arXiv:2303.14314v1 [cs.PL] 25 Mar 2023

用于指针程序模块化关系式验证的 WhyRel 原型

Ramana Nagasamudram[1], Anindya Banerjee[2], 和 David A. Naumann[1]

[1] 史蒂文斯理工学院，美国霍博肯
rnagasam,dnaumann@stevens.edu[2] IMDEA 软件研究所，西班牙马德里 anindya.banerjee@imdea.org

摘要。 在各种验证场景中，验证程序之间的关系是一项常见任务，例如优化变换、将程序的新版本与旧版本关联起来（回归验证），以及非干扰。然而，现有工具对作用于动态分配的可变状态的程序，尚不能很好地支持关系验证；这些工具虽然自动化程度高，却以限制可处理的程序为代价。另一方面，自动主动式工具需要用户进行更多交互，但能够验证更广泛类别的程序。本文介绍了 WhyRel，这是一种基于关系区域逻辑、用于自动主动式验证指针程序关系性质的工具。本文通过一系列验证案例研究评估 WhyRel，这些研究依赖于由 Why3 平台协调的 SMT 求解器，而 WhyRel 正是构建于该平台之上。案例研究包括证明抽象数据类型的表示独立性、展示非干扰，以及解决近期文献中的挑战性问题。

关键词： 局部推理 · 关系验证 · 自动主动验证 · 数据抽象。

1 引言

关系性质包括程序的条件等价性（如回归验证 [28]）、非干扰（即通过低不可区分关系将程序与其自身关联起来），以及敏感度 [6] 等其他要求。我们关注的问题是，为堆操作程序的关系性质进行模块化验证提供工具支持，其中包括操作不同数据表示的程序，这些表示涉及动态分配的指针结构。

通过使用框架条件进行局部推理、过程抽象（即基于关于程序所调用过程的假设进行推理）以及数据抽象（这要求基于状态的封装），可以对指针程序进行模块化推理。要证明抽象数据类型（ADT）的性质，例如表示独立性，封装至关重要，它使实现能够依赖于关于对客户端隐藏的私有状态的不变式。关系验证也涉及一种组合性，即中间执行步骤的对齐

，这使得可以使用更简单的关系不变式和规约（例如，参见 [29,17,25]）。

我们的目标是实现开发者可用的自动主动验证 [19], ，这也是 Dafny 和 Why3 等工具所倡导的。用户需要提供规范、循环不变量和断言等注释，以及用于关系验证的对齐提示。其核心思想是尽可能减少或消除用户手动调用策略进行证明搜索的需求。

对规约、循环不变量或程序对齐的自动推断有助于自动化验证，并已在一些工具中实现。但在当前最先进的技术中，这些方法仅适用于形式有限的规约和不变式（例如，仅限于线性算术），且很少支持动态分配对象。因此，推断不在本文讨论范围之内。

本文讨论的范围包括采用强封装来隐藏信息，即客户端使用的方法规约不会暴露内部表示细节；同时还包括验证客户端的模块化正确性，即其行为独立于内部表示。对于指针程序而言，在不对数据和控制结构施加过度限制的情况下实现强封装，在技术上颇具挑战。自动主动式工具依赖大量公理化来生成验证条件（验证条件）；为了获得高度保证，这些验证条件应根据程序和规约的定义性操作语义加以论证。

本文介绍 WhyRel，这是一种用于对指针程序的关系性质进行自动主动验证的原型。源程序以命令式语言编写，支持共享可变对象（但不支持子类型）、动态分配和封装。断言语言是一阶语言；为表达关系性质，它还包含一些构造，用于关联两个程序中变量的值和指针结构。WhyRel 基于关系区域逻辑 [1], ，这是区域逻辑 [4,2] 的一种关系扩展。区域逻辑通过使用 动态框架条件 [15]（这些条件描述了作用于堆的命令的足迹）提供了一种灵活的局部推理方法。验证需要显式地推理内存区域及其随计算推进而发生的变化；其灵活性则源于能够在同一一阶框架下表达部分关系和分离等概念。

封装通过一种称为动态边界的动态框架来规定：它是一种足迹，用于描述模块的内部位置。要确保封装，只需保证客户端不会直接修改或更新模块边界内的位置。关于关系逻辑 [1], 有详细的可靠性证明，而我们的原型忠实地实现了该逻辑。

WhyRel 构建于 Why3 平台[3] 之上，用于演绎式程序验证。该平台提供了验证 WhyML 程序的基础设施。WhyML 是 ML 的一个子集， [7] 支持幽灵代码和非确定性选择。其断言语言是一种多态一阶逻辑，并扩展支持代数数据类型以及递归定义和归纳定义的谓词 [11]。Why3 会为 WhyML 生成验证条件，随后可使用种类繁多的

[3] Why3 发行版可在以下网址获取：https://why3.lri.fr/。

定理证明器来证明这些验证条件，包括 Coq 和 Isabelle 等交互式证明助手，以及 Vampire、Alt-Ergo 和 Z3 等一阶定理证明器和 SMT 求解器。

WhyRel 主要用作 Why3 的前端。用户提供程序、规约、注释；对于关系验证，还需提供关系规约，以及使用乘积程序专用语法指定的对齐。WhyRel 将源程序转换为 WhyML，并进行大量编码，以忠实地刻画关系区域逻辑中形式化定义的堆模型和细粒度框架化。与该逻辑相关的验证条件以中间断言和引理的形式引入，供用户证明。验证借助 Why3 提供的功能完成，主要通过 IDE 查看并解除验证条件。

我们通过在 WhyRel 中开展的一系列案例研究评估了这一方法；在这些研究中，我们完全依靠 SMT 求解器来解除证明义务。本文的主要贡献是开发了一种用于堆操作程序关系验证的工具，并将其应用于具有挑战性的案例研究。所形式化的示例展示了关系区域逻辑在对齐、表达堆关系以及利用封装进行关系推理方面的有效性。

结构安排。 第2节以栈抽象数据类型为例，重点介绍如何在 WhyRel 中为程序及其关系性质编写规约。第3节讨论程序对齐的示例。第4节概述 WhyRel 的设计，第5节介绍使用该工具的经验。第6节讨论相关工作，第7节总结全文。

2 WhyRel 概览

程序与规范。 WhyRel 提供了一个轻量级模块系统，用于组织定义、程序和规约。开发内容由接口和实现接口的模块构成。此外，为了进行关系验证，WhyRel 引入了双模块这一概念，稍后将对此进行介绍，用于关联两个（一元）模块中的方法实现。

我们将通过`STACK` 接口（见图 1）介绍 WhyRel 中规约的若干方面。该接口描述了一个由装箱整数构成、支持 push 和 pop 操作的栈。接口首先声明全局变量 对象池 和容量，以及 `Cell` 和 `Stack` 类中客户端可见的字段。变量 对象池 的类型为 `rgn`， 区域 是引用的集合；该变量用于描述那些在概念上由实现栈接口的模块所拥有的对象。容量的类型为 整数 ，表示栈大小的上限。用于装箱整数的 `Cell`类声明了一个字段 `val`，用于存储一个整数。`Stack` 类声明了三个字段：类型为区域的 `rep` 用于跟踪表示该栈的对象；类型为 整数 的`size` 用于存储栈中元素的数量；类型为 整数列表 （数学整数的列表）的幽灵字段 抽象值 用于跟踪栈的抽象表示，并用于规约。实现该接口的模块之后可以细化类定义：例如，一个使用某个

接口 STACK =公开 对象池:rgn /* rgn：引用的集合 */ 公开 容量:int 类 Cell {val:int} 类 Stack {rep:rgn; size:int;幽灵 抽象值:intlist} /* 封装位置 */ 边界 {capacity, pool, pool‘any, pool‘rep‘any}公共不变式 stkPub = ∀ s: Stack ∈ pool. 0 ≤ s.size ≤ capacity∧ (∀t: Stack ∈ pool. s ≠ t ⇒ s.rep ∩ t.rep ⊆ {null}) ∧ ... 方法 Cell(self: Cell) : unit ... 方法 getVal(self: Cell) : int ... 方法 Stack(self: Stack) : unit ensures {self ∈ pool} ... 方法 push(self: Stack, k: int) : unit requires {self ∈ pool ∧self.size < capacity} ensures {self.abs = cons(k,old(self.abs)) ∧ ...} /* 实现允许的堆效应 */ effects {rw {self}‘any, self.rep‘any, alloc; rd self,capacity} 方法 pop(self: Stack) : Cell requires {self ∈ pool ∧self.size > 0} ensures {self.size = old(self.size)-1} ensures {result.val = hd(self.abs) ∧ self.abs = tl(old(self.abs))}

图 1：栈抽象数据类型的 WhyRel 接口

链表实现可能会扩展 栈 类，并添加字段 head 来存储指向该列表的引用。

WhyRel 通过动态模块边界支持模块粒度的堆封装，这些边界描述模块内部的位置。位置可以是变量，也可以是堆位置 $o.f$，其中 o 是对象引用， f 是其字段。在 WhyRel 中，模块边界在接口中指定，并强制要求客户端不得直接读取或写入边界所描述的位置，而只能通过模块方法进行操作。对于我们的栈示例，动态边界为 capacity、pool、pool‘any、pool‘rep‘any；它使用像表达式和 any 数据组来表示。给定一个区域 G 和一个类类型的字段f ，像表达式 G ‘f 表示一个区域，其中包含 G中所有非空引用 o 的 $o.f$ 位置，而 f 是 o的有效字段。如果 f的类型为 region，则 G‘f 表示对集合 G中所有 o对应的引用集 $o.f$ 取并集。对于基本类型的 f ，例如 int 或 intlist， G‘f 为空区域。数据组 any 用于抽象具体字段名：表达式 pool‘any 是 pool‘val、.....、pool‘abs的语法糖。直观地说，图 1 中的动态边界表示，客户端不得直接读取或写入 capacity、pool、pool 中对象的任何字段，以及 pool 中任意 Stack 的 rep 中对象的任何字段。

虽然封装是在模块层面进行规约的，但可以通过模块不变式来规约更细粒度的分离或局部性。栈接口定义了一个公共不变式stkPub ，它断言pool 中所有Stack对象的rep 字段彼此不相交。这个惯用法可用于确保修改一个对象不会影响另一个对象表示中的任何位置。客户端在验证期间可以依赖公共不变式，但实现该接口的模块——

模块 Client =方法 prog（n：整数）：整数 requires { 0 ≤ n < capacity ∧ ... } effects { rw 分配, pool, pool‘any, pool‘rep‘any； rd n, capacity }= var i: 整数 in var c: Cell in var stk: Stack in stk :=new Stack; Stack(stk); while (i < n)do push(stk,i); i:=i+1 done; i := 0; while (i < n) do c:=pop(stk); result:=result+ getVal(c); i:=i+1done; 方法 prog (n: 整数|n: 整数) : (整数|整数) /* prog 的关系规约 */ requires { n ≐ n ∧ Both(0 ≤ n < capacity ∧ ...) } ensures { result ≐ result }

图 2：STACK 的示例客户端及用于等价性的关系规约

接口必须确保这些不变式由模块方法保持。此外，模块可以定义捕获内部状态条件的私有不变式；只要这些不变式仅引用封装位置，即指定的边界 框架，客户端便无需对其进行推理 [14]。

最后，STACK 接口为初始化方法（方法 Cell 和 Stack）定义了规约，并为客户端可见的方法 getVal、push 和 pop 定义了公开规约。注意，栈初始化器确保将 self 添加到边界中（通过后置 self ∈ pool），而栈操作要求 self 属于边界（通过前置 self ∈ pool）。push 和 pop 的规约是标准的，使用“old”表达式来精确捕获字段更新。WhyRel 的断言语言是一阶的，并包括指向断言 $x.f = e$ 以及对区域执行的操作（如子集和成员关系）等构造。除前置条件和后置条件外，每个方法还在效应子句中注有框架条件，用于约束实现的堆效应。允许的效应通过对位置或位置集合进行读写（rw）或读取（rd）来表达，而这些位置或位置集合由区域描述。例如，push 的效应子句规定，实现可以读取或写入 self 的任意字段，以及 self.rep 中任意对象的任意字段。特殊变量 alloc 用于表示 push 可以动态分配对象。

在我们的开发中，我们构建了两个实现图 1 中接口的模块：一个使用数组，ArrayStack ；另一个使用链表，ListStack。两者都依赖于封装状态上的私有不变式，用以描述其指针表示的约束，以及这些表示与抽象值（即栈对象的数学抽象）之间的关系。例如，ListStack的私有不变式指出，对象池中任意栈的链表里的Cell 值，都与抽象值中存储的值相对应。

示例客户端、等价性规范与验证。 现在来看一个示例客户端 prog，如图 2 所示。该程序使用栈计算总和 $\Sigma_{i=0}^{n} i$，尽管实现方式有些迂回。prog 的框架条件提到了 STACK 的边界，不过这没有问题，因为客户端遵循 WhyRel 的封装规则，只通过调用 STACK 接口中声明的方法来修改封装位置。对于这个客户端，我们的目标是证明——

方法 prog（n：整数 ｜ n：整数）：（整数 ｜ 整数）=变量 i：整数 ｜ i：整数 中 变量 c：Cell ｜ c：Cell 中 变量 stk：栈 中 ⌊ stk：=new Stack ⌋；⌊ Stack(stk) ⌋；当 (i ＜ n) ｜ (i ＜ n) 时 ⌊ push(stk,i) ⌋；⌊ i:=i+1 ⌋ 完成；⌊ i:=0 ⌋；当 (i ＜ n) ｜ (i ＜ n) 时 ⌊ c:=pop(stk) ⌋；⌊ result:=result+getVal(c) ⌋；⌊ i:=i+1 ⌋ 完成；

图 3：栈客户端示例的对齐

在再次链接到 栈 的两种实现之一时建立等价性。设 左侧程序为链接到 `ArrayStack`的客户端，右侧 程序为链接到`ListStack`的客户端。等价性通过图 2 所示的关系规约表达。为简洁起见，在描述关系规约时，我们省略框架条件。

此关系规约关联两个版本的`prog`；记法(n:整数 ｜ n:整数)用于声明两个版本都将`n` 作为参数。前置关系要求输入相等：`n` $\doteq$ `n` 表示左侧的`n` 值等于右侧的`n` 值。我们使用 ($\doteq$) 而非 ($=$)，以区分左侧和右侧的值[4]。关系规约要求相关联的两个状态满足客户端的一元前置条件，如`Both(...)`所示。后置关系`result` $\doteq$ `result`断言返回值相等。在 WhyRel 中，关系规约刻画一种对终止性不敏感的性质：从满足前置关系的状态开始执行时，相关联程序的终止执行将得到满足后置关系的状态。

WhyRel 支持两种验证关系性质的方法。第一种方法是将其转化为证明相关程序的函数性质。例如，只要证明`prog` 确实计算出前`n` 个非负整数的和，那么客户端分别链接到两种栈实现时的等价性便显而易见。

然而，对于更复杂的程序和关系性质，这种方法并不适用；它也无法让我们利用相关程序之间的相似性，或通过关系规约进行模块化推理。另一种方法是利用两个程序之间便于推理的对齐来证明关系性质。在 WhyRel 中，对齐通过语法形式双程序 表示，它将两个程序中的关注点配对，以便我们能够协同推理它们的效应。如果所选对齐在充分 的意义上涵盖了相关程序的所有执行对，那么该对齐的关系性质便蕴含底层程序之间的相应关系。

prog 的双程序如图 3 所示。它所捕获的对齐是最大的：客户端一个版本中的每个控制点都与另一版本中的自身配对。构造 $(C|C')$ 将左侧的命令 C 与右侧的命令 C' 配对，而 同步 形式 $\lfloor C \rfloor$ 是 $(C|C)$ 的语法糖；例如，prog 的双程序通过 ⌊stk：= 新建 栈⌋对齐了两次内存分配。此外，该双程序还以同步方式对齐了两个循环，语法为当 e ⧸e，执行 . . . 结束。这种对齐将左侧的一次循环迭代与

[4] 请特别注意，$x \doteq y$ 与 $y \doteq x$并不相同

双模块 REL_STACK (ArrayStack |ListStack) =耦合关系 stackCoupling = ∀ s: 栈 ∈ 对象池 | s: 栈 ∈ 对象池。 s ≐ s ⇒s. 抽象值 ≐ s. 抽象值 ∧ ... 方法 Stack(self: 栈 | self: 栈) : (单元 |单元) 确保 {self ≐ self ∧ ...} =/* Stack 的双程序 */ 方法 push(self: 栈 | self: 栈) : (单元 | 单元) 要求 {self ≐self ∧ ... } 确保 {self.abs ≐ self.abs ∧ ... } =/* push 的双程序 */ 方法 出栈(self:栈 | self:栈) : (Cell |Cell) 要求 {self ≐ self ∧ 两者 (self ∈对象池) ∧ 两者 (self.size > 0)} 确保 {... ∧ result.val ≐ result.val} = /* 出栈的双程序 */

图 4：栈双模块；节选

右侧执行一次循环迭代，并要求循环守卫条件保持一致：此处，左侧的 `i < n` 为真，当且仅当右侧的 `i < n` 为真。在循环体中，使用同步构造对齐栈操作调用，以便通过指明应使用 `push` 和`pop` 的关系规约，实现关系性质的模块化验证。

为了证明图 3 中双程序的规约（见图 2），我们按如下方式推理：内存分配后`stk` 在两侧都被初始化为空栈。第一个锁步对齐循环将整数从 0、.....、`n` 依次压入栈中，并保持如下不变式：两侧的 `i` 相等，且两个栈所代表的数学抽象相等，即 `i ≐ i ∧ stk.abs ≐ stk.abs`。第二个锁步对齐循环将栈中元素出栈并递增 `result` ，其不变式是栈抽象和`result`保持一致，关键的合取项为`result ≐ result`。这足以确立所需的后置关系。值得注意的是，这些循环不变量很容易证明——其中只包含变量之间的相等关系——而且我们不必推理两个栈中的确切内容。

栈的关系规约与验证。 上述推理依赖于我们知道`ArrayStack` 和 `ListStack` 中的方法实现是等价的。我们需要为 push 编写关系规约：对于相关输入，两个栈所表示的内容相同；对于出栈，则说明对于相关输入，返回的 `Cell` 值相同。

图 4 展示了一个双模块，`REL_STACK`，用于关联`STACK`的两种实现。它包含栈操作的关系规约，以及用于验证的双程序。该双模块维护一个耦合 关系，用于关联两个栈实现所采用的数据表示。具体而言，此处的耦合关系指出，对象池中相关联的栈表示相同的抽象值。请注意，关系公式中的量词绑定的是成对变量；而栈耦合关系 中的相等式`s ≐ s` 并非严格的指针相等，而是表示对应关系。严格的指针相等条件过强，因为它不允许将内存分配建模为非确定性操作，也不允许相关联的程序采用不同的分配模式。在幕后，WhyRel 会维护一个部分双射 π ，用于关联所比较的两个状态中已分配的引用。

```
方法 mult(n: 整数, m: 整数) =
  i := 0;
  当 (i < n) 执行j:=0;
    当 (j < m) 执行
      结果 := 结果+1; j :=j+1
    已完成; i: = i+1 已完成;
```

```
方法 mult(n:整数 , m:整数) =
  i : = 0;
  当 (i < n)执行
    结果 : =result+m;
    i: = i+1
  完成;
```

图 5：简单乘法例程的两个版本

关系$x \doteq y$指出，当x 和y 是指针时，左侧状态中的x 与右侧状态中的y 相对于 π 相对应，即 $\pi(x) = y$。

初始化器 栈 的关系规约确保 `self` $\doteq$ `self`，这是 `push` 和 出栈规约所要求的。与其他不变式一样，耦合关系旨在由边界框定，并且相关模块方法必须保持这些关系。封装允许隐藏耦合关系，使客户端无需对其进行推理。

完成栈的开发并验证其客户端两个版本等价性的步骤如下：(i) 在 WhyRel 中构建 栈 接口，其中包含客户端可依赖的公共不变式，以及用于指定封装位置的边界；(ii) 开发两个实现该接口的模块，`ArrayStack`和 `ListStack`，并验证其实现符合 栈 接口规约，同时依赖用于描述封装状态条件的私有不变式；(iii) 提供一个关联这两个栈模块的双模块，并证明栈操作等价，同时依赖用于描述两个模块所用指针结构之间关系的耦合关系；(iv) 根据 栈 中给出的规约验证客户端，并证明其遵守 WhyRel 的封装机制；最后 (v) 为客户端开发一个双模块，并使用栈方法的关系规约验证等价性。

3 对齐模式

精心选择的对齐有助于分解关系验证，从而可以使用简单的关系断言和循环不变量。本节将介绍一些双程序示例，它们所捕捉的对齐并非最大对齐，不同于第 2 节中的 栈 客户端示例。这里我们不形式化双程序的语法，而是展示一些有代表性的示例。在讨论这些示例时，我们会省略框架条件以及其他与对齐无关的内容。

不同的控制结构。 Churchill 等人 [8] 开发了一种技术，通过程序轨迹的状态依赖对齐来证明程序的等价性。他们指出了一个对等价性检查而言颇具挑战的问题，如图 5 所示：该问题比较了两个控制流不同的乘法过程。对于关系验证的自动化方法而言，他们的示例颇具挑战，因为需要将左侧无界数量的循环迭代 m 与右侧单次迭代对齐。

```
过程 mult(n: 整数, m: 整数 | n: 整数, m:整数) : (整数 | 整数) ⫤⌊ i := 0 ⌋;
while (i < n) | (i < n) do invariant { i ≕ i ∧ result ≕ result } ( j := 0;
while (j < m) do result := result+1; j := j+1 done|result := result+m ); assert
{ ⦇result = old(result)+m⦈};⌊ i := i+1 ⌋ done;
```

图 6：图 5 中示例的双程序

```
方法 sumpub (l: 列表) : 整数 =
  p:=l.head; s:=0;
  当 (p ≠ 空值) 执行
    如果 p.pub 则
      s:=s+p.val
    end;
    p:=p.nxt
  完成;
  结果:=s;
```

```
方法 sumpub (l: 列表 |l: 列表) : 整数 =
  ⌊ p:=列表头 ⌋; ⌊ s:=0 ⌋;
  当 (p ≠ 空值) | (p ≠ 空值) 。
    ⦃ ¬ p.pub ⦄ | ⦃ ¬ p.pub ⦄ 做
    ( 如果p.pub 那么 s:=s+p.val 结束;
      p:=p.nxt
    | 如果p.pub 那么 s:=s+p.val 结束;
      p:=p.nxt)
  完成; ⌊ 结果: =s ⌋;
```

图 7：对链表中的公开元素求和：程序与对齐

为证明等价性，我们根据一个关系规约验证图 6 所示的双程序，其前置关系为n ≕ n ∧ m ≕ m，后置关系为结果 ≕ 结果；也就是说，输入一致则输出也一致。与图 3 所示的栈客户端双程序不同，这里体现的对齐并非最大化——实际上，由于控制结构不同，这种对齐无法实现。我们仍通过将外层循环同步对齐，并将左侧内层循环与右侧对结果 的赋值对齐，利用了二者的相似之处。

一个简单的关系循环不变式，断言i 和结果 一致，就足以证明等价性。要证明这是一个不变式，我们需要确认左侧内层循环的效果是将结果 增加m，从而在内层循环结束后保持结果 相等。图 6 中，左侧内层循环后的断言体现了这一点。记号 $\langle\!|P|\!\rangle$（分别为 $|\!\rangle P|\!\langle$）用于说明一元公式 P 在左侧（分别在右侧）状态中成立。

条件对齐循环。 迄今为止的示例都涉及步调一致对齐的循环，要求循环迭代之间一一对应。然而，这一条件往往过于严格。WhyRel 还支持其他循环对齐模式，包括考虑数据值条件的模式。例如，考虑图 7 所示的程序：它遍历一个链表，并计算所有标记为公开的元素之和，标记信息存储在每个元素的公开字段中。该程序满足以下非干扰性质，其关系规范为：

```
方法 sumpub(l: 列表 | l: 列表) : (整数 | 整数)要求 { 两
者(listpub(l,xs)) ∧ xs ≕ xs }确保 { 结果 ≕结果 }
```

这里 listpub(l,xs) 是一个谓词，断言从列表指针 l 可达的公开值序列l 由 xs表示，xs 是一个由整数构成的数学列表。直观而言，该规约刻画了这样一种性质：sumpub 的结果不依赖于输入列表 l 中非公开元素的值。证明程序恰好计算出公开元素之和：结果 = sum(xs)，即可推出所需的非干扰性质。然而，为了展示 WhyRel 对非锁步对齐的支持，我们将通过有条件地对齐 sumpub 的两个副本中的循环来证明非干扰（见图 7）。

对齐方式如下：如果 p 在一侧是非公开节点，就在该侧执行一次循环迭代，同时暂停另一侧的迭代；如果 p 在两侧都是公开的，就同步执行两个循环的迭代。这样，只有当两侧都访问公开节点时，才会递增 s ；关系前置条件保证这些节点的值相同。双程序通过额外的注释来表达这种对齐，这些注释称为 对齐守卫 。它们是一般的关系公式，用于表达触发仅左侧、仅右侧或同步迭代的条件。左侧对齐守卫 {¬ p.pub} 表示，当左侧的 p 不是公开节点时，执行仅左侧的循环迭代。右侧对齐守卫表达了类似的条件，即右侧的 p 不是公开节点。当两个对齐守卫都为假时，即 Both(p.pub) 为真时，迭代便同步进行。

这个双程序将 ∃xs⌊xs。两侧的(listpub(p,xs)) ∧xs≐xs ∧s≐s 作为循环不变量，从而推出所需的后置关系。该不变量指出，p 在两侧都指向同一序列的公开值，该序列由listpub(p,xs) 捕获；此外，双方对迄今计算出的总和s 达成一致。在验证过程中，我们必须证明，对齐后的循环在仅执行左侧、仅执行右侧以及同步迭代时，均能保持该不变量。由于采用了对齐，s的值仅在同步迭代期间更新，因此很容易证明其保持不变。对于单侧迭代，推理依赖于这样一个事实：p 所指向的公开值序列保持不变。

4 编码与设计

我们使用 OCaml 实现 WhyRel，并借助 Why3 提供的库来构造 WhyML 解析树。源程序在转换为 WhyML 之前会先经过解析和类型检查。在转换之前，WhyRel 会执行各种检查和变换：其中首要的一项是检查客户端是否遵守封装，以及用户提供的双程序是否适用。与关系区域逻辑相关的证明义务会以 WhyML 程序中的中间断言和需要用户证明的引理形式生成。本节概述实现中的若干方面，重点介绍向 WhyML 的转换。

编码程序状态。 引用由抽象的 WhyML 类型 引用 表示，并带有一个特殊元素 空值。对引用值唯一支持的操作是相等性；WhyRel 不处理指针算术。区域被编码为幽灵状态，使用一个提供数学集合的库

```
/* 类定义 */
类Cell {
  val: 整数;
  幽灵 rep: rgn;}
类Node {
  curr: Cell;
  nxt: Node; }
/* 全局变量 */
公开 对象池: rgn
```

```
type reftype = Cell |Node (*类名*) type heap =
{mutable val: map reference int; mutable g
host rep : map reference Rgn.t; mutable curr:
map reference reference; mutable nxt: map
reference reference } type state ={mutable
alloct: map reference reftype; mutable heap:
heap; mutable ghost pool:rgn } invariant {¬
(Map.mem null alloct) ∧...} (* G 的公理化‘nxt *)
function img_nxt : state → Rgn.t → Rgn.t
axiom img_nxt_ax : ∀s, r, p. Rgn.mem p (img_
nxt s r) ⇔ ∃ q. s.alloct[q] = Node ∧
Rgn.mem q r∧ p = s.head.nxt[q]
```

图 8：状态编码：左侧为 WhyRel 源代码，右侧为 WhyML 中的编码。

由 Why3 实现。区域上的集合运算是原生支持的，我们还对像表达式进行公理化：对于每个字段 f，WhyRel 会生成一个 Why3 函数符号<style id="2">img_<style id="5">f，并附带一个用于刻画 G<style id="8">‘f 含义的公理。

程序状态使用 WhyML 记录进行编码。图 8 给出了一个示例。`state` 类型至少包含两个名为 `alloct`和 `heap`的可变组件。`alloct` 组件存储从引用到对象类型的映射，并跟踪已分配对象；`heap` 本身是一个记录，其中每个源程序字段对应一个可变组件，用于存储从引用到值的映射。值集合包括引用、Why3 数学类型（如数组和列表）、区域，以及 `int` 和 `bool`等基本类型。此外，`state` 类型为源程序中的每个全局变量都包含一个可变字段，用于存储相应类型的值。`state` 类型带有一个 WhyML 不变式，用于描述良构性。该不变式包含以下条件：`null` 永远不会被分配、没有悬空引用，以及类型约束；例如，`Node` 的 `nxt` 字段本身也是一个 `Node`。

一元程序与效应的翻译。 WhyRel 将一元程序翻译为作用于状态编码的 WhyML 函数。修改堆的命令被建模为对显式状态参数的更新，而局部变量、参数以及特殊的 `result` 变量则使用 WhyML 引用单元进行编码。对象参数使用 `reference` 类型和类型假设进行建模。控制流语句的翻译很直接。对于包含循环的程序，WhyRel 还会在生成的 WhyML 函数中添加 diverges 子句，表示该函数可能不终止，从而避免生成用于证明终止性的验证条件。虽然 Why3 支持对完全正确性进行推理，但我们只关注部分正确性。图 9 展示了一个翻译示例。

鉴于我们对状态的编码，转换框架条件时需要谨慎。例如，图 9 所示方法 `m` 的写入集将包含 rw{c}‘`val`

```
方法 m（c: Cell ，i: 整数) : 整数
  要求 { c.val ≥ 0 }
= 当 (i ≥ 0) 执行
    不变式 { c.val ≥ 0}
    c.val: = c.val+i;
    i := i-1
  完成;
  结果 :=c.val
```

```
let m (s:状态) (c:引用) (i:整数) : 整数
diverges requires { s.alloct[c] =Cell }
requires { s.heap.val[c] ≥ 0 }= let
result = ref 0 in let c = ref cin let i =
ref iin while (!i ≥ 0) do invariant {
s.heap.val[!c] ≥ 0 } (* c.val := c.val +
i *) s.heap.val ←Map.add !c (
s.heap.val[!c]+!i) s.heap.val; i :=!
i-1done;result := s.heap.val[!c]; !result
```

图 9：程序转换示例：左侧为 WhyRel 程序，右侧为 WhyML 转换结果；省略了框架条件。

这是由于对对象 `c` 的字段 `val` 进行了写入和读取。相应地，在 Why3 转换中，`s.heap` 的组件`val` 会被更新；因此，在 Why3 中指定该函数需要添加注解 写入集 `{s.heap.val}` 。然而，这并不是我们想要的粒度，因为它意味着可以写入任意引用的字段 `val` 。因此，WhyRel 会为方法 `m` 生成一个额外的后置条件：`wr_framed_val (old s) s (Rgn.singleton c)`，其中

```
谓词 wr_已框定_val (s: 状态) (t: 状态) (r:rgn) = ∀ p: 引用。s.alloct[p] = Cell ∧ p ∉ r
 ⇒ s.heap.val[p] = t.heap.val[p]
```

有了这个后置条件，`m` （在 WhyML 中）的调用者便可以确信，只有 {`c`} 中引用的 `val` 字段会被修改。

双程序。 WhyRel 将双程序转换为乘积程序；具体而言，它们是作用于一对状态的 WhyML 函数[5]。转换之前，它会执行充分性检查，以确保双程序是良构的。请记住，这里的充分性是指，底层一元程序的所有计算都由其对齐双程序涵盖。充分性可确保，关于双程序的关系判断蕴含底层一元程序之间预期的关系。WhyRel 执行的检查是语法层面的，并通过对双程序进行投影⁀操作来定义。给定一个双程序 CC，左投影 CC （而⁀右投影 CC）提取左侧（相应地，右侧）的一元程序。例如， ⌊`c.f:=g`⌋; (`x:=c.f | skip`) 的左投影是`c.f:=g; x:=c.f` ，其右投影是 `c.f:=g`。对于充分性，给定一元程序 C 和 C' 及其对齐双程序 CC，只需检查⁀ ⁀$CC \equiv C$ 和 $CC \equiv C'$ [1]是否成立。

[5] 实际上，生成的 WhyML 函数作用于一对状态，以及对这些状态中分配的引用进行双射重命名。这是为了适应诸如 $x \doteq y$ 这样的关系公式，其中 x 和 y 是引用。不过，这个额外参数与我们这里的讨论无关，因此我们不提及它。

$\mathcal{B}$⟦C |C'⟧(Γ_l, Γ_r) ˆ= $\mathcal{U}$⟦C⟧(Γ_l); $\mathcal{U}$⟦C'⟧(Γ_r) $\mathcal{B}$⟦⌊m(x |y)⌋⟧(Γ_l, Γ_r) ˆ= $apply(\Phi(\text{m})\ [\Gamma_l.\text{st};\ \Gamma_r.\text{st};$ $\mathcal{E}$⟦x⟧(Γ_l); $\mathcal{E}$⟦y⟧(Γ_r))$\mathcal{B}$⟦⌊C⌋⟧(Γ_l, Γ_r) ˆ= $\mathcal{B}$⟦C |C⟧(Γ_l, Γ_r)$\mathcal{B}$⟦C; C'⟧(Γ_l, Γ_r) ˆ= $\mathcal{B}$⟦C⟧(Γ_l, Γ_r); $\mathcal{B}$⟦C'⟧(Γ_l, Γ_r)$\mathcal{B}$⟦变量 x:T |x:T' 中 CC⟧(Γ_l, Γ_r) ˆ= let x_l = def(T) in let x_r = def(T') in$\mathcal{B}$⟦CC⟧($[\Gamma_l \mid \text{x}: x_l]$, $[\Gamma_r \mid \text{x}: x_r]$)$\mathcal{B}$⟦if E |E' then CC else DD⟧(Γ_l, Γ_r) ˆ= assert $\{\mathcal{E}$⟦E⟧(Γ_l) $= \mathcal{E}$⟦ E'⟧(Γ_r)}; if $\mathcal{E}$⟦E⟧(Γ_l) then $\mathcal{B}$⟦CC⟧ else $\mathcal{B}$⟦DD⟧$\mathcal{B}$⟦while E |E' do DD⟧(Γ_l, Γ_r) ˆ= while $\mathcal{E}$⟦ E⟧(Γ_l) do 不变式 $\{\mathcal{E}$⟦E⟧(Γ_l) $= \mathcal{E}$⟦E'⟧(Γ_r)$\}\mathcal{B}$⟦CC⟧(Γ_l, Γ_r)$\mathcal{B}$⟦while E |E'. $\mathcal{P} \mid \mathcal{P}$ do DD⟧(Γ_l, Γ_r) ˆ=while ($\mathcal{E}$⟦E⟧(Γ_l) $\vee\ \mathcal{E}$⟦E'⟧(Γ_r)) do invariant $\{\mathcal{A}\}^{\swarrow}$if ($\mathcal{E}$⟦E⟧($\Gamma_l$) $\wedge\ \mathcal{F}$⟦$\mathcal{P}$⟧(Γ_l, Γ_r)) then $\mathcal{U}$⟦CC⟧(Γ_l) else if ($\mathcal{E}$⟦E'⟧(Γ_r) $\wedge\ \mathcal{F}$⟦$\mathcal{P}'$⟧(Γ_l, Γ_r)) then $\mathcal{U}$⟦$^{\rightharpoonup}$CC⟧(Γ_r) else $\mathcal{B}$⟦CC⟧(Γ_l, Γ_r) where $\mathcal{A} \equiv (\mathcal{E}$⟦E⟧($\Gamma_l$) $\wedge\ \mathcal{F}$⟦$\mathcal{P}$⟧(Γ_l, Γ_r)) $\vee\ (\mathcal{E}$⟦E'⟧(Γ_r) $\wedge\ \mathcal{F}$⟦$\mathcal{P}'$⟧(Γ_l, Γ_r)) $\vee(\neg\mathcal{E}$⟦E⟧(Γ_l) $\wedge\ \neg\mathcal{E}$⟦E'⟧(Γ_r)) $\vee\ (\mathcal{E}$⟦E⟧(Γ_l) $\wedge\ \mathcal{E}$⟦E'⟧(Γ_r))

图 10：双程序的翻译（节选）

双程序的翻译如图 10 所示。翻译函数$\mathcal{B}$ 接收一个双程序和一对上下文（Γ_l、Γ_r），并将其转换为 WhyML 程序。除了将 WhyRel 标识符映射为 WhyML 标识符外，上下文还会存储有关生成的 WhyML 程序所作用的状态参数的信息。与 $\mathcal{B}$类似，函数 $\mathcal{U}$ 将一元程序翻译为 WhyML 程序，$\mathcal{E}$将表达式翻译为 WhyML 表达式，而 $\mathcal{F}$则将一组受限的关系公式翻译为 WhyML 表达式。双程序不要求底层一元程序作用于互不相交的变量集合；然而，这意味着 WhyRel 必须在翻译过程中执行适当的重命名。变量块（变量 `x:T|x:T'` 中 `CC`）的翻译体现了重命名：上下文 Γ_l（分别是 Γ_r）会扩展为 $[\Gamma_l \mid \text{x}: x_l]$，将 `x` 映射到重命名后的副本 x_l（分别是 Γ_r 扩展后绑定 `x`: x_r）。

在翻译（$C|C'$）时，依次组合 C 和 C' 的一元翻译。Sync $\lfloor C \rfloor$ 的处理方式类似，是（$C|C$）的语法糖，但方法调用的情况除外。通过对齐方法调用，可以对关系性质进行过程模块化推理，这表明要利用与该方法关联的关系规约。WhyRel 会将这些翻译为对相应 WhyML 乘积程序的调用，并使用全局方法上下文（Φ，见图 10）。由于翻译后的乘积程序作用于状态对，生成的 WhyML 调用会将 Γ_l.`st` 和 Γ_r.`st`（分别为左侧和右侧状态参数的名称）作为额外参数。

控制流语句的乘积构造需要生成额外的证明义务。对于对齐的条件语句，WhyRel 会引入一个断言，要求守卫一致。锁步对齐循环的处理方式

也类似；守卫一致性必须是不变式。对于条件对齐循环，生成的循环体会体现对齐守卫$\mathcal{P} \mid \mathcal{P}'$所指示的模式：如果左侧（分别是右侧）的守卫为真且 $\mathcal{P}$ （分别是 $\mathcal{P}'$）成立，则仅执行左侧（分别是右侧）的一次迭代；否则，同步执行一次迭代。通过要求条件 $\mathcal{A}$ 为不变式来确保充分性。该条件表明，在两侧都终止之前，循环可以同步执行迭代，也可以仅在一侧执行迭代。在关系区域逻辑中，对齐守卫 $\mathcal{P}$ 和 $\mathcal{P}'$可以是任意关系公式。然而，条件对齐循环的编码采用条件语句，根据这些对齐守卫进行分支。在 Why3 中，只有限制 $\mathcal{P}$ 和 $\mathcal{P}'$ 才能实现这一点；例如，限制它们不包含量词。WhyRel 支持包含一致性公式、单侧指向断言、单侧布尔表达式以及常见布尔连接词的对齐守卫。

封装的证明义务。为确保封装的可靠性，WhyRel 会对源程序进行分析。该分析包含两个部分：一是进行静态检查，确保客户端程序不会直接写入模块边界中的变量；二是生成中间断言，表达客户端堆更新的足迹与模块边界划定的区域彼此不相交。对于具有公有/私有不变式的模块，WhyRel 还会生成一个引理，指出模块边界框定了该不变式，即该不变式仅依赖于边界所表示的位置。对于耦合关系，也会进行同样的处理，但需要考虑相关两个模块各自的边界。关系区域逻辑的一项技术条件要求边界随着计算推进而单调增长；通过在生成的程序中引入适当的后置条件，也可确保满足该条件。

5 评估

我们通过一系列案例研究来评估 WhyRel，这些研究体现了本文开篇强调的挑战性问题。示例包括表示独立性、循环分块等优化 [5], ，以及近期关系验证文献中的其他内容（包括 [9] 和 [21]）。其中一些示例（如第3节所述）涉及针对变化的对齐方式（包括依赖于数据的对齐方式）进行推理。我们的表示独立性示例包括：证明迪杰斯特拉单源最短路径算法分别与两个优先队列实现相连接时的等价性，这需要针对指针结构之间的细粒度耦合进行推理；以及证明克鲁斯卡尔最小生成树算法分别与实现并查集的不同模块相连接时的等价性，这需要建立耦合关系，使两个版本所表示的划分相等。对于所有示例，均使用 Alt-Ergo、CVC4 和 Z3 等 SMT 求解器来解决验证条件。在配备英特尔酷睿 i5-6500 处理器和 32 GB 内存的机器上，利用 Why3 的已保存会话功能重放大多数开发成果的证明，耗时不到 30 分钟。

本工作的一个主要目标是探究，能否以一种适合基于SMT的自动化处理的方式，验证堆操作程序的关系性质

；在很大程度上，我们认为 WhyRel 给出了一个令人期待的答案。该工具实现了关系区域逻辑，并表明，即使是其针对封装的额外证明义务，也可以用一阶断言编码。事实上，借助 WhyRel 对案例研究进行探索，对设计关系区域逻辑的证明规则起到了关键作用。

以区域逻辑的方式推理堆效应通常很简单，而且使用 SMT 可以快速证明验证条件。然而，要证明 WhyRel 生成的技术引理（这些引理涉及说明模块边界能够保持私有不变式和耦合），需要付出相当多的手动工作。这些引理通常涉及对像表达式的推理，而像表达式又涉及存在量词以及对区域进行的非平凡集合运算。鉴于我们对状态和区域的编码，SMT 求解器似乎难以解决这些目标。手动证明需要应用一系列 Why3变换（或证明策略），并引入中间断言。我们推测，使用专用求解器 [23] 或采用不同的堆编码 [24]可以缓解这一问题。

我们对带类型的程序状态进行编码时遇到的另一个问题是，会生成大量与状态良构性相关的验证条件。这些验证条件占据了证明重放时间的很大一部分。Why3 程序直接操作我们这种最小类型化的状态表示，而每次堆更新都必须保持一个不变式，该不变式规定了已分配引用的类型约束（见图 8）。利用 Why3 对模块抽象的支持 [12] 或许可以缓解这一问题。另一种方法是使用假设，这些假设可由 WhyRel类型检查器和翻译器的正确性加以证明。[6]

除了这些与验证相关的挑战之外，我们还注意到，与分离逻辑等其他形式体系相比，区域逻辑中的规约往往较为冗长 [4]。

6 相关工作

WhyRel 的设计以关系区域逻辑为蓝本，该逻辑在 [1] 中提出。那篇论文概述了 WhyRel，并通过介绍在该工具中验证的一小组示例，阐明形式逻辑的若干方面；但它没有全面介绍该工具，也未深入探讨编码细节。论文对该逻辑给出了全面的可靠性证明，并展示了 WhyRel 生成的验证条件及其执行的检查如何与关系证明规则的义务紧密对应。这篇论文建立在一系列区域逻辑相关工作的基础上 [4,2,3]。VERL 工具实现了早期版本的一元区域逻辑，但不包含封装机制，并被用于评估一种针对区域的判定过程 [23]。

对于指针程序的局部推理，分离逻辑是一种有效而优雅的形式化方法。对于关系验证，基于 Iris 分离逻辑并在 Coq 证明助手中构建的 ReLoC [13], 除了支持许多其他特性外，还支持动态分配和并发等语言特性。不过，据我们所知，目前还没有基于分离逻辑的自动主动式关系验证器。

[6] Boogie 验证语言提供了“自由 requires”和“自由 ensures”语法，正是为了表达这类假设。

人们已在各种上下文中探索用于关系验证的对齐。在 WhyRel 中，双程序语法描述了基于控制流的对齐，同时也通过使用对齐守卫支持循环的数据依赖对齐（如第3节所述）。Churchill 等人 [8] 通过使用控制流自动机表示的数据依赖对齐，开发了一种用于等价性检查的技术，并用该技术证明了一组向量化编译器变换和手工优化代码基准的正确性。Unno 等人 [30] 处理了范围广泛的关系问题，包括 k-安全性和共同终止性；他们将对齐和不变式表示为约束满足问题，并采用类似 CEGIS 的技术求解。Shemer 等人 [25] 提出了一组基准，Unno 等人的工作应用于这些基准；Shemer 等人则开发了一种用于等价性和回归验证的技术。上述两项工作都将对齐表示为转换系统，并推断关系不变式和对齐条件。推断依赖求解器，因此必须限制程序，使其适用于这些求解器。Barthe 等人 [6] 提出了一种颇有前景的方法，将关系验证转化为证明轨迹逻辑中的公式；轨迹逻辑是一种使用一阶定理证明器的多排序一阶逻辑。在轨迹逻辑中，可以针对轨迹表达条件，包括不同时间点之间的关系，而无需诉诸对齐本身。

Sousa 和 Dillig 开发了 Descartes [26]，用于通过隐式乘积构造自动推理 Java 程序的 k-安全性质，并采用他们称为笛卡尔霍尔逻辑的逻辑。Pick 等人 [22]进一步推进了这项工作，开发了检测对齐的新技术。基于交互式 KeY 工具的 REFINITY [27]工作台可用于推理 Java 程序的转换；堆推理依赖动态框架，而关系验证则通过考虑抽象程序来进行。其他相关工具还包括基于 Boogie、能够以语言无关的方式模块化推理程序差异的 SymDiff [18]，以及用于 C 程序回归验证的 LLRêve [16]。Eilers 等人 [10] 开发了一种用于非干扰的乘积程序编码，有助于进行过程模块化推理。他们使用 VIPER 工具链验证了大量基准示例。

7 结论

本文介绍了 WhyRel，这是一种用于指针程序关系验证的原型，支持动态框架化和基于状态的封装。该工具忠实地实现了关系区域逻辑，并展示了如何在一阶框架下对其证明义务（包括与封装相关的义务）进行编码。我们借助 Why3 提供的 SMT 支持，在 WhyRel 中完成了若干有代表性的示例，并相信这些示例表明，区域逻辑及其关系变体适合自动化。

致谢 我们感谢匿名的 TACAS 评审人和工件评估人员提出的详尽反馈与建议，这些意见使本文得到了重大改进。我们感谢 Seyed Mohammad Nikouei，他构建了 WhyRel 的初始版本，为当前版本的设计提供了指导。

Nagasamudram 和 Naumann 的部分研究经费由美国国家科学基金会项目（编号 1718713）资助。Banerjee 在美国国家科学基金会任职期间开展的研究得到了该基金会的支持。本文所表达的任何观点、研究结果、结论或建议均属于作者个人，并不一定反映美国国家科学基金会的观点。

数据可用性声明 WhyRel 的源代码以及使用该工具执行的所有示例均可在 Zenodo 上获取，标识符为 https://doi.org/10.5281/zenodo.7308342 [20]。

引用

1. Banerjee, A., Nagasamudram, R., Naumann, D.A., Nikouei, M：具有数据抽象和动态框架化的关系程序逻辑。ACM《编程语言与系统汇刊》**44**(4), 25:1–25:136（2022）。https://doi.org/103551497 21145./。Banerjee, A., Naumann, D.A.：全局不变式的局部推理，第二部分：动态边界。《ACM学报》**60**(3), 19:1–19:73（2013）。https://doi.org/102485981 31145./。Banerjee, A., Naumann, D.A., Nikouei, M：对含纯方法调用的规格进行框架化逻辑分析。ACM《编程语言与系统汇刊》**40**(2), 6:1–6:90（2018）4. Banerjee, A., Naumann, D.A., Rosenberg, S：全局不变式的局部推理，第一部分：区域逻辑。《ACM学报》**60**(3), 18:1–18:56（2013）。https://doi.org/102485982 51145./。Barthe, G., Crespo, J.M., Kunz, C：超越2-安全性：用于关系程序验证的非对称乘积程序。载于：《计算机科学逻辑基础》国际研讨会。《计算机科学讲义》，第7734卷，第29–43页（2013）。6. Barthe, G., Eilers, R., Georgiou, P., Gleiss, B., Kovács, L., Maffei, M：使用轨迹逻辑验证关系性质。载于：《计算机辅助设计中的形式化方法》。第1708894277 7178 2019102391920193–(). https://doi.org/./FMCAD..。Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A：Why3：统御所有证明器的牧羊人。载于：Boogie 2011：第一届中间验证语言国际研讨会。第53–64页（2011）。8. Churchill, B.R., Padon, O., Sharma, R., Aiken, A：用于等价性检查的语义程序对齐。载于：ACM编程语言设计与实现会议。第1027–1040页（2019）。9. Eilers, M., Müller, P., Hitz, S：模块化乘积程序。载于：《编程语言与系统》，欧洲编程研讨会。第502–529页（2018）。10. Eilers, M., Müller, P., Hitz, S：模块化乘积程序。ACM《编程语言与系统汇刊》**42**3324783 11131337 2020101145(), :–:(). https://doi.org/./。Filliâtre, J.C.：一个统御万物的逻辑。载于：计算机演绎会议。施普林格（2013）。12. Filliâtre, J.C., Paskevich, A：Why3中的抽象与泛化性。载于：形式化方法、验证与确认的应用：验证原理。第122–142页。施普林格国际出版（2020）。13. Frumin, D., Krebbers, R., Birkedal, L：ReLoC：一种用于细粒度并发的机械化关系逻辑。载于：IEEE逻辑与计算机科学研讨会。第442–451页（2018）

14. Hoare, C.A.R.：数据表示正确性证明。Acta Informatica 1, 271 00289507 15 281 1972 10 1007 – ( )。https://doi.org/ . /BF 。Kassios, I.T.：动态框架：不受限制地支持框架化、依赖关系和共享。载于：形式化方法。《计算机科学讲义》，第4085卷，第 268 283 2006 10 1007 11813040 _ 19 – （）。
16 https://doi.org/ . / 。Kiefer, M., Klebanov, V., Ulbrich, M：使用编译器中间表示进行关系程序推理：结合静态验证与动态分析。《自动推理杂志》60, 337–363（2018）。17. Kovács, M., Seidl, H., Finkbeiner, B：用于验证2-超安全性属性的关系式抽象解释。载于：ACM CCS（2013）。18. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H：SYMDIFF：一种适用于命令式程序、与语言无关的语义差异工具。载于：计算机辅助验证。第 712 717 2012 10 1007 978 3 642 31424 7 _ 54 –
19 （）。https://doi.org/ . / - - - - 。Leino, K.R.M., Moskal, M：实用的自动主动验证。载于：Ball, T., Shankar, N., Zuck, L.（编）《可用验证研讨会》（2010），http://fm.csl.sri.com/UV10/submissions/uv2010 _ submission _ 20.pdf 20. Nagasamudram, R., Banerjee, A., Naumann, D.A：用于指针程序模块化关系式验证的WhyRel原型（2022年11月）。https://doi.org/10 7308342 21 5281. /zenodo. . Naumann, D.A.：关系式霍尔逻辑三十七年：对其原理与历史的评述。载于：第九届形式化方法、验证与确认应用国际研讨会（ISOLA），第二部分。《计算机科学讲义》，第12477卷，第93–116页（2020），https://doi.org/10.1007/978-3-030-61470-6 _ 7，扩展版见https://arxiv.org/abs/2007.06421。22. Pick, L., Fedyukovich, G., Gupta, A：在关系验证中利用同步性与对称性。载于：计算机辅助验证。第164–182页（2018）。23. Rosenberg, S., Banerjee, A., Naumann, D.A：区域逻辑的判定过程。载于：验证、模型检测与抽象解释国际会议。《计算机科学讲义》，第7148卷，第379–395页（2012）。24. Schmid, G.S., Kuncak, V：证明与反驳含共享可变数据的程序。CoRR abs/2103 07699 25 07699 2021 2103。（），https://arxiv.org/abs/ . 。Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y：属性导向的自组合。载于：计算机辅助验证。第161–179页（2019）。26. Sousa, M., Dillig, I：用于验证k-安全性属性的笛卡尔霍尔逻辑。载于：ACM编程语言设计与实现会议。第57–69页（2016）。27. Steinhöfel, D：使用REFINITY对程序变换规则建模并证明。载于：亚洲编程语言与系统研讨会 APLAS。《计算机科学讲义》，第12470卷，第311–319页（2020）。https://doi.org/ 10 1007 978 3 030 64437 6 _ 16 . / - - - - 。Strichman, O., Godlin,
28 B：回归验证——一种实用的程序验证方法。载于：《经验证的软件：理论、工具与实验》（VSTTE），第496–501页（2008）。29. Terauchi, T., Aiken, A：作为安全性问题的安全信息流。载于：静态分析研讨会（SAS）。《计算机科学讲义》，第3672卷，第352–367页（2005）。30. Unno, H., Terauchi, T., Koskinen, E：基于约束的关系式验证。载于：计算机辅助验证。《计算机科学讲义》，第12759卷，第742–766页。施普林格（2021）。https://doi.org/10.1007/978-3-030-81685-8 _ 35