用于 Go 死锁检测的协程类型 Flow 扩展

Qiqi Jason Gu Lixue Liu Wei Ke澳门理工大学中
国澳门特别行政区，中国 qiqi.gu@cvut.cz,
lixue.liu@mpu.edu.mo,wke@mpu.edu.mo

arXiv:2602.19686v1 [cs.PL] 23 Feb 2026

摘要—**协程作为一种抽象的编程构造，是函数的一种推广，能够暂停执行，并在之后恢复。协程类型是一种行为类型，用于对协程之间的交互进行建模，其交互由单个接收操作和随后单个让出操作组成。协程类型已应用于模型驱动工程、智能合约和测试用例生成。我们为协程类型提出一种 Flow 扩展，使其能够对包含多个接收操作和让出操作的协程进行建模。相应地，我们修订了协程类型的归约规则。为展示 Flow 扩展的实用性，我们提出一种类型系统，将 Go 编程语言中的表达式映射为协程类型。如果归约结果为 0，则两个通道操作配对正确，且程序不存在死锁。**

我们选择 Go，是因为它是分布式系统中一种流行的编程语言，但 Go 中常见的一类错误是因错误使用并发特性而导致的死锁。我们重点关注 Go 中最常用的语义：无缓冲通道，以及关键字`go` **和** `defer`**。我们的 Flow 扩展和类型系统能够识别通道与 goroutine 交互中的 17 种模式，包括接收方和发送方不匹配、嵌套 goroutine 等。我们还集成了 Z3 SMT 求解器，以考虑条件执行和类型继承。其他静态或动态死锁检测器在某些模式下会崩溃或给出错误预测。因此，我们的基于类型的死锁分析器不仅填补了基于值的检测领域中的空白，也对现有检测器起到了补充作用。**

索引词—**Golang，并发，协程，进程演算，死锁，静态分析**

I. 引言

协程作为一种抽象的编程构造，是函数的一种泛化形式，可以暂停执行，并在之后恢复。为了研究进程演算领域中一组协程的集体行为，Gu 和 Ke 提出了协程类型 [1]，它们是用于协程的行为类型表达式。这些协程类型已应用于模型驱动工程中的 [2], 智能合约以及测试用例生成 [3]。

协程类型要求协程在让出任何内容之前，必须先完整接收数据。然而，将协程类型应用于许多支持并发的现实世界编程语言时，这会成为一种限制。因此，本文提出了一个针对协程类型的 Flow 扩展，以克服这一限制。我们的 Flow 扩展允许让出和接收操作以任意顺序进行，且次数不限。该扩展还包括约束类型 τ/ρ，类似于通信顺序进程中的布尔条件 如果 ρ 那么 τ [4]。我们的约束类型增强了协程类型的表达能力，使我们能够考虑通道数据中的类型继承，以及协程的其他条件激活方式。归约规则也针对新的语义进行了调整。

为了检验其适用性并解决现实问题，我们创建了一种 Flow 类型系统，将 Go 语言（简称 Golang ）中的表达式映射到带 Flow 的协程类型。这样，我们就能够从行为类型的角度检查 Go 程序是否无死锁。我们选择 Go，是因为它广泛用于构建可扩展的 Web 应用、区块链以及其他分布式系统。在 Go 中，协程称为 goroutine。

Golang 提供了丰富的并发特性 [5], ，但我们的类型系统只能为其中一部分赋予类型。列表 1 展示了可映射到带 Flow 的协程类型的 Golang 特性，包括关键字 `go` 和 `defer`，以及无缓冲通道。关键字`go` 接收函数调用或匿名子程序调用，并启动一个 goroutine。以关键字 `defer`开头的语句会将函数的执行推迟到外围函数返回时。

Go 中的通道是静态类型的管道，这意味着管道发送和接收的数据都属于同一类型。同时，一个 goroutine 可以访问多个通道，以便输入和输出。通道通过语法`make(chan` t, n) 创建，其中 t 是类型， n 指定通道的缓冲容量。如果 n 为 0 或省略，通道就会成为无缓冲通道，这意味着发送操作会一直等待，直到在该通道上执行接收操作，反之亦然。通道上的发送操作和接收操作使用同一个运算符 `<-`。一方面，当它像`x <- y`这样用作二元运算符时，`x` 是通道，`y` 是要发送的值。另一方面，当它像 `<- y`这样用作一元运算符时，`y` 就成为通道，而整个表达式的求值结果是接收到的值。在列表 1 中，第 4 行创建了一个整数通道；第 6 行启动一个延迟执行的 goroutine 来读取该通道；第 9 行向通道发送一项数据。

带 Flow 的协程类型旨在分析 Golang 中通道与 goroutine 交互导致的 17 种死锁情况。我们使用算法分析了 4 个真实世界程序文件，并准确识别出死锁情况。此外，我们还运行了两种最先进的静态死锁分析器和一种动态分析器，但它们无法正确

列表 1：包含通道创建、发送、接收以及关键字的 Go 代码片段go 和 defer

```
1   导入 "fmt"
2
3   函数main() {
4     通道: = 创建(整数通道)
5
6     defer 函数() {fmt.Print(<-通道)}()
7     go 函数() {
8       如果为真{
9         通道 <- 20
10      }
11    }()
12  }
```

预测 21 个案例中部分案例的死锁情况。

总而言之，本文的主要贡献如下：1）我们对先前发表的协程类型提出了 Flow 扩展，以便对协程灵活的接收和让出行为进行建模。2）我们提出了一组类型规则，将 Go 语言中的关键并发特性映射到 Flow 扩展中的类型表达式。3）我们的 Flow 扩展弥补了传统死锁分析器的不足；后者无法分析 11 种死锁模式和 3 个现实案例。

本文其余部分的组织结构如下。首先，我们在第二节回顾相关工作。随后，在前期研究的基础上，第三节对协程类型提出 Flow 扩展，并修订归约规则。接下来，第四节介绍一种 Flow 类型系统，将 Golang 表达式映射到协程类型，以检测死锁。我们还在本节将我们的算法与其他工具进行了比较。第五节简要介绍带 Flow 的协程类型及其相应死锁检测算法的局限性。最后，我们在第六节作出总结。

II. 相关工作

预测死锁本质上是在解决停机问题，而停机问题已被证明是不可判定的 [6]。不过，我们完全可以对有限形式的死锁进行推理 [7]。并发错误表现为阻塞执行或非阻塞执行 [8]。阻塞是指所有进程都没有进展，这种情况称为死锁。如果只有少数进程没有进展，则称为部分锁死 [9]。

本节首先介绍抽象解释，这是一种从程序中提取某些属性并以抽象方式分析这些属性的方法。我们关注的属性可以是死锁自由、终止、神经网络是否能抵御扰动 [10]，等等。如果我们关注的是类型信息，抽象解释就会专门用于类型系统，其中的类型不一定是整数、布尔值等标准类型，也可以是行为类型或会话类型 [11]，用于对程序的抽象行为进行建模。根据编程语言是静态还是动态，类型系统会在静态或动态阶段使用。除了类型系统之外，静态分析或动态分析还有其他方法来确定程序的并发属性。

A. 抽象解释

抽象解释提供了一个通用框架，用于近似程序行为并推导出适用于自动推理的有限模型。它会减少程序 [6, p 21]的可能状态。一个简单的例子是分析整数算术的符号，判断结果为正（+）、负（–）还是未知（±）。Veileborg 等人 [12]将 Go 程序中的通道抽象为状态，并利用指向关系绘制通道使用方式的图。 [13]将 C++ 程序转换为建模语言 Promela，然后查找并发问题。

通信顺序进程（CSP） [14] 和通信系统演算（CCS） [4] 都是进程演算。CSP 支持条件，并使用感叹号（!）发送数据，使用问号（?）接收数据。给定 Golang源代码，Almeida和Antonio [15]会生成一种称为 MiGo [16]的中间表示。随后，MiGo 会被转换为 CCS 表示。他们忽略了 Go 的 `select` 语句。接下来，他们会执行死锁检测和消解算法。Coelho [17] 将`select` 语句建模为 CCS 中的外部选择，并开发了一个名为 GoDDaR 的 Go 模块。

B. 类型系统

如果从程序中抽象出类型，研究的就是类型系统。Hosoya 等人 [18] 创建了正则表达式类型，用于检查 XML 文件是否符合语法。

Typed Lua [19]旨在为 Lua 语言引入可选类型，并涵盖其协程。虽然它的类型系统支持类型收窄，但不会为协程构造整体指定具体类型。此外，在 Typed Lua 类型系统中，协程的返回类型不能依赖于输入类型。CorDuroy [20]是一种基于简单类型 λ 演算、用于协程的静态类型语言。Godel [21] 使用行为类型对 Go 程序的 SSA 形式建模，并研究其活性和通道安全性属性。评估使用了一组由 22 个程序组成的小型样本。Godel 的方法有一个局限：它不支持在 for 循环中生成新线程的程序。Bar- clay 等人 [22] 发明了一种基于协程的动态类型语言，用于代码原型开发。这种语言没有函数，只有协程，而且所有协程都是可调用对象。

C. 静态分析

静态工具在无需编译器或执行代码的情况下分析源代码。静态分析往往较为保守，以避免状态爆炸问题 [23]。

GCatch [24] 指示 Microsoft Z3 [25]，a SMT 求解器求解一个等于死锁状态的方程。如果找到解，该解就是导致死锁的条件。Staticcheck [26] 和 Vet [27] 会对 Go 源代码进行静态分析。由于检查规则数量有限，它们

在少数情况下发现并发错误。Zhang 等人发明了一款闭源工具 GoDetector [28]，它不仅能检测通道相关错误，还能检测 WaitGroup 相关错误。JaDA 是一款用于 Java 字节码 [29]的死锁分析器。它在 25 个基准测试中的 247 个 Java 文件上进行了评估。

D. 动态分析

动态方法会在程序执行期间收集信息。这种方法适用于分析对象程序的一次执行，但无法研究程序尚未表现出的行为。动态分析可最大限度地减少假阳性和假阴性，但这种方法会带来执行开销。

内存泄漏通常在运行时被检测到 [30]。Liang 等人 [31] 拦截程序发出的 API 调用，以判断该程序是否具有恶意。

Golang 提供了 `runtime/trace` [32]包。开发者在想要诊断的代码

`trace.Start(traceFile)` and `trace.Stop()`

周围插入相关代码。`traceFile` 会被写入磁盘，然后命令行

`go tool trace traceFile` [33]

工具会将该文件可视化。该工具链提供的信息不足以定位并发错误，因此 GoAT [34], 作为一个基于 GOPATH 的项目，会对工具链打补丁以记录更多事件，并设计另一个静态组件来可视化和检查跟踪输出。与 GoAT 类似，BinGo [8] 是一种结合运行时和静态分析的混合方案。它不会修改 Go 执行二进制文件，而是由其运行时组件向地址空间注入代码。遗憾的是，其源代码不可用。类似地，Concuerror [35] 将动态分析与静态分析结合起来，以查找 Erlang 中的错误。

最简单的死锁检测直接由 Go 运行时自身实现。如果所有 goroutine 都已暂停，其内置死锁检测器就会打印堆栈跟踪，然后终止程序 [36]。Uber 的 GoLeak [37]会在主函数（即根 goroutine）未能在预设时间内结束时判定发生了死锁。

III. 协程类型的 Flow 扩展

我们的行为类型语法定义见图 1。斜体名称——序列类型、积类型和具体类型——来自基础协程类型 [2], , 不属于 Flow 扩展。具体类型是没有内部结构的简单类型，例如整数或 StringBuilder。序列 Θ 是扁平且满足结合律的，也就是说 $\langle\langle\tau_1,\tau_2\rangle,\tau_3\rangle = \langle\tau_1,\langle\tau_2,\tau_3\rangle\rangle$，其中两侧都可化简为 $\langle\tau_1,\tau_2,\tau_3\rangle$。特定类型 τ 的序列写作 Θ_τ。要表示序列的长度 n ，写作 $\Theta_\tau\ \tau^n$。

Flow 扩展中的主要类型是协程定义类型 D 和协程实例类型 I。协程定义类似于函数定义。协程实例是函数的一次执行。根据参数的不同，协程实例可能表现出多种不同的输入和输出

$\omega ::=$	Flow 项 Ω
$\mid !\tau$	产出项
$\mid ?\tau$	接收项
$\tau ::=$	类型 T
$\mid \tau/\rho$	受约束类型 Ξ
$\mid (\tau \mid \tau)$	联合类型 Σ
$\mid [\![\omega;\ \omega;\ \cdots]\!]$	协程定义类型 D
$\mid\ [\omega;\ \omega;\ \cdots]$	协程中的类型 I
$\mid \langle\tau,\ \tau\rangle$	序列类型 Θ
$\mid (\tau,\ \tau)$	积类型 P
$\mid$整数 $\mid$字符串 $\mid\cdots$	具体类型 K

图 1：带 Flow 的协程类型的抽象语法

行为。Start : $D \to I$ 是一个类型级函数，它启动协程定义并将其转变为协程实例。定义和实例都由 Flow 项 Θ_Ω 构成，用于说明协程如何让出和接收数据。让出和接收操作构成一种协议，用于控制何时执行协程。允许自我取消，例如 $[!A;?A]$。协程定义中可以存在不交并类型，但协程实例中不能存在，因为在运行时我们总能知道实际选择的确切类型。

类型可以带有可选的布尔谓词 ρ。 ρ 中的指令特定于应用场景，可用于建模类型继承、泛型函数和条件执行。例如，

$$l : [\![?\langle x, \text{Book}\rangle; !\langle x, \text{Book}, \text{Reserve}\rangle]\!]/x <: \text{User}$$

其含义是：“l 是一个协程定义，它接收 x 、图书，并让出 x、图书和预订，其中 x 是 User 的子类。”谓词的种类取决于应用场景，不属于 Flow 扩展的一部分。公式（1）是受约束类型的归约规则。结构**0** 表示不执行任何行为。

$$\begin{aligned} \tau/\text{false} &\Rightarrow \mathbf{0} \\ \tau/\text{true} &\Rightarrow \tau \\ \tau/\rho_1/\rho_2 &\Rightarrow \tau/\rho_1 \wedge \rho_2 \\ \tau/\rho \wedge (x \mapsto v) &\Rightarrow \tau[x \mapsto v]/\rho \end{aligned} \tag{1}$$

如果 Flow 项中的类型是序列，则应将 Flow 方向指示符 ψ （让出 ! 或接收 ? ）分配给序列中的每一项，即 $\psi\langle\tau_1,\tau_2,\cdots\rangle \Rightarrow \langle\psi\tau_1,\psi\tau_2,\cdots\rangle$。

变量可以代替具体类型、整数或函数的参数。自由变量被视为常量。

Gu 和 Ke 的协程类型 [1] 定义了协程归约的最小元素，仅包含具体类型、序列，以及具有固定接收和产出顺序的协程。其中不区分协程定义和协程实例。后来，他们加入了积类型来控制归约优先级 [2], , 但在这里我们改为利用

Start 来激活协程，这在某种程度上也是一种优先级。我们仍将积类型保留为元组，因为它们可以对元素进行分组。

A. 归约规则

归约函数定义为$\circ : \Theta_I \rightarrow I$。$\circ$ 的参数必须是协程实例或返回实例的 Start 函数序列。返回类型是协程实例 I，我们不会再将其归约为 K。这样我们就知道数据是计算得出的，而不是来自静态字段。归约函数$\circ$ 由按需驱动策略运行的规则组成 [38]。

在$\circ$的求值过程中，空类型的 Flow 项被视为无用，应从协程中移除。ψ 表示通道操作 ! 或 ?，归约规则 CR1 列出了移除这些单位元的相关归约。$e_1 \Rightarrow e_2$ 表示一个表达式 e_1 求值或归约为另一个表达式或值 e_2。

Start(d) 接收协程定义 d，并将其转换为协程实例 i。请注意，协程定义可能包含不交并类型。在清单 2 中，如果忽略`defer`语句，我们得到定义对应的 s：$[\![!(\text{整数}/v < 10 \mid \text{布尔}/\neg(v < 10))]\!]$。对于实例化，`go s(20)` 会将 Start($s$) 转换为 [!布尔]，因为选择了分支 Bool/¬($v < 10$)；而`go s(2)` 会将 Start(s) 转换为 [!整数]。第四节将讨论从 if 语句到 (整数/$v < 10$ |布尔/¬($v < 10$)) 的映射，以及 Go 的`defer`语句的处理。

清单 2：函数 s 根据条件产生不同类型，可由 Z3 求解器求解

```
变量ch1整数通道 = 创建(整数通道)
变量ch2chan bool = make(chan bool)

函数s(v 整数) {
  defer func(){ ch2 <- true }()
  如果 v < 10{ch1 <-v
  } 否则{ ch2 <- true }
}

函数main() {
  启动 s(2)

  fmt.Println(<- ch1, <- ch2)
}
```

其他归约规则需要访问或操作辅助数据。因此，这里使用符号 $\vdash$ 。 $\vdash$ 左侧是由待处理类型 $\tau : T$ 和外部产出E：$\Theta_{K|P}$组成的上下文。待处理类型是协程产出但尚未被协程接收的类型。外部产出是一个待处理类型序列，其中的类型无法被任何协程接收。$\vdash$ 右侧是一个表达式，可以是类型，也可以是对归约函数的调用。与先前的工作相同 [2], , 我们使用通配符（·）表示规则中未被引用的项。例如（·, E）表示此处的待处理类型（可能是也可能不是 **0**）不会在规则的其他部分中使用。

协程实例和协程定义的“主体”是一个由 Flow 项组成的序列。我们使用 head 函数和基础协程类型中定义的 tail 函数 [2]来获取实例的第一个元素和其余元素，如图 2 所示。我们绝不会从定义中取出头部或尾部。

以下归约规则 (CR2) 至 (CR7) 与基础协程类型中的规则非常相似，但我们必须对它们进行修订，以识别新的 Flow 项Ω。

清单 3：Haskell 中的函数 **first** 和 **none**

```
首个 :: [一个] -> (一个 ->布尔值) -> (可能 一个, [一个], [一个])
first s p =
  案例中断p s 的
    (before, [])    -> (无, 之前, [])
    (之前, x:之后) -> (仅 x, 之前, 之后)

无 :: [a]->(a-> 布尔值)-> 布尔值
无 s p =
  情况 firstsp 属于
    (无, s, [])-> 真
    _-> 假
```

1）让出：我们使用基础协程类型中定义的函数 **first**，在列表 S 中查找第一个符合条件 p 的协程实例 i，记作 $(i, S_1, S_2) = \mathbf{first}(S, \lambda i.p(i))$。除了协程之外，**first** 还会返回 S_1 之前的所有元素和 S_2 之后的所有元素。清单 3 给出了它在 Haskell 中的正式定义。如果 **first** 找不到匹配的元素，则 $i = S_2 = \mathbf{0}$。这个特殊情况简写为**none**。

当不存在待处理类型，且函数 **first** 能够找到头部方向为让出的协程时，就会触发 (CR2)。随后，我们将让出的类型移入上下文。如果某个协程已耗尽其动作语句，**t**(s) 将为 **0**，因此会由 (CR1) 将其删除。

$$(\mathbf{0}, \cdot) \vdash \circ(S) \Rightarrow (\tau, \cdot) \vdash \circ(\langle S_1, \mathbf{t}(i), S_2 \rangle) \quad \text{provided} \quad (i, S_1, S_2) = \mathbf{first}(S, \lambda i.\, \mathbf{h}(i) = !\tau) \tag{CR2}$$

如果让出的类型是协程实例，或是对类型级函数 Start 的调用，(CR3) 就会对 Start 求值，并将让出的实例插入列表末尾。这模拟了广度优先搜索，并在协程不断启动自身时避免无限循环。

$$(\mathbf{0}, \cdot) \vdash \circ(S) \Rightarrow (\mathbf{0}, \cdot) \vdash \circ(\langle S_1, \mathbf{t}(i), S_2, [u; v] \rangle) \quad \text{provided} \quad (i, S_1, S_2) = \mathbf{first}(S, \lambda i.\, \mathbf{h}(s) = ![u; v]) \tag{CR3}$$

(CR4) 描述了一种死锁状态：没有协程的头部能够让出，且待处理类型为 **0**，这表明没有内容可接收。在这种情况下，归约会以结果 E 终止，后面跟着列表中的所有剩余协程。此规则适用分配律，并允许将协程作为一等公民置于另一个协程中。

$$(\mathbf{0}, E) \vdash \circ(S) \Rightarrow (\mathbf{0}, \mathbf{0}) \vdash [!E; !S] \quad \text{provided } \mathbf{none}(S, \lambda i.\, \mathbf{h}(i) = !\tau) \tag{CR4}$$

$$
\begin{aligned}
\circ\langle S_1, [\omega_1; \psi(\mathbf{0}); \omega_2], S_2\rangle &\Rightarrow \circ\langle S_1, [\omega_1; \omega_2], S_2\rangle \\
\circ\langle S_1, [\psi(\mathbf{0})], S_2\rangle &\Rightarrow \circ\langle S_1, S_2\rangle \\
\circ\langle S_1, \mathrm{Start}([\![\omega_1; \psi(\mathbf{0}); \omega_2]\!]), S_2\rangle &\Rightarrow \circ\langle S_1, \mathrm{Start}([\![\omega_1; \omega_2]\!]), S_2\rangle \\
\circ\langle S_1, \mathrm{Start}([\![\psi(\mathbf{0})]\!]), S_2\rangle &\Rightarrow \circ\langle S_1, S_2\rangle
\end{aligned}
\tag{CR1}
$$

$$
\begin{aligned}
\mathbf{h}([\omega_1;\ \omega_2;\ \cdots]) &= \omega_1 \\
\mathbf{t}([\omega_1;\ \omega_2;\ \cdots]) &= [\omega_2;\ \cdots]
\end{aligned}
$$

图 2：协程实例的头部 **h** 和尾部 **t** 的定义

2) 恢复： 当待处理类型 τ 不为**0**时，将触发恢复操作。(CR5)会选取第一个能够接收 τ的协程 i ，并恢复它。**match**： $T \times T \to \Theta$在基础协程类型中定义，是一个用于匹配两个类型（可能携带约束）的交换函数，并返回一组条件 B ，在这些条件下两个类型相等。我们知道，协程类型可以包含变量，因此返回的条件可能会绑定这些变量。例如，$\mathbf{match}(\mathrm{Int}^5, \mathrm{Int}^n) = \{n \mapsto 5\}$。如果两个类型无论如何都无法匹配，则返回 $\bot$ ，且 $\bot \wedge B = \bot$。我们对 **match** 函数的改进是借助 SMT 求解器 Microsoft Z3 为其提供支持。关于约束处理的细节见第三节-B。

(CR6) 规定，如果 S 中没有任何协程能够处理待处理类型 τ，那么就将 τ 附加到外部产出 E 中。此归约规则通常为 (CR4) 的终止铺平道路，因为 (CR4) 要求待处理类型为空。

协程取消并非 Golang 的关键部分，因为 goroutine 必须借助信号量或类似机制编写，才能实现提前终止。尽管如此，(CR7) 概述了协程如何接收其他协程，从而在基础协程类型中完整支持一等协程。 (CR7) 中的第 2 行查找头部 $\mathbf{h}(s)$ 是某个协程实例的接收模式的第一个协程，然后第 3 行将此模式与另一个协程 i' 匹配，并返回条件 B。最后，我们将该条件应用于 $\mathbf{t}(i)$，并使用集合差符号从结果中移除被接收的协程。总体而言，这条规则展现了一等协程支持的强大之处：协程的一个组成部分可以整体移除另一个协程。

基础协程类型 [2] 有一条规则会循环处理外部 yield。这条规则不利于基于通道的死锁检测。因此，我们的 Flow 扩展禁用了这条规则。请看清单 4，其中 `main()` 先 yield 字符串，再 yield 整数，但 `work()` 要求先接收整数，随后接收字符串。顺序不匹配会导致无限阻塞。如果我们在 [2], 分析器中启用“loopexternal”规则，它将报告终止，这显然不正确。清单 4 被列入第四节 A 小节中受支持的死锁模式。

总体而言，整个归约算法是确定性的，因为它按顺序检查规则，且不存在随机选择。

清单 4：数据集中的 `Out-of-order.go` 将遭遇死锁

```
函数work(cInt 整数通道, cStr 字符串通道) {
  fmt.Println(<-cInt)
  fmt.Println(<-cStr)
}

函数main() {
  cInt: =make(整数通道)
  cStr: =make(字符串通道)
  Go work(c整数,  cStr)

  cStr  <-"你好"
  cInt  <-1

  ...
}
```

收集协程定义这一步必然会终止，因为程序中的函数数量有限。然而，如果 Go 函数递归调用自身，后续的归约过程可能不会终止。这是一种带标签的无限协程，它在执行后总会恢复自身。(CR3) 采用广度优先的方式来缓解无限循环。此外，一旦主协程耗尽，整个归约过程便会中止。最后，我们的实现将归约步数限制在 500 步以内。我们此前的论文 [1] 对这些性质进行了详细分析，包括确定性和终止性。

B. 约束类型处理

每当调用**match** 时，可能需要检查某个约束类型。此类检查由 Microsoft Z3 [25] 在 Flow 扩展中完成。Z3 可识别多种逻辑运算，例如否定、合取和析取。对符号的相等性检查以及对数字的比较（小于和大于）同样是基础操作。在这些二元关系中，一侧通常是变量，另一侧是常量或另一个变量。在我们的情形中，符号就是图 1 中的具体类型。

如前所述，协程类型中的变量可以表示任意具体类型。Z3 将包含宇宙中所有具体类型的集合称为`Concrete` 排序。我们将`Concrete`排序记为$\mathbf{K}$。设 $c: T \to \Theta_K$ 是一个用于收集类型中所有具体类型的函数。图 3 展示了 c 的定义。collect 函数会在前文介绍的归约规则之前运行，因为我们需要收集程序中的各种类型。组装好的`Concrete` 排序 $\mathbf{K}$ 会被传入**match** 函数，该函数在图 4 右侧以阴影背景突出显示。

$$(\tau, \cdot) \vdash \circ(S) \Rightarrow (\mathbf{0}, \cdot) \vdash \circ\langle S_1, \mathbf{t}(i)/B, S_2\rangle \quad \text{provided} \\ (i, S_1, S_2) = \mathbf{first}(S, \lambda i.\, \mathbf{h}(i) = ?s \wedge \mathbf{match}(\tau, s) = B) \tag{CR5}$$

$$(\tau, E) \vdash \circ(S) \Rightarrow (\mathbf{0}, \langle E, \tau\rangle) \vdash \circ(S) \quad \text{provided} \\ \mathbf{none}(S, \lambda i.\, \mathbf{h}(i) = ?u \wedge \mathbf{match}(\tau, u) \neq \bot) \tag{CR6}$$

$$(\mathbf{0}, \cdot) \vdash \circ(S) \Rightarrow (\mathbf{0}, \cdot) \vdash \circ(\langle S_1, \mathbf{t}(i)/B, S_2\rangle \setminus i') \quad \text{provided} \\ (i, S_1, S_2) = \mathbf{first}(S, \lambda i.\, \mathbf{h}(i) = ?t \wedge t \in I) \\ \text{and } (i', \cdot, \cdot) = \mathbf{first}(S \setminus i, \lambda i'.\, \mathbf{match}(i', t) = B) \tag{CR7}$$

$$\begin{aligned} c([\omega_1; \omega_2; \cdots; \omega_n]) &= \bigcup c(\omega_i) \\ c([\![\omega_1; \omega_2; \cdots; \omega_n]\!]) &= \bigcup c(\omega_i) \\ c(\psi\tau) &= c(\tau) \\ c(\tau/\rho) &= c(\tau) \cup c(\rho) \\ c(\tau_1 \mid \tau_2) &= c(\tau_1) \cup c(\tau_2) \\ c(\langle \tau_1, \tau_2, \cdots, \tau_n\rangle) &= \bigcup c(\tau_i) \\ c((\tau_1, \tau_2, \cdots, \tau_n)) &= \bigcup c(\tau_i) \\ c(K) &= \{K\} \end{aligned}$$

图 3：函数 c从某个类型中找出所有具体类型

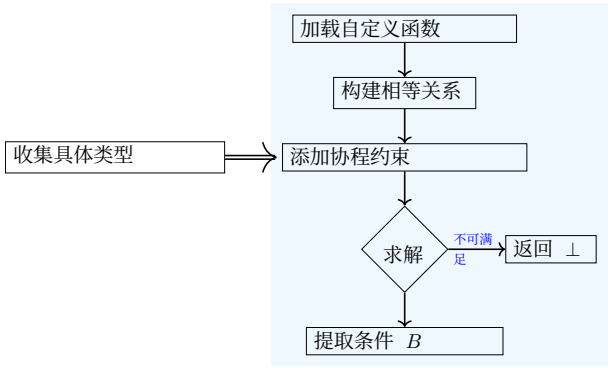

图 4：收集具体类型只运行一次。右侧的阴影框是**match** 函数的流程图

match （以及**none**）涉及构建布尔表达式、求解该表达式，并返回表达式为真时的条件。在 Flow 扩展中，变量的类型是推断得出的。如果变量的另一侧是整数，该变量的类型就是 Int。如果变量的另一侧是具体类型，该变量的类型就是 Concrete，其定义域为已收集的具体类型集合。如果两侧都是变量，它们应具有相同的类型。变量等于复合类型或**0** 是不允许的，因此会导致 $\bot$。**match** 处理的不仅是相等性检查，还包括可能出现在受约束类型 ρ 中的算术比较或自定义函数。例如，Int $/v < 10$ 中的小于号，以及 $[?\langle x, \text{Book}\rangle$ 中的子类型约束；$!\langle x, \text{Book}, \text{Reserve}\rangle]/x <: \text{User}$

虽然 Z3 附带了各种布尔和数值

清单 5：继承自定义函数的函数体

```
(forall ((x Concrete) (y Concrete))
  (令((r (非(或 (且(= x Faculty)          (= y 用户))
                (且(= x Student) (= y用户))))))
    (=> r( = (继承 x y)        假))))
```

表达式，但 Flow 扩展的用户（例如第四节介绍的 Flow 类型系统）可以注入自定义函数，例如子类型关系，以满足特定应用的需求。自定义函数必须返回布尔值。它们通常采用带有全称限定符和实质条件句的二阶逻辑。在 Go 程序中，类型继承通常通过匿名字段实现。我们用 SMT-LIB 语法定义函数头部 `inherit` 和函数体 [39]。清单 5 定义了一个函数体，其中`x` 的类型为`Concrete` ，`y` 的类型为`Concrete`。`r` 是已知子类型关系的逆，例如 Faculty 是 User 的子类，而 Student 是 User 的子类。然后，`=>` 是一个蕴含（实质条件句），表示如果`x` 和 `y` 之间不存在已知的子类型关系，`(inherit x y)` 就为假。

我们本可以保留一个持久的 Z3 上下文，并像处理「收集具体类型」那样只运行一次「加载自定义函数」，但 Z3 可能存在内存泄漏问题，因此我们必须定期释放 Z3 上下文。

当使用待处理类型 τ_1/ρ_1 和接收协程 $[?\tau_2; \cdots]/\rho_2$调用**匹配** 时，我们会构造一个布尔表达式（$\tau_1 = \tau_2$）$\wedge\rho_1\wedge\rho_2$，并让 Z3 求解。公式（1）也会用到。如果 Z3 对该布尔表达式返回“sat”，我们就从解释中提取条件 B （执行「提取条件」步骤），并将其用于 (CR5) 或 (CR7)。

Z3 是一个可满足性求解器，只会找出一组使表达式为真的变量取值。假设在 (CR5) 中，待处理类型为 $(x^1, y^j)/j < 5$，而**匹配** 将处理 $[?(x^i, y^j)$ 的头部；$?(y^j, x^i)]/j > 0$ 和 i，j 都是变量；那么我们会构造表达式 $(x^1, y^j) = (x^i, y^j) \wedge j < 5 \wedge j > 0$。Z3 会返回 $B = \{i \mapsto 1, j \mapsto 1\}$。虽然 j =1 满足该表达式，但我们不应将 j 替换为这个确切的数值，因为 j 的值尚未唯一确定。相反，我们应该将协程更新为 $[-(y^j, x^1)]/0 < j < 5$。

为了检查变量解释是否唯一，我们在初始求解后附加该解释的逆，

并再次检查。如果检查通过，则该解释不唯一，应从返回条件中排除。

IV. 用于 goroutine 死锁的 Flow 类型系统

Golang 将主函数作为根 goroutine 启动 [17,p9]，并且它可以创建新的 goroutine。如果所有 goroutine 都处于等待状态——程序进度完全停滞，Go 运行时便会认为程序永远不会结束，并报告“致命错误：所有 goroutine 都在休眠 - 死锁！”[36]

我们将新创建的 Flow 扩展应用于检测特定模式中的死锁。总的来说，我们将关键字 `go` 映射到 Start函数；将函数定义映射到协程定义；将通道的发送与接收操作映射到我们的让出与接收原语。

首先，我们解析源代码并识别函数定义。公式 (2) 检查 Golang 中的函数是否属于我们的协程函数。如果 Go 函数通过通道发送或接收数据，就会为该函数创建一个协程定义，并将发送操作标记为 $!\tau$，将接收操作标记为 $?\tau$。如果某个函数调用了另一个协程 f'，那么该函数就是一个带有 Flow 项 Inline(f') 的协程定义。如果某个函数使用关键字 `go` 启动另一个协程 f'，那么该函数也是一个带有 Flow 项 Start(f') 的协程。否则，该函数不属于 M' 的范围。接着，公式 (3) 通过求取 M' 的不动点来定义函数（或映射） M。

$$M'(f) = \begin{cases} [\![\cdot;\cdot]\!] & \text{if } f \text{ uses channels} \\ [\![\cdot;\cdot]\!] & \text{if } f \text{ calls } f' \in \mathrm{Range}(M') \\ \mathbf{0} & \text{otherwise.} \end{cases} \tag{2}$$

$$M = \mathrm{fix}(M') \tag{3}$$

函数 Inline 类似于 Start，会将协程定义转换为协程实例，但它会将新的协程实例加入当前的 Flow 项，而不是归约序列。关键字 `defer` 被视为 Inline，但其操作会插入协程 Flow 的末尾。

将 Go 函数作为协程定义类型进行类型化后，我们通过检查主函数来开始死锁检查。由于我们分析的是可执行程序而非库，因此禁用外部产出，并要求通道大小为 0。主函数也会被短路处理，因为主函数一旦结束，Go 运行时就会终止所有剩余的 goroutine。

要开始协程定义，我们会执行代码流分析（CFA） [40]，以确定参数的值和类型。如果 CFA 无法静态确定 if 条件表达式的真值，我们会将该条件表达式映射到 Flow 扩展的 ρ 部分，由 Z3 求解。形式上，`if` p $\{s_1;$ $s_2\}$ `else` $\{s_3;$ $s_4\}$ 会映射为 $(\langle s_1, s_2\rangle/p == \text{真} \mid \langle s_3, s_4\rangle/p == \text{假})$。示例请见列表 6。我们得到 $[\![\text{!整数}]\!]/1 \leqslant weekday \leqslant 3$ 和 $[\![\text{?整数}]\!]/3 \leqslant weekday \leqslant 5$。Z3 可以将 $weekday$ 的值划分为 4 组：1 到 2、3、4 到 5，以及 6 到 7。然后，我们对每组分别进行归约。显然，在星期三（3），发送和接收操作相互配对，因此不会产生锁。在周末（6 到 7），没有发送和接收操作，也不存在并发问题。在其他日期，程序会发生死锁。

列表 6：条件 goroutine 映射到 τ/ρ

```
函数main() {
  ...
  ch:= make(整数通道)
  如果 1 <=weekday&&weekday <= 3{
    启动 goroutine() {
      通道 <- 1
    }()
  }
  如果 3 <=weekday&&weekday <= 5{
    启动 goroutine() {
      fmt.Println(<-ch)
    }()
  }
  ...
}
```

我们使用代码清单 7 来说明整个工作流。该代码片段定义了一个`main` 函数和一个 `run` 函数。经过两次迭代后，M 达到不动点。函数 `run` 启动一个匿名 goroutine，因此 $M(run) = [\![\text{Start}([\![\text{!错误}]\!])]\!]$。函数 `main` 调用 `run()`，因此第一个 Flow 项是 Inline(run)，然后它从类型为错误的通道接收一个数据，因此下一个 Flow 项是 ?错误。

代码清单 7：moby#4395 的无死锁版本，其中M(主函数) $= [\![\text{Inline(运行)}; \text{?错误}]\!]$

```
func run(f func() 错误) chan 错误 {
  ch:= make(chan 错误)
  go func() {
    ch <- f()
  }()
  返回 ch
}

函数main() {
  错误: = 运行(函数() 错误 {
    返回 nil
  })
  <-错误
}
```

$$\begin{aligned} & \circ \langle \mathrm{Start}(main) \rangle \\ \Rightarrow\ & \circ \langle [\mathrm{Inline}(run); ?\mathrm{Error}] \rangle \\ \Rightarrow\ & \circ \langle [[\mathrm{Start}([\![!\mathrm{Error}]\!])]; ?\mathrm{Error}] \rangle \\ \Rightarrow\ & \circ \langle [?\mathrm{Error}], [!\mathrm{Error}] \rangle \\ \Rightarrow\ & \mathrm{Error} \vdash \circ \langle [?\mathrm{Error}] \rangle \\ \Rightarrow\ & \mathbf{0} \end{aligned}$$

图 5：moby#4395 的归约过程

图 5 展示了 moby#4395 的归约过程。Inline 将run 的 Flow 项添加到当前协程实例中，然后 Start 创建一个新协程，因此 ∘() 现在有两个参数。相应地，(CR2) 被激活并将错误放入上下文中。最后 (CR5) 和 (CR1) 运行并返回归约结果 0。由于所有类型相互抵消，这段代码是同步的。

一个发生死锁的示例如清单 4 所示。我们有 $M(work)$ =⟦?整数; ? 字符串 ⟧, $M(main)$ = ⟦Start($work$); ! 字符串 ; ! 整数 ⟧。归约结果为 [!字符串], , 这意味着这段代码会向其调用者让出字符串，但由于这是一个没有调用者的可执行程序，程序将会挂起。

A. 与其他死锁分析器的比较

我们将带 Flow 的协程类型应用于 Go 程序的死锁检测问题。对于一个通道，其发送位置可以在方法体中、普通函数内部、Start 调用中，或 Defer 调用中。共有 4 种情况。接收位置也有 4 种情况。根据这些情况，我们可以构造一种带锁的情形和一种不带锁的情形。因此，单个通道的一对发送与让出共有 32 种使用模式，尚未考虑嵌套调用。

我们并未评估全部 32 个案例，而是从 [41], [42], 及其他来源收集了 17 个单文件 Golang 程序，这些程序展示了通道和 goroutine 的典型用法。其中 10 个没有死锁，其余 7 个存在死锁。有些使用了不止一个通道，有些包含嵌套函数调用。我们的算法旨在识别这 17 种交互模式，如表 I 所示。

Yuan 等人 [43] 对著名开源仓库进行了抽样，并提取了 68 个阻塞错误。在这 68 个错误中，仅有 4 个完全由无缓冲通道和发送/接收原语引起。因此，我们使用这 4 个文件来测试我们的算法对真实世界程序的适用性。我们的协程归约算法成功识别了这 4 个案例。17 个人工案例和 4 个现实案例列于表 II。我们发布了这 21 个测试文件以及 Flow 扩展的源代码，供读者复现 [44]。

我们选取了两个静态工具和一个动态工具来检查它们的死锁检测准确性。在表 II 中，如果文件存在全局死锁，则标记为 Yes，否则标记为 No。错误的判定以粗体显示。在 P6 中，主线程会在其他 goroutine 之前退出。GOMELA 将其视为死锁，GoAT 则将其视为部分死锁（PDL）。这与死锁的一些微妙之处有关，因此我们不将这些判定标记为错误。所有这些工具均以默认设置运行。我们的算法没有可供微调的参数。

GoDDaR v2.0.2 [17] 有 8 个文件运行失败。对于 `moby#33293.go`，GoDDaR 报错“无效参数”。对于其他文件，该工具报错“递归调用”，尽管其中一些文件并不存在递归调用。我们怀疑 GoDDaR 在识别我们的测试用例时存在程序问题。

作为一种动态工具，GoAT [34] 在几乎所有情况下都是正确的。然而，GoAT 的阿喀琉斯之踵是其超时设置。如果某个被测对象完成执行所需的时间超过预定义的超时，GoAT 就会过早地判定发生死锁。这就是 GoAT 将 P10 标记为死锁的原因。GoAT 在给出所有 GDL 判定之前还必须花费长达 30 秒，而其他工具则瞬间给出判定。

GOMELA-ase21 [45], 作为 GoDel 的改进版本，因其有限的语法支持而毫不避讳地崩溃。错误消息包括“函数被声明为变量”、“在通道上发现了接收操作”、“返回一个通道”等。我们不确定 GOMELA 为何不允许对通道执行接收操作。此外，GOMELA 对 P10 不输出任何内容。

尽管 Flow 扩展在所选并发特性上表现良好，GoDDaR、GOMELA 和 GoAT 完全可能在我们这次未测试的特性上表现更出色，包括有缓冲通道、关闭通道、select 语句和循环。

V. 局限性

从语法方面来看，本文仅支持无缓冲通道、双向通道，以及关键字`go` 和 `defer`。我们识别有限数量的返回通道的内置 API，例如 `time.After`。当程序中存在其他特性时，例如仅接收或仅发送通道，Flow 类型系统会引发错误。在这种情况下，该程序需要使用其他死锁分析器。

一个重要的局限性来自语义层面。协程类型以及我们的 Flow 扩展都使用类型来建模一组协程的集体行为。由于类型表示为 τ/ρ，我们的 Flow 类型系统会将并发表达式的类型映射到 τ，并将带守卫的 if-条件表达式映射到 ρ，从而衔接纯粹基于类型的分析与纯粹基于值的分析。

清单 8：目前，我们不会区分类型相同的两个通道

```
func work(cInt 整数通道 , cStr 整数通道){
  fmt.Println(<-cInt)
  fmt.Println(<-cStr)
}

函数main() {
  cInt :=make(整数通道)
  cStr :=make(整数通道)
  Go work(c整数, cStr)

  cStr <- 2
  cInt <- 1
}
```

不过，本文不跟踪通道名称以及 if 条件表达式之外的变量。举例来说，清单 8 是对清单 4 的修改，其中涉及两个整数通道。问题的本质并未改变：程序仍会陷入死锁。我们的 Flow 类型系统为 `work()` 赋予类型 ⟦!整数;!整数⟧，并为 `main()`赋予类型 ⟦Start($work$); ?整数; ?整数⟧。由于通道名称已被剥离，归约算法认为两次让出操作和两次接收操作都正确配对，因此（错误地）判定程序不存在死锁。一

表 I：通道和 goroutine 的 17 种交互模式的描述

模式编号	简称	死锁	描述
P1	basic	No	计算两个数组的和。通过两个通道发送并接收结果
P2	basic-receiveInFunction	No	P1 的一个变体，接收操作位于另一个函数中
P3	defer2	No	嵌套`defer` 调用
P4	内联函数	No	带有 goroutine 和通道通信的匿名函数
P5	内联函数-外部变量	No	捕获外部作用域中变量的匿名函数
P6	主函数退出	No	主函数在 goroutine 完成之前结束
P7	递归主函数	No	递归 goroutine
P8	休眠接收者	No	接收方 goroutine 在通道上等待
P9	休眠发送者	No	发送方协程先等待，然后发送数据
P10	wait30	No	休眠一段时间后退出
P11	basic-receiveInFunction-extra	Yes	P1 的一个变体，接收操作位于另一个函数中，但配对有误。
P12	basic3receive	Yes	尝试接收的数据多于已发送的数据，导致死锁
P13	无发送方	Yes	尝试执行接收操作，但没有数据发送
P14	NoReceiver	Yes	尝试执行发送操作，但没有数据接收
P15	NoLiveGoroutines	Yes	包含两个 goroutine 的 P13 变体
P16	func	Yes	包含两个通道的 P13 变体
P17	乱序	Yes	两个不同类型的通道被乱序使用
P18	返回通道	Yes	一个通道从函数返回，且接收与发送未配对

表 II：我们的算法与其他 3 种工具的分析结果

程序	Ours	GoDDaR	GOMELA	GoAT
P1	No	**Err**	No	No
P2	No	**Err**	No	No
P3	No	**Yes**	**Err**	No
P4	No	No	**Err**	No
P5	No	No	**Err**	No
P6	No	**Err**	Yes	PDL
P7	No	**Yes**	**Err**	PDL
P8	No	No	No	No
P9	No	No	No	No
P10	No	No	**Err**	**Yes**
P11	Yes	**Err**	Yes	Yes
P12	Yes	**Err**	Yes	Yes
P16	Yes	Yes	Yes	Yes
P15	Yes	Yes	Yes	Yes
P13	Yes	Yes	Yes	Yes
P17	Yes	Yes	Yes	Yes
P18	Yes	Yes	**Err**	Yes
cockroachdb#25456	Yes	Yes	**Err**	Yes
grpc-go#1424	Yes	Yes	Yes	Yes
moby#33293	Yes	**Err**	**Err**	Yes
moby#4395	Yes	Yes	**Err**	Yes

种可能的修复方法是在 Flow 项中添加一个用于标识通道对象的标签。我们认为底层的协程归约算法能够处理这一点，因此将其留作未来工作，以推动相关研究继续开展。

最后，从操作层面来看，由于我们会在 ρ 部分跟踪变量名以评估条件表达式，因此我们可能会像其他静态分析工具一样遭遇状态爆炸问题。如列表 6 所示，单个变量 `weekday` 会扩展为 4 种状态。变量众多时，大型程序可能难以推理。未来，我们计划使用 SMT 求解器（例如 Z3）对整个程序进行推理，而不只是对 ρ 部分进行推理，因为 SMT 求解器擅长处理成千上万甚至数百万个变量 [46]。毕竟，死锁可能只取决于少数几个关键变量。

VI. 结论

协程是一种编程构造，能让代码更易于理解和管理，并可并行运行。协程类型 [1]是用于建模交互行为的行为类型，用于描述协程先执行单个接收操作、再执行单个让出操作的交互。我们为协程类型提出了一种 Flow 扩展，以便对包含多个接收和让出操作的协程进行建模。因此，我们修订了协程类型的归约规则。为展示 Flow 扩展的实用性，我们提出了一种 Flow 类型系统，将 Golang 表达式映射为协程类型，同时也考虑 `if` 语句中的值。Flow 类型系统旨在弥合纯粹基于类型的分析与纯粹基于值的分析之间的鸿沟。归约协程类型后，如果仍有让出和接收部分残留，程序就会表现出阻塞行为。

Go 是一种广泛应用于分布式系统的编程语言，但一种常见的错误是因错误使用并发特性而导致死锁。本文聚焦于 Go 中最常用的语义：无缓冲通道，以及 `go` 和 `defer` 关键字。如果 Golang 文件使用了不受支持的特性，我们的工具会发出警告并拒绝分析。

我们收集了 21 段正例与反例代码用于评估，因为细微的修改就可能修复一种阻塞行为。我们的算法旨在识别 17 种交互，并进一步在 4 个真实世界程序中成功预测了死锁行为。相比之下，三种最先进的死锁分析器在某些情况下会崩溃或给出错误预测。因此，我们基于类型的死锁分析器不仅填补了基于值的检测领域的空白，也补充了现有检测器。

未来，我们希望改进带 Flow 的协程类型，以表达更多语言结构，例如 Go 或其他编程语言中的有缓冲通道。我们也迫切希望对进程的活性和安全性属性进行形式化研究。最后，我们计划不再仅仅使用 Microsoft Z3 来评估 ρ，而是更广泛地使用 Z3，例如查找会导致并发错误发生的条件组合。

数据可用性

我们在 [44]发布了源代码。Flow 扩展位于`GeneratorCalculation`文件夹中；Flow 类型系统位于`Go`文件夹中；模式文件和测试用例位于`GoTests`文件夹中；构建和运行程序的方法位于`.github/workflows`文件夹中。

有关基础协程类型的源代码，请参阅 [2] 。

参考文献

[1] Q. Gu 和 W. Ke，“可组合协程的类型化”，载于《第十二届计算机技术与发展国际会议》（TechDev），2023年，第16–23页。[2] ——，“将需求模型作为协程进行类型化”，IEEE Access，第12卷，第8449–8460页，2024年。[3] L. Liu、Q. Gu 和 W. Ke，“AGTS：面向 Hyperledger Fabric 的智能合约测试套件新型自动生成方法”，Scientific Reports，第15卷，第1期，第29675页，2025年。[4] A. Roscoe，《并发理论与实践》。Prentice Hall，1998年。[5] J. Meyerson，“Go 编程语言”，IEEE software，第31卷，第5期，第104–104页，2014年。[6] F. Nielson 和 N. Jones，“抽象解释：一种基于语义的程序分析工具”，《计算机科学逻辑手册》第4卷，第527–636页，1994年。[7] A. Moller 和 M. Schwartzbach，《静态程序分析》。丹麦奥胡斯：奥胡斯大学计算机科学系，2023年。[8] C. Zhong，《BINGO：通过二进制分析定位 Go 中的并发错误》。北卡罗来纳州罗利：北卡罗来纳州立大学，2022年。[9] L. Padovani，“线性通信的基于类型的分析”，载于《行为类型：从理论到工具》。River Publishers，2017年，第193–217页。[10] G. Singh、T. Gehr、M. P¨uschel 和 M. Vechev，“用于神经网络认证的抽象域”，《ACM 编程语言论文集》，第3卷，POPL 专刊，第1–30页，2019年。[11] H. H¨uttel、I. Lanese、V. T. Vasconcelos、L. Caires、M. Carbone、P.-M. Deni´elou、D. Mostrous、L. Padovani、A. Ravara 等，“会话类型与行为契约基础”，ACM Computing Surveys（CSUR），第49卷，第1期，第1–36页，2016年。[12] O. H. Veileborg、G.-V. Saioc 和 A. Møller，“使用局部化抽象解释检测 Go 程序中的阻塞错误”，载于《第37届 IEEE/ACM 自动化软件工程国际会议》。纽约：计算机协会，2022年，第1–12页。[13] J. L˚ang 和 I. Prasetya，“模型检测 C++软件框架：案例研究”，载于《第27届 ACM 欧洲软件工程会议与软件工程基础研讨会联合会议》。纽约：计算机协会，2019年，第1026–1036页。[14] R. Milner，《通信系统演算》。柏林：Springer，1980年。[15] A. Almeida 和 A. Ravara，“并发程序中死锁的自动解决”，硕士论文，里斯本 NOVA 大学，2021年。[16] J. Lange、N. Ng、B. Toninho 和 N. Yoshida，“隔离 Go：基于通道编程的活性与安全性”，ACM SIGPLAN Notices，第52卷，第1期，第748–761页，2017年。[17] J. G. G. Coelho，“Go 程序中死锁的自动检测与解决”，硕士论文，里斯本 NOVA 大学，2022年。[18] H. Hosoya、J. Vouillon 和 B. C. Pierce，“XML 的正则表达式类型”，ACM 编程语言与系统汇刊（TOPLAS），第27卷，第1期，第46–90页，2005年。[19] A. M. Maidl、F. Mascarenhas 和 R. Ierusalimschy，“Typed Lua：Lua 的可选类型系统”，载于《动态语言与应用研讨会》。纽约：计算机协会，2014年，第1–10页。[20] K. Anton 和 P. Thiemann，“协程类型化”，载于《函数式编程趋势：第11届国际研讨会》。柏林：Springer，2011年，第16–30页。

[21] J. Lange、N. Ng、B. Toninho 和 N. Yoshida，《使用行为类型对 Go 中的消息传递进行静态验证的框架》，载于第40届国际软件工程大会论文集，2018年，第1137–1148页。[22] D. Barclay，《一种协程语言》，博士学位论文，巴斯大学，英国巴斯 BA2 7AY，2009年。[23] T. Leesatapornwongsa 和 H. S. Gunawi，《Samc：一种用于在分布式系统中查找海森堡错误的快速模型检查器》，载于国际软件测试与分析研讨会。美国纽约州纽约市：计算机协会，2015年，第423–427页。[24] Z. Liu、S. Zhu、B. Qin、H. Chen 和 L. Song，《自动检测并修复 Go 软件系统中的并发错误》，载于第26届 ACM 编程语言与操作系统架构支持国际会议。美国纽约州纽约市：计算机协会，2021年，第616–629页。[25] L. De Moura 和 N. Bjørner，《Z3：一种高效的 SMT 求解器》，载于系统构造与分析的工具和算法国际会议。德国柏林：Springer，2008年，第337–340页。[26] D. Honnef。（2024年）Staticcheck——高级 Go代码检查工具。 [在线]。网址：https://github.com/dominikh/go-tools [27] Golang。（2024年）vet命令。[在线]。网址：https://pkg.go.dev/cmd/vet [28] D. Zhang、P. Qi 和 Y. Zhang，《GoDetector：检测 Go 中的并发错误》，IEEE Access，第9卷，第136 302–136 312页，2021年。[29] A. Garcia、C. Laneve 等，《JaDA——Java死锁分析器》，《行为类型：从理论到工具》，第169–192页，2017年。[30] A. Gosain 和 G. Sharma，《动态程序分析技术与工具综述》，载于第3届智能计算前沿：理论与应用国际会议（FICTA）2014论文集：第1卷。Springer，2015年，第113–122页。[31] G. Liang、J. Pang 和 C. Dai，《一种基于行为的恶意软件变种分类技术》，《国际信息与教育技术期刊》，第6卷，第4期，第291页，2016年。[32] G. Packages。（2024年）trace包。[在线]。网址：https://pkg.go.dev/runtime/trace [33] ——。（2024年）trace命令。[在线]。网址：https://pkg.go.dev/cmd/trace [34] S. Taheri 和 G. Gopalakrishnan，《GoAT：Go的自动化并发分析与调试工具》，载于国际工作负载特征研讨会。康涅狄格州斯托尔斯：IEEE，2021年，第138–150页。[35] M. Christakis、A. Gotovos 和 K. Sagonas，《系统化测试：检测Erlang程序中的并发错误》，载于2013年IEEE第六届软件测试、验证与确认国际会议。IEEE，2013年，第154–163页。[36] N. Ng 和 N. Yoshida，《通过全局会话图合成对并发 Go 进行静态死锁检测》，载于第25届编译器构造国际会议。美国纽约州纽约市：计算机协会，2016年，第174–184页。[37] Uber。（2017年）goleak——goroutine泄漏检测器。[38] M. P. Papazoglou、P. Georgiadis 和 D. G. Maritsas，《编程语言 Simula概述》，《计算机语言》，第9卷，第2期，第107–131页，1984年。[39] C. Barrett、A. Stump、C. Tinelli 等，《SMT-LIB标准：第2.0版》，载于第8届可满足性模理论国际研讨会论文集，第13卷，英国爱丁堡，2010年，第14页。[40] C. H. Flemming Nielson、Hanne Riis Nielson，《程序分析原理》。德国柏林：Springer，1999年，第3章“基于约束的分析”，第141页。[41] Y. Inaura，《Golang——理解通道、缓冲区、阻塞、死锁和顺畅运行的goroutine》，2023年。[在线]。网址：https://gist.github.com/YumaInaura/8 52 73 7 361745 568 3 4 37 d e dac dc bf c c ba [42] *101* 2019 [ ] T. Liu，《Go》。加拿大维多利亚：Leanpub，。在线。网址：https://go101.org [43] T. Yuan、G. Li、J. Lu、C. Liu、L. Li 和 J. Xue，《Gobench：真实世界Go并发错误的基准测试套件》，载于IEEE/ACM代码生成与优化国际研讨会。韩国首尔：IEEE，2021年，第187–199页。[44] Blind。论文源代码。[在线]。网址：h t t p s://drive.google.com/file/d/1yFShrbRLIbhAwUqTv7aPcsPo7Ked2J7t/view?usp=sharing [45] N. Dilley 和 J. Lange，《Go中消息传递并发的有界验证》。英国坎特伯雷：肯特大学，2022年。

[46] E. ´Abrah´am 和 G.Kremer，《算术理论的 SMT 求解：理论与工具支持》，载于*2017*年第*19*届符号与数值算法科学计算国际研讨会（*SYNASC*）。IEEE，2017年，第1–8页。