arXiv:2602.22075v1 [cs.PL] 25 Feb 2026

# RustyDL：一种 Rust 程序逻辑

Daniel Drodt(✉)[0000−0003−3036−8220] 和 Reiner Hähnle[0000−0001−8000−7613]

达姆施塔特工业大学，德国达姆施塔特
{daniel.drodt,reiner.haehnle}@tu-darmstadt.de

**摘要。** Rust 是一种现代编程语言，凭借强大的类型系统保证内存安全并杜绝数据竞争。我们介绍 RustyDL——一种 Rust 程序逻辑，作为面向 Rust 的自动交互式演绎验证工具的基础。与其他全部基于转换为中间语言的工具不同，RustyDL 直接在源代码层面上推理 Rust 程序。面向 Rust 的源代码级程序逻辑，对于人在回路中（HIL）式验证至关重要，因为这种验证方式允许证明高度复杂的函数性质。我们讨论了为 HIL 式验证设计程序逻辑和演算时面临的 Rust 特有挑战，并针对每种挑战提出解决方案。我们以原型的形式，对一个广为人知的演绎验证工具 KeY 的 Rust 实例进行了概念验证，从而展示了这些想法。

**关键词：** 演绎验证 · 程序逻辑 · Rust验证

## 1 引言

Rust 是一种现代编程语言，拥有强大的类型系统，无需垃圾回收即可保证内存安全。为此，它依赖类型系统来管理所有权、引用和借用，从而防止数据竞争。Rust 变得非常流行，并开始应用于 Linux 内核 [24,32]。在这类安全关键型应用场景中，程序的正确性至关重要，因此形式化验证发挥着重要作用。

过去几年中，人们提出了多种用于对 Rust 代码进行形式化验证的工具。大多数现有工具 [4,9,12,22] 采用的通用方法是，将 Rust 代码和形式化规约转换为中间语言，例如 Viper [26] 或 Why3 [15], ，而现有验证工具支持这些中间语言。随后，这些中间代码被用于生成一阶验证条件，并由 SMT 求解器完成证明。

这种基于翻译的方法能够相对轻松地实现演绎验证工具，因为它建立在现有工具链之上。它还提供了高度的自动化。这种架构的缺点有：首先，必须信任从 Rust 及其规约到中间语言的（高度非平凡的）翻译；其次，当验证尝试失败时，很难将 SMT 求解器的输出与正在验证的代码和规约联系起来：验证工程师无法直接与证明交互，也无法手动应用证明步骤。

在证明的完全自动化与对验证过程的完全控制之间，现有的 Rust 验证工具正趋向于前者。另一方面，KeY [7] 或 KIV [30] 等验证工具提供了明确的证明对象，并允许用户检查和应用任意证明步骤，同时保留强大的自动化能力。这是通过在源代码 层面采用程序逻辑实现的，通常以序列式演算进行公理化。这种“人在回路中”（HIL）的源代码级方法支持细粒度控制。因此，人们得以验证（并发现其中的错误）高度复杂且未经修改的 Java 源代码库 [8,11]。迄今为止，基于翻译到中间语言的系统尚未实现这一点。

目前，还没有面向 Rust、基于源代码的 HIL 演绎验证器。因此，一个自然而然的研究问题随之产生，这也是本文要回答的问题：能否将基于源代码的 HIL 验证应用于 Rust？需要满足哪些条件？如何在源代码程序逻辑中对 Rust 的类型系统和行为（包括所有权和引用）建模？因此，我们的主要成果是用于基于源代码的 HIL 验证的程序逻辑 RustyDL。我们表明，带有少量常见扩展的动态逻辑非常适合 Rust验证。作为概念验证，我们还提供了一个基于 KeY 验证器的实现原型，名为 $Rusty\ KeY$。

第 2 节介绍 Rust 的背景知识，并引入我们的 Rust 程序逻辑。第 3 节定义 RustyDL 的序列演算。第 4 节将我们的方法与相关工作进行比较。我们在第 5 节总结全文。

## 2 Rust 程序逻辑

Rust 是一种复杂的语言，拥有丰富的类型系统并支持并发。它分为两种子语言：*安全 Rust* 对开发者有所限制，但提供了强有力的内存安全保证；而*不安全 Rust* 保证较弱，但赋予开发者更多控制权。后者必须使用关键字 **unsafe** 标记。

我们聚焦于安全 Rust，原因如下：(1) 大多数 Rust 函数都是安全的；(2) 对安全 Rust 建模为验证带来了机遇（也带来了挑战）：在这种情况下，无需使用分离逻辑或动态框架等机制；(3) 可以假设不安全 Rust 已封装起来，也就是说，它仅限于代码中可单独分析和验证的一部分 [5]。此外，我们在此聚焦于 Rust 的一个子集，主要包括基本值、共享引用和可变引用，以及数组。结构体和枚举可以采用与数组或引用类似的方式建模。联合体、智能指针和特质留待未来工作处理。

### 2.1 Rust 编程语言

我们简要回顾理解本文所需的 Rust 特性。有关 Rust 的详细信息和完整内容，请参阅 [21,28]。Rust 应用程序和库被组织成称为 *crate* 的单元，它们可以依赖其他 crate。每个 crate 都被划分为包含函数、类型定义和实现的模块。

**清单 1。** Rust 中的共享引用和可变引用

```
1 letn:u32 = 1;2 letx: &u32 =&n;3 println!("{}", *x); //打印"1"
45 fn increase_lower(a: &mut u64, b:&mut u64,amount:
u64) {6 let lower = if a < b { a }else { b };7 *lower = *
lower + amount;8 }
```

本文关注的 Rust 类型包括 布尔类型、整数类型 i8、……、i128（有符号整数）、u8、……、u128（无符号整数）、空元组类型 ()以及数组类型，例如 [i16; 10]。Rust 数组的长度固定（此处为 10），且其长度是类型的一部分。

Rust 的独特之处在于其类型系统会强制执行所有权和引用。所有权意味着每个值都恰好由一个所有者（例如变量或数组字段）拥有。所有者超出作用域后，该值便会从内存中移除，也就是被丢弃。这一特性是 Rust 内存安全的基础。值的所有权会在赋值时转移——也就是说，该值会被移动。

示例 *1*。 清单 1 中的函数 increase_lower 接收两个可变引用 a、b 作为参数，比较它们的值，将较小值对应的引用存入 lower，并将该值增加 amount。请注意，在 Rust 中，条件语句、循环和代码块等控制结构都是表达式。第 6 行的赋值会将 a 或 b 移动到 lower。因此，a 或 b 中有一个不再处于已初始化状态，而 lower 现在包含被移动的值。在第 6 行之后读取 a 或 b 会导致编译时错误。这确保只读取已初始化内存。

内存占用较小的类型（例如整数类型）不遵循移动语义。它们的值被复制，而不是在赋值时被移动。用 Rust 的术语来说：它们实现了Copy trait。本文中唯一未实现该 trait 的类型是可变引用以及可变引用的数组。

由于仅通过一个所有者访问某个值通常限制过大，Rust 提供了引用。共享（或不可变）引用（&）允许对借用值进行只读访问，而可变引用（&mut）允许读写访问。如清单 1 的前三行所示，类型为u32 的共享引用的类型是&u32。借用值1 可通过引用x 使用解引用运算符 *x读取。当借用x处于活跃状态时，我们称n被借用。对于可变引用，借用值不仅可以被读取，还可以使用解引用运算符进行修改，见示例的倒数第二行。

Rust 编译器——更具体地说，其借用检查器——确保程序遵循某些规则，从而保证不会发生数据竞争：(1) 不会直接写入被借用的所有者；(2) 对被可变借用的所有者，只能通过引用读取

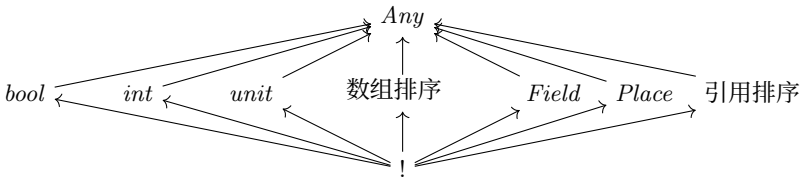


**图 1.** RustyDL 的排序层级

；(3) 在程序的任意时刻，每个所有者要么只被可变借用一次，要么可以被不可变借用任意多次；(4) 借用值不会被丢弃。

Rust 中的引用具有相关联的生命周期，这在复杂程序中是必要的。我们的逻辑不考虑生命周期，因此在此省略。

在下文中，我们不仅考虑完整的 crate，也考虑 Rust 程序的片段。定义 1 定义了合法程序片段的集合，也就是我们的逻辑能够处理的 Rust 片段。

**定义 1（合法程序片段）**。设 *Crt* 是一个有效的（即经 *Rust* 编译器认可的）*Rust crate*。合法程序片段是 *Crt* 中一系列 *Rust* 语句 $p$ （末尾可选带一个 *Rust* 表达式），其中局部变量$\mathsf{a}_1, \ldots, \mathsf{a}_n$ 的 *Rust* 类型为 $\mathsf{T}_1, \ldots, \mathsf{T}_n$，使得将该函数插入

$$\mathsf{fn}\ \mathsf{f}(\mathsf{mut}\ \mathsf{a}_1\colon \mathsf{T}_1, \ldots, \mathsf{mut}\ \mathsf{a}_n\colon \mathsf{T}_n)\ \{\ \{\ p\ \};\ \}$$

*Crt* 的入口点后，会得到一个符合 *Rust* 编译器要求的合法程序。

片段 $p$ 被包裹在一个块表达式中。我们添加分号以确保函数不返回任何内容，从而简化类型检查。

### 2.2 Rusty 动态逻辑

动态逻辑是一种程序逻辑，由 [27] 提出，并且非常适合程序验证 [2]。我们介绍一种针对 Rust 定制的动态逻辑，称为 RustyDL，它是类型化一阶逻辑的扩展。我们的定义基于 JavaDL（用于 Java 的动态逻辑）的定义，参见 [1]。

图 1 展示了 RustyDL 的排序层级，即逻辑类型的层级。$Any$ 是所有类型的超排序，而“永不排序”! 是任何项都不能具有的底排序。类型bool 映射为$bool$，单元类型() 映射为$unit$。有符号和无符号整数类型映射为数学整数$int$ （关于如何实现精确模型，参见第 3.4 节）。对于每种数组类型 [T;N] ，我们创建一个数组排序 $[T; N]$。此外，对于每种类型 $A$，我们定义一个共享引用排序$\mathrm{Ref}^s_A$ 和一个可变引用排序 $\mathrm{Ref}^m_A$。Rust 类型&T 和&mut T 按预期映射到相应的引用排序。

我们要求签名中包含零元谓词 true 和 false、用于表示相等的二元谓词 $\doteq (Any,\ Any)$、所有常见的整数函数和谓词（例如 $+$，$-$，$\cdot$，$>$），以及类型为 布尔类型 的常量 真 ， 假 。

我们同时使用 true 和 真，以区分永真式和真布尔值：前者是公式，后者是项，因此二者可能出现在不同的情境中。false 和 假 的情况也类似。

RustyDL 将经典一阶逻辑项集扩展为包含程序变量，例如 a : 整数，其中 整数 是 a 的类型。这些变量与常量类似，但其值依赖于状态。与一阶逻辑变量不同，程序变量可以出现在程序中，但不能对程序变量进行量化。

我们将一阶公式集扩展为包含模态算子 $\langle p\rangle\phi$（“菱形算子”）和 $[p]\phi$（“方框算子”），用于程序片段 $p$ 和公式 $\phi$。直观而言，对于确定性程序，$\langle p\rangle\phi$ 表示 $p$ 会终止，并且在执行后得到的状态中，$\phi$ 成立，即完全正确性。反之，$[p]\phi$ 成立，当且仅当 $p$ 不会终止，或 $\phi$ 在结果状态中成立，即部分正确性。

论域 $\mathcal{D}$ 上的克里普克结构 *Kripke structure* $\mathcal{K} = (\mathcal{D}, \mathcal{I}, \mathcal{S}, \rho)$ 及解释 $\mathcal{I}$，形式化了模态公式所表达的状态变化，并依据程序可能引发的状态转换 $\rho$，对状态集 $\mathcal{S}$ 进行排序。$\mathcal{S}$ 中的状态将程序变量映射到 $\mathcal{D}$ 中的元素。$\mathcal{I}$ 与一阶逻辑中一样，对函数和关系进行求值。对于程序片段 $p$，$\rho(p)$ 是从 $\mathcal{S}$ 到 $\mathcal{S}$ 的偏函数，用于定义 $p$ 引起的状态变化。仅当 $p$ 正常终止时，才定义这种变化，参见定义 2。

**定义 2（RustyDL 克里普克结构）**。设 *Crt* 是一个有效的 *Rust crate*。对于从 *Crt* 派生的签名，RustyDL 克里普克结构是一个二元组

$$\mathcal{K} = (\mathcal{S}, \rho)$$

其中

– $\mathcal{S}$ 是一个无限的一阶结构集合 $s = (D, \mathcal{I})$，称为状态，并且满足以下性质：所有状态的论域 $D$ 和函数符号与谓词符号的解释 $\mathcal{I}$ 均相同。– $\rho$ 是一个函数，它将合法程序片段 $p$映射为一个偏函数$\mathcal{S} \rightharpoonup \mathcal{S}$，且当且仅当在状态 $s_1$ 中求值 $p$ 正常终止于 $s_2$时，$\rho(p)(s_1) = s_2$ 成立。如果从 $s_1$开始执行 $p$ 未能正常终止，则 $\rho(p)(s_1)$ 未定义。请注意，我们假设 *Rust* 程序具有确定性。

为简洁起见，转移函数 $\rho$ 未予定义，因为它取决于 Rust 的形式语义。我们的演算规则为 $\rho$提供公理语义。定义 3 给出了相对于克里普克结构 $\mathcal{K}$的项和公式的语义，其中包括程序变量和模态。

**定义 3（RustyDL 项和公式的语义）**。设 *Crt* 是一个有效的 *Rust crate*，$\mathrm{VSym}$ 是逻辑变量符号的集合，$\mathcal{K} = (\mathcal{D}, \mathcal{I}, \mathcal{S}, \rho)$ 是一个用于建模 *Rust* 程序的克里普克结构，$s \in \mathcal{S}$ 是一个状态，$\beta : \mathrm{VSym} \to \mathcal{D}$ 是一个变量赋值。

对于每个 *RustyDL* 项 $t$，我们通过求值函数 $val_{\mathcal{K},s,\beta}$ 归纳定义其求值，方式与一阶逻辑完全相同；其中，对程序变量的求值定义为，对于每个程序变量 a，$val_{\mathcal{K},s,\beta}(\mathsf{a}) = s(\mathsf{a})$。

对于每个 *RustyDL* 公式 $\phi$，我们定义其满足关系，记作 $(\mathcal{K}, s, \beta) \models \phi$，与一阶逻辑中的定义完全相同，但增加了以下两种情况：

*1.* $(\mathcal{K}, s, \beta) \models [p]\phi$ 当且仅当 $\rho(p)(s)$ 未定义，或 $\rho(p)(s) = s'$ 且 $(\mathcal{K}, s', \beta) \models \phi$；
*2.* $(\mathcal{K}, s, \beta) \models \langle p\rangle\phi$ 当且仅当 $\rho(p)(s) = s'$ 存在且 $(\mathcal{K}, s', \beta) \models \phi$。

在 RustyDL 中，可以表达 Rust 代码的各种属性。程序 $p$ 关于前置条件和后置条件 前置条件，后置条件 的部分正确性写作前置条件 $\rightarrow [p]$后置条件。此公式等价于霍尔三元组 $\{$前置条件$\}\,p\,\{$后置条件$\}$ [18]，且当且仅当对于每个状态 $s$，只要前置条件 在 $s$ 中成立，对程序 $p$在 $s$ 中求值要么不终止，要么在所得状态 $s'$中后置条件 成立，该公式才有效。相比之下，前置条件 $\rightarrow \langle p\rangle$后置条件 表达的是完全正确性。与霍尔逻辑不同，动态逻辑对所有运算都封闭。因此，例如，后置条件可以包含其他模态，以表达程序等价、精化等性质 [2]。

为了获得一种可用且能高效建模状态变化的 RustyDL 演算，我们提供了一种比称为更新的模态更细粒度的状态转换机制。定义 4 将更新 [1]扩展至 JavaDL，并加入变异更新。

**定义 4（更新，更新应用）**。我们通过以下方式归纳定义集合Upd（与 *[1]*相比，第二个子句是新增的）：

*1.* 对每个程序变量 $a:A$ 以及每个类型为 $A$的项 $t$，$(a := t) \in \mathit{Upd}$；*2.* 对每个类型为 $\mathit{Ref}_A^n$ 的项 $t_1$ 和类型为 $A$的 $t_2$，$(t_1 \leftarrow\!* \, t_2) \in \mathit{Upd}$；*3.* $\mathit{skip} \in \mathit{Upd}$；*4.* 对所有更新 $u_1$和 $u_2 \in \mathit{Upd}$，$(u_1 \parallel u_2) \in \mathit{Upd}$；*5.* 对所有更新 $u_1$ 和 $u_2 \in \mathit{Upd}$，$(\{u_1\}\, u_2) \in \mathit{Upd}$。

我们将项的归纳定义扩展为：对于所有更新 $u \in$Upd 和类型为 $A$的项 $t$，加入类型为 $A$ 的项 $\{u\}\,t$。公式的归纳定义扩展为：对于所有更新 $u \in$ Upd 和公式 $\phi$，加入公式 $\{u\}\,\phi$。

更新在语法上被应用 于项和公式。更新应用会改变项（公式）求值时所处的状态。更新使我们能够将大型程序片段分解为一系列小的确定性状态变化（见下文定义 5）。

例 *2.* 设 x，y 为类型 i32 的程序变量，且 z 的类型为 &**mut** i32。我们给出更新及其直观含义的例子：

– skip 表示不发生状态变化的更新。– 基本更新 $(\mathrm{x} := \mathrm{x} + 1)$ 表示一种状态变化，即程序变量 x 递增 1。更新应用 $\{\mathrm{x} := \mathrm{x} + 1\}\mathrm{x}$ 在语义上等同于表达式 $\mathrm{x} + 1$。– 并行 更新 $(\mathrm{x} := \mathrm{y} \parallel \mathrm{y} := \mathrm{x})$ 表示一种状态变化，即 x 和 y 交换值。这种并行更新表示彼此之间没有干扰的状态变化。在一次并行更新中，同一变量可以被多次更新；此时，最后一次更新会覆盖先前的更新。$-(\mathrm{z} \leftarrow\!* \, \mathrm{y} - 1)$ 是一种变异 更新，表示将 z 借用的位置设为 $\mathrm{y} - 1$。

变异更新对于在我们的逻辑中表示可变引用至关重要。要在 RustyDL 中对后者建模，我们不仅需要涉及值，还需要涉及被借用的位置（出借方）。为此，我们使用类型位置（这一概念取自 Rust 编译器 [29]）。对于每个程序变量 x，我们定义一个唯一的常量 $\llbracket \mathsf{x} \rrbracket$ :位置。为了对数组字段的位置建模，我们为每种类型 $A$ 和 $n \in \mathbb{N}$ 添加一个函数 $arrPlc$: $\mathsf{Ref}^m_{[A;n]} \times$ 整数 $\to$ 位置，直观而言，该函数将每个字段映射到其位置。我们将 $\llbracket t, t' \rrbracket$ 记作 $arrPlc(t, t')$。因此，项$mref$(*pl*) 的类型为 $\mathsf{Ref}^m_A$，其中 $pl$ : 位置是类型为 $A$ 的出借方的位置。第 3.5 节将展示如何使用这些可变引用。

示例 *3.* 设 x : 整数 和 a : [整数; 4] 为程序变量。可变引用 $mref(\llbracket \mathsf{x} \rrbracket)$ 表示 x 被可变借用，而 $mref(\llbracket mref(\llbracket \mathsf{a} \rrbracket)\ 1 \rrbracket)$ 表示数组 a 的索引 1 处的字段被可变借用。

变异更新是我们逻辑的核心扩展。它们自然地融入我们的动态逻辑，并且能够以很少的概念开销处理可变引用。其他方法，例如显式建模内存或活动借用集合，一旦用于逻辑中，很快就会变得笨重且计算复杂。

定义 5 对上述直观描述的更新语义及其应用进行了形式化。

**定义 5（RustyDL 更新的语义）**。设 *Crt* 为一个有效的 *Rust crate*，$\mathcal{K} = (\mathcal{S}, \rho)$ 为 *Crt* 的一个克里普克结构，$s \in \mathcal{S}$ 为一个状态，$\beta : \mathrm{VSym} \to D$ 为一个变量赋值。赋值函数 $val_{\mathcal{K},s,\beta} : \mathrm{Upd} \to (\mathcal{S} \to \mathcal{S})$ 定义如下：

$$val_{\mathcal{K},s,\beta}(\mathsf{a} := t)(s')(\mathsf{b}) = \begin{cases} val_{\mathcal{K},s,\beta}(t) & \text{if } \mathsf{b} = \mathsf{a} \\ s'(\mathsf{b}) & \text{otherwise} \end{cases} \quad \text{for all } s' \in \mathcal{S}, \mathsf{b} \in \mathrm{ProgVSym}$$

$$val_{\mathcal{K},s,\beta}(t_1 \mathrel{*\!\!\to} t_2)(s')(\mathsf{b}) = \begin{cases} val_{\mathcal{K},s,\beta}(t_2) & \text{if } val_{\mathcal{K},s,\beta}(t_1) = val_{\mathcal{K},s,\beta}(mref(\llbracket \mathsf{b} \rrbracket)) \\ s'(\mathsf{b}) & \text{otherwise} \end{cases}$$
$$\text{for all } s' \in \mathcal{S}, \mathsf{b} \in \mathrm{ProgVSym}$$

$$val_{\mathcal{K},s,\beta}(\mathtt{skip})(s') = s' \quad \text{for all } s' \in \mathcal{S}$$
$$val_{\mathcal{K},s,\beta}(u_1 \parallel u_2)(s') = val_{\mathcal{K},s,\beta}(u_2)(val_{\mathcal{K},s,\beta}(u_1)(s')) \quad \text{for all } s' \in \mathcal{S}$$
$$val_{\mathcal{K},s,\beta}(\{u_1\}\, u_2) = val_{\mathcal{K},s',\beta}(u_2) \text{ where } s' = val_{\mathcal{K},s,\beta}(u_1)(s)$$

我们扩展 *RustyDL* 项和公式语义的归纳定义（定义 *3*），加入

$$val_{\mathcal{K},s,\beta}(\{u\}\, t) = val_{\mathcal{K},s',\beta}(t) \text{ where } s' = val_{\mathcal{K},s,\beta}(u)(s)$$

*and*

$$(\mathcal{K}, s, \beta) \models \{u\}\, \phi \text{ iff } (\mathcal{K}, s', \beta) \models \phi \text{ where } s' = val_{\mathcal{K},s,\beta}(u)(s) \ .$$

例 *4.* 设 x 是一个类型为 i32 的程序变量。显然，$\mathrm{x} \doteq 0 \to \mathrm{x} > 0$ 不可满足。然而，对右侧应用更新 $\mathrm{x} := \mathrm{x} + 1$ 后，该公式成立：

$$\models \mathrm{x} \doteq 0 \to \{\mathrm{x} := \mathrm{x} + 1\}\,(\mathrm{x} > 0).$$

将上述更新与以下模态进行比较：

$$\mathrm{x} \doteq 0 \to \langle \mathrm{x} = \mathrm{x} + 1; \rangle (\mathrm{x} > 0).$$

观察其差异颇具启发：更新是完全 状态变化，它们保证正常终止。因此，它们类似于显式替换。另一方面，模态则可能不终止。加法可能导致溢出，从而引起异常终止： $\rho(p)$是部分的。

## 3 RustyDL 演算

回想一下，程序片段 $p$相对于前置条件前置 和后置条件后置的函数式完全正确性，是由 RustyDL 公式前置 $\to \langle p \rangle$后置的有效性来表达的。这可以在 RustyDL 的序列演算中证明：通过推导序列式“$\Longrightarrow$ 前置 $\to \langle p \rangle$后置”来表明有效性。

下文给出的演算是可靠的：本节中的所有规则均已在正在进行的工作中相对于形式语义 [14] 被证明为可靠。

### 3.1 RustyDL 序列演算的设计中的挑战

我们以程序逻辑的规则来建模 Rust 的独特特性及其语义，力求保持人类可读的形式，以便进行人工交互。为此，我们基于 Rust 编译器 [29] 中仅略作简化并规范化的高级中间表示（HIR）开展工作。该表示接近源码级 Rust，但宏已展开，且诸如不同形式的循环等结构已规范化。

我们确定了本逻辑必须应对的 Rust 五项核心特性：（1）所有权，（2）整数语义，（3）数据类型，例如数组、结构体，（4）共享引用与可变引用，包括对数组字段的引用，以及（5）函数调用与循环可能无界的行为。出于篇幅考虑，我们仅呈现本逻辑及其实现所支持的 Rust 片段中的一个必要子集。下文讨论的这一子集展示了我们的逻辑与演算如何建模 Rust 的特性，同时它也是处理复杂问题所必需的最小范围。

### 3.2 符号执行演算

序列演算包含常见的命题规则和一阶规则，以及一套相对完备的数学整数规则。由于公式中可能包含模态，我们需要一种方法来简化并分解程序

片段，直到其中只剩下一阶公式。我们遵循[1,7] 来设计实现符号执行的规则模式。

考虑一个 if-else 表达式。它的执行会根据守卫条件的值分成两条路径。由于我们处理的是符号化程序变量，演算必须涵盖所有可能的路径，并拆分当前证明目标。

如下方的规则 ifElseSplit 所示，规则中包含模式变量： $\Gamma$、$\Delta$ 代表 RustyDL 公式集合， $\phi$ 代表一个 RustyDL 公式， $\mathcal{U}$ 代表一个可选的更新，$se$ 代表一个“简单表达式”（字面量或程序变量），{$be_0$} 和 {$be_1$} 代表块表达式。此外，程序模态中包含一个上下文，表示当前关注的程序片段，其中有非活动程序前缀$\pi$ 和剩余程序 $\omega$。通常， $\pi$ 与左花括号匹配，而 $\omega$ 与剩余语句和右花括号匹配。这个上下文使下方的规则能够匹配任意程序片段，只要其中下一个要执行的表达式是 if-else 表达式，即使后面还有其他语句，或者它是块表达式的开头。应用规则时，所有模式变量都会被实例化为相应类型的具体对象。

$$\text{ifElseSplit}\ \frac{\Gamma, \mathcal{U}(se \doteq TRUE) \Longrightarrow \mathcal{U}\langle \pi\ \{be_0\}\ \omega\rangle\phi, \Delta \quad \Gamma, \mathcal{U}(se \doteq FALSE) \Longrightarrow \mathcal{U}\langle \pi\ \{be_1\}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \texttt{if}\ se\ \{be_0\}\ \texttt{else}\ \{be_1\}\ \omega\rangle\phi, \Delta}$$

该规则对以 if-else 表达式开头的程序进行符号执行，其中守卫条件简单（因此无副作用）。结论中的证明目标被化简为对应于两条执行路径的两个前提。前提中的序列式表示我们知道每个分支中守卫条件 $se$ 的值：在 then 分支中，我们知道 $se$ 为 $TRUE$；在 else 分支中，我们知道它为 $FALSE$。对表达式（而非语句）进行匹配，使我们能够用一条规则处理语句中出现的赋值（x = y;），以及例如作为代码块值的赋值（{x = y}）。

如果 if 的条件是复杂表达式，即既不是字面量也不是程序变量，我们就通过引入一个新的辅助变量来对该条件进行符号执行。在规则 unfoldIfElse 中，$nse$ 表示“非简单表达式”，而 x 是一个尚未使用的程序变量。

$$\text{unfoldIfElse}\ \frac{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \{\texttt{let x} = nse\texttt{; if x } \{be_0\}\ \texttt{else } \{be_1\}\}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \texttt{if}\ nse\ \{be_0\}\ \texttt{else}\ \{be_1\}\ \omega\rangle\phi, \Delta}$$

为了在演算中对状态变化建模，我们采用更新。规则 copy如下，用于匹配将简单表达式赋给程序变量的任意赋值。应用该规则会在模态之前添加一个表示该赋值状态变化的更新，并将该赋值的原始表达式替换为单元表达式 ()。不出所料，下列规则仅适用于 Copy 类型的字面量或变量，即不具有移动语义。导致移动的赋值需要单独的规则。

$$\text{copy}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\texttt{x} := se\}\langle \pi\ ()\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \texttt{x} = se\ \omega\rangle\phi, \Delta}\quad se \text{ is a literal or implements Copy}$$

后续应用复制规则，会在模态之前累积更新，这些更新表示程序片段控制流中某一特定路径上的状态变化。累积的更新会被化简，并最终应用于后置条件。因此，该演算包含更新化简 规则。例如，将基本更新 x:= $t$（其中 x 为程序变量， $t$ 为项）应用于程序变量 x，对应于将 x 代换为 $t$。按我们的记法：

$$\mathsf{applyOnPV} \quad (\{\mathsf{x} := t\}\,\mathsf{x}) \rightsquigarrow t$$

完整的更新化简规则集请参见附录 A.1。

为去除规则 copy 留下的单元表达式，以及诸如 {2; a + b} 中的 2 这样的表达式，我们使用规则 simpleExprStmt。该规则会移除由简单表达式组成的表达式语句，因为这些语句不会影响程序状态。

$$\mathsf{simpleExprStmt}\ \frac{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ se;\ \omega\rangle\phi, \Delta}$$

例 *5.* 设 x、y、z 是类型为 i32 的程序变量。考虑序列式 x ≐ z ⟹ ⟨x = x + y; y = x - y;⟩(y ≐ z)。应用 copy，然后两次应用 simpleExprStmt，得到序列式 x ≐ z ⟹ {x := x + y} {y := x − y} ⟨⟩(y ≐ z)（暂不考虑溢出，相关内容将在第 3.4 节讨论）。该序列式可化简为（此规则未展示）x ≐ z ⟹ {x := x + y} {y := x − y} (y ≐ z)；经过更新应用和化简后，得到 x ≐ z ⟹ (x + y − y ≐ z)，这是一个有效的一阶序列式。

### 3.3 所有权

考虑赋值 x = y，其中 y 的类型未实现 Copy，例如可变引用。那么，y的值会从 y 移到 x。我们在逻辑中对这种行为进行建模：移动之后，在公式中无法通过 x访问 y 的值。一种逻辑建模方法是显式使用未初始化值。这种方法的缺点是，每次访问变量时都必须检查它是否包含这种特殊值。这与使用多值逻辑来建模未定义值时产生的问题类似 [16]。更高效的做法是依赖欠规格化，直接使用斯科伦化（称为 [1]中的匿名化更新）。move 规则与 copy类似，但它会将y更新为一个类型匹配的 $c$ 新常量：虽然y 仍然具有某个值，但除了它的类型外，我们对它一无所知。

$$\mathsf{move}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathsf{x} := \mathsf{y} \parallel \mathsf{y} := c\}\langle\pi\ ()\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \mathsf{x} = \mathsf{y}\ \omega\rangle\phi, \Delta}$$
$$\mathsf{y} : A,\ A \text{ is not } \mathsf{Copy} \text{ and } c : A \text{ a fresh constant}$$

### 3.4 处理整数与溢出

Rust 提供多种具有不同优化级别的编译模式，其中最常见的是调试模式 和发布模式。前者会执行更多检查，但性能较低。这些模式也会影响整数运算。

像 a + b 这样的表达式可能发生溢出；根据 Rust 的语义，这种情况并非预期行为，并会在调试模式下导致恐慌 [28]。在发布模式下，溢出检查会被省略，值会回绕。为了进行验证，我们关注调试语义，因此 RustyDL 必须证明安全性，即不存在溢出。规则 assignAddU32 用于匹配将两个简单表达式的和赋给具有 Rust 类型 x 的程序变量 u32的情形。该规则将证明分为两种情况。如果加法结果在 u32 类型的取值范围内，则将该和转换为更新；否则会触发恐慌。

$$\text{assignAddU32}\quad \frac{\Gamma,\mathcal{U}\ inU32(se_1+se_2) \Longrightarrow \mathcal{U}\{\mathsf{x} := se_1+se_2\}\langle\pi\ ()\ \omega\rangle\phi,\Delta \quad \Gamma,\mathcal{U}\neg inU32(se_1+se_2) \Longrightarrow \mathcal{U}\langle\pi\ \mathsf{panic!}()\ \omega\rangle\phi,\Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \mathsf{x} = se_1+se_2\ \omega\rangle\phi,\Delta}$$

$inU32$ 谓词的语义是对常量 u32::MIN（$=0$）与 u32::MAX（$=2^{32}-1$）之间的边界检查。

### 3.5 引用规则

回想一下，程序片段 $p$ 只有在 Rust 编译器（包括借用检查器）接受它时才有效。因此，我们的演算无需确保 $p$遵守借用检查器的规则，相反，它可以依赖其保证。

如何在 RustyDL 中为引用建模？一种方法是增加一个保留的程序变量，或一个幽灵变量，称之为借用，用于记录哪些变量借用了哪些其他变量，以及这些借用是共享的还是可变的。因此，借用表示任一时刻的活动借用集合。随后，如有必要，可将引用规则显式地作为关于借用的公理或规则引入。然而，活动借用的显式模型必须在每一次状态变化之后保持最新。如果我们遇到像 x = y 这样简单的语句，就不能使用 copy，因为我们可能还必须更新借用：如果 x 或 y 已被借用呢？那么必须从借用中移除这些变量的所有活动借用。这种方法并不能从 Rust 的借用检查器中获益，实际上等于重新实现了它。

因此，我们不显式建模活跃借用，而是将引用表示为逻辑项。对于共享引用，这很简单。遇到 x =&y 时，我们知道，只要 x 仍然有效，x 所借用的值就不能更改。因此，我们不必关心该值来自何处，只需使用 y 的值：

$$\text{borrowShared}\quad \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathsf{x} := sref(\mathsf{y})\}\langle\pi\ ()\ \omega\rangle\phi,\Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \mathsf{x} = \&\mathsf{y}\ \omega\rangle\phi,\Delta}$$

y 的当前值由 $\Gamma,\Delta$ 或 $\mathcal{U}$ 持有。对 y 的任何进一步更改（必然发生在 x 被丢弃或重新赋值之后）都不会影响 x。由于 x 不等同于 y，我们不会将 y 赋给 x，而是使用 $sref$(y)，即一个共享引用项。函数 $sref$ 接受一个类型为 $A$ 的项，并返回一个类型为 $\mathrm{Ref}_A^{\mathrm{s}}$ 的项。

例 *6.* 考虑序列式 $\Longrightarrow \langle \mathsf{y} = 2; \mathsf{x} = \&\mathsf{y}\rangle(\mathsf{x} \doteq sref(2))$。先应用规则 copy，再应用 borrowShared，得到

$$\Longrightarrow \{\mathsf{y} := 2\}\,\{\mathsf{x} := sref(\mathsf{y})\}\,\langle\rangle(\mathsf{x} \doteq sref(2))\ .$$

读取共享引用的值由函数 $*^s_A : \mathrm{Ref}^s_A \to A$ 完成，针对每个类型 $A$（为便于阅读，下文省略）。规则 derefShared 接受将 y 的解引用赋值给 x，并将其转换为带有 $*^s$ 的更新。化简规则 derefOfSharedRef 随后将项 $t$ 的共享引用的解引用转换回 $t$。

$$\textsf{derefShared}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathsf{x} := *^s(\mathsf{y})\}\langle \pi\ ()\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \mathsf{x} = \text{*}\mathsf{y}\ \omega\rangle\phi, \Delta}\quad \mathsf{x} : A, \mathsf{y} : \mathrm{Ref}^s_A,\ A \text{ is Copy} \qquad \textsf{derefOfSharedRef}\ \ *^s(sref(t)) \rightsquigarrow t$$

由于共享引用只考虑值，因此对于数组的共享引用，其处理方式完全相同。

可变引用有所不同，我们不能将其视为借用时的借用值：如果 x 是对 y 的可变借用，那么可能会出现类似 *x = $se$ 的赋值，这会改变 y 的值，而非 x 的值。因此，x 的值包含出借方，我们将对 *x 的赋值视为对该位置的赋值。

考虑借用变量。遇到可变借用时，例如在 x =&mut y 中，应用规则 borrowMut 会为 x 添加一个更新 $mref(\llbracket \mathsf{y} \rrbracket)$，表示 x 借用了 y：

$$\textsf{borrowMut}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathsf{x} := mref(\llbracket \mathsf{y} \rrbracket)\}\langle \pi\ ()\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \mathsf{x} = \text{&mut}\ \mathsf{y}\ \omega\rangle\phi, \Delta}$$

为了读取可变引用所借用的值，我们添加一元的依赖于状态的运算符 $*^m_A$，它接受一个类型为 $\mathrm{Ref}^m_A$ 的项，其自身类型为 $A$，并声明相应的查找规则 derefOfMutReference $*^m(mref(\llbracket \mathsf{x} \rrbracket)) \rightsquigarrow \mathsf{x}$。该运算符依赖于状态，以体现任何表示状态变化的更新。这在 x 曾被变更时可能是必要的。为了更新出现在 $*^m$ 中的变量 x，同时保留可能影响出借方的其他状态变化，我们使用规则 applyElemOnDeref $\{u_1 \parallel \mathsf{x} := t \parallel u_2\}\ *^m(\mathsf{x}) \rightsquigarrow \{u_1 \parallel u_2\}\ *^m(t)$。

此外，我们还需要一条规则，通过可变引用 x 来变更一个值。我们希望更新的不是 x 的值，而是出借方的位置 $\llbracket \mathsf{y} \rrbracket$。因此我们引入了变异更新 $t_1 \mathrel{*\!\!\to} t_2$，表示将项 $t_2$ 写入 $t_1$借用的位置。

$$\textsf{mutate}\ \frac{\Gamma \Longrightarrow \mathcal{U}\{\mathsf{x} \mathrel{*\!\!\to} se\}\langle \pi\ ()\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \text{*}\mathsf{x} = se\ \omega\rangle\phi, \Delta}$$

示例 7。考虑序列式 $\Longrightarrow \langle \mathsf{x}\ =\ 1; \mathsf{y}\ =\ \text{&mut}\ \mathsf{x}; \text{*}\mathsf{y}\ =\ 3;\rangle\phi$，其中程序变量为 x : 整数、 y : $\mathrm{Ref}^m_{int}$。应用规则 Copy 、 borrowMut 和 mutate 后，得到

$$\Longrightarrow \{\mathsf{x} := 1\}\,\{\mathsf{y} := mref(\llbracket \mathsf{x} \rrbracket)\}\,\{\mathsf{y} \mathrel{*\!\!\to} 3\}\,\langle\rangle\phi\ .$$

为了简化此类变异更新，我们声明两条简化规则。第一条规则 applyOn Mutating 将更新 $u$ 应用于变异更新。与基本更新不同（参见附录 A.1 中的表 1），$u$ 会应用于 $\mathrel{*\!\!\to}$ 的两侧。

规则 mutatingToElementary 会将变异更新转换为基本更新；对于该变异更新，已知其左侧为某个程序变量 a 的 $mref$(a)，转换结果为 a:= $t$。

applyMutating
$$\{u\}(t_1 \mathrel{*\!\!\to} t_2) \rightsquigarrow \{u\}t_1 \mathrel{*\!\!\to} \{u\}t_2$$

mutatingToElementary
$$mref(\llbracket \mathsf{a} \rrbracket) \mathrel{*\!\!\to} t \rightsquigarrow \mathsf{a} := t$$

示例 *8.* 继续示例 7，化简前两个更新后可得

$$\Longrightarrow \{\mathsf{x} := 1 \parallel \mathsf{y} := mref(\llbracket \mathsf{x} \rrbracket)\}\,\{\mathsf{y} \mathrel{*\!\!\to} 3\}\,\langle\rangle\phi$$

由于位置 $\llbracket \mathsf{x} \rrbracket$ 是一个常量，它不受更新影响。进一步化简可得 $\Longrightarrow \{\mathsf{x} := 1 \parallel \mathsf{y} := mref(\llbracket \mathsf{x} \rrbracket) \parallel mref(\llbracket \mathsf{x} \rrbracket) \mathrel{*\!\!\to} 3\}\,\langle\rangle\phi$. 使用 mutatingToElementary 后，我们得到 $\Longrightarrow \{\mathsf{x} := 1 \parallel \mathsf{y} := mref(\llbracket \mathsf{x} \rrbracket) \parallel \mathsf{x} := 3\}\,\langle\rangle\phi$，由此可见，$\mathsf{y} \mathrel{*\!\!\to} 3$ 被 $\mathsf{x} := 3$ 替换。最后，我们将被覆盖的更新化简掉，得到 $\Longrightarrow \{\mathsf{y} := mref(\llbracket \mathsf{x} \rrbracket) \parallel \mathsf{x} := 3\}\,\langle\rangle\phi$。

### 3.6 数组访问的建模

为了对数组访问进行建模，我们为每种数组类型 $[S; n]$ 使用两个函数：函数 $get_{[S;n]} : [S; n] \times Field \to S$ 接收一个数组和一个类型为 $Field$ 的项，并求得该数组在由该字段表示的索引处的值。函数 $set_{[S;n]} : [S; n] \times Field \times S \to [S;\ n]$ 接收一个数组、一个表示索引 $i$ 的字段以及一个值 $t$，并创建一个新的数组项，其中索引 $i$ 处的值被替换为 $t$。类型 $Field$ 允许对索引进行抽象：函数 $idx$ :整数 $\to Field$ 将整数转换为字段。类型 $Field$ 也用于元组字段和结构体字段。当前方法遵循 JavaDL 和数组理论 [25]。区别在于，在 RustyDL 中，只有数组索引可能存在别名，而数组元素不会。程序中的数组访问通过符号执行来处理：如果索引越界，我们就以恐慌继续；否则，我们通过更新对成功的访问进行建模。

示例 *9.* 设 a : [整数; 4] 且 i、j : 整数 为程序变量。项

$$get_{[int;4]}(set_{[int;4]}(set_{[int;4]}(\mathsf{a}, idx(\mathsf{i}), 1), idx(\mathsf{j}), 2), idx(\mathsf{i}))$$

表示先将索引 i 处的值设为 1，再将索引 j 处的值设为 2，之后访问 a 在索引 i 处的值。假设 i 和 j 都是有效索引，当且仅当 i ≒j 时，该项的值为 2；否则为 1。

我们将类似 $get_{[S;n]}(t_0, idx(t_1))$ 的项美化打印为 $t_0[t_1]$，并省略 $set$ 的类型。示例 9 中的项可以排版为 $set(set(\mathsf{a}, idx(\mathsf{i}), 1), idx(\mathsf{j}), 2)[\mathsf{i}]$。

对于数组索引的读取和写入，我们使用上述函数。下面的规则 copy ArrayIdx 对读取数组 a 在索引 i 处的值并将结果存入 x 进行建模。我们将证明拆分开来，以检查可能导致恐慌的越界错误。类似的 move 版本在此省略。

**清单 2**。一个计算 a 和 b 的乘积并将其存入 n 的程序。由于我们使用带值的 **break**，结果也就是循环的值。

let mut n = 0; let old b = b; 循环 { if b == 0 { break n; } n += a; b −= 1; }$\Gamma, \mathcal{U}(\mathsf{i} < 0 \lor \mathsf{i} \geq n) \Longrightarrow \mathcal{U}\langle\pi \text{ panic!() } \omega\rangle\phi, \Delta$

copyArrayIdx $\Gamma, \mathcal{U}(0 \leq \mathsf{i} \land < n \Longrightarrow \mathcal{U}\{ = [\,]\}\langle\pi \quad \omega\rangle\phi \ \Delta\, \mathsf{i})$

$\Gamma \Longrightarrow \mathcal{U}\langle\pi = [\,]\ \omega\rangle\phi \ \Delta$

x : a i () , x a i , x : $S$, a : $[S;\ n]$, $S$ 是 Copy

---

类似地，规则 writeArrayIdxCopy 处理将值 x 写入数组 a 在索引 i 处的操作。我们添加一个更新来改变 a 的值。

$$\text{writeArrayIdxCopy}\ \frac{\begin{array}{l}\Gamma, \mathcal{U}(\mathsf{i} < 0 \lor \mathsf{i} \geq n) \Longrightarrow \mathcal{U}\langle\pi \text{ panic!() } \omega\rangle\phi, \Delta \\ \Gamma, \mathcal{U}(0 \leq \mathsf{i} \land \mathsf{i} < n) \Longrightarrow \mathcal{U}\{\mathsf{a} := set(\mathsf{a}, idx(\mathsf{i}), se)\}\langle\pi\ ()\ \omega\rangle\phi, \Delta\end{array}}{\begin{array}{c}\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \mathsf{a}[\mathsf{i}] = se\ \omega\rangle\phi, \Delta \\ \mathsf{a} : [S; n],\ S \text{ is Copy}\end{array}}$$

数组字段的引用遵循与第 3.5 节相同的总体方法，但更为复杂，因为无效的数组边界可能导致恐慌，因此必须进行检查。详情见附录 A.2。

### 3.7 循环不变量与循环作用域

函数调用和循环会引入潜在的无界计算，因此，演绎式程序验证需要某种形式的归纳原理才能达到相对完备性。我们的方法旨在全面处理函数调用和循环，其对前者的处理主要遵循 [1]。由于篇幅所限，我们仅在附录 B 中讨论这一点。

相反，我们将较为详细地讨论循环：虽然循环不变量是一个标准概念，但对于可能因**break** 或恐慌而终止的循环，其规则会变得极其复杂。我们采用相对较新的循环作用域 [31] 概念，给出能够处理非标准控制流的简洁不变式规则。（为简便起见，我们不考虑标签；不过，本文提出的方法也能处理标签。）

考虑清单 2 中的程序，其中 a、b、old_b 是类型为 u64的程序变量。循环是一个表达式，并且具有一个值：最终一次迭代中 n 的值。为了安全地近似未知的循环迭代次数，我们使用循环不变式：一个 RustyDL 公式$inv$，它在首次迭代之前以及之后的每次迭代之后都成立。忽略溢出的情况下，要证明清单 2 中循环的功能正确性，一个足够强的循环不变式$inv$ 就是该公式（无需用 $0 \leq \mathsf{b}$强化不变式，因为这可由 b的类型推知）：

$$\mathsf{n} \doteq \mathsf{a} \cdot (\mathsf{old_b} - \mathsf{b}) \land \mathsf{b} \leq \mathsf{old_b} \tag{1}$$

Rust 编译器会将所有循环转换为清单 2 中那样的无条件循环，因此霍尔 [18], 迪杰斯特拉 [13], 或 JavaDL [1],的不变式规则并不适用：它们都会根据守卫条件的值进行分类讨论。这里无法做到这一点，因为我们无法提取循环守卫。为了解决这个问题，并支持重定向控制流的语句（例如 **break** 和 **return**），我们使用一种辅助控制语句：带有索引变量 循环作用域 $\circlearrowleft_{\mathsf{x}} p \circlearrowleft$，其索引变量 $\mathsf{x}$ 和主体 $p$ 是一个 Rust 表达式，其中 $\mathsf{x}$ 是一个类型为 布尔类型的新程序变量。

直观地说，循环作用域表示一次通用的循环迭代，并将退出循环的原因记录在索引变量x中。例如，当通过**break**退出循环时，$\mathsf{x}$ 被设置为真。当循环并非以这种方式退出时，我们将$\mathsf{x}$ 设置为假。与循环本身一样，RustyDL中的循环作用域是表达式，并且可以具有一个值，即其所表示的循环的值。我们在循环不变式规则中使用循环作用域，以符号执行一次通用的循环迭代，并记录其终止方式。规则 loopScopeInvBox 如下匹配方框模态中的循环。它将证明分成两个分支。

$$\text{loopScopeInvBox}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \mathcal{U} inv, \Delta \\ \Gamma \Longrightarrow \mathcal{U}\mathcal{V}\Big(inv \rightarrow [\circlearrowleft_{\mathsf{x}}\, body;\ \mathsf{continue};\circlearrowleft]\big( \\ \qquad (\mathsf{x} \doteq TRUE \rightarrow [\pi\omega]\phi) \wedge (\mathsf{x} \doteq FALSE \rightarrow inv)\big)\Big), \Delta\end{array}}{\Gamma \Longrightarrow \mathcal{U}[\pi\ \mathsf{loop}\ body\ \omega]\phi, \Delta}$$

第一个分支确保进入循环时$inv$ 成立。第二个分支将循环体放入带有新索引变量x的循环作用域中，并在假定不变式成立的前提下对循环体进行一次符号执行。然后读取索引变量x的值，以确定循环是退出（$\mathsf{x} \doteq TRUE$），还是执行进一步的迭代（$\mathsf{x} \doteq FALSE$）。在第一种情况下，我们必须证明后置条件 $\phi$ 成立；在第二种情况下，则必须证明$inv$ 仍然有效。

该规则执行一个通用的循环迭代，其中 $\Gamma \cup \Delta$ 中出现的程序变量值未知。因此，$\Gamma \cup \Delta$ 中此类变量的值必须被“遗忘”。这是通过所谓的匿名化更新$\mathcal{V}$ [1] 实现的：为$body$中写入的任何变量赋予一个新的未知值。

索引变量 $\mathsf{x}$ 在对循环作用域进行符号执行时，由处理 **break**、**continue**、**return**等的规则设定。我们通过规则 breakValue和 continue 以及一个尾随程序 $p$ 来说明这一机制；另见 [31]。注意，breakValue 返回循环作用域的值$se$。

$$\text{breakValue}\ \frac{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \{\mathsf{x} = \mathsf{true};\ se\}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \circlearrowleft_{\mathsf{x}} \mathsf{break}\ se;\ p \circlearrowleft\ \omega\rangle\phi, \Delta} \qquad \text{continue}\ \frac{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \mathsf{x} = \mathsf{false}\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle\pi\ \circlearrowleft_{\mathsf{x}} \mathsf{continue};\ p \circlearrowleft\ \omega\rangle\phi, \Delta}$$

示例 *10*。考虑清单 2 中的程序。令 $\phi \equiv \mathsf{n} \doteq \mathsf{a} \cdot \mathsf{old_b}$，并令 $inv$如式 (1) 所示。在对赋值进行符号执行并化简后，我们得到

$$\Longrightarrow \{\mathsf{n} := 0 \,\|\, \mathsf{old_b} := \mathsf{b}\}\, [\mathsf{loop}\ \{\ \ldots\ \}]\phi.$$

应用 loopScopeInvBox 得到两个序列：首先是“初始时有效”的情况：

$$\Longrightarrow \{\mathsf{n} := 0 \,\|\, \mathsf{old_b} := \mathsf{b}\}\, (\mathsf{n} \doteq \mathsf{a} \cdot (\mathsf{old_b} - \mathsf{b}) \wedge \mathsf{b} \leq \mathsf{old_b})$$

化简后为 $\Longrightarrow 0 \doteq \mathsf{a} \cdot (\mathsf{b} - \mathsf{b}) \land \mathsf{b} \leq \mathsf{b}$。整数化简可以证明这一分支。
loopScopeInvBox 的第二个分支得到：

$$\Longrightarrow \{\mathsf{n} := 0 \parallel \mathsf{old_b} := \mathsf{b}\}\,\{\mathsf{n} := c_n \parallel \mathsf{b} := c_b\}\,\Big(inv \to [\circlearrowleft_{\mathsf{x}}\ \mathsf{if\ b == 0}\ \ldots \circlearrowleft] \underbrace{\big((\mathsf{x} \doteq \mathit{TRUE} \to [\,]\phi) \land (\mathsf{x} \doteq \mathit{FALSE} \to inv)\big)}_{\psi}\Big) \tag{2}$$

对（2）中的 if 进行化简和符号执行后，会得到两个分支（3）、（4），其中 $u \equiv$ old_b :=b $\parallel$n := $c_n \parallel$b := $c_b$and $inv' \equiv \{u\}\, inv$：

$$inv', c_b \doteq 0 \Longrightarrow \{u\}\,[\circlearrowleft_{\mathsf{x}}\ \mathsf{break\ n;\ n+=a;\ b-=1;\ continue;}\ \circlearrowleft]\psi \tag{3}$$

$$inv', c_b \neq 0 \Longrightarrow \{u\}\,[\circlearrowleft_{\mathsf{x}}\ \mathsf{n+=a;\ b-=1;\ continue;}\ \circlearrowleft]\psi \tag{4}$$

对（3）应用 breakValue 会丢弃剩余的循环体和循环作用域：

$$inv', c_b \doteq 0 \Longrightarrow \{u\}\,[\{\mathsf{x = true;\ n}\}]\psi \tag{5}$$

因此，我们知道在此分支中 x 为真，从而可以将序列式化简为

$$inv', c_b \doteq 0 \Longrightarrow \{u\}\,(\mathsf{n} \doteq \mathsf{a} \cdot \mathsf{old_b}) \tag{6}$$

应用更新后得到（7），通过算术化简即可轻松证明。

$$c_n \doteq \mathsf{a} \cdot (\mathsf{b} - c_b) \land c_b \leq \mathsf{b}, c_b \doteq 0 \Longrightarrow c_n \doteq \mathsf{a} \cdot \mathsf{b} \tag{7}$$

（4）的证明步骤与此类似。

### 3.8 演算规则的实现

我们基于 KeY 工具实现了 RustyDL 及其演算，[1] 并利用了该系统的通用数据结构和规则匹配算法。该证明系统并非基础性的；我们依赖整数、序列等理论。

所有规则均使用一种领域特定语言实现，该语言用于在一种称为“taclets”的序列演算中编写示意性的程序逻辑规则 [7]。这些规则可以独立于系统其余部分进行审查、检查，甚至在一定程度上通过机械方式证明。此外，也有人致力于证明这些规则相对于程序语义是可靠的，并验证这些变换规则 [3,33]。

KeY 系统久经时间考验。因此，我们认为，其核心值得信赖，且实现忠实地体现了上述演算规则。目前，该实现仍是一个原型，能够通过手动和自动方式证明若干示例，展示了多种 Rust 特性，尤其是借用、循环、数组、元组以及（部分）枚举。最具启发性的是对二分查找的验证：该证明包含 4260 次规则应用，用时 2.1 秒。

[1] 请参阅实现和示例：https://github.com/Drodt/key/tree/rusty。

## 4 相关工作

我们的演绎验证方法受到 KeY [1]的启发。近年来，另有多种工具被开发出来，用于验证 Rust 程序。*Creusot* [12]会将安全 Rust 程序转换为中间语言和验证工具 *Why3* [15]，后者生成验证条件，再由 SMT 求解器予以证明。可变引用通过预言 [20], 建模：引用被丢弃后，实际上会将其值写回出借方。*Prusti* 工具 [4] 与 Creusot 类似，但基于分离逻辑，并会转换为 *Viper* [26]。*Aeneas* [17] 会将安全 Rust 转换为多个函数式后端，从而完全消除可变引用。所有这些工具都依赖于向中间语言的转换，且不允许交互；而我们的动态逻辑则支持 HIL 交互模式。

*Verus* [22] 基于验证条件生成和 SMT 求解，支持对不安全 Rust 进行验证。目前尚不清楚如何将 RustyDL 扩展到不安全 Rust； [6] 所采用的“混合”方法是一个有力的竞争方案。

对引用进行逻辑建模的另一种方案是使用权限 [10]，以处理对内存位置的读取权限和读写权限。KeY 工具在 [19], 中实现了权限机制，但我们在此选择了可变更新，因为它们开销较低，而且可以轻松扩展现有的更新机制。

## 5 结论与未来工作

我们介绍了 RustyDL——一种适用于 Rust 子集的动态逻辑，以及一种用于证明 RustyDL 公式有效性的顺序演算。我们提出了以合理复杂度对 Rust 关键特性进行逻辑建模的方案。这表明，KeY 用于 HIL 程序验证的方法可以应用于 Rust。

此处介绍的 RustyDL 演算已在 KeY 验证工具之上实现，作为一项概念验证。我们利用该实现成功验证了多个 Rust 函数，包括包含循环和引用的函数；然而，要进行更全面的评估，还需等待自动化程度足够高。

因此，未来工作可分为两个方向：首先，扩展 Rust 子集，纳入特质、迭代器、模式匹配和泛型函数。这项工作耗时，但相对直接。还应考虑验证不安全 Rust，不过必须谨慎权衡验证的范围。不安全 Rust 很可能需要比本文所介绍的更明确的内存模型；但我们仍希望该模型尽可能精简，并在可行时参考安全版本。对此还需进行全面研究。其次，我们正在改进实现：完善类似于 JML [23] 的规约语言，并实现一个图形用户界面。

## 引用

1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.（编）：《演绎式软件验证——KeY手册：从理论到实践》。计算机科学讲义第 10001 卷，Springer（2016）。https://doi.org/10.1007/978-3-319-49812-6。Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Ulbrich, M.：《动态逻辑的多种用途》。载于：Ernst, G., Güdemann, M., Knapp, A., Nafz, F., Ortmeier, F., Ponsar, H., Schellhorn, G., Schiendorfer, A.（编）《探寻 Bug 所在之处——献给 Wolfgang Reif 65 岁生日的论文集》。计算机科学讲义，第 15765 卷，第 56–82 页。Springer, Cham（2025 年 6 月）。https://doi.org/10.1007/978-3-031-92196-4 4 3 。Ahrendt, W., Roth, A., Sasse, R.：《针对重写语义自动验证 Java 验证变换规则》。载于：Sutcliffe, G., Voronkov, A.（编）《编程、人工智能与推理逻辑》。第 412–426 页。Springer, 柏林、海德堡（ 2005 10 1007 11591191 _ 29 ）。https://doi.org/。 4 /。Astrauskas, V., Bílý, A., Fiala, J., Grannan, Z., Matheja, C., Müller, P., Poli, F., Summers, A.J.：《Prusti项目：Rust 的形式化验证》。载于：Deshmukh, J.V., Havelund, K., Perez, I.（编）《NASA 形式化方法——第 14 届国际研讨会，NFM 2022，2022 年 5 月 24–27 日，美国加利福尼亚州帕萨迪纳，论文集》。第 88–108 页。计算机科学讲义第 13260 卷，Springer（2022）。https://doi.org/10.1007/978-3-031-06773-0 5 5 。Astrauskas, V., Matheja, C., Poli, F., Müller, P., Summers, A.J.：《程序员如何使用不安全 Rust？》ACM 程序设计语言论文集。4(OOPSLA)，136:1–136:27（2020）。https://doi.org/10.1145/3428204。Ayoun, S.É., Denis, X., Maksimović, P., Gardner, P.：《一种半自动 Rust 验证的混合方法》。arXiv 预印本 arXiv:2403.15122（2024）。7. Beckert, B., Bubel, R., Drodt, D., Hähnle, R., Lanzinger, F., Pfeifer, W., Ulbrich, M., Weigl, A.：《Java 验证工具 KeY：教程》。载于：Platzer, A., Rozier, K.Y., Pradella, M., Rossi, M.（编）《形式化方法》。第 597–623 页。计算机科学讲义第 14934 卷，Springer Nature Switzerland，Cham（2025）。https://doi.org/10.1007/978-3-031-71177-0 32 8 。Beckert, B., Sanders, P., Ulbrich, M., Wiesler, J., Witt, S.：《形式化验证高效排序器》。载于：Finkbeiner, B., Kovács, L.（编）《系统构造与分析工具和算法——第 30 届国际会议，TACAS 2024，作为欧洲软件理论与实践联合会议 ETAPS 2024 的一部分举行，2024 年 4 月 6–11 日，卢森堡市，卢森堡，论文集，第一部分》。第 268–287 页。计算机科学讲义第 14570 卷，Springer（ 2024 10 1007 978 3 031 57246 3 _ 15 ）。https://doi.org/。/----。Blanc, A.L., Lam, P.：《Rust 验证领域综述》。CoRR **abs/241001981 1001981** 2024 10 48550 2410.（）。https://doi.org/。/ARXIV.。。Boyland, J.：《分数权限》。载于：Clarke, D., Noble, J., Wrigstad, T.（编）《面向对象编程中的别名：类型、分析与验证》，计算机科学讲义，第 7850 卷，第 270–288 页。Springer（2013）。https://doi.org/10.1007/978-3-642-36946-9 10 9 11 。De Gouw, S., Rot, J., De Boer, F.S., Bubel, R., Hähnle, R.：《OpenJDK 的 java.utils.Collection.sort() 存在缺陷：最好、最坏与最糟的情况》。载于：Kroening, D., Pasareanu, C.（编）《第 27 届计算机辅助验证国际会议（CAV）论文集》，旧金山。计算机科学讲义，第 9206 卷，第 273–289 页。Springer（2015 年 7 月）。12. Denis, X., Jourdan, J.H., Marché, C.：《Creusot：用于 Rust 程序演绎式验证的铸造厂》。载于：Riesco, A., Zhang, M.（编）《ICFEM 2022，论文集——

论文集。第 90–105 页。计算机科学讲义第 13478 卷，Springer（2022）。https://doi.org/10.1007/ 978-3-031-17244-1 613 ¯. Dijkstra, E.W., Scholten, C.S.：《谓词演算与程序语义》。计算机科学专著，Springer，纽约州纽约市，第 1 版（1990）。https: //doi.org/10 5 14 1007 978 1 4612 3228 . / - - - - . Din, C.C., Hähnle, R., Henrio, L., Johnsen, E.B., Pun, V.K.I., Tapia Tarifa, S.L.：并发编程语言的局部抽象、全局具体语义。《编程语言与系统汇刊》**46**(1), 3:1–3:58（2024 年 3 月 3648439 15 10 1145 3）。https://doi.org/ . / . Filliâtre, J.C., Paskevich, A.：Why——程序与证明器交汇之处。载于：Felleisen, M., Gardner, P.（编），第 22 届欧洲编程研讨会论文集。计算机科学讲义，第 7792 卷，第 125–128 页。Springer（2013 年 3 月）16. Hähnle, R.：形式规约语言中的多值逻辑、部分性与抽象。《IGPL逻辑期刊》**13**(4), 415–433（2005）17. Ho, S., Protzenko, J.：Aeneas：通过函数式翻译验证 Rust。《ACM程序设计论丛》**6**3547647 18 711 741 2022 10 1145（ICFP），–（）。https://doi.org/ . / . Hoare, C.A.R.：计算机程序设计的公理基础。《ACM通讯》**12**363259 19 10 576 580 1969 10 1145 363235（），–（1969 年 10 月）。https://doi.org/ . / . . Huisman, M., Mostowski, W.：用于并发推理的权限记账符号化方法。载于：Grosu, D., Jin, H., Papadopoulos, G.（编），第 14 届并行与分布式计算国际研讨会，ISPDC 2015，塞浦路斯利马索尔，2015 年 6 月 29 日至 7 月 2 日。第 165–174 页。IEEE 计算机学会（2015）。https://doi.org/10 26 20 1109 2015 . /ISPDC. . . Jung, R., Lepigre, R., Parthasarathy, G., Rapoport, M., Timany, A., Dreyer, D., Jacobs, B.：未来属于我们：分离逻辑中的预言变量。《ACM程序设计论丛》**4**3371113 21 45 1 45 32 2020 10 1145（POPL），: – :（）。https://doi.org/ . / . Klabnik, S., Nichols, C.：《Rust 编程语言》。No Starch Press，旧金山，加利福尼亚州，第 2 版（2022 年 12 月），https://doc.rust-lang.org/stable/book/ 22. Lattuada, A., Hance, T., Cho, C., Brun, M., Subasinghe, I., Zhou, Y., Howell, J., Parno, B., Hawblitzel, C.：Verus：使用线性幽灵类型验证 Rust 程序。《ACM程序设计论丛》**7**(OOPSLA1), 286–315（2023）23. Leavens, G.T., Poll, E., Clifton, C., Cheon, Y., Ruby, C., Cok, D., Müller, P., Kiniry, J., Chalin, P., Zimmerman, D.M., Dietl, W.：《JML参考手册》（2013 年 5 月），http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf 24. Li, H., Guo, L., Yang, Y., Wang, S., Xu, M.：Rust-for-Linux 的实证研究：成功、不满与妥协。载于：Bagchi, S., Zhang, Y.（编），2024 年 USENIX年度技术会议论文集，USENIX ATC 2024，美国加利福尼亚州圣克拉拉，2024 年 7 月 10–12 日。第 425–443 页。USENIX协会（2024），https://www.usenix.org/conference/atc24/presentation/li-hongyu 25. McCarthy, J.：迈向计算的数学科学。载于：信息处理：1962 年第二届 IFIP大会论文集，德国慕尼黑，1962 年 8 月 27 日至 9 月 1 日。第 21–28 页。North-Holland（1962）26. Müller, P., Schwerhoff, M., Summers, A.J.：Viper：用于基于权限的推理的验证基础设施。载于：Jobstmann, B., Leino, K.R.M.（编），验证、模型检测与抽象解释。第 41–62 页。Springer Berlin Heidelberg，柏林，海德堡（2016）27. Pratt, V.R.：关于Floyd-Hoare逻辑的语义考量。载于：第 17 届计算机科学基础年度研讨会，美国得克萨斯州休斯顿。第 109–121 页。IEEE，洛斯阿拉米托斯，加利福尼亚州（1976）。https://doi.org/10.1109/SFCS.1976.27

28. Rust Community：Rust 参考手册。参考资料，Rust 基金会（2024），https: // doc.rust-lang.org/stable/reference/introduction.html 29. Rust Community：Rust 编译器开发指南（2025），https:// rustc-dev-guide.rust-lang.org/ 30. Schellhorn, G., Bodenmüller, S., Bitterlich, M., Reif, W.：使用 KIV 进行软件与系统验证。载于：Ahrendt, W., Beckert, B., Bubel, R., Johnsen, E.B.（编），《软件逻辑》。《形式化方法精选》——献给 Reiner Hähnle 六十岁生日的论文集。计算机科学讲义，第 13360 卷，第 408 436 2022 10 1007 978 3 031 08166 8_20 – 页。31 Springer（）。https://doi.org/ . / - - - - . Steinhöfel, D., Wasser, N.：一种用于分析含非标准控制流循环的新不变式规则。载于：Polikarpova, N., Schneider, S.A.（编），集成形式化方法——第 13 届国际会议，IFM 2017，意大利都灵，2017 年 9 月 20–22 日，会议论文集。第 279–294 页。计算机科学讲义第 10510 卷，Springer（2017）。https://doi.org/ 10 1007 978 3 319 66845 1_18 . / - - - - . Thompson, C.：Rust 如何从 32 一个业余项目发展为全球最受喜爱的编程语言。《麻省理工科技评论》**126(2)**（2023 年 2 月），https://www.technologyreview.com/2023/02/14/1067869/ 33. Trentelman, K.：使用 Bali 证明 JavaCard DL taclet 的正确性。载于：第三届 IEEE 软件工程与形式化方法国际会议论文集。第 160–169 页。SEFM '05，IEEE 计算机学会，美国（2005 年 9 月）。https://doi.org/10.1109/SEFM.2005.37

# A 演算规则

## A.1 更新简化规则

参见表 1。

## A.2 数组引用规则

考虑规则 ruleborrowMutIdxArr：我们希望获取由 a 引用的数组中索引 i 处的可变引用。同样，我们会执行边界检查。我们将 x 更新为位置 a[i] 的可变引用。

$$\text{borrowMutIdxArr}\quad \frac{\Gamma, \mathcal{U}(\mathsf{i} < 0 \lor \mathsf{i} \geq n) \Longrightarrow \mathcal{U}\langle \pi\ \mathsf{panic!}()\ \omega\rangle\phi, \Delta \quad \Gamma, \mathcal{U}(0 \leq \mathsf{i} < n) \Longrightarrow \mathcal{U}\{\mathsf{x} := \mathit{mref}(\llbracket \mathsf{a}, \mathsf{i} \rrbracket)\}\ \langle \pi\ ()\ \omega\rangle\phi, \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \mathsf{x} = \&\mathsf{mut}\ \mathsf{a}[\mathsf{i}]\ \omega\rangle\phi, \Delta}$$
$$\mathsf{a} : \mathrm{Ref}^m_{[A;n]}, \mathsf{x} : \mathrm{Ref}^m_A, \mathsf{i} : \mathit{int}$$

示例 *11.* 设 a : [整数; 4] , x : $\mathrm{Ref}^n_{[int;4]}$, y : $\mathrm{Ref}^m_{int}$ 是程序变量。我们从以下序列式开始

$$\Longrightarrow \langle \mathsf{x} = \&\mathsf{mut}\ \mathsf{a};\ \mathsf{y} = \&\mathsf{mut}\ \mathsf{x}[0];\ {*}\mathsf{y} = 8;\rangle\phi\ .$$

对程序片段进行完整符号执行得到序列式

$$\Longrightarrow \{\mathsf{x} := \underbrace{\mathit{mref}(\llbracket \mathsf{a} \rrbracket)}_{r_1}\}\ \{\mathsf{y} := \mathit{mref}(\llbracket \mathsf{x}, 0 \rrbracket)\}\ \{\mathsf{y} \mathrel{*\!\!\to} 8\}\ \phi\ .$$

$$
\begin{aligned}
\{u_1 \parallel \mathsf{a} := t_1 \parallel u_2 \parallel \mathsf{a} := t_2 \parallel u_3\}\, t &\rightsquigarrow \{u_1 \parallel u_2 \parallel \mathsf{a} := t_2 \parallel u_3\}\, t \\
&\quad \text{where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd} \\
\{u_1 \parallel \mathsf{a} := t' \parallel u_2\}\, t &\rightsquigarrow \{u_1 \parallel u_2\}\, t \\
&\quad \text{where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd}, \mathsf{a} \notin \mathit{fpv}(t) \\
\{u_1\}\{u_2\}\, t &\rightsquigarrow \{u_1 \parallel \{u_1\}\, u_2\}\, t \\
&\quad \text{where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd} \\
\{u \parallel \mathtt{skip}\}\, t &\rightsquigarrow \{u\}\, t \text{ where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd} \\
\{\mathtt{skip} \parallel u\}\, t &\rightsquigarrow \{u\}\, t \text{ where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd} \\
\{\mathtt{skip}\}\, t &\rightsquigarrow t \text{ where } t \in \mathrm{Trm}_A \cup \mathrm{Fml} \cup \mathrm{Upd} \\
\{u\}\, v &\rightsquigarrow v \text{ where } v \in \mathrm{VSym} \\
\{u\}\, f(t_1, \ldots, t_n) &\rightsquigarrow f(\{u\}\, t_1, \ldots, \{u\}\, t_n) \text{ where } f \in \mathrm{FSym} \cup \mathrm{PSym} \\
\{u\}\, (\text{if } \phi \text{ then } t_1 \text{ else } t_2) &\rightsquigarrow \text{if } \{u\}\, \phi \text{ then } \{u\}\, t_1 \text{ else } \{u\}\, t_2 \\
\{u\}\, \neg\phi &\rightsquigarrow \neg\{u\}\, \phi \\
\{u\}\, (\phi \bullet \psi) &\rightsquigarrow \{u\}\, \phi \bullet \{u\}\, \psi \text{ where } \bullet \in \{\wedge, \vee, \rightarrow, \leftrightarrow\} \\
\{u\}\, \mathcal{Q}\ A\ v;\ \phi &\rightsquigarrow \mathcal{Q}\ A\ v;\ \{u\}\, \phi \text{ where } \mathcal{Q} \in \{\forall, \exists\}, v \notin \mathit{fv}(u) \\
\{u\}\, (\mathsf{a} := t) &\rightsquigarrow \mathsf{a} := \{u\}\, t \\
\{u\}\, (t_1 \mathrel{*\!\!\rightarrow} t_2) &\rightsquigarrow \{u\}\, t_1 \mathrel{*\!\!\rightarrow} \{u\}\, t_2 \\
\{u\}\, (u_1 \parallel u_2) &\rightsquigarrow \{u\}\, u_1 \parallel \{u\}\, u_2 \\
\mathit{mref}(\llbracket \mathsf{a} \rrbracket) \mathrel{*\!\!\rightarrow} t &\rightsquigarrow \mathsf{a} := t \\
\mathit{mref}(\llbracket \mathit{mref}(\llbracket \mathsf{a} \rrbracket), t_1 \rrbracket) \mathrel{*\!\!\rightarrow} t_2 &\rightsquigarrow \mathsf{a} := \mathit{set}(\mathsf{a}, t_1, t_2) \\
\{\mathsf{a} := t\}\, \mathsf{a} &\rightsquigarrow t
\end{aligned}
$$

**表 1**。更新的化简规则

化简前两个更新后，我们得到如下序列式。请注意，我们在 y 的值中使用了存储于 x 中的可变引用。

$$\Longrightarrow \{\underbrace{\mathsf{x} := r_1 \parallel \mathsf{y} := \mathit{mref}(\llbracket r_1, 0 \rrbracket)}_{u_1}\}\, \{\mathsf{y} \mathrel{*\!\!\rightarrow} 8\}\, \phi$$

合并这些更新后，得到以下序列式：

$$\Longrightarrow \{u_1 \parallel \mathit{mref}(\llbracket r_1, 0 \rrbracket) \mathrel{*\!\!\rightarrow} 8\}\, \phi$$

接下来，我们使用 mutatingArrToElementary 来处理变异更新：

$$\Longrightarrow \{u_1 \parallel \mathsf{a} := \mathit{set}(\mathsf{a}, \mathit{idx}(0), 8)\}\, \phi \ .$$

## B 应用函数契约

考虑函数调用 $\mathsf{r} = \mathsf{f}(\mathsf{p}_1, .., \mathsf{p}_n)$，其中 f 是一个函数，其参数为 $\mathsf{a}_1$: $\mathsf{T}_1$, ..., $\mathsf{a}_n$: $\mathsf{T}_n$ ; $\mathsf{p}_1, \ldots, \mathsf{p}_n, \mathsf{r}$ 是类型兼容的程序变量。在 RustyDL 中，处理此类调用有两种方式。

首先，我们可以“内联”函数 f 的函数体，为参数创建局部变量，在语法上将返回表达式替换为对 r 的赋值，等等。这种方法无法处理无界递归，会导致对同一 Rust 代码的重复符号执行，因此常常造成状态爆炸。

$$\text{useFnContract}\ \frac{\Gamma \Longrightarrow \mathcal{U}\mathcal{V}(pre \wedge paramsInRange \wedge trm \prec \mathsf{mby}), \Delta \quad \Gamma \Longrightarrow \mathcal{U}\mathcal{V}\mathcal{W}(post \wedge inRange_{\mathsf{T}}(\mathsf{res}) \rightarrow \langle \pi\ \mathsf{r} = \mathsf{res}\ \omega \rangle \phi), \Delta}{\Gamma \Longrightarrow \mathcal{U}\langle \pi\ \mathsf{r} = \mathsf{f}(se_1, \ldots, se_n)\ \omega \rangle \phi, \Delta}$$

位置

– fn f($\mathsf{a}_1$: T1, . . ., $\mathsf{a}_n$: $\mathsf{T}_n$) −> T 是一个函数，– ($pre, post, trm$) 是 f 的契约，– $\mathcal{V} := \{ \mathsf{a}_1 := se_1 \parallel \cdots \parallel \mathsf{a}_n := se_n \} \in$ Upd 初始化参数，– $\mathcal{W} := \{\mathsf{b}_1 \twoheadrightarrow c_1 \parallel \cdots \parallel \mathsf{b}_m \twoheadrightarrow c_m\}$ 是一个匿名化更新，– $\mathsf{b}_1 : \mathsf{Ref}^n_{A_1}, \ldots, \mathsf{b}_m : \mathsf{Ref}^n_{A_m}$ 恰好是 f 中具有可变引用类型的参数，– $c_1, \ldots, c_m$ 是新常量，– 并且若这是递归调用，则 mby 是终止见证。

**图 2.** 函数契约规则

其次，我们可以使用合适的声明式抽象，在不对 f 的函数体进行符号执行的情况下推理其行为。为此，定义 6 引入了函数的契约这一概念。

**定义 6（函数契约）**。设 *Crt* 是一个有效的 *Rust crate*，且fn f($\mathsf{a}_1$: $\mathsf{T}_1, \ldots,$ $\mathsf{a}_n$: $\mathsf{T}_n$) −> T 是 *Crt* 中的一个函数。三元组 (*pre, post, trm*) 是 f 的一个契约，由以下部分组成：*(1)* 公式 *pre*，作为前置条件；*(2)* 公式 *post*，作为后置条件；*(3)* 一个可选项 *trm*，作为终止见证。这些组成部分可以访问 f 的参数 $\mathsf{a}_1 : \mathsf{T}_1, \ldots, \mathsf{a}_n : \mathsf{T}_n$；此外，后置条件还可以访问 f 的返回结果 res: T。

图 2 中的 useFnContract 规则用函数的契约替换函数调用，并假定该契约正确；这一点必须单独证明。

该规则有两个前提：第一个前提说明，在将形式参数更新为实际参数之后，函数的前置条件在当前符号状态中得到满足。除此之外，我们还要证明终止性见证严格递减，其中 $\prec$是$trm$ 类型上的一个良基二元关系。这对于递归调用是必要的：如果$trm$ 在每次递归调用时都严格递减，那么良基性可以确保递归有界，并且调用会终止。

第二个前提允许我们假设函数的后置条件成立，并继续证明。然而，如果我们传递可变引用，被调用的函数可能会更改状态的某些部分。因此，我们添加匿名化更新$\mathcal{W}$，为所有可能被变更的位置赋予新常量。更多说明请参见第 3.7 节。

例 *12.* 考虑递归函数 fn mul(a: u64, b: u64, c: &mut u64)，其契约为 (真 $\doteq$ $*^m$(c) a · b, —)。它计算 a 乘以 b 的结果，并将结果写入 c。

假设目标序列式具有以下形式，其中程序变量为 x、y 的类型为 u64，z 的类型为 &mut u64，以及 r 的类型为 ()：

$$\mathrm{inRange}_{\mathsf{u64}}(\mathsf{x}), \mathrm{inRange}_{\mathsf{u64}}(\mathsf{y}), \mathrm{inRange}_{\mathsf{\&mut\ u64}}(\mathsf{z}) \Longrightarrow \langle \mathsf{r = mul(x, y, z);} \rangle \phi$$

应用 fnUseContract 会得到以下序列式作为第一个前提：

$$\begin{aligned} & \mathrm{inRange}_{\mathsf{u64}}(\mathsf{x}), \mathrm{inRange}_{\mathsf{u64}}(\mathsf{y}), \mathrm{inRange}_{\mathsf{\&mut\ u64}}(\mathsf{z}) \\ \Longrightarrow & \{\underbrace{\mathsf{a := x} \parallel \mathsf{b := y} \parallel \mathsf{c := z}}_{\mathcal{V}}\} \left(\mathrm{true} \wedge \mathit{paramsInRange}\right) \end{aligned}$$

这很容易证明，因为所有参数都在各自类型的取值范围内。第二个序列式稍作简化后如下：

$$\begin{aligned} & \mathrm{inRange}_{\mathsf{u64}}(\mathsf{x}), \mathrm{inRange}_{\mathsf{u64}}(\mathsf{y}), \mathrm{inRange}_{\mathsf{\&mut\ u64}}(\mathsf{z}) \\ \Longrightarrow & \mathcal{V}\{\mathsf{c} \mathrel{*\!\!\rightarrow} d\} \left(*^{m}(\mathsf{c}) \doteq \mathsf{a} \cdot \mathsf{b} \rightarrow \langle \mathsf{r = res;} \rangle \phi\right) \end{aligned}$$

其中， $d$ 是一个类型为 整数的新常量，用于匿名化 c 的出借方。

现在可以在后续证明 $\phi$时，利用 c 包含 a 与 b 的乘积这一事实。