167 个错误和 29 个警告。代码的第一位数字表示引发该代码的阶段（表4）。

每条消息都会说明位置、未通过的事实以及通常相应的修复方案，正如上面的 E6003 和 E4002 所示。项目网站为每个代码提供一个页面；如果测试套件中有触发该代码的程序，页面中还会包含该程序。对于拼写错误的名称，系统会根据它与作用域中的名称之间的编辑距离给出建议。标志 `-diagnostics-json` 会将相同的事实写入记录，供工具使用（第 4.4 节）。标志 `-audit` 会列出受信事实（第6.6 节）。

状态： [已测试]。196 个代码中有 22 个从未发出过，而且有些错误没有代码（#3227）。JSON 记录没有注明写入该记录的编译器版本。

5.4 确定性与可复现性

在 `-j` 路径下，编译器的输出不取决于线程数。全局标识符来自模块路径和名称，因此不取决于工作顺序（第 4.9节）。

持续集成要求在 1 个和 8 个线程下，以及采用顺序调度时，生成相同的 MLIR 字节。它还要求在 1 个和 8 个线程下生成相同的内部流。有一项测试是在哈希映射的顺序泄漏到输出之后添加的。另一项测试检查输出是否受当前目录影响。持续集成中的 grep 会阻止编译器源代码中出现锁和全局原子变量。vxc –version 会标明发布构建的提交。

状态： [已测试], , 但仍有一个缺口。在 2026年11月22日之前，两个文件可以定义同名宏，而且每次运行时选用的宏都可能不同。现在，一个文件只能看到自己的宏以及其导入项中的宏（#3768）。编译器会读取命令行中未列出的输入，例如环境变量（#2250）。grep 采用词法方式，因此会漏掉某些惰性初始化的形式。接口文件取决于其源的路径，因为全局标识符是对模块路径进行哈希得到的。没有测试检查工具链的两次构建是否具有相同的字节。

5.5 工具

vx-format 根据词元格式化文件，并保持无法进行词法分析的文件不变。配置文件可以设置缩进，而 –check 会打印差异。语言服务器 vx-analyzer 会在文件发生变化时发布诊断信息。它还会响应悬停、转到定义和格式化请求。另有 Visual Studio Code 扩展和 Emacs 模式。驱动程序 vx-opt 会运行 vx MLIR 方言的各遍，就像 mlir-opt 会运行上游方言的各遍一样。

状态： vx-format 和 vx-opt [已测试]；vx-analyzer [部分支持]。语言服务器声明支持查找引用，但不会响应该请求（#2509）。其诊断流程会运行解析器、名称解析和类型检查，但不会执行借用、放置或容量检查。没有测试向它发送请求（#2510）。持续集成中没有针对编辑器扩展的测试。

5.6 项目如何保持其声明可验证

只有自身文档始终属实，编程语言才可维护。Vx 采用五种做法来确保这一点。

可能失败的测试。 测试套件包含 2,530 个 Vx 程序。约 1,300 个必须能够编译，其中约 820 个还会在持续集成中运行。约 890 个必须因给定的诊断信息而失败，其余则用于测试警告、模块和优化。在这 2,530 个程序中，有 2,095 个使用 FileCheck 检查输出。此外，还有约 1,400 个 Rust 测试函数。

项目规则要求每个新测试都由人工故意破坏一次，以确认它确实能够失败。测试注释是一项声明，规则要求将其作为声明加以验证。

程序的第二个版本。 第 2 节（P5）列出了这些版本：内核的 CPU 副本、编译时 块的运行时孪生体，以及一个差分模糊测试器。该模糊测试器有六个生成器。它通过两个代码生成器和一个等效的 Rust 程序分别运行每个生成的程序，统计堆分配，并对结果不一致的程序进行缩减。它找出了六个问题中的错误，持续集成会以每个生成器 100 个种子的配置运行其中五个生成器。

持续集成中的形式化模型。 一个工作流构建 60 个 Lean 文件：共 18,446 行、777 条定理，且没有`sorry`，也没有 公理。它会在 herd7 中运行 42 个指令序列测试，并检查 Lean 与 herd7 在不含分支的 36 个测试上是否一致。它还会对三个运行时 TLA+ 模型运行 TLC 模型检查器。另一个工作流会对编译器为四个小型程序生成的 LLVM IR 运行 GenMC。第 6.2 节说明这些模型涵盖的内容。

经过检查的文档。 一个脚本会解析文档和网站中的每个代码块，并运行标记为可运行的代码块。本书直接收录了测试套件中的 49 个程序，因此这些程序不会过时。另一个脚本会生成诊断代码索引，持续集成会检查该索引是否为最新版本。

面向人类和 *AI* 代理的规则。 Vx 的许多部分都是借助 AI 编码代理编写的：3,273 次提交中有 855 次（26%）附有 AI 模型的共同作者署名。项目对代理和人的规则写在同一个文件中。每次提交都由人类完成。

这些规则要求使用受控文体的简明英语，并要求提供一个在修复方案实施前会失败的测试。它们制定了基准测试政策：使用绝对数字、选择最佳基线、至少运行三次，并将失败结果与成功结果并列呈现。每项特征还必须更新语言语义和教程。本书对 Vx 所保证内容的各项陈述，都会说明这项承诺止于何处。

实践不足之处： 持续集成没有图形处理器，也没有 macOS 测试作业。因此，GPU 内核和 Apple 后端只能手动运行。用于比较两个代码生成器在整套测试中的表现的作业每周运行两次。该作业始于 2026年11月10日，而截至 11月23日的四次运行均失败。五个解析器模糊测试无法失败，因为它们丢弃了被测代码的结果（#3639）。

规则规定，分支保护会阻止向主分支推送。但代码仓库并未强制执行这一规则，11月有四次提交直接推送到了主分支。此外，README 中有几项说法本报告不会重复，因为代码目前尚不支持这些说法。例如，前端是“无锁”的，接缝契约经过“形式化验证”，以及差分测试套件会将每项检查与 CUDA 配对。这样的配对共有六组，而其中的 CUDA 部分需要手动运行。

6 安全

本报告以狭义方式使用“安全”一词。如果某些类别的缺陷在安全代码中不可能发生，那么编程语言就有助于提升安全性。仍可能出现这些缺陷的代码必须明确标记，且规模应尽可能小。Vx 致力于消除四类问题：内存错误、数据竞争、未定义的算术行为，以及对设备无法寻址的内存进行读取。它还限制程序各部分能够执行的操作。在安全代码中，File::open 和标准库的套接字调用都需要相应权限。本节将介绍每项检查以及这一权限限制。随后将介绍可信计算基、工具链的安全性，以及 719726333 处的差距。

6.1 安全代码中的内存安全

第 3.5 节中的检查赋予安全代码这些属性，但第 6.8节中的情况除外：

- 值移动后不得再使用（E4001），且不得通过共享引用进行写入（E4006）；
- 同一位置不得同时存在独占引用和其他引用（E4002– E4004）；

- 不会出现生命周期长于其所指对象的引用：在函数内部通过块区域进行检查，在函数调用之间则通过返回来源摘要和借用子句进行检查（E4005、E4016）；
- 不会出现未初始化值：每个 let 都必须有一个值，而 ::uninit() 需要使用不安全代码；
- 不会发生重复释放：编译器会插入每个 Drop，程序不能将 drop 作为方法调用（E4012）。

原始指针、外部调用和 假设 都需要 不安全代码。第 3.9 节的指针子句允许安全代码调用接受原始指针的函数，前提是调用方能够证明这些子句成立。该项目将 Rust 借用检查器的 78 份缺陷报告移植到 Vx 中。他们发现了 14 个缺陷，并且项目修复了所有缺陷（#1501）。

边界检查。 常量索引或边界为常量且会越出张量的循环，会在编译时产生错误（E8003）。`Vec::get` 始终会进行检查。除非程序使用 `-fbounds-checks` 构建，否则运行时不会检查其他张量索引；该选项仅涵盖 CPU 代码。这是当今内存安全方面最大的缺口（第 6.8 节）。

6.2 并发安全

本书用一句话阐明这一承诺：能够通过编译的程序，在不安全代码之外不会发生数据竞争。数据竞争指的是，在同一时间对同一个位置进行两次访问，其中至少一次是写入，且并非两次访问都是原子操作。编译器通过第4.6 节所述的每任务借用检查，以及通过 `Send` 和 `Sync`，来强制落实这一承诺。表5 列出了项目如何检查这一承诺，以及由此推导出的其他承诺。《结构化任务》论文详述了相关内容 [Kumar 2026g]。

这些证明针对的是模型，本书对此也有说明。Lean 检查器并非 vxc 中的检查器。它使用足迹，也就是任务读取和写入的 place，而 vxc 处理的是整个变量。因此，vxc 会拒绝某些模型接受的程序，而且没有测试会对这两个检查器进行比较。只有 GenMC 会检查编译器生成的代码。GPU 测试用例在 H100 上运行时，没有打印出弱结果；但对照测试也没有打印出任何结果，因此这些运行只能提供较弱的证据。

运行时竞争检测器弥补了不安全代码中的这一缺口。标志 –sanitize=thread 会连接 ThreadSanitizer，而工作流每隔一天在每个启动任务的程序上运行它。

6.3 算术与契约

Vx 中不存在未定义整数算术，除非移位量大于或等于位宽。第 3.1 节给出了相关规则：对 `+`、`-` 和 `*`采用回绕；除以零以及 `usize` 溢出则停止。由于契约检查器编码了相同的回绕行为，程序无法通过溢出证明错误的后置条件，借助一个

表 5：Vx 对并发的承诺，以及如何检查每项承诺。“模型”指证明所针对的小型语言或运行时，而非编译器。

承诺	检查
不安全代码之外不存在数据竞争。	针对包含任务、引用、原始指针的模型语言的 Lean 证明，通道和互斥锁：模型的检查器所接受的每个程序都不存在数据竞争。
一个不存在数据竞争且仅使用顺序一致原子操作的程序，其运行方式就好像其任务轮流执行一样（数据竞争自由⇒顺序一致性）。	适用于任意规模执行的 Lean 证明；herd7 对每个测试用例规则；使用 RC 在 memalloy 中手动运行的比较11 [Lahav 等人 2017]，最多六个事件；对于 AArch64，最多五个。
锁、recv 和 Arc 的最后一次 Drop 看到它们必须看到的写入。	每一对都有 Lean 证明；GenMC [Kokologiannakis 和 Vafeiadis 2021] 在四个程序编译后的 LLVM IR 上，于每次执行中 RC11 所允许的。
就绪的任务不会永远等待；select 不会使通道饥饿。	关于调度器队列和 select 的 Lean 证明。
当每个任务都在等待，什么都没有可以唤醒一个，运行时会报告一个死锁。	证明死锁始终是死锁的 Lean 证明；对运行时等待与唤醒的 TLA+ 模型进行 TLC 运行时的等待与唤醒。
在图形处理器上，原子操作的含义相同。	作用域的 herd7 模型；Vx 中的测试程序在图形处理器上手动运行。

溢出（列表 8）。一个发生溢出的编译时块不会产生任何值，并会被拒绝，因此溢出无法折叠成错误的常量。

6.4 放置安全性

E6003 会在安全代码和不安全代码中，拒绝设备读取设备无法寻址的空间。当前有两个未解决问题指出了此项检查的缺漏（#1809、#3638）。未声明连接关系的空间之间无法进行 Copy（E6002）。指向一个空间的指针不能充当指向另一个空间的指针（E6029）。容量检查会拒绝声明的机器无法容纳的放置（E6009、E6010、E6027）。这些问题如今会在运行时表现为非法地址或内存不足故障。

一个小型差分测试套件将六个 Vx 程序与在 A100 上运行的 CUDA 程序配对。在大多数配对中，CUDA 会在运行时失败，而 Vx 会在编译时拒绝该程序。CUDA 部分只由人工运行过一次。这些检查的可靠性取决于机器文件，因此 P4 要求为每张图提供源文件。

6.5 权限

内存安全可以防止因写入错误位置而导致的错误，但无法阻止程序执行其获准执行的操作。一个真实案例说明了这一点：一款面向 Qwen 模型的 Vx 应用，其会话模式会从请求中获取文件路径并打开该文件。因此，请求可以写入进程有权写入的每个文件（qwen-vx #105）。第 6.1–6.4 节中的检查无法发现这个问题，因为其中的每次内存访问都是正确的。该应用使用 C 函数 `fopen` 打开文件，而它属于可信代码。因此，现有权限也无法阻止这个问题。单文件夹权限正是为此类情况设计的（#3750）。

第 3.10节的能力限制了哪些代码可以打开文件或套接字。除以下限制外，保证如下：没有不安全代码、没有外部函数、没有外部实现且没有未经验证的`mlir!` 块的代码无法获得权限。这类代码只有持有相应权限时，才能调用`File::open` 或打开套接字。因此，没有权限的请求解析器不能调用`File::open` 或打开套接字。权限是一个普通参数，因此函数签名会将函数获得的每项权限都显示为参数（P2）。闭包或结构体的字段也可以携带权限。

这项保证在 719726333 有七项限制：

• 它仅在可信代码之外成立，而且只有在安全代码具备内存安全时才成立。 不安全代码、C 函数或图形处理器运行时中的内存漏洞，都可能造成进程能够做到的任何事情。安全代码中的漏洞若可达表 7 的一个缺口，也可能造成同样后果。
• 任何文件都可以将 C 函数声明为安全函数，并且无需不安全代码块即可调用它。一个探针以这种方式声明了 C 函数`system`，并运行了 Shell 命令。`-audit` 会列出每个此类声明，而`-no-trusted-code` 会在软件包中拒绝此类声明（第6.6节）。
• 权限范围很广。Files 会打开进程能够打开的每个文件，而 Network 会打开每个连接。

• 进程、内存映射和本机代码目前没有权限。代码通过 C 函数访问它们，而 C 函数属于可信代码。
• 标准库的某些部分仍然在无权限的情况下运行。std::llama 中的三个加载器会读取它们从路径中获取的文件。在 CUDA 构建中，std::collective::join_job 会加载由环境变量指定的软件库，并写入和读取由另一个变量指定的文件。问题 #3750 跟踪了这些部分。

• 有三个漏洞会让安全代码获得无人授予的权限或句柄。当某个字段的类型是参数的类型时，泛型结构体可以实现 Copy，因此权限会被复制（#3861）。安全代码可以从原始指针构建 File 或套接字，并使用进程持有的每个描述符（#3848）。一个探针就曾以这种方式写入标准输出。此外，main 文件中的函数可以调用 main，从而创建一个新的 Root（#3865）。

• 程序无法收回权限。只要所有者持有权限，函数就可以一直保留对该权限的借用引用。所有者通常是 main，因此借用的权限可以在整个运行期间持续使用。

6.6 受信基

每个 Vx 程序都信任编译器、其运行时和工具。表 6 列出了它们的规模。程序还拥有自己的可信代码，`-audit` 会列出这些代码。每个`unsafe` 块、`unsafe fn`、`unsafe impl`、`extern impl`、外部函数和未经验证的`mlir!` 块各占一行。编译器无需证明即可信任的每个事实也各占一行。这类事实可以是`assume`，也可以是外部函数的`ensures`。它还可以是`pub fn`的指针子句，或调用在`unsafe` 代码中所信任的子句。对于一个包含一个外部函数、一个带有`unsafe` 读取的循环以及两条`assume` 语句的小型程序，它会打印：

```
audit.vx:4:3:可信代码：`extern` 函数 `c_buffer`，它运行 C 代码 audit.vx:5:11：`extern`函数 `c_buffer`
ensures `valid(return, n)`，编译器信任它 audit.vx:13:21：可信代码：一个 `unsafe`块 audit.vx:19:3：可信代
码：一个 `unsafe`块 audit.vx:19:12：一个 `assume`信任 `valid(p, 4)`audit.vx:20:3：可信代码：一个 `
unsafe`块 audit.vx:20:12：一个 `assume`信任 `initialized(p, 4)`
```

最后一行统计了程序所使用模块中 core 和 std 的可信代码：对于这个程序，共有 10 个不安全代码块、4 个不安全函数和 8 个 mlir!代码块。–audit-library 也会列出这些内容。该列表不会显示编译器为接收 Root 的 main 创建的不安全代码块。

标志 –no-trusted-code NAME 会将软件包 NAME 中每次使用可信代码的行为视为错误（E5003），正如 Rust 中的 #![forbid(unsafe_code)] 一样。标志 –package NAME=DIR 指定软件包的文件夹。使用该软件包的程序负责提供这些标志。因此，软件包的新版本无法在不产生错误的情况下添加可信代码。Vx 目前还没有清单，所以这是编译器的标志，而不是软件包的设置。

该列表涵盖程序及其库的代码，但不涵盖编译器、运行时或机器文件，而程序也信任这些内容。可信代码的契约设计仍未定案（#1902）。

表 6. 719726333 版本的可信计算基。行数统计的是受跟踪文件；“不安全代码”统计的是源代码出现次数。

Part	规模与备注
编译器（Rust）	149 个文件，共 152,249 行；包含 18 个不安全代码块和一个不安全的 extern 代码块，主要用于 MLIR 和 LLVM C API
vx MLIR 方言（C++）	18,869 行；仅降级遍就有 14,941 行
运行时（C、C++、Objective-C++）	13,174 行：CUDA、Metal、CoreML、英特尔、远程工作进程
任务运行时（Rust）	8,575 行；有 308 个不安全代码块，没有任何一个附有安全性注释
标准库（Vx）	34,437 行代码，分布在 63 个文件中；434 个不安全代码块、53 个不安全函数、33 个不安全实现、8 个外部实现、约 180 个 mlir! 块、15 个外部块
外部工具	LLVM 和 MLIR 22.1.8（通过哈希锁定版本）、Enzyme（锁定版本）、Z3（PATH 中的任意版本）、CUDA、Metal、CoreML、204 个 Rust crate

6.7 工具链和部署

安装程序会检查每个版本的 SHA-256 sum；如果缺少 sum，则会拒绝继续。sum 来自同一版本，因此可以检测下载文件是否损坏，但无法检测版本是否被替换。这些版本没有签名，也没有软件物料清单、安全策略文件，且未对 Rust 依赖项进行审计。构建会将 LLVM 和 Enzyme 固定到确切的提交。调试构建会在 AddressSanitizer 和 UndefinedBehaviorSanitizer 下运行运行时的测试。

任务运行时支持 AArch64（GCS）和 x86-64（CET）的影子栈。当操作系统启用它们时，每个任务栈都会获得一个属于自己的影子栈。持续集成在 QEMU 下测试 AArch64 路径。编译器不会标记其输出以请求影子栈，因此这是兼容性支持，而不是强化保护。编译器不添加其他强化措施，例如栈保护器。

远程工作进程在所有接口上通过明文 TCP 进行监听，且不进行身份验证。部署时必须在可信网络中运行它，或通过隧道运行；文档目前尚未说明这一点。

6.8 Gapsat 719726333

该调查在发布构建上运行了小型探测程序，首次针对提交 `a06269cad`（11月20日），随后再次针对 main 提交`719726333`。在这两次提交之间，三个缺口的探测停止了。一个从地址加载数据的 `mlir!`块会触发 E8016 (#3790)。一个移动捕获值的闭包只能运行一次（E4001、#3751）。并且只有类型所属的文件才能使其成为`Copy`（E3066、#3774），因此 #3743 的探测不再能够编译。其中两个缺口以另一种形式依然存在。在同一行包含两个操作的 `mlir!` 块会规避 E8016 (#3864)，而 #3861 会复制一个`Vec`，所以 #3743 仍未解决。表7 列出了安全代码仍然不安全的位置。

同一区域还有两个较小的缺陷处于开放状态。类型可以同时是 `Drop` 和 `Copy`，这样一来它的析构就永远不会运行（#3742）。此外，结构体不变量是一条隐私规则，而不是证明。因此，类型所在文件中的代码可以构造出破坏该不变量的值。第 10 节列出了其他区域中的缺口。

7 实现

编译器 `vxc` 使用 Rust 编写。它读取一个程序，并在提供机器文件时读取该文件。它输出 MLIR [Lattner 等人 2021], LLVM IR [Lattner 和 Adve 2004],目标文件、PTX、Metal 源代码或 SPIR-V 映像。本节将沿着程序的处理流程对其进行介绍（图 1）。

前端。 词法分析器和解析器（11,595 行）为每个模块构建语法树。宏在名称解析之前的独立阶段中展开。

解析器会为每个条目分配一个 256 位的 GID（第 4.9 节）。这四个 64 位字分别存放模块、符号、泛型或区域部分，以及标志。前两个字是以下内容的 FNV-1a 哈希：

Tab表 7. 719726333 处安全代码尚不安全的位置，来自对发布构建运行的探针 .

Gap	探针显示的结果	问题
带有 Drop 的结构体持有一个引用，该引用指向先被释放的值。	它读取已释放的内存，并打印出错误的值对于 77 和 7。	#3784
对一个没有 _ arm 结束一个函数。	该函数执行到末尾后退出，并返回 0 , or a 导致崩溃的空引用。	#3785
运行时边界检查默认关闭。	写入张量末尾之后的位置会成功，然后程序因总线错误信号而停止。	#3791
@ 位于两个 [上? , ?]内部大小不同的张量。	它会越过张量末尾读取，即使使用了 -fbounds-checks。	#3792
一个泛型结构体，其字段的类型是一个参数，例如 Twice<T>，可以是 Copy。	安全代码会复制 Files 权限。复制的 Vec 读取已释放的内存：7 的 6952。	#3861
安全代码从原始指针构建一个文件。	它在没有权限的情况下写入标准输出，然后程序停止。	#3848
一行中包含两个操作的 mlir! 块，第二个是加载操作。	检查只读取第一个操作。一个安全函数从地址 16 加载并停止，段错误。	#3864
文件中的一个函数调用了 main。	每次调用都会创建一个新的 Root，并拥有其全部权限。	#3865

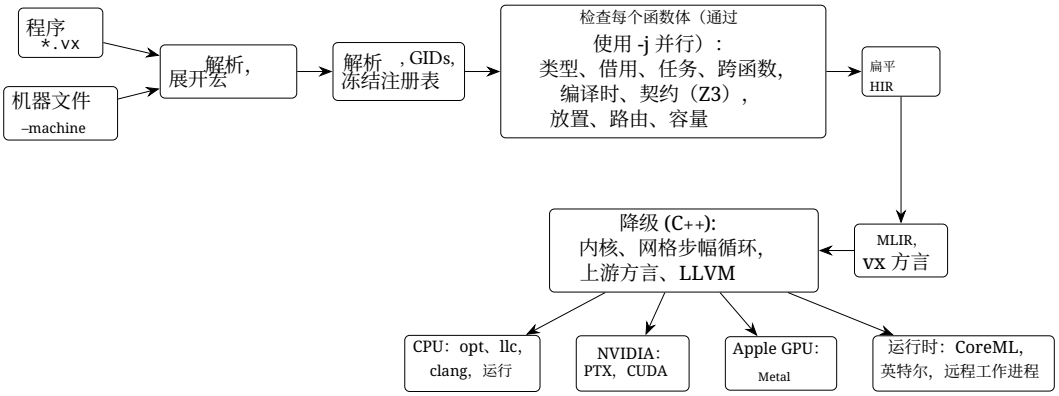

图 1. 程序经由 vxc 的路径。注册表会在检查任何函数体之前冻结（P2），因此不同函数体的检查互不依赖。

模块路径和名称，因此 GID 会在每次运行中标识同一个条目。它们不是密码学哈希；如果两个类型布局具有相同的 GID 但名称不同，注册表就会拒绝它们。随后，签名、内存布局和实现的注册表便会冻结。

检查器。 检查器（src/hir，78,301 行）会依据冻结的注册表检查每个函数体。它会推导表达式的类型，检查移动、借用和析构，检查任务与跨循环行为，求值编译时块，并依据机器检查放置、路由和容量。其证明器会编写 SMT-LIB 查询，并以固定的资源限制将 Z3 作为子进程运行。随后，它会将每个函数体降级为扁平的指令流，即扁平 HIR。

代码生成。 两个代码生成器生成 MLIR。默认的生成器从扁平 HIR 开始。较旧的生成器从语法树开始。默认生成器会拒绝尚未支持的构造，

并将模块交给较旧的生成器。门禁会在每个拉取请求上比较它们的输出，并每周两次运行完整测试套件（第5.6节）。

两者都会生成 vx 方言的操作，这是一组用于张量、传输、设备区域和通道操作的 MLIR 操作。其降级遍使用 C 编写（该方言有 18,869 行代码）。它会将每个区域上的 spawn 提取为内核，并将跨循环转换为 GPU 方言中的网格步幅循环。其余部分则降级为上游方言和 LLVM。

后端和运行时。对于 CPU，vxc 会运行 mlir-translate、opt、llc 和 clang，随后连接并运行程序。当程序使用 grad 时，Enzyme 会在 opt 内部运行。对于 NVIDIA GPU，MLIR GPU 流水线会生成 PTX，运行时再通过 CUDA 驱动程序加载它。对于 Apple GPU，打印器会将 GPU 方言转换为 Metal 着色语言。CUDA、Metal、CoreML、Intel GPU 和远程工作进程的运行时共有 13,174 行 C、C 和 Objective-C 代码。以 Rust 编写的任务运行时（8,575 行）负责管理调度器、等待、任务栈和死锁监视器。

标准库。 该库包含 63 个文件，共 34,437 行 Vx 代码。core 模块包含选项、Result、迭代器、比较、格式和原子操作。它还包含一个 12,428 行的数学库，该项目将其从 LLVM 的 C 库移植而来。std 模块包含输入和输出、文件、网络功能、时间、向量、字符串、哈希映射、张量、同步、集合通信和 std::nn。约 180 个 mlir!代码块和 15 个外部块将其连接到 MLIR 操作和 C。

规模和历史记录。 该代码仓库从5月14日到2026年11月23日共有3,273次提交，涉及18个作者姓名；其中一位作者完成了97%的提交。它有3,569个受跟踪文件：其中2,651个是测试，175个是 Markdown 文档，共33,731行。0.0.1至0.0.5版本于9月10日至11月19日期间发布。许可证为 Apache 2.0，附带 LLVM 例外条款。

8 状态和证据

表 8 汇总了本报告中每项特征的状态。状态术语沿用第 1 节的定义。最后一列列出了主要证据：测试目录、手动运行、形式化模型或配套研究。

配套研究。 五项研究在真实硬件上或以 scale 进行测试，以验证本报告的主张，并且每项研究都说明了自身的限制。

- 真实 *GPU* 上的容量。 编译器在任何租用之前，就预测出 12 种大语言模型服务配置中哪些能够在 H100 和 A100 上运行。全部 24 次启动结果都与预测一致。引擎所需容量比预测值多 1.1 至 1.9 吉比字节，而固定预留 7% 覆盖了每次准入的启动 [Kumar2026b]。
- 机器文件与硬件对比。 一项研究冻结了机器文件的预测，然后租用硬件并进行测量。研究报告指出了声明错误的地方，例如缺少小型 Copy 的启动延迟 [Kumar, 2026e]。
- 并行前端。前端在每种线程数下都给出了相同的输出。八月构建在 48 个核心上实现了 5.9 的加速比，输出为 MLIR 文本；九月的一次运行则达到了 3.9 至 4。5 [Kumar 2026a]。
- 编译时求值。一个生成器生成了 57,950 个程序，其中包含编译时块，没有一个被折叠成错误的值 [Kumar 2026f]。
- *GPU* 循环嵌套。这项研究将来自三个基准测试套件的 38 个内核改写为跨循环。检查器接受了 27 个，拒绝了 11 个；被拒绝的内核中有三个存在真实的数据竞争 [Kumar2026d]。

表8。719726333 中每个特征的状态及其主要证据。

特征	状态	证据和限制
核心语言		
类型，数字，结构体，枚举，张量	[已测试]	前端和后端测试；128 位打印失败
模式匹配	[部分完成]	没有守卫、或模式或嵌套模式；#3785
泛型、特征	[已测试]	类型的文件中的实现（E3066）；Copy 缺口（#3861、#3742）；实现顺序（#3787）
闭包	[已测试]	移动捕获变量的闭包只能运行一次（E4001）
移动、借用、析构	[已测试]	130 个失败的借用测试；#1501；#3784
借用子句	[已测试]	E4016 测试；–print-borrow-clauses
线性值	[设计]	代码是仿射的
编译时、宏	[经过测试]	持续集成中的 13 个运行时孪生项；Must-Fold 生成器 [Kumar2026f]
契约、循环不变式	[已测试]	137 个程序使用它们；感知回绕的编码
结构体不变量，已验证	[部分]	隐私规则；未证明
不安全代码、指针子句、外部函数接口	[已测试]	安全外部函数在调用时不需要不安全代码
mlir!	[已测试]	在不安全代码之外，只有已验证代码块（E8016）；两项操作在一行中绕过了检查（#3864）
能力：文件、网络	[已测试]	Forge 测试；Twice<T> 复制一个（#3861）；#3848；std::llama
单文件夹权限	[已设计]	#3750；进程、内存映射或本机代码尚无设计
可扩展		
机器文件	[已测试]	16 个文件；257 个测试文件使用 –machine；来源信息未经检查
放置、可见性、路由	[已测试]	E6003 出现在 21 项测试中；与 Bellman–Ford 算法进行路径对比
容量、峰值、跨调用	[已测试]	两块图形处理器上 24/24 次启动 [Kumar 2026b]；#3632
接缝检查	[已测试]，偏差为 de-故障	三项测试；只判断“是否为宽松模式”（#662）
用户传输主体	[部分]	仅指向多处理器内存的边
spawn 于	[已测试]	每个后端；仅设备 0（E6042）
任务、通道、互斥锁、Arc，原子操作	[已测试]	x86-64 上的持续集成；在三种运行时下运行
select	[已测试]	不同类型的源：通道、套接字、信号量，区间，截止时间
死锁监视器	[部分]	仅在每个线程都等待时报告；标出一个 select
跨循环、竞态和屏障规则	[已测试]	静态检查；38 个内核语料库 [Kumar 2026d]
NVIDIA GPU 上的内核	[手动测试]	A100、A6000、H100、H200 运行；持续集成中的 PTX 文本
Apple GPU 上的内核	[手动测试]	仅限 macOS
Apple 神经引擎	[部分]	四个预构建模型；其他区域使用 CPU；#3793
Intel GPU	[部分]	跨循环、通道例程和 SPIR-V 中的原子操作；单卡运行；无持续集成
AMD GPU、Cortex-M7	[设计]	仅机器文件
张量、@	[已测试]	[的内维度？，?] 未经检查 (#3792)
自动微分	[已测试]	仅限标量
std::nn	[手动测试]	Apple GPU；CPU 在持续集成中进行测试
集合通信	[部分]	一个秩
并行前端，确定性	[已测试]	字节完全一致的输出门槛；3.9–5。9× 在 48 个核心上 [Kumar 2026a]；选择加入
远程工作进程	[已测试]，环回	一次双机运行；无身份验证
NUMA 绑定	[手动测试]	一次 c5.metal 运行
可维护性		
模块、接口	[已测试]	接口取决于源路径
vx-api	[已测试]	17 项测试；未考虑契约和可见性
可见性级别	[设计]	仅限 pub fn 导出功能
包管理器	[设计]	#1275
诊断信息、JSON、错误页面	[已测试]	196 个代码，其中 22 个从未发出
vx-format、vx-opt	[已测试]	
vx-analyzer	[部分完成]	没有借用或放置诊断；#2509、#2510
安全		
无数据竞争，数据竞争自由—顺序一致性	已针对一个模型证明	Lean（总计 777 个定理）、herd7、memalloy；未与 vxc 连接

9 相关工作

Vx 融合了四个研究方向的思想。前两个方向分别是带所有权的系统编程语言，以及面向并行和异构机器的编程语言；另外两个方向分别是语言内部的验证，以及使语言的接口和声明保持可检查性的工具。本节将 Vx 分别置于这四个方向中进行讨论。

系统编程语言。 Rust [Matsakis 和 Klock 2014] 将所有权和借用引入了系统编程语言，而 RustBelt 则为其类型系统提供了形式化基础 [Jung 等人。 2018]Vx 的借用规则源自 Rust，但在三个方面有所不同。首先，它没有生命周期语法，而是从被调用函数处读取返回的引用可能指向的位置（第 3.5 节）。其次，它没有 `dyn`，没有 `impl Trait` 结果，也没有特化，因此签名就是完整的契约。第三，其整数算术在每次构建中都有定义，但移位宽度为类型宽度或更大时除外。

Zig 的 编译时 [Zig 软件基金会 2026]和 C++ `constexpr` [Dos Reis 和 Stroustrup2010] 都在编译器中运行代码。与 Zig 类似，Vx 会拒绝无法折叠的块。Vx 还允许块命名一个运行时值，前提是结果不依赖于该值。最接近的先前工作是混合部分求值 [Shali 和 Cook 2011] 以及分阶段计算 [Taha 和 Sheard 1997]。

Mojo [Modular Inc. 2023] 旨在为新型硬件编写系统代码，而 Carbon [Carbon 语言项目 2026] 则致力于打造 C 的继任者++。Mojo 在指针类型中指定地址空间，但这两种语言都不会在程序中指定容量或机器声明。Miri 等工具 [Jung 等人 2026] 会在运行时查找 不安全代码 Rust 中的未定义行为。Vx 目前还没有这样的解释器，而它的 `-audit`标志则采用静态列表。

并行和异构编程语言。CUDA [英伟达公司 2026], OpenCL [Stone 等人2010] 和 SYCL [Khronos SYCL 工作组 [未注明日期]] 都在指针上指定地址空间。但指向设备内存的指针和指向主机内存的指针在运行时仍可能混用，而且容量并非程序的一部分。Chapel 的 locale [Chamberlain 等人，2007 年；Chapel 项目 2021] 和 X10 的 place [Charles 等人 2005] 描述了数据在集群中的位置。Legion 和 Regent [Bauer 等人，2012 年；Slaughter 等人 2015] 通过映射器在运行时将逻辑区域映射到内存，而 Sequoia [Fatahalian 等人 2006] 则将存储层次结构纳入程序。Kokkos [Edwards 等人 2014] 在 C++ 中提供可移植的并行模式。Vx 与这些系统目标相同，但它会在编译时根据源文件所声明的机器检查放置、可见性和容量。

在内核编程语言中，Halide [Ragan-Kelley 等人 2013]将算法与其调度分离，Triton [Tillet 等人 2019] 对块进行编程，而 TVM [Chen 等人 2018]编译张量程序。Exo [Ikarashi 等人 2022] 和 Exo-GPU [Akeley 等人 2026] 允许程序员在检查机制下重写调度。Exo-GPU 通过在程序员选择的问题规模下解释调度来检查调度的等价性，而同步则由程序员编写。Descend [Köpcke 等人 2024] 是最接近的内核编程语言。其类型跟踪内存空间和线程层级的层级结构，其借用检查器排除了内核中的数据竞争。相反，Vx 的跨越循环保留了循环嵌套的顺序语义，而编译器会插入屏障。

Futhark [Henriksen 等人 2017] 将带有嵌套并行性的函数式编程语言编译到图形处理器上。GPUVerify [Betts 等人 2012] 和 Faial [Cogumbreiro 等人 2021] 会在程序员编写内核后检查其中的数据竞争。Vx 将同一检查中可判定的部分构建到编程语言中，并拒绝其余部分。

确定性并行 Java [Bocchino 等人 2009] 利用效应来证明并行任务不会相互干扰。Habanero-Java [Cavé 等人 2011] 和 Cilk [Frigo 等人 1998] 提供的任务不会超出其作用域，Vx 任务 块也不会。ML5 [VII 等人 2008] 以及 Murphy 等人的模态 λ 演算 [VII 等人 2004]为分布式程序中运行的代码赋予类型。

编程语言中的验证。Dafny [Leino 2010], F* [Swamy 等人 2016] 和 Liquid Haskell [Vazou 等人 2014] 使用 SMT 求解器检查契约或精化类型。对于 Rust，Prusti [Astrauskas 等人 2019], Creusot [Denis 等人 2022], Verus [Lattuada 等人 2023] 和 Kani [VanHattum 等人2022] 添加规格说明或有界模型检测。Vx 的契约弱于这些系统：它们涵盖整数事实和指针子句，而非一般的功能正确性。不过，这些契约是每个程序都会使用的编译器的一部分，其编码所采用的回绕模型与生成代码相同。CompCert [Leroy 2009]证明编译器正确；Vx 则证明模型的性质（第 6.2 节），只有 GenMC 会检查编译器输出是否符合其中一个模型。这一主张要小得多。Vx 所依据的内存模型研究包括 RC11 [Lahav 等人 2017], herd7 [Alglave 等人2014], memalloy [Wickerson 等人 2017] 以及 GenMC [Kokologiannakis 和 Vafeiadis； 2021], 其证明则以 Lean 4 [de Moura 和 Ullrich 2021]编写。

能力。Dennis and Van Horn 将能力命名为 [Dennis and Horn 1966]。Miller 的对象能力模型要求，权限只能通过程序持有的引用传递 [Miller2006]。Joe-E [Mettler et al. 2010] 和 Wyvern [Melicher et al. 2017] 限制编程语言，使模块只能从使用它的代码获得权限。Pony 为 main 提供环境权限值，程序由此获得使用文件的权限 [Pony 开发者 2026]。Austral 为启动程序的函数提供根能力 [Borretti 2026]。Capsicum [Watson et al. 2010],、WASI [WebAssembly 社区组 2026] 以及 Rust crate cap-std [Bytecode Alliance 2026]会将路径替换为进程已经持有的文件夹句柄。Vx 采用了 Pony 和 Austral 的根能力。与 Austral 一样，它将每项权限设为只有一个模块能够构建的类型。Austral 的能力是线性的，因此程序必须使用每项能力。Vx 的权限是仿射的，因此程序可以无错误地丢弃某项权限。Joe-E 和 Wyvern 移除了所有不经过能力便通向操作系统的路径。Vx 目前尚未做到这一点。可信代码仍可调用操作系统，而 –audit 会列出这些代码（第 6.5 节）。

接口、版本与测试。 Elm 的软件包工具根据其类型计算版本发布的版本变更 [Elm 2026], ，而 `cargo-semver-checks` 则为 Rust 完成同样的工作 [Gruevski 等人 2026]。工具`vx-api` 根据冻结接口完成同样的工作。两个代码生成器的差分测试遵循 McKeeman [McKeeman 1998]和 Csmith [Yang 等人 2011] 的做法。自动微分使用 Enzyme [Moses 和 Churavy 2020], ，而契约使用 Z3 [de Moura 和 Bjørner 2008]。

10 限制与未来工作

本报告描述了一种年轻语言的一次提交，其中大部分限制已经在各项特征旁边说明。本节汇总其中最为重要的限制。

安全性。 表 7 列出了 safecode 目前尚不安全的位置。仍待确定的最重要设计决策，是运行时边界检查的默认设置（#3791）。该领域的其他待处理事项包括 `Copy` 缺口（#3861、#3742）、结构体不变量的证明，以及针对可信代码的契约（#1902）。

权限。文件和套接字需要权限，但权限范围很广，而且程序无法收回权限。进程、内存映射和本机代码尚无权限，标准库中的一些加载器会在没有权限的情况下读取文件。声明为安全的外部函数可以在调用时无需权限，也无需使用不安全代码，便可访问操作系统。

证明与编译器。 Lean 证明、herd7 模型和 TLA+ 模型针对的是模型，而不是 vxc。编译器的任务检查器保存完整变量，而模型使用足迹，且没有测试对二者进行比较。目前没有关于放置、传输、容量或接缝的形式化模型（#1209），也没有针对跨循环竞争规则的机械化证明。接缝检查只判断一次跳转是否宽松（#662）。

硬件覆盖范围。 持续集成在 x86-64 Linux CPU 上运行。GPU 后端、Apple 后端和 NUMA 绑定仅通过手动方式运行。有一项测试会在 pocl（面向 CPU 的 OpenCL）上运行 Intel GPU 测试的 SPIR-V 映像，但持续集成目前尚未使用 pocl（#3731）。大多数手动运行结果只记录在提交消息和问题中，没有文件保存这些测试结果。神经引擎运行四个预构建模型。AMD 和微控制器目标有机器文件，但没有代码生成器。成本模型没有每次传输的延迟数据，也没有争用数据。

编程语言与工具。 模式匹配缺少守卫、或模式和嵌套模式。可见性只有一个级别，也没有包管理器，vx-api 也不会比较契约。语言服务器不会运行借用或放置检查。并行前端需选择加入，且有些标志无法与之配合使用。机器文件无法导入另一个机器文件。

后续工作。 接下来的步骤由这些限制得出：

- 默认启用安全的边界检查；
- 通过差分测试将任务检查器连接到其 Lean 模型，然后将其迁移到足迹；
- 为放置和容量建立形式化模型；
- 在真实硬件上通过持续集成运行图形处理器和 macOS 套件；
- 添加可见性级别和包管理器；
- 添加单文件夹权限，以及进程和本机代码的权限；
- 让代码审查中能够轻松找到每次对安全外部函数的调用。

11 结论

Vx 将三类事实纳入程序文本。第一类是数据存放的位置，以及运行代码的机器。第二类是函数向其调用方承诺的内容。第三类是编译器在没有证明的情况下所信任的内容。

这些事实都写在文本中，因此编译器可以在程序运行前检查它们。它还可以并行检查函数体，并准确指出某项更改破坏了什么。这正是标题中三种特质的共同根源。这种编程语言具有可扩展性，因为同一份文本可以在不同机器、核心和图形处理器线程上运行。它具有可维护性，因为签名就是函数的契约，只有一种明确说明的例外，而且诊断信息会指出出了什么问题。它具有安全性，因为安全代码具备所有权、竞态和放置检查。此外，在安全代码中，File::open 和标准库的套接字调用都需要权限标志。标志会列出程序中的可信代码。

报告还指出了各项特质尚未实现之处。我们认为，描述一种编程语言时应说明它的不足之处，以便读者检验这些主张是否符合代码。

引用

David Zhao Akeley、Yuka Ikarashi 和 Jonathan Ragan-Kelley。2026。Exo-GPU：面向张量核心的安全、命令式、可由用户调度的编程。arXiv：2609.16389 [cs.PL] https://arxiv.org/abs/2609.16389 发布于 2026-09-14；arXiv 评论：“未在会议上发表”。Jade Alglave、Luc Maranget 和 Michael Tautschnig。2014。Herding Cats：弱内存的建模、模拟、测试与数据挖掘。《*ACM* 编程语言与系统汇刊》36，2（2014），1–74。doi:10.1145/2627752 Vytautas Astrauskas、Peter Müller、Federico Poli 和 Alexander J. Summers。2019。利用 Rust 类型进行模块化规范与验证。《*ACM* 编程语言汇刊》3，OOPSLA（2019），1–30。doi:10.1145/3360573Michael Bauer、Sean Treichler、Elliott Slaughter 和 Alex Aiken。2012。Legion：利用逻辑区域表达局部性与独立性。载于*SC*。doi:10.1109/SC.2012.71 Adam Betts、Nathan Chong、Alastair Donaldson、Shaz Qadeer 和 Paul Thomson。2012。GPUVerify：GPU 内核验证器。载于*ACM* 面向对象编程系统、语言与应用国际会议论文集（*OOPSLA '12*）。ACM，113–132。doi:10.1145/2384616.2384625 Robert L. Bocchino、Vikram S. Adve、Danny Dig、Sarita V. Adve、Stephen Heumann、Rakesh Komuravelli、Jeffrey Overbey、Patrick Simmons、Hyojin Sung 和 Mohsen Vakilian。2009。用于确定性并行 Java 的类型与效应系统。载于

第 24 届 ACM SIGPLAN 面向对象编程系统、语言与应用会议（OOPSLA）论文集。ACM，97–116。 doi:10.1145/1640089.1640097 Fernando Borretti。2026。Austral 语言规范。 https://austral-lang.org/spec/spec.html。访问日期：2026-11-24。Bytecode Alliance。2026。cap-std：Rust 标准库的基于能力的版本。https://github.com/ bytecodealliance/cap-std。访问日期：2026-11-24。Vincent Cavé、Jisheng Zhao、Jun Shirako 和 Vivek Sarkar。2011。Habanero-Java：老 X10 的新冒险。载于第 9 届 Java 编程原理与实践国际会议论文集（PPPJ）。ACM，51–61。doi:10.1145/2093157.2093165 Bradford L. Chamberlain、David Callahan 和 Hans P. Zima。2007。并行可编程性与 Chapel 语言。《高性能计算应用国际期刊》21，3（2007）。 doi:10.1177/1094342007078442 Chapel 项目。2021。Chapel 1.25 技术说明：区域模型。 https://chapel-lang.org/docs/1.25/technotes/localeModels. html Philippe Charles、Christian Grothoff、Vijay Saraswat、Christopher Donawa、Allan Kielstra、Kemal Ebcioğlu、Christoph von Praun 和 Vivek Sarkar。 2005。X10：一种面向对象的非一致集群计算方法。载于 OOPSLA。doi:10.1145/1094811.1094852 Tianqi Chen、Thierry Moreau、Ziheng Jiang、Lianmin Zheng、Eddie Yan、Haichen Shen、Meghan Cowan、Leyuan Wang、Yuwei Hu、Luis Ceze、Carlos Guestrin 和 Arvind Krishnamurthy。2018。TVM：面向深度学习的端到端自动优化编译器。载于第 13 届 USENIX 操作系统设计与实现研讨会（OSDI 18）。USENIX 协会，578–594。 https://www.usenix.org/conference/osdi18/presentation/chen Tiago Cogumbreiro、Julien Lange、Dennis Liew Zhen Rong 和 Hannah Zicarelli。2021。组合式检查 GPU 内核的数据竞争自由。载于计算机辅助验证（CAV 2021），Alexandra Silva 和 K. Rustan M. Leino（编）。施普林格，403–426。doi:10.1007/978-3-030-81685-8_19 Leonardo de Moura 和 Nikolaj Bjørner。2008。Z3：高效的 SMT 求解器。载于系统构造与分析工具和算法（TACAS）（《计算机科学讲义》，第 4963 卷）。施普林格，柏林、海德堡，337–340。 doi:10.1007/978-3-540-78800-3_24 Leonardo de Moura 和 Sebastian Ullrich。2021。Lean 4 定理证明器与编程语言。载于自动演绎——CADE 28（《计算机科学讲义》，第 12699 卷）。施普林格，625–635。 doi:10.1007/978-3-030-79876-5_37 Xavier Denis、Jacques-Henri Jourdan 和 Claude Marché。2022。Creusot：用于 Rust 程序演绎验证的工具平台。载于形式化方法与软件工程（ICFEM）。施普林格，90–105。 doi:10.1007/978-3-031-17244-1_6 Jack B. Dennis 和 Earl C. Van Horn。1966。多程序计算的编程语义。《ACM 通讯》9，3（1966），143–155。doi:10.1145/365230.365252 Gabriel Dos Reis 和 Bjarne Stroustrup。2010。系统编程语言的通用常量表达式。载于 2010 年 ACM 应用计算专题讨论会论文集（SAC）。ACM，纽约，美国，2131–2136。 doi:10.1145/ 1774088.1774537 H. Carter Edwards、Christian R. Trott 和 Daniel Sunderland。2014。Kokkos：通过多态内存访问模式实现众核性能可移植性。《并行与分布式计算杂志》74，12（2014）。 doi:10.1016/j.jpdc.2014.07.003 Elm。2026。elm-package：用于分享 Elm 库的命令行工具。 https://github.com/elm-lang/elm-package。README；已由 Elm 0.19 的 elm bump/elm diff 命令取代。访问日期：2026-10-16。Kayvon Fatahalian、Daniel Reiter Horn、Timothy J. Knight、Larkhoon Leem、Mike Houston、Ji Young Park、Mattan Erez、Manman Ren、Alex Aiken、William J. Dally 和 Pat Hanrahan。2006。Sequoia：存储层次结构编程。载于 2006 年 ACM/IEEE 超级计算会议论文集（SC）。doi:10.1145/1188455.1188543 Matteo Frigo、Charles E. Leiserson 和 Keith H. Randall。1998。Cilk-5 多线程语言的实现。载于 ACM SIGPLAN 1998 年编程语言设计与实现会议论文集（PLDI）。ACM，212–223。doi:10.1145/277650.277725 Predrag Gruevski 等。 2026。cargo-semver-checks：扫描 Rust crate 中的 Semver 违规。https://github.com/ obi1kenobi/cargo-semver-checks。访问日期：2026-10-16。Troels Henriksen、Niels G. W. Serup、Martin Elsman、Fritz Henglein 和 Cosmin E. Oancea。2017。Futhark：采用嵌套并行与原地数组更新的纯函数式 GPU 编程。载于第 38 届 ACM SIGPLAN 编程语言设计与实现会议论文集（PLDI '17）。ACM，556–571。 doi:10.1145/3062341.3062354 Yuka Ikarashi、Gilbert Louis Bernstein、Alex Reinking、Hasan Genc 和 Jonathan Ragan-Kelley。2022。通过外部编译实现高效的硬件加速器编程。载于第 43 届 ACM SIGPLAN 编程语言设计与实现国际会议论文集（PLDI）。doi:10.1145/3519939.3523446 Ralf Jung、Jacques-Henri Jourdan、Robbert Krebbers 和 Derek Dreyer。2018。RustBelt：巩固 Rust 编程语言的基础。《ACM 编程语言汇刊》2，POPL（2018），66:1–66:34。doi:10.1145/3158154 Ralf Jung、Benjamin Kimock、Christian Poveda、Eduardo Sánchez Muñoz、Oli Scherer 和 Qian Wang。2026。Miri：实用的 Rust 未定义行为检测。《ACM 编程语言汇刊》10，POPL（2026），1383–1411。doi:10.1145/3776690

Khronos SYCL 工作组。 [n. d.]。SYCL 2020 规范（第 12 次修订）。
https://registry.khronos.org/SYCL/specs/sycl- 2020/html/sycl-2020.html。访问于 2026 年 10 月 28 日。
Michalis Kokologiannakis 和 Viktor Vafeiadis。2021。GenMC：弱内存模型的模型检查器。载于《计算机辅助验证》（CAV 2021）（《计算机科学讲义》，第 12759 卷）。施普林格，427–440。doi:10.1007/978-3-030-81685-8 20_Bastian Köpcke、Sergei Gorlatch 和 Michel Steuwer。2024。Descend：一种安全的 GPU 系统编程语言。《ACM 编程语言汇刊》8，PLDI（2024）。doi:10.1145/3656411 Aditya Kumar。2026a。Agree Without Locking：一种确定性并行编译器前端。预印本。Aditya Kumar。2026b。面向异构机器上大语言模型服务的编译时 GPU 内存检查。预印本。Aditya Kumar。2026c。Vx 中的编译时：设计、可靠性与测试。技术报告。Aditya Kumar。2026d。Erasable Parallelism：保留顺序语义的 GPU 循环嵌套。预印本。Aditya Kumar。2026e。一种在异构系统编程语言中携带来源信息的机器模型。预印本。Aditya Kumar。2026f。Must-Fold：能够访问运行时变量的编译时块。预印本。Aditya Kumar。2026g。Structured Tasks and Footprints：一种系统编程语言中的数据竞争自由性，及其实现与机械化验证。预印本。Aditya Kumar。2026h。Vx：一种面向舰队级编程的编程语言。技术报告。Aditya Kumar。2026i。Vx：一种面向异构计算的系统编程语言。https://vxlang.org。版本0.0.5。源代码位于 https://github.com/vx-lang/Vx。Ori Lahav、Viktor Vafeiadis、Jeehoon Kang、Chung-Kil Hur 和 Derek Dreyer。2017。在 C/C++11 中修复顺序一致性。载于第 38 届 ACM SIGPLAN 编程语言设计与实现会议论文集（PLDI）。618–632。
doi:10.1145/3062341.3062352 Chris Lattner 和 Vikram Adve。2004。LLVM：用于程序分析与转换的编译框架。载于国际代码生成与优化研讨会（CGO）。75–86。doi:10.1109/CGO.2004.1281665 Chris Lattner、Mehdi Amini、Uday Bondhugula、Albert Cohen、Andy Davis、Jacques Pienaar、River Riddle、Tatiana Shpeisman、Nicolas Vasilache 和 Oleksandr Zinenko。2021。MLIR：扩展面向领域专用计算的编译器基础设施。载于 2021 年 IEEE/ACM 国际代码生成与优化研讨会论文集（CGO）。IEEE，2–14。
doi:10.1109/CGO51591.2021.9370308 Andrea Lattuada、Travis Hance、Chanhee Cho、Matthias Brun、Isitha Subasinghe、Yi Zhou、Jon Howell、Bryan Parno 和 Chris Hawblitzel。2023。Verus：使用线性幽灵类型验证 Rust 程序。《ACM 编程语言汇刊》7，OOPSLA1（2023），286–315。doi:10.1145/3586037 K. Rustan M. Leino。2010。Dafny：一种用于功能正确性验证的自动程序验证器。载于《编程、人工智能与推理逻辑》（LPAR）。施普林格，348–370。doi:10.1007/978-3-642-17511-4_20 Xavier Leroy。2009。对实用编译器进行形式化验证。《ACM 通讯》52，7（2009），107–115。doi:10.1145/1538788. 1538814 Nicholas D. Matsakis 和 Felix S. Klock。2014。《Rust 语言》。《ACM SIGAda《Ada Letters》》34，3（2014），103–104。doi:10.1145/2692956.2663188 William M. McKeeman。1998。软件差分测试。《数字技术期刊》10，1（1998），100–107。Darya Melicher、Yangqingwei Shi、Alex Potanin 和 Jonathan Aldrich。2017。一种用于权限控制的基于能力的模块系统。载于第 31 届欧洲面向对象编程大会（ECOOP）（LIPIcs，第 74 卷）。20:1–20:27。doi:10.4230/LIPIcs.ECOOP.2017.20 Adrian Mettler、David Wagner 和 Tyler Close。2010。Joe-E：一种面向安全的 Java 子集。载于网络与分布式系统安全研讨会（NDSS）。
https://www.ndss-symposium.org/ndss2010/joe-e-security-oriented-subset-java/ Mark Samuel Miller。2006。鲁棒组合：迈向访问控制与并发控制的统一方法。博士论文。约翰斯·霍普金斯大学。Modular Inc. 2023。Mojo 编程语言。https://www.modular.com/mojo。William S. Moses 和 Valentin Churavy。2020。无需为机器学习重写外部代码，自动合成快速梯度。载于《神经信息处理系统进展》33（NeurIPS）。arXiv:2010.01709。英伟达公司。2026。CUDA C++ 编程指南。https://docs.nvidia.com/cuda/cuda-c-programming-guide/。访问于 2026 年 10 月 23 日。Jonathan Ragan-Kelley、Connelly Barnes、Andrew Adams、Sylvain Paris、Frédo Durand 和 Saman Amarasinghe。2013。Halide：一种用于优化图像处理流水线中的并行性、局部性和重复计算的编程语言与编译器。载于 PLDI。Amin Shali 和 William R. Cook。2011。混合部分求值。载于 2011 年 ACM 面向对象编程系统、语言与应用国际会议论文集（OOPSLA）。ACM，纽约，美国，375–390。
doi:10.1145/2048066.2048098

Elliott Slaughter。计算机逻辑区域的高效编程语言。载于《国际高性能计算、网络、存储与分析会议论文集》（SC）。doi:10.1145/2807591.2807629John E. Stone、David Gohara 和 Guochun Shi。2010。OpenCL：面向异构计算系统的并行编程标准。《计算机科学与工程》12，3（2010）。doi:10.1109/MCSE.2010.69Nikhil Swamy、Cătălin Hriţcu、Chantal Keller、Aseem Rastogi、Antoine Delignat-Lavaud、Simon Forest、Karthikeyan Bharga- van、Cédric Fournet、Pierre-Yves Strub、Markulf Kohlweiss、Jean-Karim Zinzindohoue 和 Santiago Zanella-Béguelin。2016。F* 中的依赖类型与多单子效应。载于第 43 届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集（POPL）。ACM，256–270。doi:10.1145/2837614.2837655Walid Taha 和 Tim Sheard。1997。使用显式注释的多阶段编程。载于 1997 年 ACM SIGPLAN 部分求值与基于语义的程序操作研讨会论文集（PEPM）。ACM，美国纽约州纽约市，203–217。doi:10.1145/258993.259019Carbon 语言项目。2026。Carbon 语言：C++ 的实验性后继者。GitHub 代码仓库。https: //github.com/carbon-language/carbon-lang。访问于 2026 年 8 月。Pony 开发者。2026。对象能力。Pony 教程，https://tutorial.ponylang.io/object-capabilities/object- capabilities.html。访问于 2026-11-24。Philippe Tillet、H. T. Kung 和 David Cox。2019。Triton：用于分块神经网络计算的中间语言与编译器。载于 MAPL。Alexa VanHattum、Daniel Schwartz-Narbonne、Nathan Chong 和 Adrian Sampson。2022。在 Rust 中验证动态特征对象。载于第 44 届国际软件工程会议：软件工程实践论文集（ICSE-SEIP）。ACM，321–330。doi:10.1145/3510457.3513031Niki Vazou、Eric L. Seidel、Ranjit Jhala、Dimitrios Vytiniotis 和 Simon Peyton Jones。2014。Haskell 的精化类型。载于第 19 届 ACM SIGPLAN 函数式编程国际会议论文集（ICFP）。ACM，269–282。doi:10.1145/2628136.2628161Tom Murphy VII、Karl Crary 和 Robert Harper。2008。使用 ML5 进行类型安全的分布式编程。载于可信全球计算（TGC）。doi:10.1007/978-3-540-78663-4_9 Tom Murphy VII、Karl Crary、Robert Harper 和 Frank Pfenning。2004。用于分布式计算的对称模态 λ 演算。载于第 19 届 IEEE 计算机科学逻辑研讨会论文集（LICS）。Robert N. M. Watson、Jonathan Anderson、Ben Laurie 和 Kris Kennaway。2010。Capsicum：面向 UNIX 的实用能力。载于第 19 届 USENIX 安全研讨会。https://www.usenix.org/conference/usenixsecurity10/capsicum-practical-capabilities-unixWebAssembly 社区组。2026。WASI：WebAssembly 系统接口。https://wasi.dev/。访问于 2026-11-24。John Wickerson、Mark Batty、Tyler Sorensen 和 George A. Constantinides。2017。自动比较内存一致性模型。载于第 44 届 ACM SIGPLAN 编程语言原理研讨会论文集（POPL）。190–204。doi:10.1145/3009837.3009838Xuejun Yang、Yang Chen、Eric Eide 和 John Regehr。2011。发现并理解 C 编译器中的错误。载于第 32 届 ACM SIGPLAN 编程语言设计与实现会议论文集（PLDI）。ACM，美国纽约州纽约市，283–294。doi:10.1145/1993498.1993532Zig 软件基金会。2026。Zig 语言参考：编译时表达式。https://ziglang.org/documentation/ master/。行为检查使用 Zig 0.17.0。

一页了解 Vx

本附录总结了 Vx 在 719726333 处的语法，位于 Vx 书籍的快速参考之后。每一行都是编译器接受的一种形式。

B 本报告中的阶乘是如何得出的

三位读者审阅了`a06269cad`所对应的代码仓库。这是 2026年11月20日的一次提交。第二轮审查则逐项对照 `719726333`处的 main 核实每项事实。一位读者审阅核心语言，一位审阅异构和并行特性，另一位审阅工具和项目。他们逐一统计受跟踪文件，并阅读每项文档声明背后的代码和测试。他们还分别列出了文档声明超出代码实际支持范围的地方。随后，我们从 `719726333`的克隆构建了编译器：在一台 Apple M 系列计算机上，使用 Rust 1.98 和 LLVM 22 进行发布构建。我们在该计算机上运行了表 7 中的每个代码清单、每条引用的消息和每个探针。表 2 中的准入判定来自同一构建。程序与本报告的

Part	形式
程序	一个文件就是一个模块；导入 core::option；会引入模块所声明的内容；fn main() -> i32 启动程序，其 Result 就是退出代码。
声明	常量 N : i64 = 4; 函数 add(a : i64, b : i64) -> i64 { .. } 结构体 Point { x : i64, y : i64 } 枚举 Shape { Square(i64), Rect(i64, i64) } 特征 Describe { fn size(self : &Self) -> i64; } 实现 Describe 用于 Shape { .. }，位于 Shape 的文件中；外部实现在另一个文件中
泛型	结构体 Pair<T> fn largest<T : 描述>(a : &T) -> i64 fn dot<const N : i32>(a : 张量<f32, [N]>) -> f32；泛型类型始终写入其参数：Vec<i64>::new()、Option<i64>::Some(3)。
数字	i4–i64、u4–u64、usize、f16、bf16、f32、f64、布尔类型、八位和四位浮点数；没有隐式转换 (x as i64)； + - *包装；/ 除以零和 usize溢出会停止程序 .
Data	元组 (i64, f32) 以及 let (a, b) = t;; 引用 &T、&mut T；原始指针 *const T、*mut T 位于 Memory::X 中；张量Tensor<f32, [2, 3]>，张量<f32, [?, ?]>，张量<f32, [4], 内存::高带宽内存>；Option、Result、Vec、Box。
语句	let x = 1; let mut y : i64 = 2; 如果 c { .. } 否则 { .. }（也是一个表达式） 遍历 i 从 0 到 n 当 loop match v { Shape::Square(s) => { .. }, _ => { .. } } let v = 解析(s)?;
闭包	|x : i64| x * 2; 参数类型已写明；捕获模式遵循函数体。
机器	内存 高带宽内存 { 容量：80 吉比字节， 带宽：3.35 太字节/秒，... }
拓扑	设备 { 内存： ...， 可见： [...]， 层级： [...]， 传输 A -> B：63 GB/s }
放置	传输(x, 内存::高带宽内存) 传输_异步(x, M).wait() spawn(设备拓扑) { .. }
图形处理器循环	for y in 0..64 across 线程块 { for x in 0..300 across 线程束(256) { .. } }; 线程束求和(&part, n) 在线程束循环之间。
任务	let r =任务 { let h = spawn { .. }; h.join() }; 令 (tx, rx) = 通道<i64>(n); select2(&jobs, &quiet) 会给出第一个就绪项的索引源 互斥锁、信号量、Arc、原子操作。
编译时	编译时 { 阶乘(10) } if comptime ... 宏_规则 name { ($x : 表达式) => { .. }; }，以 name!(..) 的形式调用。
契约	要求 x < 100 确保子句 return > x 循环不变式 区间 { lo : i64, hi : i64, 不变式 lo <= hi, } 结构体 assert(c) 借用子句 确保子句 !可达(return, *k) 指针子句 要求有效(p, n).
不安全代码和外部函数接口	unsafe { .. } 不安全函数 不安全实现 Send for T（在另一个文件中，不安全外部实现） 假设 有效(p, n); extern "C" { fn abs(x : i32) -> i32; } 外部代码块中的安全函数 用于导出的 pub fn mlir!(...).
输出	打印(x) 打印并换行!("{} and {}", a, b) 打印错误信息并换行! 恐慌("..") abort()。
编译器	vxc file.vx（编译并运行） –machine F –host 默认 –操作 生成 MLIR | 生成 LLVM |生成目标文件 -j N –diagnostics-json –audit –verify-seams –print-borrow-clauses -fbounds-checks –sanitize=线程。

源文件一同保存。一项结果见第 4 节，来自较早的 0.0.4 构建版本，文中对此有所说明：监视器未报告的一次死锁。我们未能复现的硬件结果，例如图形处理器运行结果，来自配套论文或代码仓库中的记录。文中均对每项结果作了说明。