组合符号执行的端到端理论：从表达力丰富的规约到可满足性模理论（扩展版）

SHIVANANDAN TAMIL KUMARAN*，伦敦帝国学院，英国ANDREAS LÖÖW，伦敦大学皇家霍洛威学院，英国PHILIPPA GARDNER，伦敦帝国学院，英国

arXiv:2611.28916v2 [cs.PL] 24 Nov 2026

组合符号执行（CSE）是一种强大的程序验证和缺陷查找技术，但现有的 CSE 形式化理论尚未充分弥合理想化模型与现实工具之间的鸿沟。我们在 Rocq 证明助理中，为构建基础扎实、贴近现实的 CSE 端到端理论取得了进展。我们的理论从组合符号执行流程的两端提升了现实性：它支持包含代数数据类型、逻辑函数和引理的表达力丰富的前端；同时还提供了与符合 SMT-LIB 标准的 SMT 求解器之间经过验证的连接。我们的贡献包括：(1) 一套能够捕捉实用验证模式的 CSE 理论；(2) 一套独立、可复用的 SMT-LIB 标准 Rocq 机械化实现；(3) 一种经过正确性证明的编码，将组合符号执行后端查询转换为 SMT-LIB 查询，并考虑语义差异且利用类型信息进行优化；(4) 通过提取经过验证的 SMT-LIB 编码，并将其与 CSE 工具 Gillian 自带的编码一同用于处理大多数求解器查询，为后端提供了实证依据。

CCS 概念分类：• **计算理论 → 逻辑与验证；自动推理；分离逻辑；程　　序推理；• 软件及其工程 → 软件形式化验证。**

附加关键词与短　　语：符号执行、分离逻辑、专　　约、可满足性模理论、SMT-LIB

ACM 引用格式：
Shivanandan Tamil Kumaran、Andreas Lööw 和 Philippa Gardner。2027年。《组合式符号执行的端到端理论：从表达力丰富的规范到可满足性模理论（扩展版）》。*ACM 程序设计语言会议录*11，POPL，第 7 篇文章（2027年1月），55 页。https://doi.org/10.1145/3871709

1 引言

组合符号执行（CSE）提供符号执行 [3]，用于创建和使用基于分离逻辑（SL）的函数规约 [44, 50] 及其衍生形式，例如错误性分离逻辑（ISL） [48] 和隐式动态框架 [56]。分离逻辑推理使 CSE 能够使用可复用的函数规约，并验证和综合自身规约。CSE 既适用于用于半自动验证的过近似（OX）正确性分析，也适用于欠近似

（UX）用于自动真实错误查找的错误性分析；现代 CSE 工具包括用于 OX 的 VeriFast [27],Viper [40], 和 CN [47]、用于 UX 的 Infer-Pulse [29]，以及同时用于两者的 Gillian [32, 36]。

图 1 展示了现代 CSE 工具的架构。CSE 引擎对所支持的编程语言进行符号执行。其面向用户的前端 由用户指定的注释组成

*通讯作者

作者联系信息：Shivanandan Tamil Kumaran，伦敦帝国学院，伦敦，英国，shiva.tamil-kumaran21@imperial.ac.uk；Andreas Lööw，伦敦大学皇家霍洛威学院，埃格姆，英国，andreas.loow@rhul.ac.uk；Philippa Gardner，伦敦帝国学院，伦敦，英国，p.gardner@imperial.ac.uk。



本作品采用知识共享署名 4.0 国际许可协议授权。©2027年版权归所有者/作者所有。ACM 2475-1421/2027/1-ART7https://doi.org/10.1145/3871709

例如逻辑数据类型（列表）、逻辑函数（数学上的列表反转函数 †）、表示谓词（列表，定义见图 2）以及函数规约（针对反转[1]的分离逻辑规范）。所需的其他注释取决于分析类型：例如，在验证中，函数规约由人工提供；而在使用双向溯因进行缺陷查找时，函数规约会被自动综合。另一端的 后端 负责一阶推理，并将其卸载给遵循 SMT-LIB 标准的 理论模可满足性 （SMT）求解器 [6], , 例如 Z3 [14] 或 cvc5 [4]；空间推理（关于堆内存的所有权）则保留在 CSE 引擎内部。

近期工作提出了若干 *CSE* 理论：带有正确性证明的 CSE 引擎数学模型。其中最完整的是轻量级 VeriFast (FVF) [28],，这是 VeriFast 的一种简化且依赖工具的模型；以及 Dardinier 等人 [13]和 Lööw 等人 [32, 34],提出的理论，分别大致借鉴了 Viper 和 Gillian，并力求实现工具无关性。

现有的 CSE 理论都未达到形式化方法的黄金标准：同时具备基础性、现实性和端到端特性。此处基础性 指在交互式定理证明器（如 Rocq、Isabelle/HOL 或 Lean）中进行机械化；现实性 指能够涵盖相应未验证工具的足够多特性，以支持现实世界任务；而端到端 指使用与相应工具相同的标准化、规范完善的接口和/或语言

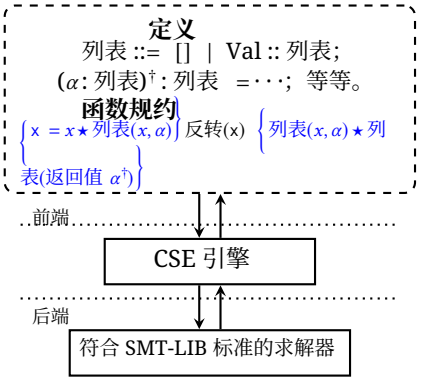


图 1. 现代 CSE 工具的架构。

。达到这一标准的最知名项目是 CompCert [31]：它在 Rocq 中进行了机械化；具备足够的现实性，涵盖了许多现实世界中的优化过程并能生成高效代码；并且是端到端的，因为它以标准 C 语言（的一个较大子集）为输入，并输出标准汇编语言。

我们在让 CSE 达到这一黄金标准方面取得了重大进展。以往的 CSE 理论 [13, 28,34] 具备基础性，但未能做到现实且端到端：其前端对于现实世界任务而言过于有限，其后端也未与符合 SMT-LIB 标准的 SMT 求解器建立形式化联系，而是依赖玩具求解器或假定存在的 SL 求解器等。我们提出的理论在 Rocq 中进行了机械化，涵盖了现实世界验证案例研究中使用的许多 CSE 特性，并通过对 SMT-LIB 标准进行新的 Rocq 机械化，以及将本理论中的表达式编码为 SMT-LIB 的经验证编码，实现了直至标准 SMT 求解器的端到端保证。求解器是我们唯一采用公理化处理的组件；我们仅假定它满足标准规定的要求。因此，遵循本理论的工具只需信任其求解器符合 SMT-LIB 标准，无须信任其他任何方面。除此之外，我们的结果仅依赖 Rocq 内核和经典公理。[2]

我们详细说明了以往工作的前端和后端差距（第 1.1 节和 1.2节），并总结我们的贡献（第 1.3节）。我们的主要贡献是弥合后端差距，但必须同时弥合这两方面的差距，因为前端添加到表达式中的所有内容都必须编码为 SMT-LIB。

1.1 CSE 前端差距（现实性差距）

现代 CSE 工具，尤其是验证工具，支持种类繁多且表达力丰富的特性。一个现实的 CSE 理论不仅必须定义这些特性，还必须至关重要地证明它们是可靠的。为了说明现有 CSE 理论在前端方面的不足，我们回到图 1中的列表反转函数，并使用`reverse(x){r := reverse′(x, null); return r}`实现，该函数使用一个辅助函数

[1]与往常一样，逻辑变量 x 记录 x 的旧值，而专用程序变量 ret 记录返回值。[2]排中律、构造性不定描述、函数外延性、证明无关性以及实数公理，均来自 Rocq 的标准库。

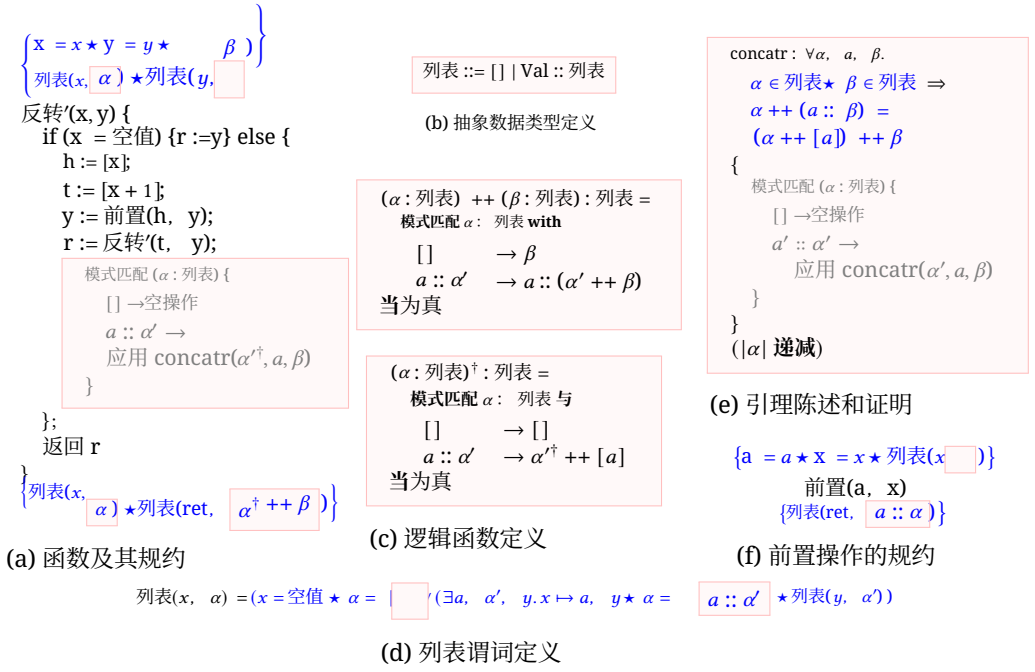


图 2. 使用我们的 CSE 理论验证一个规约的示例，采用了以往理论所不具备 免内容过于繁杂，我们省略了用于折叠和展开用户定义谓词的幽灵命令，以及列表长度函数的定义 $|\cdot|$（因为它与所示逻辑函数类似）。

reverse′，它将 x 反转并前置到一个累加器上。图 2 展示了 reverse′ 及其完整的验证设置，并着重说明了以往 达性特性。为了验证 reverse′（图 2a），证明开发者会为列表内容定义一个代数数据类型（ADT）（图 2b），并定义其上的逻辑函数，例如连接（++）和反转（†）（图 2c）。列表谓词（图 2d）是一个归纳定义的空间谓词，用来断言 x 指向一个堆分配的链表，其内容为 α。reverse′ 和 prepend 的规约使用了这些定义。在通过符号执行验证 reverse′ 的过程中，对 prepend 和 reverse′ 的调用会根据其规约更新符号状态：给定指针 x 处表示逻辑列表$\alpha = a :: \alpha'$ 的链表，以及指针 y 处表示 β的链表，将 a 前置到 β，再将 α' 反转到前置后的列表中，得到 $\alpha'^{\dagger}$ ++ $(a :: \beta)$。要确立 reverse′ 的最终后置条件，需要归纳事实 $\forall \alpha, a, \beta.\ \alpha \in \text{List} \star \beta \in \text{List} \Rightarrow \alpha ++ (a :: \beta) = (\alpha ++ [a]) ++ \beta$。由于 SMT 求解器缺乏归纳推理能力，CSE 工具支持引理（图 2e）。开发者提供一个引理和一个用于触发其应用的幽灵命令 apply。

难点在于证明这些特性的可靠性，并在同一形式体系中完成这一证明。例如，反转′ 在递归调用处使用自身的规约进行验证，此时该规约尚未得到确立，因此可靠性证明必须打破这种循环性。类似地，concatr 的归纳证明会应用 concatr 本身，因此理论必须证明其接受的所有证明都是良基的，例如通过检查用户提供的度量（此处为 $|\alpha|$）是否在每次应用时递减。

先前工作。 以往的 CSE 理论采用表达能力有限的前端语言，不支持用户定义的数据类型、逻辑函数和引理：FVF 仅支持基本算术和布尔运算；Dardinier 等人 [13] 和 Lööw 等人 [34] 对此仅作了小幅扩展。此前没有任何理论能够验证

规约含义独立于引擎的递归函数。因此，这些理论无法对现实的验证任务进行建模。

1.2 CSE 后端鸿沟（端到端鸿沟）

前端表达能力越强，后端编码就越复杂：随着前端日益丰富，它与 SMT-LIB 之间的语义鸿沟也随之扩大。具体而言，用户定义函数暴露了 CSE 的操作语义与 SMT-LIB 的声明式语义之间的不一致。在操作层面，CSE 中的函数可能无法终止，但在 SMT-LIB 中仍有其含义：例如，定义 $f(x) = f(x)$ 不会终止，但在 SMT-LIB 中对应于公理 $\forall x.\, f(x) = f(x)$，而该公理并未对 f 施加任何约束。因此，求解器可能会任意解释 f，使其尽管按预期不会终止，仍能被“求值”。部分操作会加剧这种不一致。CSE 通常将除法或数据类型析构器在某些输入上的行为视为报错，而 SMT-LIB 假设函数均为全函数，并将此类情况留作未指定。因此，求解器可能会为出错的情况赋予任意值（例如，将 $1/0 = 42$ 解释为某个值），从而产生虚假的可满足性结果。

类型判断带来了另一处不匹配。SMT-LIB 采用多排序和静态类型，因此与 SMT 交互时必须将组合符号执行值强制转换为 SMT-LIB 排序。对于动态类型的 CSE 工具，这一差距最为明显：例如，Gillian 的前端采用动态类型，以支持动态类型语言；其渊源可追溯至 JavaScript 验证器 JaVerT [20]。但即使是静态类型的 CSE 工具也需要强制转换，因为它们的类型不一定与 SMT-LIB 排序一致。无论哪种情况，与 SMT 交互时都必须排除 SMT 模型中包含的、在 *CSE* 逻辑中无法表示的值：如果使用 SMT-LIB 的无界整数类型对 256 位无符号整数进行编码，就必须排除诸如 -1 或 2^{256}这样的求解器值，尽管它们是有效的 SMT-LIB 整数。

先前工作。现有理论未能充分考虑与 SMT 的交互。FVF 为其断言语言实现了一个玩具判定过程，绕过了 SMT-LIB。Dardinier 等人将其形式化工作与 Parthasarathy 等人联系起来 [45]，通过 Boogie 建立了端到端的关联，但没有涉及 SMT 交互。Lööw 等人假设其断言语言配有一个分离逻辑求解器。因此，先前没有任何 CSE 理论提供经过验证的、与现成 SMT 求解器的连接。

1.3 贡献

我们基于 Lööw 等人 [34] 的工作，构建了一套机械化 CSE 理论，大幅缩小了我们所指出的真实性差距和端到端差距。Lööw 等人的理论是唯一一种能够创建并使用函数规约的理论，这些规约的含义由分离逻辑满足关系给出，且独立于引擎；这种独立性对于我们的端到端研究目标至关重要。Lööw 等人的理论还独特地涵盖了 SL 正确性分析和 ISL 错误性分析；我们的扩展保留了这两者。

前端扩展（第 4 节）。我们的 CSE 理论具有一个表达力丰富的前端语言，支持用户定义的代数数据类型（列表，图 2b），以及与之关联的匹配表达式（**模式匹配** α： 列表 **，其中** $\cdots$，图 2c）和命令级匹配（模式匹配 (α： 列表) $\{\cdots\}$，图 2a）；逻辑函数（++ 和 $\div$，图 2c）；引理的定义与证明（`concatr`，图 2e）；以及递归函数验证和基于度量的完全正确性论证的可靠性证明。

SMT-LIB机械化（第5节）。为了陈述我们后端扩展的正确性，我们首次逐行机械化了 SMT-LIB 2.7，并将其作为可复用的 Rocq 软件包提供。它涵盖：

(1) SMT-LIB语言的语法和语义；(2) SMT-LIB理论的定义以及SMT-LIB理论的组合；(3) CSE引擎最常使用的SMT-LIB理论：标准理论Core、HO-Core、实数_整数和 Strings（核心部分），以及常用但非标准的理论Seq（即序列理论）。

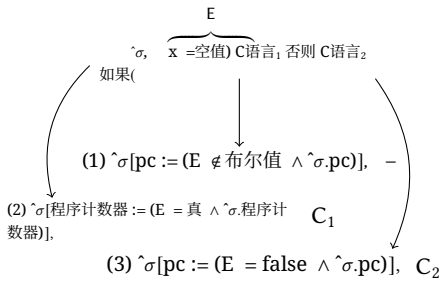


图 3. reverse′ 的部分符号执行树。符号状态被视为一条记录，pc 指“路径条件”。

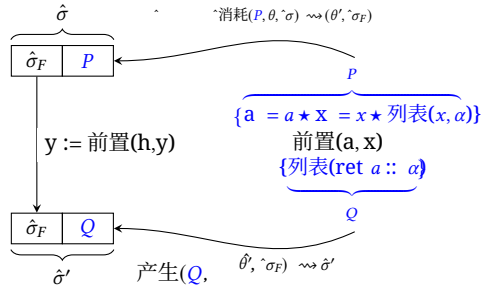


图 4. 由消耗和产生操作支撑的功能组合性。

后端扩展（第 6节）。基于我们的 SMT-LIB 机械化，我们的 CSE 理论包含一种经验证的编码，可将组合符号执行状态满足关系和蕴含查询编码为 SMT-LIB 查询，涵盖：

（1）我们所有的富有表现力的规约特性，例如用户定义数据类型和函数；（2）将 CSE 的部分表达式求值转换为 SMT-LIB 的全表达式求值；（3）将 CSE 的动态类型表达式转换为 SMT-LIB 的静态类型表达式；（4）可表示性条件，确保满足条件的 SMT-LIB 模型对应于 CSE 值和部分逻辑解释。

我们还验证了两种基于类型的编码优化：

（1）受渐进类型系统 [55]（具有不完整的类型信息）以及编译器利用类型信息进行高效代码生成 [23, 30], 的启发，我们将具有已知类型的子表达式映射到原生 SMT-LIB 类型；（2）受类型错误相关工作 [49], 的启发，我们利用已知类型，在无需查询底层 SMT 求解器的情况下，提前排除不可满足查询。

实证评估（第7节）。我们使用 VeriFast 和 Gillian 中的示例评估前端；对于后端，我们将经验证的编码器提取到 OCaml，并将其连接到 Gillian 的求解器循环。一个桥接器将 Gillian 的 75% 查询转换为我们的理论；其余查询大多包含我们的理论尚不支持的负整数。对于两个后端都能回答的查询，经验证的编码器与 Gillian 的结果一致，但有 14 个查询例外，因为两者的表达式语言含义不同。除少数已知原因导致的长尾情况（1.5%）外，求解器运行时间相当。

2 概述

我们概述 Lööw 等人的理论 [34], ，然后介绍我们的前端和后端扩展。

2.1 背景：组合式符号执行速成课程

Lööw 等人的理论使用一种小型示例语言，其大步语义定义在由变量存储和堆组成的具体状态 σ 上。判断式 σ，C $\Downarrow_\gamma o$：σ' 表示：在状态 σ中，使用函数定义 γ对命令 C 求值，会产生结果 o 和最终状态 σ'。

CSE 引擎判断式。 CSE 引擎定义在符号状态 $\hat{\sigma}$ 之上，并以一阶路径表达式记录分支约束。符号语义由判断式$\hat{\sigma}$,C语言 $\Downarrow^m_\Gamma o : \hat{\sigma}'$定义，其解释方式与具体语义类似，但有两点不同：语义由模式参数化（$m \in$ {OX,UX}）；上下文 Γ 提供可调用函数规约，而非 γ所提供的函数体。图 3 展示了分支

以及路径条件更新，说明了图 2a 中`if (x = 空值) ··· else ··`中的反转′：条件的求值会出现以下情况：(1) 执行失败或产生非布尔值，结果为错误 *err*；(2) 产生true，结果为成功 *ok*；或 (3) 产生false，结果为 *ok*。

OX 和 *UX* 健全性。CSE 引擎的关键性质是 OX 和 UX 健全性；此处我们给出简化说明，并在第 3 节给出精确定义。令 $\sigma \models \hat{\sigma}$ 表示具体状态 σ 满足符号状态 $\hat{\sigma}$。那么 OX 健全性是指：当 $o = ok$ 或 *err*时，

$$\sigma, \mathrm{C} \Downarrow_\gamma o : \sigma' \land \sigma \models \hat{\sigma} \Rightarrow \exists \hat{\sigma}'.\ \hat{\sigma}, \mathrm{C} \Downarrow_\Gamma^{\mathrm{OX}} o : \hat{\sigma}' \land \sigma' \models \hat{\sigma}',$$

也就是说，每个具体执行都有一个与之对应的符号执行。具体执行永远不会陷入停滞，因为所有错误都是显式结果。UX 健全性保证了具体执行与符号执行之间的反向关系：当 $o = ok$ 或 *err*时，

$$\hat{\sigma}, \mathrm{C} \Downarrow_\Gamma^{\mathrm{UX}} o : \hat{\sigma}' \land \sigma' \models \hat{\sigma}' \Rightarrow \exists \sigma.\ \sigma, \mathrm{C} \Downarrow_\gamma o : \sigma' \land \sigma \models \hat{\sigma},$$

也就是说，每个表示某次符号执行结果的具体状态，都可由相应的具体执行到达。OX 健全性支持验证：具体行为包含于符号行为之中，因此具体程序错误会在符号层面有所反映，而所有成功的符号路径都能排除具体程序错误。UX 健全性支持错误查找：符号程序错误对应于一个具体程序错误。

消耗和产生操作。CSE 中的功能组合性源于符号状态与函数规约中使用的分离逻辑断言之间的联系。Lööw 等人通过对消耗和产生操作进行数学描述，刻画了 VeriFast、Viper 和 Gillian 等工具中的消耗与产生实现，从而描述这一接口：消耗操作会对与某个断言相对应的符号状态进行模式匹配并将其移除；产生操作则反向地将其添加。这些操作实现了函数规约验证、函数调用推理以及谓词折叠/展开。图 4 展示了如何在对图 2a 中的调用 ^ y := prepend(h, y) 进行符号执行时使用这些操作。在替换 θ 下，该替换将规约的逻辑变量a和 x映射到参数 h 和 y 的符号值；从 $\hat{\sigma}$中消耗 P会移除 y 处的列表。由此 ^ ^ 得到框架 $\hat{\sigma}_F$和替换 θ'；后者通过将 α 映射到 y 处列表的内容而扩展了 θ。随后，在 $\hat{\sigma}_F$上使用 θ' 产生 Q，通过添加返回的列表得到 $\hat{\sigma}'$；该列表现在位于 y 处，其内容为 $\hat{\theta}'(a :: \alpha)$，即将 h 的值前置到原有内容所得的值。函数规约验证与此过程对偶：(1) 将 P产生到一个空符号状态中；(2) 对函数体进行符号执行；(3) 从最终状态中消耗 Q。这三个步骤均成功即验证了该规约。

2.2 前端扩展

我们将前端扩展与现有工具和理论进行比较；见表 1。

非扩展项：总逻辑函数与部分逻辑函数；静态类型逻辑与动态类型逻辑；以及用户定义谓词。对于前三列，我们沿用 Lööw 等人的设计选择和 Rocq 实现。正如引言中所述，带有部分操作的动态类型逻辑与 SMT-LIB 之间存在较大的差距。尽管 JavaScript 和 Python 等动态类型语言广受欢迎，但大多数基于分离逻辑的分析工具仅针对静态类型语言。我们所知的唯一例外是 Gillian [36], 及其前身 JaVerT [20], ，它们可以分析 JavaScript。基于 Viper 构建的 Nagini [16], 可以分析 Python，但仅适用于类型化程序。

用户定义的逻辑数据类型和函数。以往的 CSE 理论缺少用户定义的逻辑代数数据类型，例如图 List 中的 2b，以及函数，例如图 ++ 中的 2c。我们在 Lööw 等人的工作基础上扩展了 ADT 构造器应用（例如 $a :: \alpha$）、逻辑函数应用（例如 $\alpha^\dagger$ ++ β）、ADT 模式匹配表达式（例如 **match** α： List **with** ··· ），以及命令级 ADT 匹配（例如`match(α: List){...}`）。这带来了 Rocq 方面的挑战（第 4 节），但主要工作在于 SMT-LIB 编码（第 6 节）。

表1. CSE 工具和理论中的前端特性。对于工具，条目（例如 ✓ 和 ✗）表示是否支持某项特性；对于理论，则表示该特性是否由理论论证。特性列涵盖逻辑函数是总函数还是部分函数（逻辑函数）、逻辑是静态类型还是动态类型（类型判断）、用户定义谓词（UDP）、用户定义逻辑类型（UDLT）、用户定义逻辑函数（UDLF）、函数规约的应用与验证（规约），以及引理的应用与验证（引理）。

CSE 工具	逻辑函数	类型判断	UDP	UDLT	UDLF	规约	引理
VeriFast [27]	总计	静态	✓	ADTs	✓	✓	✓
CN [47]	总计	静态	✓	ADTs	✓	✓	✓[1]
Viper [40]	部分	静态	✓	域[2]	✓	✓	✗[3]
Gillian [36]	总计[4]	动态	✓	✗	✗	✓	✓
CSE 理论							
FVF [28]	总计	静态	✓	✗	✗	✓[5]	✗
Dardinier 等人 [13]	部分	静态	✗	✗	✗	✗[6]	✗
Lööw 等人 [34]	部分	动态	✓	✗	✗	✓[7]	✗
本文	部分	动态	✓	ADTs	✓	✓	✓

[1] 引理必须编码为函数体为空的 C 函数，这些函数可以导出到 Rocq 并在那里完成证明。[2] 我们不再进一步讨论域。Viper 提供了一个抽象数据类型插件，可以自动将代数数据类型转换为域。[3] 不提供特殊的引理支持，引理必须编码为函数。[4] 这并非基于原则的设计决策，工具可能会偏离这一点。[5] 该理论不为独立于引擎的函数规约赋予含义，这意味着规约无法在引擎中导入或导出。此外，也不支持验证终止性。[6] 该理论不包含函数；inhale 和 exhale 仅应用并验证循环和并行分支规约。[7] 该理论无法为递归函数的验证提供理论依据。

函数规约的应用与验证。 Lööw 等人对函数调用和验证的论述是 CSE 理论中最完整的：函数规约被用于函数调用处（图 4），验证结果是在 SL 中成立的函数规约。该论述缺少递归函数验证的可靠性证明；对此，我们借鉴 深度索引，并改编 Nipkow 关于霍尔逻辑的工作中的深度归纳法 [42]。

引理应用与验证。 用户定义的代数数据类型需要归纳推理，但此前没有 CSE 理论对 引理 （例如图 2e）及其幽灵命令进行形式化。引理应用与验证仿照函数调用和验证：引理是函数，其函数体仅由表示证明的幽灵命令组成。图 2e 给出了检查 `concatr` 的归纳证明是否良基所需的度量，类似于检查函数终止性。我们再次采用 [42], 基于 度量 的方法：用户为引理参数提供一个度量，此处为 $|\alpha|$，该度量必须在每次归纳应用中递减。然后通过对该度量进行归纳即可得到可靠性。我们也将这一论证应用于函数终止性，即 完全正确性。

2.3 后端扩展

后端负责处理符号状态满足关系，$sat(\hat{\sigma})$ 表示 $\exists \sigma,\ \sigma \models \hat{\sigma}$，以及蕴含查询：$\hat{\sigma} \models \hat{E}$ meaning $\hat{E}$ 在 $\hat{\sigma}$ 的所有模型中都为真。其主要任务包括：(1) 当 $\neg sat(\hat{\sigma})$ 时，剪除状态为 $\hat{\sigma}$ 的分支；(2) 在消耗过程中检查蕴含关系 $\hat{\sigma} \models E$。

OX 验证与 UX 真正缺陷查找中的分支剪除有所不同。OX 健全性要求探索所有 可满足分支，因此剪除不可满足分支是可靠的，但并非必需；这能通过控制指数级分支来提升性能，并防止由不可满足错误分支导致的假阴性验证失败。UX 健全性要求探索仅有的 可满足分支，因此

剪除不可满足分支是必需的。在图 3中，引擎针对 x =空值进行分支判断：错误分支不可实现，因为 x 始终作为函数参数存在，因此 OX 剪除可避免虚假的验证失败，而 UX 剪除可避免误报缺陷。

此前的 CSE 理论都无法将此类查询编码为 SMT-LIB：例如，Lööw 等人假设存在一个独立的 oracle 来处理 可满足($\hat{\sigma}$) 和 $\hat{\sigma} \models \hat{E}$，尽管 $\hat{\sigma}$和 $\hat{E}$是特定于引擎的构造。我们通过对 SMT-LIB 进行 Rocq 机械化，以及将符号表达式编码为 SMT-LIB 查询的经验证编码来解决这一问题。我们为符号状态扩展了一个类型环境 T，其中记录了状态已蕴含的逻辑变量的类型。该编码是一个判断式 $T \vdash E \quad t : \tau \mid \Phi$，它将 E在 T下编码为一个排序为 τ的 SMT-LIB 项 t；只要侧条件 Φ 成立，该项就能忠实地表示 E。我们使用该编码定义表达式可满足性检查 checkSat($\hat{E}$)、OX 和 UX 状态可满足性检查 checkSat(OX/UX，$\hat{\sigma}$)，以及蕴涵检查 checkEntails($\hat{\sigma}$，$\hat{E}$)。OX 和 UX 的健全性可归结为这些检查的以下性质，其中 $[\![\hat{E}]\!]_\theta$ 在逻辑解释 θ 下对 $\hat{E}$ 求值，该解释将逻辑变量映射为值：

$$\text{(OX)} \quad [\![\hat{E}]\!]_\theta = \mathsf{true} \Rightarrow \mathsf{checkSat}(\hat{E}) \in \{\mathsf{SAT}, \mathsf{UNKNOWN}\}$$

$$\text{(UX)} \quad \mathsf{checkSat}(\hat{E}) = \mathsf{SAT} \Rightarrow \exists \theta.\, [\![\hat{E}]\!]_\theta = \mathsf{true}.$$

将符号表达式、其值和逻辑解释称为源程序端，编码解决了四个问题。首先，它将用户定义的代数数据类型和逻辑函数映射到 SMT-LIB 签名中。其次，源语言求值可能出错，而 SMT-LIB 函数是总函数，并且在相应参数上未作充分规定，因此侧条件会排除这些未充分规定的 SMT-LIB 求值。第三，源表达式是动态类型的，而每个 SMT-LIB 项都必须具有单一且静态已知的排序。因此，编码器采用类型导向翻译，并基于渐进类型系统 [55], 为每个表达式指定一个排序。它将类型未知的值包装在一个 SMT-LIB 代数数据类型中，该类型为每种类型设置一个构造器；在必要时，还会在该类型与原生排序之间进行强制转换。同时，T它允许类型已知的变量使用原生排序，并让检查过程在调用求解器之前检测出某些类型错误。第四，查询会将 SMT-LIB 模型限制为可表示为源值的值，因为某些目标排序对源值进行了过度近似：例如，用整数编码自然数时，整数可能为负。

3 背景：CSE 理论

我们介绍 Lööw 等人 [34]的 CSE 理论中，构建前端（第 4 节）和后端（第 6 节）扩展所需的部分；更多细节见附录 A。该 CSE 理论仅以内存模型为参数，并通过代数方式表达：使用一个部分交换幺半群，其内存操作满足分离逻辑中熟悉的框架性质和可靠性性质。我们的 Rocq 机械化保留了这种参数化特性，但此处采用标准线性内存，因为参数化仅关系到符号状态可满足性和蕴含关系（第 6.7 节）。

3.1 语法

我们采用图5（左）所示的标准演示语言，其中包括值 Val、类型 Type、[3] （程序）变量 PVar、（程序）表达式 PExp，以及带有内存操作的（程序）命令 PCmd：查找 x：= [E], 变异 [E] ： =E, 分配 x：=new(E)，以及释放 free(E)。

哪些程序注释与分析相关，取决于具体分析；例如，UX 缺陷查找无需任何注释。我们在图中展示了验证 OX 函数规约所需的注释。5 (右)：逻辑变量 LVar ，与程序变量不同；逻辑表达式 LExp，其构造方式与 PExp 类似，但还包括逻辑变量和类型成员关系；用于折叠和展开的幽灵命令 GCmd

[3]像 布尔值 和 自然数，列表（τ）是内置类型，不同于图 2b中的用户定义代数数据类型 列表。

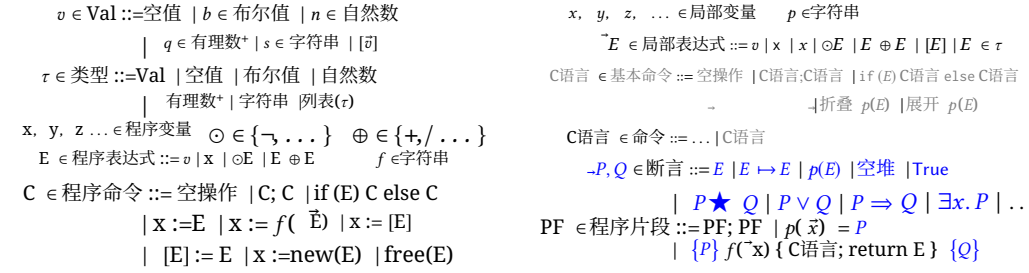


图5. 不带注释（左）和带注释（右）的编程语言语法。箭头表示列表，例如，$\vec{v}$ 是值列表，$\vec{E}$是（程序）表达式列表。

用户定义谓词；扩展了幽灵命令的命令 Cmd；断言 Asrt，我们给出其中一个有代表性的片段。与工具中的常见做法一样，该片段省略了全称量化，因为 SMT 求解器难以处理它；而存在量化变量则会在通过匹配计划消费时实例化 [33]；以及程序片段 PrgFrg，由谓词[4] 和函数定义组成。我们使用程序片段而非完整程序，因为 SL 分析可以分别分析各个函数。下文中，pv(·) 和 lv(·) 分别表示表达式、命令、断言或（后文中的）符号状态所包含的程序变量和自由逻辑变量。

3.2 具体语义

与分离逻辑领域的通常做法一样，我们的具体操作语义使用部分具体内存。具体状态是由具体变量存储 $s : \mathrm{PVar} \rightharpoonup_{fin} \mathrm{Val}$（从程序变量到值的部分有限映射）和部分具体内存 μ组成的对 $\sigma = (s, \mu)$。不同实例的内存各不相同：理论要求它们构成一个部分交换幺半群，且内存操作满足 OX 和/或 UX 框架性质。本文采用分离逻辑的标准线性内存模型，$\mu : \mathrm{Nat} \rightharpoonup_{fin} \mathrm{Val}$，该模型与 OX 分析兼容。[5]

程序表达式 $\mathrm{E} \in \mathrm{PExp}$ 由标准求值函数 $[\![\mathrm{E}]\!]_s = v$ 求值，且仅对存储求值；也就是说，程序表达式求值不会访问或修改堆。如果求值出错，例如除以零，则记作 $[\![\mathrm{E}]\!]_s = \lightning$；我们将可能出错的求值称为部分。

命令 $\mathrm{C} \in \mathrm{Cmd}$ 的大步语义采用判断式 $\sigma, \mathrm{C} \Downarrow_\gamma o : \sigma'$，其中 $\mathrm{outcome}_o ::= \mathrm{ok} \mid \mathrm{err} \mid \mathrm{miss}$ 表示执行成功、语言错误或缺失资源错误（下文将讨论），而 γ 是函数实现上下文：一个为函数调用提供定义的部分函数，即 $\gamma(\mathrm{f}) = (\vec{\mathrm{x}}, \mathrm{C}, \mathrm{E})$，通常写作 $\mathrm{f}(\vec{\mathrm{x}})\ \{\ \mathrm{C};\ \mathrm{return\ E}\ \} \in \gamma$。这些规则在常规操作语义的基础上扩展了预期的语言错误和缺失资源错误规则（出错的执行会在特殊程序变量 err 中记录错误描述），并忽略用户注释，因此幽灵命令 GCmd 是空操作。由于错误是显式结果，语义不会陷入停滞；UX 缺陷查找需要报告这些错误。作为一个代表性案例，我们给出最

[4]我们对用户定义谓词的介绍经过了简化。在完整理论中，谓词参数分为输入和输出参数。两者的区别用于通过匹配计划进行框架推断 [33]，，此处无需对此展开讨论。[5]标准 ISL 模型 [48]，$\mu : \mathrm{Nat} \rightharpoonup_{fin} (\mathrm{Val} \uplus \{\varnothing\})$，与之类似，其中符号 $\varnothing$ 表示某个内存单元已被释放，因此 UX 框架属性成立；它可用于双向溯因等 UX 分析。

有趣的查找规则，展示了各种结果类型：

$$\frac{[\![\mathrm{E}]\!]_s = n \qquad \mu(n) = v}{(s,\mu), \mathrm{x} := [\mathrm{E}] \Downarrow_\gamma \text{满足} : (s[\mathrm{x} \mapsto v], \mu)} \qquad \frac{[\![\mathrm{E}]\!]_s = v \qquad ^{v \notin \text{自然数}}}{(s,\ \mu),\ \mathrm{x} := [\mathrm{E}] \Downarrow_\gamma \text{错误} : (s[\text{错误} \mapsto [\text{“类型”}]],\ \mu)}$$

$$\frac{[\![\mathrm{E}]\!]_s = n \qquad ^{n \in \text{自然数}} \qquad n \notin \mathrm{dom}(\mu)}{(s,\ \mu),\ \mathrm{x} := [\mathrm{E}] \Downarrow_\gamma \text{缺失} : (s[\text{错误} \mapsto [\text{“MissingCell”}\ n]],\ \mu)}$$

ok 和*err* 判断式的查找规则并不令人意外。*miss* 判断式的查找规则或许不太熟悉。它源于缺失资源，例如 $n \notin \mathrm{dom}(\mu)$。必须将缺失资源错误与其他错误区分开来，因为它们不满足框架性质：例如，对一个单元格进行框定，可能会将一次未命中的查找变成成功的查找。它们对双向溯因也很有用；双向溯因用于自动 UX 错误查找，通过添加缺失资源来修复缺失资源错误，在此处即为在 *n*添加一个单元格。

3.3 断言与程序逻辑的语义

该理论建立在一系列关于断言和程序逻辑的定义之上，详见附录。A。我们回顾最重要的定义，以确立记号。断言的满足关系记作 $\theta, (s,\ \mu) \models P$，其中 $\theta : \mathrm{LVar} \rightharpoonup_{fin} \mathrm{Val}$ 是一个为逻辑变量赋予语义的逻辑解释，这是标准的。逻辑表达式通过 $[\![E]\!]_{\theta,\ s}$ 求值；它在程序表达式求值的基础上扩展了对逻辑变量的处理 θ；对于不含程序变量的逻辑表达式，我们记作 $[\![E]\!]_\theta$。给定 σ，$\mathrm{C} \Downarrow_\gamma o : \sigma'$ 和 $\theta, (s,\ \mu) \models P$，我们可以分别使用 SL 和 Iris分离逻辑中的四元组定义 OX 和 UX 判断，其形式如下：

$$\gamma \models \{P\}\ \mathrm{C}\ \{ok : Q_{ok}\}\ \{err : Q_{err}\} \qquad \text{and} \qquad \gamma \models [P]\ \mathrm{C}\ [ok : Q_{ok}]\ [err : Q_{err}],$$

前置条件为 P，后置条件为 Q_{ok}，$Q_{err} \in$ 断言，分别对应成功终止和错误终止。CSE 所采用的统一 OX 与 UX 推理方法要求使用这些四元组。具体细节见附录 A。对于本文而言，使用传统的分离逻辑三元组$\{P\}\ \mathrm{C}\ \{Q\}$即可；它是分离逻辑四元组的语法糖，其中 $Q_{ok} = Q$ 且 $Q_{err} = \mathsf{false}$，表示不会发生错误执行。符号 $\Gamma \in 2^{Spec}$ 表示函数规范上下文，其中 $Spec$ 是函数四元组的集合。最后，当函数实现上下文下 Γ 的所有规约均有效时，判断式 $\models (\gamma, \Gamma)$ 成立 γ。

3.4 符号引擎

现在，我们定义该语言的符号执行引擎，也称为其符号语义。

符号状态。引擎的符号状态记为 $\hat{\sigma}$，它是建立在符号 $\hat{}$ 表达式 $E \in \mathrm{SExp} \subset$ 局部表达式之上的结构；局部表达式是指不包含程序变量的逻辑表达式。$\hat{}$ 符号状态 $\hat{\sigma}$ 是一个元组 $(\hat{s},\ \hat{\mu},\ \mathrm{P},\ \hat{\pi})$，其中包含符号变量存储 $\hat{s}$：$\mathrm{PVar} \rightharpoonup_{fin} \mathrm{SExp}$；符号 $\hat{}$ 内存 $\hat{\mu}$：$\mathrm{SExp} \rightharpoonup_{fin} \mathrm{SExp}$；符号谓词 P，即符号用户定义谓词的多重集，$\vec{\hat{}}$ 其中符号谓词的形式为 $p(E)$，而 $p \in \mathrm{Str}$ 是谓词名称；以及路径条件 $\hat{\pi} \in \mathrm{SExp}$，它描述了符号执行过程中施加的约束。我们将符号状态视为一条记录：对于符号状态 $\hat{\sigma}$ 的符号存储，记作 $\hat{\sigma}.\mathrm{sto}$；对于 $\hat{\sigma}$，记作 $\hat{\sigma}[\mathrm{pc} := \hat{\pi}']$，其中其路径条件被替换为 $\hat{\pi}'$；其他字段同理。符号状态的满足关系记为 $\hat{\theta}, (s,\ \mu) \models (\hat{s},\ \hat{\mu},\ \mathrm{P},\ \hat{\pi})$，其定义如下：

$$\exists \mu_1, \mu_2.\ \mu = \mu_1 \uplus \mu_2 \wedge \theta, s \models_{\mathrm{Sto}} \hat{s} \wedge \theta, \mu_1 \models_{\mathrm{Mem}} \hat{\mu} \wedge \theta, \mu_2 \models_{\mathrm{Pred}} \hat{\mathcal{P}} \wedge [\![\hat{\pi}]\!]_\theta = \mathsf{true}.$$

$\hat{}$ 其中，关系 $\theta,\ s \models_{\mathrm{Sto}} \hat{}s$、$\theta,\ \mu \models_{\mathrm{Mem}} \hat{}\mu$ 和 $\theta,\ \mu \models_{\mathrm{Pred}} \mathrm{P}$ 的定义符合预期；特别地，我们有 $\theta,\ \mu \models_{\mathrm{Mem}} \{\hat{}E_a^1 \mapsto \hat{}E_e^1,\ \ldots\ldots,\ \hat{}E_a^n \mapsto \hat{}E_e^n\} \Leftrightarrow \mu = \uplus_{i=1}^n \{[\![\hat{}E_a^i]\!]_\theta \mapsto [\![\hat{}E_e^i]\!]_\theta\}$。我们称一个符号

状态 $\hat{\sigma}$ 是可满足的，记作 $sat(\hat{\sigma})$，当 $\exists \theta,\ s,\ \mu.\ \theta,\ (s,\ \mu) \models \hat{\sigma}$。此外，如果一个符号$\hat{}$表达式 E 在符号状态 $\hat{\sigma}$ 的所有模型下求值均为真，我们就称该表达式可由该符号状态蕴含：$\hat{\sigma} \models \hat{E} \Leftrightarrow (\forall \theta,\ s,\ \mu.\ \theta,\ (s,\ \mu) \models \hat{\sigma} \Rightarrow [\![\hat{E}]\!]_\theta = \text{真})$。

引擎判断式。 CSE 引擎判断式 $\hat{\sigma}$，C语言 $\Downarrow_\Gamma^m o : \hat{\sigma}'$ 定义在符号状态之上。[6]模式 $m \in \{\mathrm{OX},\ \mathrm{UX}\}$ 会根据所需分析的类型切换引擎的行为。符号结果在具体结果的基础上增加了中止，用于表示一种无法保证对应于具体错误的分析失败，例如在所选函数规约不适用时就可能发生这种情况。在这些规则中，$[\![\mathsf{E}]\!]_{\hat{s}} \Downarrow \hat{E}$ 表示对程序表达式进行符号求值E，即用符号值替换 $\hat{s}$ 中的程序变量。查找的成功和资源缺失的符号规则如下：

$$\frac{[\![\mathsf{E}]\!]_{\hat{s}} \Downarrow \hat{E}_l \qquad \hat{\mu}(\hat{E}'_l) = \hat{E} \qquad \hat{s}' = \hat{s}[\mathsf{x} \mapsto \hat{E}] \qquad \hat{\pi}' = (\hat{E}_l = \hat{E}'_l \wedge \hat{\pi})}{(\hat{s},\ \hat{\mu},\ \hat{\mathcal{P}},\ \hat{\pi}),\mathsf{x} := [\mathsf{E}] \Downarrow_\Gamma^m \text{好} : (\hat{s}',\ \hat{\mu},\ \hat{\mathcal{P}},\ \hat{\pi}')}$$

$$\frac{[\![\mathsf{E}]\!]_{\hat{s}} \Downarrow \hat{E}_l \qquad \hat{s}' = \hat{s}[\mathtt{err} \mapsto [\text{“MissingCell”}, \hat{E}_l]] \qquad \hat{\pi}' = (\hat{E}_l \in \mathrm{Nat} \wedge \bigwedge_{\hat{E}'_l \in \mathrm{dom}(\hat{\mu})} \hat{E}_l \neq \hat{E}'_l \wedge \hat{\pi})}{(\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}), \mathsf{x} := [\mathsf{E}] \Downarrow_\Gamma^m \mathit{miss} : (\hat{s}', \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}')}$$

引擎可靠性。 回顾一下，Lööw等人的主要结果是 OX 和 UX 健全性结果，其简化形式见第 2 节。OX 健全性的完整陈述对 C语言 源程序 $\hat{\sigma}$ 的符号执行采用中止缺失侧条件。首先，每条可满足路径都必须不会中止，因为中止属于分析失败，而非具体结果。其次，如果一条路径报告缺失资源，那么最终状态中必须不含任何剩余谓词资源；否则，所谓的缺失资源仍可能隐藏在其中。于是，OX 健全性表述如下：

定理 3.1（OX 健全性）。假设 $\models (\gamma, \Gamma)$。

> 若 σ，C语言 $\Downarrow_\gamma o : \sigma'$ 且 $\theta, \sigma \models \hat{\sigma}$，并且从 C语言 $\hat{}$满足中止未命中条件 σ
> 则 $\exists \hat{\sigma}', \theta'.\ \hat{\sigma}, \mathrm{C} \Downarrow_\Gamma^{\mathrm{OX}} o : \hat{\sigma}'$ 且 $\theta|_{\mathrm{lv}(\hat{\sigma})} = \theta|_{\mathrm{lv}(\hat{\sigma})}$ 且 $\theta, \sigma' \models \hat{\sigma}'$

引擎的消耗和产生操作各自都有健全性结果，可用于证明例如函数调用的健全性。此处省略细节。

3.5 函数验证

Lööw 等人在其引擎的基础上，实现了基于双向溯因的 OX 函数验证分析和 UX 缺陷查找分析。我们回顾前者，并将其扩展到递归函数（第4节）。它由判断式 $\Gamma \vdash \{P\}\,\mathsf{f(\tilde{}x)\ \{\ C;\ return\ E\ \}}\ \{Q\}$ 给出，该判断式接受一个具有前置条件 P 和后置条件 Q 的函数规约、一个函数实现 $\mathsf{f(\tilde{}x)\ \{\ C;\ return\ E\ \}}$，以及一个包含已知函数规约的上下文 Γ。它通过产生 P、对函数进行符号执行并消耗 Q，来检查规约是否符合实现。对于每个可满足的分支，为了排除内存泄漏，消耗 Q 后不得有任何内存残留。形式化定义见附录 A。他们的结果仅适用于非递归函数，因为递归函数自身的规约必须在验证前添加到 Γ 中，而他们的证明并未涵盖这一点。

4 扩展前端：面向用户的新功能

大多数前端扩展沿用了 CSE 工具和霍尔逻辑中确立的设计。难点在于，要在统一的形式体系中证明这些扩展的可靠性，这需要大量 Rocq 工程工作；还要将它们编码为 SMT-LIB，而这会限制其设计。我们从基于扩展理论的工具用户视角介绍这些扩展，并逐步构建到图 2 所示的验证任务。

[6]Lööw 等人 在判断式中使用预言机来建模天使式非确定性；为便于展示，我们省略了这些内容 .

这些扩展引入了新的上下文，例如 ADT 声明上下文，如 γ 和 Γ，见第 3节。由于许多判断会间接依赖这些上下文，我们仅在相关时标注上下文索引。

4.1 递归函数验证

验证 反转′（图 2a）时的第一个局限是，Lööw 等人 [34]并未验证递归函数的函数验证过程。我们通过将 Nipkow[42]的霍尔逻辑可靠性证明改编到 CSE 中来解决这个问题，下面将对此进行说明。我们证明的结果如下：

定理 4.1（递归函数验证正确性）。令 $\Gamma' \triangleq \Gamma \cup \{\{P\}\, \mathsf{f}(\vec{\mathsf{x}})\, \{Q\}\}$：

$$\begin{array}{ll} \textit{If} & \models (\gamma, \Gamma)\ \textit{and}\ \mathsf{f}(\vec{\mathsf{x}})\,\{\,\mathsf{C}; \mathtt{return}\ \mathsf{E}\,\} \in \gamma\ \textit{and}\ \Gamma' \vdash \{P\}\ \mathsf{f}(\vec{\mathsf{x}})\,\{\,\mathsf{C}; \mathtt{return}\ \mathsf{E}\,\}\ \{Q\} \\ \textit{then} & \models (\gamma, \Gamma') \end{array}$$

与 Lööw 等人的结果不同之处在于突出显示的 Γ'：函数体中的递归调用使用 f 自身的规约进行符号执行。[7] 为这些调用提供依据需要 $\models (\gamma, \Gamma')$，这正是该定理的结论；他们的证明没有处理这种循环性的机制。Nipkow 在对霍尔逻辑进行机械化时，通过按函数调用深度对语言语义编制索引，打破了相应的循环性，同时不对程序逻辑编制索引：这样，归纳假设便涵盖了发生在更小深度处的递归调用。

遵循 Nipkow 的做法，我们只为具体语义编入索引，而不为符号语义编入索引。深度索引的具体语义 $\Downarrow_\gamma^n$ 与 $\Downarrow_\gamma$ 完全相同，唯一的区别是增加了深度索引 $n \in$ 自然数，它限定了执行命令所需的函数调用深度。除函数调用规则外，所有规则都原样传递该索引；函数调用规则则将索引增加 1。我们用一条顺序规则和成功的函数调用规则来说明，并突出显示其中的索引；它们与未编入索引的规则唯一的区别就是这些索引：

$$\frac{(s,\mu), \mathsf{C}_1 \Downarrow_\gamma^{\boxed{n}} ok : (s',\mu') \qquad (s',\mu'), \mathsf{C}_2 \Downarrow_\gamma^{\boxed{n}} o : (s'',\mu'')}{(s,\mu), \mathsf{C}_1; \mathsf{C}_2 \Downarrow_\gamma^{\boxed{n}} o : (s'',\mu'')}$$

$$\frac{\begin{array}{c}\mathsf{f}(\vec{\mathsf{x}})\,\{\,\mathsf{C}; \mathtt{return}\ \mathsf{E}'\,\} \in \gamma \qquad [\![\vec{\mathsf{E}}]\!]_s = \vec{v} \\ \vec{\mathsf{z}} = \mathsf{pv}(\mathsf{C}) \setminus \{\vec{\mathsf{x}}\} \qquad s_p = \emptyset[\vec{\mathsf{x}} \mapsto \vec{v}][\vec{\mathsf{z}} \mapsto \mathtt{null}] \\ (s_p,\mu), \mathsf{C} \Downarrow_\gamma^{\boxed{n}} ok : (s_q,\mu') \qquad [\![\mathsf{E}']\!]_{s_q} = v' \qquad s' = s[\mathsf{y} \mapsto v']\end{array}}{(s,\mu), \mathsf{y} := \mathsf{f}(\vec{\mathsf{E}}) \Downarrow_\gamma^{\boxed{n+1}} ok : (s',\mu')}$$

基于这一语义，我们定义深度索引的函数规约有效性，$\models_n (\gamma, \Gamma)$，其定义与$\models (\gamma, \Gamma)$类似，但使用 $\Downarrow_\gamma^n$。对于 $n \neq 0$，所有规约都必须对函数调用深度小于 n 的执行有效，也就是说，深度索引为 $n-1$；$\models_0 (\gamma, \Gamma)$ 空泛地成立。

有了这些准备，我们现在可以证明定理 4.1。首先，我们证明 OX 健全性（定理 3.1）的深度索引版本，其中用 $\models_n (\gamma, \Gamma)$ 代替 $\models (\gamma, \Gamma)$，并用 $\Downarrow_\gamma^n$代替 $\Downarrow_\gamma$：如果函数上下文在深度 n之前（不包括该深度）有效，那么符号执行在深度 n（包括该深度）以内是过近似可靠的。定理 4.1 并未提及 n，因此要对 n进行归纳，我们利用以下事实：带索引和不带索引的具体语义一致，即 $\sigma、\mathsf{C} \Downarrow_\gamma o$：$\sigma' \Leftrightarrow \exists n.\ \sigma、\mathsf{C} \Downarrow_\gamma^n o$：$\sigma'$，以及 $\models (\gamma, \Gamma) \Leftrightarrow \forall n.\ \models_n (\gamma, \Gamma)$。因此，只需对 n进行归纳，证明 $\forall n.\ \models_n (\gamma, \Gamma')$。基础情形是空泛成立的。在归纳情形中，归纳假设 $\models_n (\gamma, \Gamma')$、深度索引的 OX 健全性，以及产生与消耗的可靠性表明，函数体的每个深度为 n 的具体执行都遵循其规约；因此 $\models_{n+1} (\gamma, \Gamma')$。

4.2 代数数据类型

接下来，我们添加代数数据类型（代数数据类型），使用户能够定义例如 列表 （图 2b）。这一扩展在概念上很简单，但需要大量的绑定子工程：我们的机械化

[7]我们只考虑直接递归；Nipkow 的论证也适用于相互递归，而相互递归不会增加实质性的复杂性。

局部无名表示 [10], 需要能够一次打开多个绑定符的机制，而且我们必须在各处传递局部闭合侧条件。这里我们聚焦于面向外部的扩展。

我们为程序片段 PrgFrg 扩展了代数数据类型声明：PrgFrg ::= ... |($\delta = c_1(\vec{\tau}_1) \mid \cdots \mid c_n(\vec{\tau}_n)$)，其中 $\delta \in$ Str 是代数数据类型名称，每个 $c_i(\vec{\tau}_i)$ 都是一个构造器。这些声明会生成一个代数数据类型声明上下文 Δ : Str $\rightharpoonup_{fin}$ Type* ×Str，将构造器映射到其参数类型及构造出的代数数据类型。我们将 $c(\vec{\tau}) : \delta \in \Delta$记作：当 $\Delta(c) = (\vec{\tau}, \delta)$ 时成立，并将 $\delta \in \Delta$ 记作某个构造器构造出 δ 时成立。我们将类型扩展为包含代数数据类型名称，Type ::= ... | δ，并如下定义 Wf (τ)：(1) τ既不是列表类型，也不是代数数据类型；(2) τ = List(τ') 且 Wf (τ')；或 (3) $\tau = \delta$ 且 $\delta \in \Delta$。我们假设 Δ 中所有构造器的参数类型都是良构的。值中加入了构造器值 $c(\vec{v})$，其类型为 δ，当 $c(\vec{\tau}) : \delta \in \Delta$且 $\vec{v} \in \vec{\tau}$时成立。格式错误的类型不可居留。在 Rocq 中，值现在依赖于类型判断，而类型判断又依赖于值；Rocq 无法将它们定义为相互归纳—归纳类型。因此，我们将值定义为依赖对：由一个预值（可能格式错误的值）和一个良构性证明组成，并假设证明无关性（见附录 B.3）。

在断言层，代数数据类型允许我们在列表谓词中写入 [] 和 $a :: \alpha$（图 2d）。我们将逻辑表达式局部表达式 LExp （但不包括程序表达式）扩展为包含构造器应用 $\vec{\text{structor applications}}$ c(E)、测试器 E is c，以及匹配 match E : δ with $p_i \to \text{E}_i|_{i=1}^n$，其中每个模式 p_i 要么是变量 x，要么是构造器模式 $c(\vec{\text{x}})$。求值按预期扩展；对于匹配类型错误，返回 $\lightning$；现在它还依赖于 Δ，按惯例省略。幽灵命令新增 match (E : δ) { $p_i \to \text{C}_i|_{i=1}^n$ }，其符号语义（图 6）在所有模式上进行分支。构造器模式 $c_j(\vec{x})$ 对应的分支会将被匹配表达式等于 $c_j(\vec{x})$ 且不匹配任何更早模式这一条件添加到路径条件中。其他规则见附录。 B。

$$\frac{[\![E]\!]_{\hat{\sigma}.\text{sto}} \Downarrow \hat{E} \quad p_j = c(\vec{x}) \text{ for } 1 \le j \le n \quad p_k \text{ not a variable binder for } 1 \le k < j \quad \hat{\pi} = (c(\vec{x}) = \hat{E}) \wedge \bigwedge_{k=1}^{j-1} \neg(\hat{E} \text{ is } c_k) \quad \hat{\sigma}' = \hat{\sigma}[\text{pc} := \hat{\pi} \wedge \hat{\sigma}.\text{pc}] \quad \hat{\sigma}', \text{C}_j \Downarrow_\Gamma^m o : \hat{\sigma}''}{\hat{\sigma}, \texttt{match}\ (E : \delta)\ \{\ p_i \to \text{C}_i|_{i=1}^n\ \} \Downarrow_\Gamma^m o : \hat{\sigma}''}$$

图 6. `match`的符号语义。

4.3 引理

借助代数数据类型，我们能够表述图 2a，但 SMT 求解器缺乏对用户定义的代数数据类型进行归纳推理的能力。因此，我们允许在 CSE 中陈述并证明引理。例如，图 2e 中的证明对 $\alpha = []$ 和 $\alpha = a' :: \alpha'$ 进行分支：基础情形由路径条件推出；在归纳情形中，$\alpha \in$ List 给出 $\alpha' \in$ List，因此可以应用引理，用 $\alpha' ++ (a :: \beta) = (\alpha' ++ [a]) ++ \beta$强化路径条件。

引理定义与应用。我们将引理语句 l : $\forall \vec{\text{x}}. P \rightleftharpoons Q${C} ($M$递减度量) 加入程序片段 PrgFrg，其中 l 是引理名称，$\vec{\text{x}}$ 是全称量化逻辑变量，$P, Q \in$断言分别是引理的左侧和右侧，而 $\rightleftharpoons \in \{\Leftarrow \not\Rightarrow\}$ 表示蕴含方向。其余部分用于验证：C 是证明，即一个幽灵命令，它将满足假设的状态转换为满足结论的状态；而 $M \in$ 局部表达式是一个递减度量，用于确保 C 中对 l 的递归应用是良基的。我们还将 apply l(E) 加入幽灵命令 C，以参数 E 应用引理 l。

引理上下文与引理语义。引理为 CSE 引擎引入了一个新的上下文，即引理上下文 Λ。引擎判断式会将它与 Γ 一并携带，记作 $\Downarrow_{\Gamma\Lambda}^m$。它将每个 l 映射到其陈述 $\forall \vec{x}. P \rightleftharpoons Q$；或者在验证 l 时（下文将讨论），映射到附带度量的陈述 $\forall \vec{x}. P \rightleftharpoons Q$($M$ **递减**)。引理验证通过后，其证明 C 和度量 M都不会保留在上下文中，这与 Γ 携带函数规约但不携带其实现的情形相同。引理有效，$\models \forall \vec{x}. P \rightleftharpoons Q$，当且仅当对于所有满足 $\{\vec{x}\} \subseteq \text{dom}(\theta)$ 的 θ, σ，都有 $\theta, \sigma \models P \overset{\Rightarrow}{\Leftarrow} \theta, \sigma \models Q$，与任何度量无关。当且仅当上下文中的所有陈述都有效时，该上下文才有效。

引理应用规则。新的 apply 命令与函数调用类似（形式化规则见附录 B）：它消耗 P 并产生 Q。我们已扩展引擎的 OX 和 UX 健全性证明，使其涵盖这一规则：以往支持引理的工具只能处理 OX 推理；我们的规则也支持 UX，其中 $\Rightarrow$ 引理可应用于 OX，$\Leftarrow$ 可应用于 UX，而 $\Leftrightarrow$ 则适用于两者，这与不正确性逻辑中最初提出的 OX/UX 对称性相符 [43]。

引理验证。引理验证会检查由幽灵命令 C语言 表示的归纳证明。它由判断式 $\Lambda \vdash 1 : \forall \vec{x}. P \gtrdot Q$ {C语言} (M **递减**) 给出，它接收一个引理上下文 Λ 和一个引理定义。（形式化定义同样见附录 B。）简而言之，验证遵循与函数验证相同的三个阶段，并具有相同的成功条件：生成 P，对 C语言 进行符号执行，并消耗 Q。由于引理必须对每个满足其假设的状态都成立，而这只有 OX 推理才能保证，因此无论方向如何，每个引理都在 OX 模式下验证为 $\Rightarrow$ 引理：通过交换两边得到的 $\Leftarrow$ 引理，以及由两个 $\Leftrightarrow$ 引理组成的 $\Rightarrow$ 引理，每个方向各有一个证明。

为排除诸如 荒谬 : 真 $\Rightarrow$ 假 { 应用荒谬() } 这类无效的自我论证证明，我们再次借鉴 Nipkow [42]，并调整了他在霍尔逻辑中针对递归过程的全正确性证明，将度量用作良基关系。验证引理时，它在引理上下文中的条目会携带其度量，(1) $= \forall \vec{x}. P \Rightarrow Q$($M$ **递减**)，而应用 规则会检查每次应用时度量是否递减；在图 2e 中，它会检查 $|\alpha'| < |\alpha|$在 $\alpha = a' :: \alpha'$ 下。与 Nipkow 的良基关系不同，后者是元逻辑中定义在状态上的关系；度量是逻辑表达式，因此 CSE 引擎可以通过可满足性模理论检查其是否递减。验证通过后，引理会在不带度量的情况下添加到上下文中：此时它对度量的任何取值都有效，因此后续应用无需检查度量是否递减。

我们使用两个定义和一个辅助定理来证明引理验证正确性（下文的定理 4.3）。首先，与 Nipkow 的浅嵌入关系不同，度量可能无法求值（↯），或产生非自然数的值；我们将此类结果规范化为可能的最小度量：$\mathrm{numify}(\llbracket M \rrbracket_{\theta,s}) = n$ if $\llbracket M \rrbracket_{\theta,s} = n$ for some $n \in$ 自然数, and 0 otherwise。其次，度量有效性 $\models_n \Lambda$要求每个带度量的 $\Rightarrow$引理在小于 n 的范围内有效：$\forall \theta, s, \mu. \{\vec{x}\} \subseteq \mathrm{dom}(\theta) \wedge \mathrm{numify}(\llbracket M \rrbracket_{\theta,s}) < n \Rightarrow \theta, (s, \mu) \models P \Rightarrow \theta, (s, \mu) \models Q$，其余所有引理则按通常方式有效。最后，对于幽灵命令，我们需要以下按度量索引的 OX 健全性版本（定理 3.1）（参见按深度索引的可靠性，第 4.1 节）：

定理 4.2（Ghost OX 可靠性）。假设 $\models_n \Lambda$，且所有带度量的引理共享相同的度量表达式 M。[8]

> 若 θ，$(s, \mu) \models \hat{\sigma}$ 且 $\mathit{numify}(\llbracket M \rrbracket_{\theta,s}) = n$，并且定理 *3.1* 的中止未命中条件对来自 $\hat{\sigma}$ 的 C 成立，则 $\exists \hat{\sigma}', \theta'. \hat{\sigma}, C \Downarrow^{\mathrm{OX}}_{\Gamma\Lambda} \mathit{ok} : \hat{\sigma}'$ 且 $\theta|_{\mathrm{lv}(\hat{\sigma})} = \theta|_{\mathrm{lv}(\hat{\sigma})}$ 且 $\theta', (s, \mu) \models \hat{\sigma}'$

证明概要。关键情况是引理应用。由于 $\models_n \Lambda$，该引理仅对度量小于 n的状态有效。初始度量 n 连同度量递减检查，可确保在每次归纳应用时都满足这一条件。□

定理 4.3（引理验证正确性）。令 $\Lambda' \overset{\text{定义}}{=} \Lambda[\text{引理} \mapsto \forall \vec{x}. P \Rightarrow Q(M \textbf{递减})]$:
如果$\models \Lambda$，且 Λ 中没有任何引理带有度量，并且 $\Lambda' \vdash 1 : \forall \vec{x}. P \Rightarrow Q$ {C}（M **递减**）那么 $\models \Lambda[1 \mapsto \forall \vec{x}. P \Rightarrow Q]$
通过对每个已验证的方向分别应用这一点，即可得出 $\Leftarrow$ 和 $\Leftrightarrow$ 情形。

[8]只有正在验证的引理带有度量，因此这并不构成限制。该度量用于标记这个引理，因为与函数验证共用的 CSE 引擎并不知道自己正在验证哪个引理。

证明概要。 应用 $\models \Lambda \Leftrightarrow \forall n.\ \models_n \Lambda$ 并对 n归纳法：基础情形显然成立；归纳情形由归纳假设 $\models_n \Lambda'$、定理 4.2以及 消耗 和生成的可靠性即可推出。 □

完全函数正确性。 基于度量的同一技术可将函数验证扩展到完全正确性。函数规约可以附带一个度量，由一种新的引擎模式检查该度量在每次调用时是否递减；在此模式下进行验证，还能证明函数从满足其前置条件的每个状态出发都会终止。详见附录。B。

4.4 逻辑函数

图 2 的最后一个要素是逻辑函数，包括图 2c 中的 $\circ^{\dagger}$ 和 ++ 。我们将逻辑表达式 LExp 扩展为包含应用 $f(\mathrm{E})$，并将程序片段 PrgFrg 扩展为包含定义$f(\vec{\mathrm{x}} : \vec{\tau}) : \tau = \mathrm{E}$，其中 $\mathrm{E}^{\mathrm{pre}}$；参数 $\vec{x}$和 具有类型 $\vec{\tau}$， τ是返回类型，E 是函数体，$\mathrm{E}^{\mathrm{pre}}$ 是前置条件；函数体和前置条件中不包含程序变量，且其中的逻辑变量均属于 $\vec{x}$。逻辑函数还需要一个新的上下文，即逻辑函数上下文 Ψ，它将每个逻辑函数标识符映射到其定义。我们假设 Ψ 中所有逻辑函数的参数类型和返回类型都是良构的。

终止性与前置条件的设计。 递归逻辑函数除了报错之外，还可能以另一种方式导致求值失败：它可能发散，此时 $[\![E]\!]_{\theta,s}$ 没有结果，甚至不是 $\lightning$。当求值产生一个值或 $\lightning$时，我们称其终止 ； 部分性始终指报错，而非发散。为了可靠地将逻辑函数翻译为可满足性模理论函数，逻辑函数必须终止。前置条件 E^{pre} 刻画了函数体能够无错误终止的参数：当前置条件求值为真时，函数体必须求值得到某个 some $v \in \tau$。编码器还会生成侧条件，要求每个被调用函数的前置条件成立，以防部分定义或未充分规定的情况产生虚假的可满足结果。对于递归函数，这些义务可能需要归纳证明：例如，计算倒数列表的函数要求每个元素都非零。我们不让编码器推断这类事实，而是在逻辑中将前置条件设为一等要素，并可像在 Viper 中那样通过引理证明它们。

形式化定义。当 $[\![f(\vec{\mathrm{E}})]\!]_{\theta,s} = v$ 成立时，求值满足：(1) $f(\vec{\mathrm{x}} : \vec{\tau}) : \tau = \mathrm{E}$，当 $\mathrm{E}_{\mathrm{pre}} \in \Psi$；(2) $[\![\vec{\mathrm{E}}]\!]_{\theta,s} = \vec{v}$，其中 $v_i \in \tau_i$对每个 i都成立；(3) $\theta' \,\mathrm{def}{=}\, \{\vec{x} \mapsto \vec{v}\}$；(4) $[\![\mathrm{E}_{\mathrm{pre}}]\!]_{\theta'} =$为 true；以及 (5) $[\![\mathrm{E}]\!]_{\theta'} = v$with$v \in \tau$。否则， $f(\mathrm{E})$ 的求值结果为 $\lightning$，除非其中某次求值不终止。按照我们的约定，不将 Ψ 作为显式索引添加到 $[\![\cdot]\!]_{\theta,s}$。

可满足性模理论的可靠性依赖于前置条件能正确刻画每个函数体在哪些情况下会正常终止。Ψ 当且仅当对所有 $f(\vec{x} : \vec{\tau}) : \tau = E$ **when** $E_{\mathrm{pre}} \in \Psi$，它都是有效的，记作 $\models \Psi$：

$$\forall \vec{v} \in \vec{\tau}.\ \theta = \{\vec{x} \mapsto \vec{v}\} \Rightarrow (\exists r.\ [\![E_{\mathrm{pre}}]\!]_\theta = r) \wedge ([\![E_{\mathrm{pre}}]\!]_\theta = \mathrm{true} \Rightarrow \exists v \in \tau.\ [\![E]\!]_\theta = v).$$

第一个合取项使得前置条件之外的应用求值为 $\lightning$ ，而不是发散。二者共同确保对有效的 Ψ 求值总会终止。此外，当且仅当前置条件不成立时，应用才会报错：如果前置条件求值为真，根据有效性，函数体不会报错；否则，根据定义，应用会报错，即便函数体本身不会报错。可满足性模理论的 SMT-LIB 编码需要这一点，因为它会用被调用函数的前置条件来约束每个应用：为了保证 OX 健全性，这个约束不得排除任何不会报错的求值。

在 Rocq 中，由于求值可能不会终止，因此不能再将其直接定义为函数。我们将其定义为一个关系，用 $\models \Psi$ 来描述状态与项之间的关系；然后，在假设 $\models \Psi$成立的情况下，使用希尔伯特 ε 算子将其转化为函数（参见附录B.3）。

扩展逻辑函数上下文。 机械化直接假设 $\models \Psi$。检查需要证明，每个新函数体在其前置条件下都能无错误地终止，而这

仅靠度量无法做到：例如 $|\cdot|$ 这样的度量本身就是一个逻辑函数，其定义需要一个度量。结构递归打破了终止性证明中的这个循环，但要证明函数体不会出错，可能仍需用到归纳法，因此也需要一个带度量的引理。这留待未来工作处理。

5 SMT-LIB 与我们的机械化

SMT 求解器用于检查一阶公式在给定背景理论集合下的可满足性。SMT-LIB [5] 提供了一种通用的求解器输入语言和理论目录，但其规范以散文和非形式化数学的方式编写。我们首次对 SMT-LIB 进行了机械化：逐行机械化 SMT-LIB 2.7（2026年9月9日发布的版本，也是目前的最新版本），涵盖其语法与语义、理论定义以及若干标准理论。与我们开发的其余部分一样，该机械化使用经典公理。

SMT-LIB 有两项主要制品：语言，为脚本和公式定义语法与语义；以及理论，规定类型和函数及其预期含义。SMT-LIB 标准定义了该语言 [6], 2.7 版本约 100 页。[9] 近期版本添加了代数数据类型和匹配表达式（2.6，2017），随后又加入函数类型和函数抽象，以支持高阶风格推理，同时保留一阶量化（2.7，2025）。[10] 另一方面，理论由发布在 SMT-LIB 网站上的理论文件定义。由于篇幅所限，我们不讨论这些理论文件；感兴趣的读者可参阅我们的机械化工作。

完成机械化后，我们对 SMT-LIB 标准留下了积极的印象：在研究过程中，我们没有发现任何重大缺陷，这或许是因为该标准已经相对形式化（例如，与一般编程语言标准相比）。尽管如此，在尝试证明标准中规定的一些语义属性时，我们发现了一些疏漏，并向维护者提出了这些问题；其中一些已得到确认，并将在未来版本中修正。更多信息请参阅我们的机械化成果。

5.1 语法

图 7a 给出了 SMT-LIB 项的语法，其中包括变量、命名函数应用（当多态性导致结果类型不明确时，会标注其结果类型）、量词和绑定符、let 表达式以及代数数据类型匹配。请注意，在此处以及本文其余部分中，我们使用深红色来区分 SMT-LIB 的句法构造和语义构造。图 7b 对比了 SMT-LIB 与传统一阶逻辑的语法。两者有两个重要区别。首先，SMT-LIB 是一种多排序（“有类型”）逻辑，而传统一阶逻辑不区分排序（“无类型”）。其次，SMT-LIB 在句法上并不区分项和公式：SMT-LIB 中的公式只是排序为布尔值的项。因此，即使是布尔联结词，在句法上也是函数应用，不过为了便于阅读，我们将它们写作中缀形式。我们用 t 表示任意项，用 φ 表示布尔值排序的项，并用 Φ 表示一组布尔值排序的项。

SMT-LIB 词汇由签名定义。签名 Σ 包含排序符号 Σ^S、这些排序符号的元数以及函数符号 Σ^F。非零元排序符号构成参数化排序：如果 $s \in \Sigma^S$ 的元数为 n，且 τ_1、……、τ_n 是排序，那么 $s(\tau_1$、……、$\tau_n)$ 就是一个排序。每个签名都包含排序 Bool 和箭头排序 $\tau_1 \to \tau_2$，即函数抽象 $\lambda(x : \tau).\ t$ 的排序。集合 sorts(Σ) 涵盖基于 Σ 的排序符号及其所指定元数的良构排序集合。每个函数符号都有一个或多个秩，记作 $\mathrm{f} : \tau_1$、……、τ_n；$\tau \in \Sigma$（适用于非恒定符号），而常量则仅记作 $\mathrm{f} : \tau \in \Sigma$。签名还包含一个排序映射，即从变量到排序的部分映射：当 Σ 将 x 映射到排序 τ 时，我们记作 $x : \tau \in \Sigma$。自 2.6 版起，签名还会区分

[9] 我们的工作涵盖 SMT-LIB 的语义层，该层赋予查询数学含义，并定义逻辑的可满足性。该标准还定义了基于 S 表达式的词法层和结构层，但我们的工作不需要这些内容。[10] SMT-LIB 3.0 预计将以归纳构造演算（CIC）[46], 这一完备的高阶逻辑为基础。

$$
\begin{aligned}
t \in \text{项} ::= {} & x \mid f(\ \vec{t}\ \mid (f:\tau)\ (\ \vec{t}) \\
& \mid \forall (x:\ \tau).\ t \mid \exists (x:\tau).\ t \mid \lambda (x:\ \tau).\ t \\
& \mid \textbf{令}\ \ x_i = t_i|_{i=1}^{n}\ \textbf{in}\ t \\
& \mid \textbf{模式匹配}\ \ t\ \textbf{与}\ \ p_i \to t_i|_{i=1}^{n}
\end{aligned}
$$

(a) SMT-LIB 项语法。

$$
\begin{aligned}
t ::= {} & x \mid f(\ \vec{t}) \\
\phi ::= {} & P(t_1, \ldots, t_n) \mid t_1 = t_2 \\
& \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \ldots \\
& \mid \forall x.\, \phi \mid \exists x.\, \phi
\end{aligned}
$$

(b) 传统一阶语法。

图 7. SMT-LIB 语法与传统一阶语法的比较。前者使用显式排序注释（$x:\tau$），并将所有表达式（包括量化公式）都视为项。

ADT 的函数符号：构造子、选择器和测试器。每个构造子都配有一个布尔值测试器，以及针对其每个字段的一个选择器，其特殊语义将在下文给出。

良类型性，$\Sigma \vdash t:\ \tau$会参考变量排序和函数秩，并在绑定符下更新排序映射。SMT-LIB 脚本仅断言布尔值-排序的项。

5.2 语义

一个结构 A 通过定义全域 A、将每种排序 τ 解释为一个论域$\tau^{\mathbf{A}} \subseteq A$，并将每个带秩函数符号 f：$\tau_1, \ldots, \tau_n$；$\tau$解释为一个全函数 f A :${\tau_1}^{\mathbf{A}} \times \cdots \times {\tau_n}^{\mathbf{A}} \to \tau^{\mathbf{A}}$，来给出语义。函数符号不能被解释为部分函数。该标准还固定了一些论域，包括布尔值$^{\mathbf{A}}$ = Bool = {true, false}以及 $(\tau_1 \to \tau_2)^{\mathbf{A}}$，即所有全函数的集合${\tau_1}^{\mathbf{A}} \to {\tau_2}^{\mathbf{A}}$。

将全域表示为 A 单个 Rocq 归纳类型行不通，因为箭头排序的论域包含其他论域之间的所有全函数，而构造子若以作用于自身的函数为参数，其归纳类型会被 Rocq 的严格正性检查拒绝。因此，我们改用基于塔斯基式全域的宇宙设计模式 [11], ，其基础是塔斯基风格的全域 [37]:

1 不动点解释（论域：排序 ->类型）（参数：排序列表）（返回值：排序）：类型：=2 对参数进行模式匹配：3 | [] => 论域 返回值4 | 参数 :: 参数' => 论域 参数 -> 解释 论域 参数' 返回值 5 结束。6 7 记录结构：= { 8 论域：排序 ->类型；9 论域_布尔值：论域 布尔值_排序 =布尔值；10 论域_映射：对任意 s1、s2，论域（映射_排序 s1 s2） =（论域 s1 ->论域 s2）；11 解释：函数 ->对任意（参数：排序列表）（返回值：排序），解释 论域 参数 返回值；12 }。

一个结构**A** 在 Rocq 中由`structure`表示，其字段`domain` 是解码函数，它将每个代码（即一种语法排序）映射到其载体类型，也就是 $\tau^{\mathbf{A}}$的对应项。其字段`interp` 为每个函数符号及其每个秩指定一个柯里化 Rocq 函数，该函数在相应载体类型之间进行映射，是$f^{\mathbf{A}}$的对应项。字段`domain_bool` 和 `domain_map` 用于强制满足上述约束。全域从不显式定义，而是隐式地由各排序论域的并集构成。

结构如何解释一个函数符号，取决于该符号以何种方式进入签名。理论定义符号从所选背景理论中获得其含义。用户-声明的符号通过`declare-fun` 引入时是未解释的，因此结构可以将其解释为声明的秩所对应的任意全函数。用户-定义的符号通过`define-funs-rec` 引入时受其函数体约束。使用参数f定义 $\vec{x}:\ \vec{\tau}$ 且函数体为 t，其中可能提及f 本身，相当于声明f 并断言其全称量化的定义方程

$\forall(\vec{x}:\ \vec{\tau})$。f（$\vec{x}$）$= t$。由于 f 被解释为全函数，因此终止性在此没有意义。一个作为程序不会终止的定义，因而可能不会对 f 施加任何约束。例如，每一种 f 的解释都满足 $\forall(x:\ \tau)$。f（x）$=$ f（x）。该方程甚至可能没有模型，例如矛盾的 $\forall(x:$ 整数)。f（x）$=$ f（x）$+ 1$，从而使包含它的每个脚本都不可满足。

对于代数数据类型，标准要求每个结构都以绝对自由的方式解释构造器。每个数据类型排序 δ 的论域 $\delta^{\mathbf{A}}$ 都是排序为 δ 的基项集合，由 $g ::= v \mid \mathrm{c}(g_1, \ldots, g_n)$ 给出，其中 v 遍历非数据类型排序的值（例如整数），c 遍历构造子。每个构造子的解释都会构造出相应的基项，即 $\mathrm{c}^{\mathbf{A}}(g_1, \ldots, g_n) = \mathrm{c}(g_1, \ldots, g_n)$。对于一个构造子为 nil : List 和 cons : Int, List; List 的整数列表数据类型，其论域 $\mathrm{List}^{\mathbf{A}}$ 因此是基项集合 {nil, cons(0, nil),cons(1, cons(0, nil)) . . . }，并且 $\mathrm{cons}^{\mathbf{A}}(0, \mathrm{nil}) =$cons(0, nil)。代数数据类型的预期性质源于不同的基项对应不同的值。构造子是单射且值域不相交；由于论域中不包含其他内容，每个值都是有限构造子树。我们的机械化要求每个数据类型排序的论域等于一个由该排序基项构成的 Rocq 归纳类型。

测试器用于检查某个值是否由其构造器构造，而选择器则从这样的值中提取字段。由于所有 SMT-LIB 函数都采用全定义解释，因此选择器也定义在由其他构造器构造的值上。这些情况是未充分规定的，因此结果排序中的任意值都可以。

签名 Σ 的赋值 θ 将变量-排序对（x，τ）部分映射到 $\tau^{\mathbf{A}}$ 的元素，并且当 $x : \tau \in \Sigma$ 时，赋值在（x，τ）上有定义。项相对于结构-赋值对 $[\![t]\!]_{\mathbf{A},\ \theta}$ 进行求值。我们用 $\mathbf{A}\ \theta \models \Phi$ 表示 $\forall \varphi \in \Phi.\ [\![\varphi]\!]_{\mathbf{A},\ \theta} =$ 为真；当 Φ 是闭项时，省略 θ。

一个理论 $\mathcal{T}$ 定义一个签名 Σ 以及一组结构 $\mathbf{A} \in \mathcal{T}$ 对其进行解释。典型的 SMT-LIB 理论要么是单一模型（例如整数），要么是满足某些公理的所有结构；公理集不必是有限的或可判定的。由于 SMT 求解器使用可配置的理论集，标准也定义了理论组合。一组布尔值排序的项 Φ 是 $\mathcal{T}$-可满足的，记作 $sat_{\mathcal{T}}(\Phi)$，当且仅当存在某个 $\mathbf{A} \in \mathcal{T}$ 以及签名的赋值 θ 满足 $\mathbf{A}\ \theta \models \Phi$。

6 扩展后端：SMT-LIB 编码

ˆ现在，我们将符号执行生成的符号表达式 $E \in \mathrm{SExp}$ 与第5节的 SMT-LIB 语义联系起来；换言之，现在我们来构建 CSE 引擎的后端。我们将符号表达式、它们的值以及逻辑解释称为编码的源程序端，以区别于 SMT-LIB 端。编码必须解决四个问题：符号表达式是动态类型的，可能是部分的，可能涉及用户定义的代数数据类型和逻辑函数，而且可能使用其论域包含无法表示为源值的值的 SMT-LIB 排序进行编码。我们按如下方式应对这些挑战。第6.2 节介绍用于检查查询的背景 SMT-LIB 理论，其中声明了用户定义的代数数据类型、逻辑函数符号、用于动态值的特定代数数据类型、用于部分逻辑解释的第二种代数数据类型，以及可表示性谓词；这些谓词精确标识可表示为源值的 SMT-LIB 值。第6.3 节定义表达式编码，即一种类型导向翻译，为每个项指定一种排序，并生成侧条件，使符号表达式的部分求值在 SMT-LIB 的全语义中得到忠实体现。第6.4 节给出 SMT-LIB 定义，以限制背景理论中声明的逻辑函数符号的解释。第6.5 节证明表达式编码的正确性。最后，第6.6 节和6.7 节定义操作可满足性检查；其查询使用可表示性谓词排除包含不可表示值的 SMT-LIB 模型，并在此基础上进行状态可满足性检查和蕴涵检查。详细定义见附录C。与第4节一样，除非必要，我们省略上下文索引（例如 Δ 和 Ψ）。

6.1 以结构化类型信息扩展的符号状态

$\hat{}$ 类型信息通常只在符号状态中隐式存在（例如，若 E 属于符号$\hat{}$内存的论域，则 $E \in$ 是自然数），但后端基于类型的优化需要将其显式表示。因此，我们扩展符号状态，使其携带一个类型环境 T： LVar $\rightharpoonup_{fin}$ 类型，并在$\hat{}T$为 x 指定类型 τ时，记作 x： $\tau \in \hat{}T\ x$。我们将类型环境可满足性纳入状态可满足性 θ，即 $(s, \mu) \models (\hat{}s, \hat{}\mu, \hat{}\mathrm{P}, \hat{}\pi, \hat{}T)$，写作 $\theta \models \hat{}T \Leftrightarrow \forall x, \tau.\, \mathrm{x}: \tau \in \hat{}T \Rightarrow \exists v.\, \theta(x) = v \wedge v \in \tau$。添加$\hat{}$x： $\tau \in T$等价于向路径条件$\hat{}\pi$中添加 $x \in \tau$，因此只有当状态已经蕴含这些条目时，才能添加它们。由此， T只会记录其余状态已蕴含的类型，但不必记录所有这些类型。省略已蕴含的类型是可靠的，但会失去基于类型的优化，因此在执行过程中应将这些类型传播到 T 中，类似于 Viper [53]中的内存合并。

6.2 背景声明与动态值

我们首先定义用于检查编码查询的 SMT-LIB 理论。该背景理论在我们已机械化的 SMT-LIB 理论基础上进行了扩展，为 Δ中的声明添加了代数数据类型排序，为 Ψ中的逻辑函数添加了函数符号，并且关键的是，为动态$\hat{}$类型值添加了包装器代数数据类型排序。由于 T 中的类型信息决定了分配给逻辑变量的排序，我们将$\mathcal{T}_{\hat{T}}$ 记作所得的背景理论，并将 $\Sigma_{\hat{T}}$ 记作其签名。

Val 代数数据类型使我们能够编码动态类型的符号表达式。它为动态值可能具有的每种类型分别提供一个构造器，如图 8 所示，因此，声明为 Val 排序的变量可以表示布尔值（使用 bool 构造）、自然数（使用 nat 构造）等。每个构造器都有常规的测试器和选择器，例如用于 nat 的 isNat 和 getNat。对于 Val 排序项 t，$\mathrm{is}_\tau(t)$ 检查 t 是否具有表示 τ 的包装器形状。它通过对 τ 分情况讨论来定义（见附录 C.3 ），例如，$\mathrm{is}_{\mathrm{Nat}}(t) = \mathrm{isNat}(t)$。

对于每个用户定义的代数数据类型 δ， $\Sigma_{\hat{T}}$都会声明一个 SMT-LIB 排序 δ，并带有一个构造器 c： 排序$(\vec{\tau})$；δ每个 $c(\vec{\tau})$： $\delta \in \Delta$，以及一个动态包装器 adt_δ，用于将其转换为 Val(图 8）。SMT-LIB 要求每种数据类型至少有一个有限值，因此我们假设 Δ 中的每个代数数据类型都是良基的，即存在一个构造器，其所有参数类型都是良基的。

这里，sort(τ) 是 SMT-LIB 排序编码 τ（图 8）。其中一些排序所允许的值多于其源类型所允许的值：例如，Nat 被编码为 Int，其中包含负整数。我们通过可表示性谓词来处理这一问题；这些布尔值函数符号在 $\Sigma_{\hat{T}}$ 中声明（图 8），用于精确标识哪些 SMT-LIB 值可以表示为源值。后端使用 define-funs-rec 定义这些谓词（第5节）。例如，reprNat 的定义方程 $\forall(x : \mathrm{Int}).\ \mathrm{reprNat}\ (x) = (x \geq 0)$ 断言了非负性。同样，reprRat 对正实数及其整数商成立；reprVal 和每个 repr_δ 则递归检查构造器负载，包括列表元素。我们在 Ψ_{repr} 中汇总这些定义方程，并在附录 C.2 中完整给出。与 is_τ 类似，我们用 $\mathrm{repr}_\tau(t)$ 表示由这些符号构成的公式，该公式说明一个属于 sort(τ) 排序的项 t 是否可表示为类型为 τ 的源值；例如，$\mathrm{repr}_{\mathrm{Nat}}(t)$ 就是 reprNat(t)。

逻辑解释 θ 是部分的（第 3 节），但 SMT-LIB 赋值对于签名中声明的变量是总的。为了表示 θ 论域之外的变量，后端使用了第二个代数数据类型 MaybeVal，其构造子包括 *some* 和 $\lightning$、测试器 *isVal* 以及选择器 *getVal*（图 8）。

$\hat{}$ T启用的优化是：对于类型已知的变量，避免使用动态包装器。如果 $\hat{}\,\hat{}\,\mathrm{x}: \tau \in T$，那么对于每个满足 $\theta \models T$ 的逻辑解释 θ， $\theta(x)$ 都有定义，且类型为 τ。因此， x 可以直接声明为其原生排序 x: sort(τ) $\in \Sigma_T^{\hat{}}$，而不会排除任何此类 θ。否则，无法保证 $\theta(x)$ 有定义，因此 x 被声明为 x: MaybeVal $\in \Sigma_{\hat{T}}$。原生排序极大地简化了编码，并通过避免对 Val 和 MaybeVal 的嵌套进行代数数据类型推理来提升求解器性能。第 6.3 节中的示例展示了生成的查询有何不同。

$\Sigma_{\hat{T}}$ 中的 Val 和 MaybeVal 声明 nullVal : 空值; Val bool : 布尔值; Val nat : 整数 ; Val rat : 实数 ; Val string : 字符串 ; Val list : Seq(Val); Val adt_{δ} : δ; Val isNull : Val; 布尔值 isBool : Val; 布尔值 isNat : Val; 布尔值 . . . getNull : Val; 空值 getBool : Val; 布尔值 getNat : Val; 整数 . . . some : Val; MaybeVal $\lightning$: MaybeVal isVal : MaybeVal; 布尔值 is$\lightning$: MaybeVal; 布尔值 getVal : MaybeVal; Val

$\Sigma_{\hat{T}}$ 中的可表示性声明

reprNat: 整数; 布尔值 reprRat: 实数; 布尔值 reprVal: Val; 布尔值 repr_{δ} : δ; 布尔值 sort(τ) = τsort(Val) = Val sort(Null) =空值 sort(Bool) =布尔值 sort(Nat) =整数 sort(Rat^{+}) = 实数 sort(Str) = 字符串 sort(List(τ)) = Seq(sort(τ)) sort(δ) = δ $\hat{}$ $\hat{}$ $\hat{}$ $\hat{}$ $T \vdash E$ $t : \tau \mid \Phi$ $T \vdash E$ $_{\tau}\, t \mid \Phi \hat{T} \vdash \hat{} E$ $t : \tau \mid \Phi$ to$_{\tau}(t, \tau, \Phi)$ = $(t', \text{sort}(\tau)\ \Phi')$ $x : \tau \in \hat{\hat{T}\hat{T}} \vdash \hat{} E$ $_{\tau}\, t' \mid \Phi' \hat{} T \vdash x$ $x : \text{sort}(\tau) \mid \emptyset x \notin \text{dom}\, \hat{T}\hat{T} \vdash \hat{} E_i$ 自然数 $t_i \mid \Phi_i \hat{} \hat{} \hat{} T \vdash x$ x : MaybeVal $\mid \emptyset$ $T \vdash E_1 / E_2$ t_1 div t_2 : 整数 $\mid \Phi_1 \cup \Phi_2 \cup \{t_2 \neq 0\} \hat{} \hat{} c(\tau_1, \ldots, \tau_n)$: $\delta \in \Delta$ $T \vdash E_i$ $_{\tau_i}\, t_i \mid \Phi_i \hat{} T \vdash c(\hat{} E_1, \ldots, \hat{} E_n)\ c(t_1, \ldots, t_n) : \delta \mid \bigcup_i \Phi_i f(\vec{x} : \vec{\tau}) : \tau = \text{E}$ when $\text{E}_{pre} \in \Psi \hat{T} \vdash \hat{} E_i$ $_{\tau_i}\, t_i \mid \Phi_i \hat{} T \vdash f(\hat{} E_1, \ldots, \hat{} E_n)\ f(t_1, \ldots, t_n) : \text{sort}(\tau) \mid \{f_{pre}(t_1, \ldots, t_n)\}$ $\cup \bigcup_i \Phi_i \neg \text{Wf}\ (\tau) \hat{T} \vdash \hat{} E$ Val $t \mid \Phi$ Wf $(\tau) \hat{T} \vdash \hat{} E \in \tau$ false: 布尔值 $\mid \emptyset \hat{T} \vdash \hat{} E \in \tau \bigwedge \Phi \wedge \text{is}_{\tau}(t)$: 布尔值 $\mid \emptyset$ toNat(t, τ, Φ) = (t', τ', Φ') to$_{\tau}(t, \tau, \Phi)$ = (t', τ', Φ') toNat(t, τ, Φ) = $(t$, 整数 $\Phi)$

if τ = 整数 (getNat (t), 整数 $\Phi \cup \{\text{isNat}\ (t)\})$ if τ = Val (getNat (getVal(t)), 整数 $\Phi \cup \{\text{isVal}(t), \text{isNat}\ (\text{getVal}(t))\})$ if τ = MaybeVal undefined otherwise $\text{to}_{\text{Val}}(t, \tau, \Phi)$ = toVal(t, τ, Φ) $\text{to}_{\text{Bool}}(t, \tau, \Phi)$ = toBool(t, τ, Φ) $\text{to}_{\text{Nat}}(t, \tau, \Phi)$ = toNat(t, τ, Φ) $\text{to}_{\delta}(t, \tau, \Phi)$ = $\text{toADT}_{\delta}(t, \tau, \Phi)$ . . .

图 8. 背景声明与编码。

对于每个逻辑函数 $f(\vec{x}:\vec{\tau}):\tau = \mathrm{E}$ when $\mathrm{E}_{pre} \in \Psi$，签名中包含一个秩为 $\mathrm{sort}(\tau_1)\ldots,\mathrm{sort}(\tau_n)$; $\mathrm{sort}(\tau)$ 的函数符号 f，一个秩为 $\mathrm{sort}(\tau_1)\ldots,\mathrm{sort}(\tau_n)$; Bool 的前置条件符号 f_{pre}，以及一个与 f 具有相同秩的未解释符号 $f_\perp$。它在前置条件之外提供一个不受约束的值，同时确保 f 在第6.4节中的定义可满足。

6.3 动态类型与部分表达式的编码

^^给定一个类型环境 T 和一个符号表达式 E，表达式编码器会返回一个^^SMT-LIB 项 t、其类型 τ 以及侧条件 Φ，记作 $T \vdash E \quad t : \tau \mid \Phi$（图 8）。它确保^当侧条件 Φ 成立时，t 能够忠实地表示 E。类型 τ 采用自底向上的方式合成。值会被编码为原生字面量，例如 n 在 Int 中，而变量 x 的类型是在 $\hat{\Sigma}_{\hat{T}}$中声明的类型，即如果 $x:\tau \in T$，则为 $\mathrm{sort}(\tau)$，否则为 MaybeVal。随后，每个运算符会确定其结果的类型，例如除法会生成 Int 类型的项，并将其操作数强制转换为所需的类型（同样为 Int）。因此，该编码器是一种遵循渐进类型系统 [55]的类型导向翻译。

在渐进类型系统中，与我们的设定一样，类型信息是部分已知的，而一种专门的动态类型（Val 在我们的情形中）表示缺少类型信息。类型系统允许在静态类型使用点出现动态类型的子表达式，反之亦然，但不允许静态类型与其使用点不一致的子表达式。当类型相等或其中一个是 $\tau \sim \tau'$ 时，^ ^我们的渐进类型系统中就满足一致性。Val，我们用 $T \vdash E : \tau$ 表示类型判断（见附录C.6）。Siek 和 Taha [55] 为他们的渐进类型演算定义了强制转换插入翻译，将源表达式转换为中间语言，其中显式表示了向动态类型及从动态类型转换的强制转换。我们的编码器执行类似的转换，而我们的强制转换与这些强制转换类似，必要时在动态Val 排序与原生排序之间进行转换。插入的强制转换可能在运行时失败，但类型安全可保证此类失败会被明确报告，而不会悄然产生错误结果；相比之下，我们的强制转换会生成侧条件，排除那些会导致强制转换失败的 SMT-LIB 模型。

例如，toNat（图 8）将一个 Val 排序项 t 强制转换为 getNat (t)，其侧条件为 isNat (t)。与所有 SMT-LIB 函数符号一样，选择器 getNat 按全函数解释，因此对于并非由 nat 构造的参数，其行为未作充分规定；例如，getNat (bool(true)) 可以被解释为任意整数。侧条件 isNat(t) 排除了这类未充分规定的求值，而这些求值对应于源表达式的类型错误求值。从 Val 和 MaybeVal 中进行强制转换时，同样会应用由相应测试器守护的选择器。to$_\tau$ 将一种编码强制转换为原生排序 $\mathrm{sort}(\tau)$，并根据 τ 分派到相应的强制转换（图 8）。我们用$\hat{T} \vdash ^\hat{E} \quad _\tau\, t \mid \Phi$ 表示类型为 τ 的 E 的^^^编码：它先在 T 下对 E 进行编码，再用 to$_\tau$ 强制转换所得结果（图 8 中的第一条规则）。反过来，toVal 则强制转换到 Val。对于原生排序的项，它会将其包装在匹配的构造器中（例如，为 Int 使用 nat），且不需要侧条件；对于 MaybeVal 项，它会在侧条件 isVal(t) 下使用 getVal 将其解包。两者的完整定义见附录C.3。

编码器是一个部分函数；对于拥有多条规则的表达式形式，这些规则的前提彼此不相交。除了未声明的构造器、代数数据类型或逻辑函数之外，只有在强制转换未定义的地方，编码器才未定义。例如，1/true 没有编码，因为 true 被编码为 布尔值排序，而 toNat 在该排序上未定义。因此，编码器会在转换输入的同时对其进行类型检查。与插入强制转换一样，它对渐进类型系统接受的每个表达式都有定义。

Th ^^^^ 定理 6.1（表达式编码器全定义性）。若 $T \vdash E : \tau$，则 $\exists t, \Phi. T \vdash E \quad _\tau t \downarrow \Phi$.

侧条件还涵盖了类型错误之外的部分性，如图 8 所示的编码器代表性案例。在除法案例中，侧条件 $t_2 \neq 0$ 排除了 SMT-LIB 对除以零的欠规范求值；这对应于源表达式求值出错。应用案例也通过侧条件 $f_{pre}(t_1,\ldots,t_n)$实现了同样的效果，其中 f_{pre}是 f的前置条件符号。这是精确的，因为应用出错当且仅当其

前置条件不成立（第 4.4节）。这里既不编码 f的函数体，也不编码其前置条件：两者都只在 f和 f_{pre}的 SMT-LIB 定义中编码一次（第 6.4节）。类型成员关系ˆ案例说明了为何不能将所有表达式侧条件都在顶层断言： $E \in \tau$ 应当ˆ在 E 出错时求值为 false，而不是消除整个模型，因此这里将侧条件折叠到主项中。

为了通过一个具体示例将上述内容具体化，考虑 $\hat{}_E$ def= $(x/y = 2)$。如果 x 和 y 按 MaybeVal 排序，编码器会生成项 getNat (getVal(x)) div getNat (getVal(y)) $= 2$，其侧条件为 isVal(x) $\wedge$isNat (getVal(x)) $\wedge$isVal(y) $\wedge$isNat (getVal(y)) $\wedge$getNat(getVal(y)) $\neq 0$。如果 ˆx：自然数 y：自然数 $\in T$，则该表达式会原生编码为 $x$div $y = 2$，其侧条件为 $y \neq 0$。ˆ因此 T 在不改变语义契约的情况下，移除了编码中的动态包装器。

6.4 逻辑函数定义

逻辑函数同样使用 `define-funs-rec`进行定义。给定逻辑函数 $f(\vec{x}: \vec{\tau}): \tau = E$ 当 E_{pre} 在 Ψ 中时，其 SMT-LIB 定义生成以下定义方程：

$$\forall(\vec{x} : \text{sort}(\vec{\tau})).\ f(\vec{x}) = ite(f_{pre}(\vec{x}), t_f, f_\perp(\vec{x})), \quad \text{where } \hat{T}_f \vdash E \rightsquigarrow_\tau t_f \mid \Phi_f$$
$$\forall(\vec{x} : \text{sort}(\vec{\tau})).\ f_{pre}(\vec{x}) = (\bigwedge_i \text{repr}_{\tau_i}(x_i) \wedge \bigwedge \Phi_{pre} \wedge \varphi_{pre}), \quad \text{where } \hat{T}_f \vdash E_{pre} \rightsquigarrow_{\text{Bool}} \varphi_{pre} \mid \Phi_{pre}$$

以及 ˆ$T_f$def= $\{x_i \mapsto \tau_i\}_{i=1}^n$。在 f 的定义中（其中 ite 是 SMT-LIB 的条件表达式），侧条件 Φ_f会被舍弃：函数体的部分性由前置条件处理。对于 f_{pre}，则保留前置条件编码中的侧条件 Φ_{pre}，并将它们与编码后的前置条件φ_{pre} 以及参数上的可表示性保护条件 repr$_{\tau_i}$(x_i) 合取。为保证这些定义存在，除了 $\models \Psi$，我们还始终假设 Ψ 的类型一致： $T_f \vdash$ E : τ 且 $T_f \vdash$ E$_{pre}$: 布尔值；根据定理 6.1，两种编码均有定义。我们将生成的公式汇总于 Ψ。

Ψ 的有效性保证，只要前置条件成立，函数体就会终止且不会出错。在其他情况下，未由前置条件保护的定义方程可能没有模型（第5节）；例如， f(n : 自然数) : 自然数 $= f$(n) + 1 当 false 时，会生成矛盾的 $\forall(n : \text{整数}).\ f(n) = f(n) + 1$，它没有模型，因此每个查询都会（不可靠地）不可满足。因此，在前置条件之外，该定义会将 f 设为未解释的 $f_\perp$；而第 6.3 节的应用编码则通过 f_{pre} 保护函数调用。 f_{pre} 中的参数可表示性保护条件同样必不可少：若以前置条件 n $<$ 0 替代 false（没有自然数满足该条件），那么 $f_{pre}(-1)$ 原本会在 SMT-LIB 整数上成立，从而再次引发矛盾。

6.5 表达式编码的正确性

我们证明表达式编码器的两个事实。首先，编码器会在生成的签名中生成类型良好的 SMT-LIB 项和侧条件。

定理 6.2（表达式编码的良类型性）。假设$\mathcal{W}f(\tau)$ 对于每个 x： $\tau \in$ ˆˆT，且 $T \vdash E$ t： $\tau \mid \Phi$。那么， $\Sigma_{\hat{T}} \vdash t$： τ，并且对于所有 $\varphi \in \Phi$， $\Sigma_{\hat{T}} \vdash \varphi$： 布尔值。

其次，该编码是可靠的。为了表述我们的可靠性定理，我们通过语义等价将源值与 SMT-LIB 值联系起来：一种关系 $v \simeq u$，用于关联源值 v（或缺失值 $\lightning$）与 SMT-LIB 值 u（完整定义见附录 C.5）。对于原生排序， $v \simeq u$ 是直接的：它关联相等的布尔值、与相等整数对应的自然数，以及与相等实数对应的有理数，并逐点递归处理列表和代数数据类型。对于 Val，它会解包动态构造器。对于 MaybeVal，它同样会解包 some，并将 $\lightning$ 与 $\lightning$关联起来。我们用 $\theta \simeq \theta$ 表示逐点提升；也就是说，对每个变量 x， $\theta(x) \simeq \theta(x)$，将未定义的 $\theta(x)$ 视为 $\lightning$。

ˆ ˆ在定理 6.3中，假设条件固定了 E 的编码和求值： Ψ 有效， E 在ˆ下求值 θ得到一个值或 $\lightning$，并且 $\theta \models T$。这些条件通过 $\theta \simeq \theta$、一个 $\in \mathcal{T}_{\hat{T}}$以及一个 $\models \Psi \cup \Psi_{repr}$将 SMT-LIB 一侧与之联系起来，从而使得一个按定义解释逻辑函数和可表示性谓词。

Theo注释 6.3（表达式编码可靠性）。假设 $\models \Psi$，并令 $r, r' \in \mathrm{Val} \cup \{\lightning\}$。

$$
\begin{array}{ll}
\textit{If} & \hat{T} \vdash \hat{E} \rightsquigarrow t : \tau \mid \Phi, \quad [\![\hat{E}]\!]_\theta = r, \quad \theta \models \hat{T}, \\
 & \theta \simeq \theta, \quad \mathbf{A} \in \mathcal{T}_{\hat{T}}, \quad \mathbf{A} \models \Psi \cup \Psi_{repr}, \\
\textit{then} & (\mathrm{OX})\ r \neq \lightning \Rightarrow \mathbf{A}, \theta \models \Phi \wedge \exists u.\ [\![t]\!]_{\mathbf{A},\theta} = u \wedge r \simeq u, \ \textit{and} \\
 & (\mathrm{UX})\ \forall r'.\ \big(\mathbf{A}, \theta \models \Phi \wedge \exists u.\ [\![t]\!]_{\mathbf{A},\theta} = u \wedge r' \simeq u\big) \Rightarrow r' = r.
\end{array}
$$

证明概要。对表达式求值进行归纳；对于逻辑函数调用，使用函数归纳法，从而获得关于函数体和前置条件的假设。两个方向同时证明：复合情形有时需要用到相反方向，尤其是类型成员关系。□

OX 方向说明：如果源语言求值没有出错，那么侧条件成立，且编码后的项求值得到一个等价的值。UX 方向说明：在侧条件成立时得到的 SMT-LIB 值，对应于相同的源结果，即使它是 $\lightning$也如此，因此 SMT-LIB 模型无法为出错的表达式赋值。因此，侧条件是精确的：它们涵盖 E 的每一种求值，同时排除仅由未充分规定的选择器或未受保护的部分操作导致的见证。该定理假设求值 $[\![\hat{}E]\!]_\theta = r$会终止，而只有上下文有效性才能保证这一点；上下文有效性要求前置条件及其约束下的函数体都能终止。

6.6 表达式可满足性检查

操作可满足性检查 checkSat$_{\hat{T}}$(ˆE) 需要可表示性保护条件，使可满足的 SMT-LIB 模型对应于逻辑解释，并排除类型错误，从而实现类型优化。

考虑操作可满足性检查的 UX 健全性；给定一个 SAT 答案，我们必须 ˆ 得出结论： E 是可满足的。要将定理 6.3 应用于一个可满足模型，必须从其 SMT-LIB 赋值 θ构造出一个满足 $\theta \simeq \theta$的逻辑解释 θ。一般而言，这是不可能的，因为 θ可能会赋予不可表示的值。例如，第 6.3 节中 $x/y = 2$ 的编码并未排除对 x和 y赋予负整数，因此该查询还需要保护条件 $\mathrm{repr}_{\mathrm{Nat}}(x)$ 和 $\mathrm{repr}_{\mathrm{Nat}}(y)$。OX 方向则相反：它从一个逻辑解释 θ出发，构造一个等价的赋值，因此只需这些保护条件允许该赋值即可。

In 一般而言，该查询会为 lv(ˆE) ∪ dom ˆT 中的每个变量断言一个保护条件，并将其收集在

$$
\mathrm{guards}_{\hat{T}}(\hat{E}) \stackrel{\mathrm{def}}{=} \{\mathrm{repr}_\tau(x) \mid x : \tau \in \hat{T}\} \cup \{\mathit{isVal}(x) \Rightarrow \mathrm{repr}_{\mathrm{Val}}(\mathit{getVal}(x)) \mid x \in \mathrm{lv}(\hat{E}) \setminus \mathrm{dom}\,\hat{T}\}.
$$

ˆ位于 dom T 之外的变量声明为 MaybeVal，因此它的保护条件只约束已定义的情形： $\lightning$仍然可用，但它包装的每个值都可表示。因此，当且仅当赋值中的每个值都可表示时，该赋值才满足这些保护条件。对于 UX，这样的赋值确定了一个逻辑解释 θ，其中 $\theta \simeq \theta$将 $\lightning$读取为 $\lightning$。对于 OX，由逻辑解释构造的赋值只会赋予可表示的值，因此满足这些保护条件。

可表示性保护条件必须位于查询的顶层，以便立即排除任何不可表示的模型。它们不是表达式侧条件，因为表达式侧条件可能需要折叠到编码的布尔项中，例如第 6.3 节的类型隶属情形。

为排除基于类型的不可满足情形而无需调用底层 SMT 求解器，我们定义了渐进式判断 $T \vdash E : \tau$的一个ˆ ˆ否定对应项，它能静态识别出必然导致类型错误的表达式。遵循双向类型错误推理 [49],$\hat{T} \dashv \hat{}E : \tau$ˆ ˆ 表明， E不可能求值为类型为 τ的值（注意，这并不只是 $T \vdash E : \tau$ 的否定）。ˆ ˆ因此，如果 $T \dashv E :$ 布尔值，可满足性检查无需调用求解器即可返回不可满足。

定理 6.4（⊣的完备性）。 $\theta \models \hat{T} \wedge \hat{T} \dashv \hat{}E : \ \tau \wedge [\![\hat{}E]\!]_\theta = v \Rightarrow v \notin \tau$

现在我们可以定义操作检查 checkSat$_{\hat{T}}$($\hat{}E$)：它会构造一个布尔查询，并调用符合 SMT-LIB 标准的求解器：

$$\text{checkSat}_{\hat{T}}(\hat{E}) = \begin{cases} \mathsf{UNSAT} & \text{if } \hat{T} \dashv \hat{E} : \text{Bool or } \exists x : \tau \in \hat{T}.\ \neg \mathcal{W}f(\tau) \\ \mathsf{solve}(\mathcal{T}_{\hat{T}}, \{\varphi\} \cup \Phi \cup \Psi & \text{if } \hat{T} \vdash \hat{E} \rightsquigarrow_{\text{Bool}} \varphi \mid \Phi \\ \qquad \cup\ \Psi_{repr} \cup \text{guards}_{\hat{T}}(\hat{E})) & \\ \mathsf{UNKNOWN} & \text{otherwise.} \end{cases}$$

⊣第一个分支的条件是 ⊣启用的优化：当 $\hat{}$ 类型系统证明 E 无法求值为布尔值时，它可以避免调用求解器。其良构性条件会拒绝格式错误的类型环境：格式错误的类型不可被实例化（第 4.2 节），因此没有 θ 满足 $\theta \models \hat{}T$。它还确保第二个分支满足定理 6.2 的前提，从而使求解器收到排序正确的查询。第二个分支会断言编码后的布尔公式、侧条件、逻辑函数定义、可表示性谓词定义以及可表示性保护条件。solve(T Φ) 可能返回 SAT、UNSAT 或 UNKNOWN。我们只假设其符合标准：对于排序正确的布尔查询，SAT 答案意味着 sat T (Φ)，而 UNSAT 答案意味着 $\neg\text{sat}_{\mathcal{T}}(\Phi)$（第 5 节）。[11]

引理 6.5（模型存在性）。如果 $\models \Psi$，那么存在**一个 A** $\in \mathcal{T}_{\hat{T}}$ 满足**A** $\models \Psi \cup \Psi_{repr}$。

定理 6.6（可满足性检查可靠性）。

$$\text{(OX)} \quad [\![\hat{E}]\!]_\theta = \text{true} \wedge \theta \models \hat{T} \ \Rightarrow\ \text{checkSat}_{\hat{T}}(\hat{E}) \in \{\mathsf{SAT}, \mathsf{UNKNOWN}\}$$
$$\text{(UX)} \quad \text{checkSat}_{\hat{T}}(\hat{E}) = \mathsf{SAT} \ \Rightarrow\ \exists \theta.\ [\![\hat{E}]\!]_\theta = \text{true} \wedge \theta \models \hat{T}.$$

证明概要。利用定理6.2 至6.4 以及求解器的合规性。对于 OX，取引理6.5中的模型。一个逻辑解释 θ 若能见证可满足性，就会导出一个 SMT-LIB 赋值 θ，且$\theta \simeq \theta$满足可表示性保护条件；随后定理6.3 给出编码后的公式和侧条件，从而得到可满足性。对于 UX，针对 lv($\hat{}E$) ⊔dom $\hat{}T$ 的保护条件排除了模型中不可表示的ˆ值，因此任何满足条件的模型都确定了一个 θ，其中有 $\theta \models T$ 和 $\theta \simeq \theta$；应用定理6.3 即可将布尔值结果传回源程序一侧。□

6.7 状态可满足性与蕴含关系

St可满足性与蕴含关系都建立在表达式检查 checkSat$_{\hat{T}}$($\hat{}E$) 之上 .

我们将 checkSat 重载用于状态可满足性：状态检查 checkSat(m, $\hat{}\sigma$) 接受一个模式 m 和一个符号状态 $\hat{}\sigma = (\hat{}s, \hat{}\mu, \text{P}, \hat{}\pi, T)$。它会对路径条件 $\hat{}\pi$ 与存储和内存中隐含的约束合取后，调用表达式检查 checkSat $\hat{}_T$。类型环境本身不需要此类约束，因为 checkSat$_{\hat{T}}$ 已经对其进行了约束：根据定理 6.6，可满足的答案会给出一个 θ，满足 $\theta \models \hat{}T$。当且仅当符号存储可满足时，存储约束 $\hat{}\pi_{\hat{s}}$ 才成立，即 $[\![\hat{}\pi_{\hat{s}}]\!]_\theta = \text{true} \Leftrightarrow \exists s.\ \theta, s \models_{\text{Sto}} \hat{}s$，其定义为$\hat{\pi}_{\hat{s}}\text{def}= \bigwedge_{\hat{}E \in \text{rng}(\hat{}s)} \hat{}E \in \text{Val}$。内存约束 $\hat{}\pi_{\hat{\mu}}$ 特定于内存模型，内存模型必须对其进行定义，以满足类似的约定$[\![\hat{}\pi_{\hat{\mu}}]\!]_\theta = \text{true} \Leftrightarrow \exists \mu.\ \theta, \mu \models_{\text{Mem}} \hat{}\mu$。对于第 3 节中的线性堆，$\hat{}\mu = \{\hat{}E_a^1 \mapsto \hat{}E_c^1, \ldots, \hat{}E_a^n \mapsto \hat{}E_c^n\}$ 表示单例堆的不相交并，因此其约束为

$$\hat{\pi}_{\hat{\mu}} \stackrel{\text{def}}{=} \bigwedge \{\hat{E}_a^i \neq \hat{E}_a^j \mid 1 \le i < j \le n\} \wedge \bigwedge \{\hat{E}_a^i \in \text{Nat} \mid 1 \le i \le n\} \wedge \bigwedge \{\hat{E}_c^i \in \text{Val} \mid 1 \le i \le n\}.$$

[11]遵循我们理论的 CSE 工具会继承这一假设，但不必盲目信任它。可满足的回答所附的模型可以提升为逻辑解释，并针对源表达式进行检查；不可满足的回答则可以附带证明，例如由 Carcara [1]检查的 cvc5 [4] Alethe 证明 [52]，。

```
pred CRawEncryptionContext(+buffer : 列表, ECKs : 列表) : (0i i<#  |buffer|) * (#rawKC  ==
l-sub(buffer, 0i, 2i)) * rawToUInt16(#rawKC, false, #keyCount) * (0i i<# #keyCount) * Elements("
Complete", buffer, 2i, #keyCount, 2i, ECKs, #ECKsLength) * toUtf8PairMap(ECKs, #utf8ECKs) *
FirstProj(ECKs, #ECKeys) * Unique(#ECKeys) * ((2i i+  #ECKsLength)  ==  |buffer|);
```

图 9. Gillian 代码库中的加密上下文谓词，经过轻微重新排版。参见文件 Gillian-C/examples/amazon/logic/EncryptionHeaderLogic.gil。

剩下的近似点是折叠谓词。在 OX 模式下，可以忽略谓词，因为 OX 健全性只要求不剪除可行执行：忽略谓词只会漏掉状态中的矛盾，绝不会引入矛盾。相反，在 UX 模式下，可满足性检查必须以一个真实状态为见证，因此只要仍有折叠谓词，检查就会返回未知，因为它们的函数体可能包含检查无法看到的约束，从而使整个状态不可满足：

$$\mathrm{checkSat}(m, \hat{\sigma}) = \begin{cases} \mathsf{UNKNOWN} & \text{if } m = \mathsf{UX} \text{ and } \hat{\mathcal{P}} \neq \emptyset \\ \mathrm{checkSat}_{\hat{\Gamma}}(\hat{\pi} \wedge \hat{\pi}_{\hat{s}} \wedge \hat{\pi}_{\hat{\mu}}) & \text{otherwise.} \end{cases}$$

仍然可以通过在状态检查之前展开折叠谓词来利用它们，例如，将列表谓词展开为空与非空情形，直到某个分支揭示出足够的纯约束。

一种简单的状态蕴含检查实现 checkEntails(ˆσ, ˆE) 会在 ˆσ 的类型环境 T 下检查：checkSat$\hat{\tau}$(ˆ$\pi \wedge \neg(E \in$ 布尔值)) = 不可满足，因此 E 必然是布尔值；并且 ˆ checkSat$\hat{\tau}$(ˆ$\pi \wedge \neg E$) = 不可满足，因此它不可能为假。

定理 6.7（状态检查可靠性）。对于每种分析，假设都支持内存模型：

$$\begin{aligned} &(\mathrm{OX}) && sat(\hat{\sigma}) \Rightarrow \mathrm{checkSat}(\mathsf{OX}, \hat{\sigma}) \in \{\mathsf{SAT}, \mathsf{UNKNOWN}\} \\ &(\mathrm{UX}) && \mathrm{checkSat}(\mathsf{UX}, \hat{\sigma}) = \mathsf{SAT} \Rightarrow sat(\hat{\sigma}) \\ &(\text{Entailment}) && \mathrm{checkEntails}(\hat{\sigma}, \hat{E}) \Rightarrow \hat{\sigma} \models \hat{E}. \end{aligned}$$

如第 2 节所述，状态可满足性检查贯穿整个符号执行过程。为了证明这类检查是可靠的，我们将其建模为幽灵命令，从而使其可以在任意时刻执行：基本命令 ::= ... | `check pc` | `check state`。幽灵命令 `check pc` 仅检查 checkSat$\hat{\tau}$(ˆπ)；`check state` 则调用完整状态检查 checkSat(m, ˆσ)。

定理 6.8（幽灵检查健全性）。幽灵命令 `checkpc` 和 `checkstate` 在 *OX* 和 *UX* 下都可靠，也就是说，它们在具体执行中都对应于空操作。

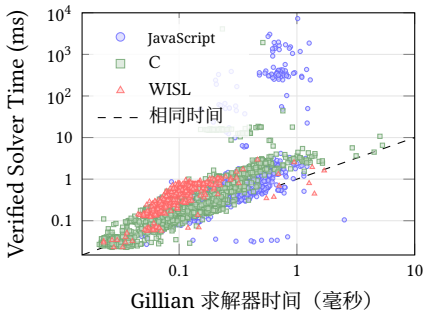
7 实证评估

到目前为止，我们的评估都是数学性的。数学证明可以回答这样一个问题：“我们所建模的 *CSE* 引擎是否可靠？”但它无法回答“建模的 *CSE* 引擎与现实世界中的 *CSE* 工具有何关联？”我们必须通过实证评估来回答这个问题。

7.1 前端评估

我们设计的理论前端能够直接、清晰地映射到现有 CSE 工具的重要特性。代数数据类型、逻辑函数、引理，以及将堆结构与其数学抽象联系起来的用户定义谓词，这些要素结合起来，能让我们实现很多目标，效果或许出人意料。我们提供了三方面的证据，表明我们的前端涵盖了实际工具前端中一个实用的片段。这并非完整的前端模型：工具使用的一些构造（例如集合）仍未纳入其中。

证据 *1*：我们的 *CSE* 理论到 *VeriFast*。为我们的理论编写的示例很容易转换为真实的 CSE 工具：例如，图 2 可直接对应到 VeriFast（见附录 D）。



语言	Runs	Qs	Tr.	Cov.	一致
JS	16	1324	985	74%	923/937
C	7	4073	2989	73%	2988/2988
WISL	9	423	416	98%	416/416
总计	32	5820	4390	75%	4327/4341

“翻译”表示桥接器翻译的查询数；“一致”表示达成一致的数量两个后端均已作答的查询中的答案。

图 10. Gillian 后端比较的数据。散点图展示了两个后端均已作答的全部 4,341 个查询的求解器时间；两轴均采用对数刻度，靠近虚线的点表示两种编码下的求解器时间相同。表格给出了相应的查询覆盖率和一致计数。

证据 *2*：*VeriFast* 到我们的 *CSE* 理论。借助代数数据类型，我们的理论能够验证许多数据结构及其算法，正如 VeriFast 的案例研究所示。例如，有序_二叉树.c 验证了有序二叉树及其操作。我们已将该示例中验证其插入函数所需的部分转换为我们的理论（见附录D）。

证据 *3*：我们的 *CSE* 理论与现实世界验证案例研究。我们的前端也能扩展到现实世界的案例研究，正如我们通过 Gillian 对 AWS Encryption SDK 消息头反序列化模块的案例研究所展示的那样。[36], 这是 Gillian 最大的案例研究，涵盖一个 JavaScript（≈200 代码行）实现和一个 C语言（≈1 千代码行）实现，并配有 ≈3.5 千代码行的规约、谓词、引理等。除了集合之外，我们的理论具备该研究所需的全部前端特性。像这种规模的研究通常会逐层构建抽象，每一层都使用我们前端的特性。例如，图9中为 C 语言研究定义加密上下文的谓词，建立在较低层级的谓词之上，例如`Unique`，它说明其列表参数不含重复项，而它自身也建立在更低层级的谓词之上。

7.2 后端评估

为了评估后端，我们使用生产级符号执行引擎 Gillian 生成求解器查询。我们扩展了 Gillian，使其将每个 SMT 查询发送到两个后端：它自己的 SMT-LIB 编码器，以及我们提取到 OCaml 中的经验证的编码器。Gillian 的分析继续依据其自身后端的答案进行，因此经验证的后端只进行观察。一个桥接器将 Gillian 的符号表达式转换为我们理论中的符号表达式。Gillian 的大多数表达式在我们的理论中都有直接对应项。只有编码器经过提取；桥接器以及将编码后的表达式发送给求解器的 OCaml 代码均未经验证。这段代码省略了第 6.6 节中的可表示性保护条件，因为 Gillian 的浮点数包含零和负数；加入这些保护条件后，比较衡量的将是这种差异，而非编码本身（见附录 D.4）。

我们运行了 32 个 Gillian 示例和案例研究，涵盖 JavaScript、C语言和 WISL（Gillian 的演示语言），其中包括 AWS 案例研究（第 7.1 节）。我们记录了两个后端的答案和求解器运行时间，使用 Z3 4.15.4，并且只对其 `check-sat` 调用进行计时。

图 10 总结了本次实验的结果。Gillian 生成了 5,820 个可满足性模理论查询，其中桥接器翻译了 4,390 个（75.4%），涵盖全部三种语言。未翻译的 1,430 个查询中，大多数包含负整数字面量，因为 Gillian 会将减法规范化为 $a-b \;\; a+(-1)\times b$，而我们的理论中的整数实际上是自然数；其余查询则使用了我们的理论不支持的量词、集合或列表操作。经验证的编码器拒绝了已翻译查询中的 48 个，这些查询全部来自 JavaScript AWS 案例研究，它们对由列表长度得出的整数进行浮点数比较；这种类型在我们的理论中并不一致，因此超出了编码器的论域（定理 6.1）。

两个后端都回答了剩余 4,342 个查询中的 4,341 个，并且其中 4,327 个的结果一致：双方均判定 2,767 个查询可满足、1,560 个查询不可满足。14 个结果不一致的查询均被 Gillian 判定为可满足、被经验证的后端判定为不可满足；这反映的是表达式语言之间的差异，而非编码之间的差异：两者都会将整数 0 强制转换为浮点数，但我们理论中的有理数为正数，因此从自然数到有理数的相应强制转换在 0 处未定义，其编码还带有要求参数为正数的侧条件。若不加此条件，所有结果不一致的查询都可满足。

大多数查询的求解器运行时间相当：使用 Gillian 的编码时，中位数为 0.09 毫秒；使用经验证的编码时，中位数为 0.16 毫秒。例外是少数 67 个查询（1.5%），均为可满足性查询，且全部来自 AWS 案例研究；使用经验证的编码时，这些查询耗时超过 100 毫秒（最长达 7.3 秒），而使用 Gillian 的编码时，耗时至多为 1.3 毫秒。经验证的编码器会精确编码列表长度，因此要判定可满足性，求解器必须构造列表；而 Gillian 的编码器会将某些长度编码为未解释函数，只有在 Gillian 引擎所维护的不变式成立时，这种做法才是可靠的。若像 Gillian 的编码器那样编码长度，这些查询的耗时便与其余查询一样（见附录D.4）。

因此，提取出的编码器与 Gillian 的求解器循环兼容：它能够处理四分之三的实际验证查询；除表达式语言含义不同的情况外，其结果与 Gillian 的后端一致；除少数原因已知的查询外，求解器时间也相当。

8 相关工作

前端工作。我们的前端涵盖了现有未经验证工具中的许多 CSE 特性，但并非全部。值得注意的缺失功能包括 Viper 的魔法 wand [12] 和迭代分离合取 [39]。我们认为，这两项功能都可以基于我们的理论实现。若要将理论扩展到并发，则需要进行更大幅度的修改。现有方法要么将并发验证简化为顺序验证（Dardinier 等人已对这种方法进行了机械化实现） [13], ，要么原生处理并发，例如尚未机械化的 Raven 验证器 [25]。与 VeriFast、Viper 和 Gillian 等独立工具不同，有些工具嵌入在交互式定理证明器中：F 中的 Steel [21]和 Pulse [15]，Rocq 中的 RefinedC [51]和 RefinedRust [22] ，以及分离逻辑之外、Lean 中的 Loom [24] 。这些工具可以复用宿主交互式定理证明器中的数学基础，包括代数数据类型和逻辑函数，而不必像我们在本文中这样重新构建。

SMT-LIB 相关工作。据我们所知，此前尚无对完整 SMT-LIB 标准进行逐行机械化的工作。我们所知唯一专门针对 SMT-LIB 的机械化工作是 Lotz 等人 [35],针对 Strings 的工作。其他 SMT-LIB 机械化工作都是相关项目的副产品，而这些项目只构建自身所需的片段。其中最重要的是 SMTCoq [2], ，这是一个用于基于 SMT 的证明自动化的 Rocq 插件；其 SMT-LIB 机械化仅作了简要描述，并将理论集合固定下来，而不支持通用的理论加载。规模较小的例子是 CoqQFBV [54], ，它仅涵盖定长位向量。据我们所知，目前没有经过验证的通用 SMT 求解器；输入格式简单得多的经过验证的可满足求解器则已相当成熟（例如 IsaSAT [8, 18, 19]）。生成证明的 SMT 求解器（如 Z3 和 cvc5）十分常见，已有 LFSC [58], 、Alethe [52], 和 eDRAT [26], 等证明格式，以及 Carcara [1]等独立检查器；但它们都没有针对 SMT-LIB 建立机械化的元理论。Isabelle 的 Sledgehammer [7], 、Rocq 的 SMTCoq [2], 和 Lean 的 lean-smt [38]也将 SMT 推理集成到了交互式定理证明器中。大多数工具使用交互式定理证明器逻辑中的证明重构，而非机械化的 SMT-LIB 语义；SMTCoq 则采用计算反射，其检查器已通过针对自身 SMT-LIB 机械化的验证。

后端工作。 鉴于目前还没有规模足够大、可用于 CSE 工具的 SMT-LIB 机械化工作，先前没有任何工作对 CSE 工具的 SMT 编码进行过机械化。除 CSE 外，对未经验证的 SMT-LIB 编码最全面的描述是 F* 的 [17]。“Destabilising” Iris [57] 为将 Iris 与 SMT 求解器连接奠定了基础，但并未对 SMT-LIB 本身建模；而 Loom [24]使用 Lean 自动化，其中部分由 SMT 驱动。验证条件生成器已得到机械化，但

SMT-LIB 连接：Parthasarathy 等人 [45] 给出了一个从 Viper 到 Boogie 的证明生成器，而 Nezamabadi 等人 [41] 给出了一个从 Dafny 到 Dafny 表达式的经验证生成器。

9 结论

我们介绍了朝着构建一种兼具基础性、现实性和端到端特性的 CSE 理论所取得的机械化进展。在 Lööw 等人的 CSE 理论基础上 [34], ，我们为其 CSE 引擎前端扩展了用于真实世界程序分析的丰富特性，并通过将其正确性已获证明的编码转换为 SMT-LIB，将后端连接到标准 SMT 求解器；这一工作建立在我们对 SMT-LIB 标准逐行进行机械化的基础之上。我们认为，未来有两个直接的研究方向。第一是可执行性：目前只有 SMT-LIB 编码可执行，而 CSE 引擎由推理规则描述，且逻辑函数上下文的有效性是被假定的，而非经过检查。第二是并发，这将需要进一步扩展前端特性，例如加入不变式。

研究中 AI 使用情况披露

我们使用 Claude（Anthropic）在 Rocq 机械化中开发证明并编写求值代码。作者编写或仔细审阅了所有定义和顶层定理陈述，且有一位作者审阅了求值代码。Rocq 会检查所有证明，我们也审查了其中的公理：它们仅依赖于第1节所列的经典公理。

数据可用性声明

我们的制品 [59] 已存档于 Figshare。它包含 Rocq 机械化实现，以及第 7 节求值所用的 VeriFast 翻译和内置经验证后端的 Gillian 分支。我们的 SMT-LIB 机械化实现也单独以软件包 `rocq-smt-lib` 的形式发布在 Rocq Packages 上：https://rocq-prover.org/p/rocq-smt-lib。

致谢

我们感谢 Shing Hin Ho 和 Alyssa Renata 指引我们了解塔斯基宇宙，它们用于我们的 SMT-LIB 机械化实现；也感谢 POPL 匿名审稿人帮助我们将表达式编码器理解为一种类型导向翻译。本研究得到了 Gardner 通过 Meta 提供的教师捐赠，以及伦敦帝国学院计算系为 Tamil Kumaran 提供的博士奖学金资助。

参考文献

[1] Bruno Andreotti、Hanna Lachnitt 和 Haniel Barbosa。2023。Carcara：一种用于 Alethe 格式 SMT 证明的高效证明检查器与 elaborator。载于 国际系统构造与分析工具及算法会议 *(TACAS)*。 doi: 10 1007 978 3 031 30823 9_19 。/ - - - -Michael Armand 、 Germain Faure 、 Benjamin Grégoire 、 Chantal Keller 、 Laurent Théry 和 Benjamin Werner。[2] 2011。通过证明见证将 SAT/SMT 求解器模块化集成到 Coq 中。载于 国际认证程序与证明会议 *(CPP)*。 doi: 10 1007 978 3 642 25379 9_12 。 / - - - -Roberto Baldoni、Emilio Coppa、Daniele Cono D'elia、[3] Camil Demetrescu 和 Irene Finocchi。2018。符号执行技术综述。 *ACM* 计算调查 51, 3 2018 10 1145 3182657 ( )。doi:./Haniel Barbosa、Clark W. Barrett、[4] Martin Brain、Gereon Kremer、Hanna Lachnitt、Makai Mann、Abdalrhman Mohamed、Mudathir Mohamed、Aina Niemetz、Andres Nötzli、Alex Ozdemir、Mathias Preiner、Andrew Reynolds、Ying Sheng、Cesare Tinelli 和 Yoni Zohar。2022。cvc5：一种通用且具备工业级实力的 SMT 求解器。载于 国际系统构造与分析工具及算法会议 *(TACAS)*。 doi: 10 1007 978 3 030 99524 9_24 。 / - - - -Clark [5] Barrett、Pascal Fontaine 和 Cesare Tinelli。n. d.。[ ] 理论模可满足性库 (SMT-LIB)。https://smt-lib.org。 [6] Clark Barrett、Pascal Fontaine 和 Cesare Tinelli。2026。*SMT-LIB* 标准：*2.7* 版。技术报告。计算机科学系，爱荷华大学。https://smt-lib.org/papers/smt-lib-reference-v2.7-r2026-09-09.pdf 2026年9月9日发布。[7] Jasmin Christian Blanchette、Sascha Böhme 和 Lawrence C. Paulson。2013。使用 SMT 求解器扩展 Sledgehammer。 自动推理期刊 51, 1 (2013)。doi:10.1007/s10817-013-9278-5

[8] Jasmin Christian Blanchette、Mathias Fleury、Peter Lammich 和 Christoph Weidenbach。2018。具备学习、遗忘、重启和增量性的经验证 SAT 求解器框架。《自动推理期刊》61, 1–4 (2018)。doi:10.1007/s10817-018-9455-7[9] Cristiano Calcagno、Dino Distefano、Peter W. O'Hearn 和 Hongseok Yang。2011。利用双向溯因进行组合式形状分析。《美国计算机学会期刊》58, 6 (2011)。doi:10.1145/2049697.2049700[10] Arthur Charguéraud。2012。局部无名表示。《自动推理期刊》49, 3 (2012)。doi:10.1007/s10817-011-9225-2[11] David Thrane Christiansen。2023。宇宙设计模式。https://leanprover.github.io/functional_编程_使用_Lean/dependent-types/universe-pattern.html [12] Thibault Dardinier、Gaurav Parthasarathy、Noé Weeks、Peter Müller 和 Alexander J. Summers。2022。魔法棒的可靠自动化。载于《国际计算机辅助验证会议》(CAV)。doi: 10 1007 978 3 031 13188 2_7 。 / - - - - Thibault [13] Dardinier、Michael Sammler、Gaurav Parthasarathy、Alexander J. Summers 和 Peter Müller。2025。翻译式分离逻辑验证器的形式化基础。《ACM 程序设计语言会议录》 9, 2025 10 1145 3704856 POPL ( )。doi: . / [14] Leonardo de Moura 和 Nikolaj Bjørner。2008。Z：一种高效的 SMT 求解器。载于《国际系统构造与分析工具及算法会议》(TACAS)。doi:10.1007/978-3-540-78800-3_24[15] Gabriel Ebner、Guido Martínez、Aseem Rastogi、Thibault Dardinier、Megan Frisella、Tahina Ramananandro 和 Nikhil Swamy。2025。PulseCore：一种用于依赖类型程序的非谓述并发分离逻辑。《ACM 程序设计语言会议录》 9, 2025 10 1145 3729311 PLDI ( )。doi: . / Marco Eilers 和 Peter Müller。。Nagini：一种用于 Python 的静态验证器。载于《国际计算机辅助验证会议》(CAV)。doi: [16] 2018 10 1007 978 3 319 96145 3_33 。 / - - - - F* 开发者。n. d.。了解 F* 如何使用 Z。https://fstar-lang.org/tutorial/book/under_the_hood/uth_smt.html.[17] [ ] 3 [18] Mathias Fleury。2019。优化经验证的 SAT 求解器。载于 NASA形式化方法研讨会 (NFM)。doi:10.1007/978- 3-030-20652-9 10_[19] Mathias Fleury 和 Peter Lammich。2023。为 IsaSAT 实现更务实的 CDCL 并面向 LLVM（短文）。载于国际自动演绎会议 (CADE)。doi: 10 1007 978 3 031 38499 8_12 。 / - - - - José Fragoso Santos、Petar [20] Maksimović、Daiva Naudžiuniene、Thomas Wood 和 Philippa Gardner。2017。JaVerT：JavaScript 验证工具链。《ACM 程序设计语言会议录》2, POPL (2017)。doi:10.1145/3158138[21] Aymeric Fromherz、Aseem Rastogi、Nikhil Swamy、Sydney Gibson、Guido Martínez、Denis Merigoux 和 Tahina Ramananandro。2021。Steel：依赖类型并发分离逻辑中的证明导向编程。《ACM 程序设计语言会议录》 5, 2021 10 1145 3473590 ICFP ( )。doi: . / Lennard Gäher、Michael Sammler、Ralf Jung、[22] Robbert Krebbers 和 Derek Dreyer。2024。RefinedRust：一种用于 Rust 程序高保证验证的类型系统。《ACM 程序设计语言会议录》 8, 2024 10 1145 3656422 PLDI ( )。doi: . / Andreas [23] Gal、Brendan Eich、Mike Shaver、David Anderson、David Mandelin、Mohammad R. Haghighat、Blake Kaplan、Graydon Hoare、Boris Zbarsky、Jason Orendorff、Jesse Ruderman、Edwin W. Smith、Rick Reitmaier、Michael Bebenita、Mason Chang 和 Michael Franz。2009。面向动态语言的基于轨迹的即时类型特化。载于编程语言设计与实现会议 (PLDI)。doi: 10 1145 1542476 1542528 。 / . Vladimir Gladshtein、George [24] Pirlea、Qiyuan Zhao、Vitaly Kurin 和 Ilya Sergey。2026。基础性多模态程序验证器。《ACM 程序设计语言会议录》 10, 2026 10 1145 3776719 POPL ( )。doi: . / Ekanshdeep Gupta、Nisarg Patel 和 [25] Thomas Wies。2025。Raven：一种基于 SMT 的并发验证器。载于《国际计算机辅助验证会议》(CAV)。doi: 10 1007 978 3 031 98668 0_4 。 / - - - - S Hitarth、Cayden Codel、Hanna Lachnitt 和 [26] Bruno Dutertre。2024。将 DRAT 扩展到 SMT。载于计算机辅助设计形式化方法会议 (FMCAD)。doi: 10 34727 2024 978 3 85448 065 5_8 。 / /isbn. - - - - Bart Jacobs、Jan Smans、Pieter [27] Philippaerts、Frédéric Vogels、Willem Penninckx 和 Frank Piessens。2011。VeriFast：一种适用于 C语言和 Java 的强大、可靠、可预测且快速的验证器。载于 NASA形式化方法研讨会 (NFM)。doi:10.1007/978- 3-642-20398-5 4_[28] Bart Jacobs、Frédéric Vogels 和 Frank Piessens。2015。轻量级 VeriFast。《计算机科学中的逻辑方法》 11, 3 2015 10 2168 11 3 19 2015 ( )。doi: . /LMCS- (:) Quang Loc Le 、 Azalea Raad 、 Jules Villard 、 Josh Berdine 、 Derek Dreyer [29] 和 Peter W. O'Hearn。2022。利用不正确性逻辑在大型程序中发现真实缺陷。《ACM 程序设计语言会议录》 6, 1 2022 10 1145 3527325 OOPSLA ( )。doi: . / Xavier [30] Leroy。1992。非装箱对象与多态类型。载于编程语言原理研讨会 (POPL)。doi:10.1145/143165.143205 [31] 2009 52, 7 2009 10 1145 1538788 1538814 Xavier Leroy。。现实编译 [32] 器的形式化验证。《美国计算机学会通讯》( )。doi: . / . Andreas Lööw、Daniele Nantes-Sobrinho、Sacha-Élie Ayoun、Caroline Cronjäger、Petar Maksimović 和 Philippa Gardner。2024。用于正确性与不正确性推理的组合式符号执行。载于欧洲面向对象编程会议 (ECOOP)。doi:10.4230/LIPIcs.ECOOP.2024.25

[33] Andreas Lööw、Daniele Nantes-Sobrinho、Sacha-Élie Ayoun、Petar Maksimović 和 Philippa Gardner。2024。组合推理中的框架推断匹配计划。载于欧洲面向对象编程会议（ECOOP）。doi: 10 4230 2024 26 。/ LIPIcs.ECOOP。 . Andreas Lööw、Seung Hoon Park、Daniele Nantes-Sobrinho、Sacha-Élie Ayoun、Opale Sjöstedt 和 Philippa Gardner。2025。面向未来700种内存模型的组合式符号执行。ACM 程序设计语言会议录 9，OOPSLA2（2025）。doi:10.1145/3763151[35] Kevin Lotz、Mitja Kulczynski、Dirk Nowotka、Danny Bøgsted Poulsen 和 Anders Schlichtkrull。2023。经过验证的验证：Isabelle 中用于字符串的 SMT-LIB。载于国际自动机实现与应用会议（CIAA）。doi:10.1007/978-3-031-40247-0 15_[36] Petar Maksimović、Sacha-Élie Ayoun、José Fragoso Santos 和 Philippa Gardner。2021。Gillian，第二部分：JavaScript 和 C 的现实世界验证。载于国际计算机辅助验证会议（CAV）。doi:10.1007/978-3-030-81688-9 38_[37] Per Martin-Löf。1984。直觉主义类型论。证明论研究，第1卷。Bibliopolis，那不勒斯。Giovanni Sambin 笔记。[38] Abdalrhman Mohamed、Tomaz Mascarenhas、Harun Khan、Haniel Barbosa、Andrew Reynolds、Yicheng Qian、Cesare Tinelli 和 Clark Barrett。2025。lean-smt：用于在 Lean 中完成证明目标的 SMT策略。载于国际计算机辅助验证会议（CAV）。doi: 10 1007 978 3 031 98682 6_11 。/ - - - - Peter Müller、Malte Schwerhoff 和 Alexander J. Summers。 。使用符号执行自动验证迭代分离合取。载于国际计算机辅助验证会议（CAV）。doi:10. 1007/978-3-319-41528-4 22_[40] Peter Müller、Malte Schwerhoff 和 Alexander J. Summers。2016。Viper：用于基于权限的推理的验证基础设施。载于国际验证、模型检测与抽象解释会议（VMCAI）。doi:10.1007/978-3-662-49122-5 2_[41] Daniel Nezamabadi、Magnus O. Myreen 和 Yong Kiam Tan。2026。Dafny 的经验证VCG和经验证编译器。载于国际认证程序与证明会议（CPP）。doi: 10 1145 3779031 3779092 。/ 。Tobias Nipkow。 。递归过程与无界非确定性的霍尔逻辑。载于国际计算机科学逻辑研讨会（CSL）。doi: 10 1007 3 540 45793 3_8 。/ - - - Peter W. O’Hearn。 。不正确性逻辑。ACM 程序设计语言会议录 4， 2019 10 1145 3371078 POPL（ ）。doi： 。/ Peter W. O’Hearn、John C. Reynolds 和 Hongseok Yang。 。关于修改数据结构的程序的局部推理。载于国际计算机科学逻辑研讨会（CSL）。doi: 10 1007 3 540 44802 0_1 。/ - - - Gaurav Parthasarathy 、 Thibault Dardinier 、 Benjamin Bonneau 、 Peter Müller 和 Alexander J. Summers。2024。迈向可信的自动程序验证器：对中间验证语言中的翻译进行形式化验证。ACM 程序设计语言会议录 8， 2024 10 1145 3656438 PLDI（ ）。doi： 。/ Christine Paulin-Mohring。 。Coq系统中的归纳定义：规则与性质。载于国际类型化λ演算及其应用会议（TLCA）。doi: 10 1007 0037116 。/BFb Christopher Pulte、Dhruv C. Makwana、Thomas Sewell、Kayvan Memarian、Peter Sewell 和 Neel Krishnaswami。2023。CN：使用分离逻辑精化类型验证系统C代码。ACM 程序设计语言会议录 7，POPL（2023）。doi:10.1145/3571194[48] Azalea Raad、Josh Berdine、Hoang-Hai Dang、Derek Dreyer、Peter O’Hearn 和 Jules Villard。2020。关于程序错误存在性的局部推理：错误性分离逻辑。载于国际计算机辅助验证会议（CAV）。doi: 10 1007 978 3 030 53291 8_14 。/ - - - - Steven Ramsay 和 Charlie Walpole。 。类型错误的程序不会求值。ACM 程序设计语言会议录 8，POPL（ 2024 2024 10 1145 3632909 ）。doi： 。/ John C. Reynolds。 。分离逻辑：一种用于共享可变数据结构的逻辑。载于计算机科学逻辑研讨会（LICS）。doi: 10 1109 2002 1029817 。/LICS。 。Michael Sammler、Rodolphe Lepigre、Robbert Krebbers、Kayvan Memarian、Derek Dreyer 和 Deepak Garg。2021。RefinedC：使用精化所有权类型自动化 C 代码的基础验证。载于编程语言设计与实现会议（PLDI）。doi: 10 1145 3453483 3454036 。/ 。Hans-Jörg Schurr、Mathias Fleury、Haniel Barbosa 和 Pascal Fontaine。 。Alethe：迈向通用 SMT 证明格式（扩展摘要）。载于定理证明证明交换研讨会（PxTP）。doi: 10 4204 336 6 。/EPTCS。 。Malte H. Schwerhoff。 。通过符号执行推进自动化的基于权限的程序验证。博士论文。苏黎世联邦理工学院。doi:10.3929/ethz-a-010835519[54] Xiaomu Shi、Yu-Fu Fu、Jiaxiang Liu、Ming-Hsien Tsai、Bow-Yaw Wang 和 Bo-Yin Yang。2021。CoqQFBV：可扩展的经认证SMT无量词位向量求解器。载于国际计算机辅助验证会议（CAV）。doi:10.1007/978-3-030-81688-9 7_[55] Jeremy G. Siek 和 Walid Taha。2006。函数式语言的渐进式类型。载于Scheme与函数式编程研讨会。

[56] Jan Smans、Bart Jacobs 和 Frank Piessens。2012。隐式动态框架。 *ACM*《编程语言与系统汇刊》
34, 1 2012 10 1145 2160910 2160911 （ ）。doi：。/。Simon Spies、Niklas Mück、Haoyi Zeng、Michael Sammler、Andrea
[57] Lattuada、Peter Müller 和 Derek Dreyer。2025。使 Iris 失稳。*ACM* 程序设计语言会议录
9, 2025 10 1145 3729284 PLDI（ ）。doi：。/Aaron Stump、Duckki Oe、Andrew Reynolds、
[58] LianaHadarean 和 CesareTinelli。2013。使用逻辑框架进行SMT 证明检查。《系统设计形式化方法》
42, 1 2013 10 1007 10703 012 0163 3 （ ）。doi：。/s - - -Shivanandan Tamil Kumaran、Andreas
[59] Löow 和 Philippa Gardner。。组合符号执行的端到端理论：从表达力丰富的规范到可满足性模理论（制品）。doi:10.6084/m9.figshare.34398291.v2
2026

A 附加背景定义

A.1 断言与程序逻辑的语义

现在我们定义断言满足关系 $\theta, (s,\mu) \models P$，它表示断言 P 由逻辑解释 θ 和具体状态 (s,μ) 满足。谓词定义 $p(\vec{x}) = P$ 来自正在分析的程序片段 PF（图 5）；我们要求参数 $\vec{x}$ 是互不相同的逻辑变量，并且主体满足 $\mathrm{pv}(P) = \emptyset$ 和 $\mathrm{lv}(P) \subseteq \vec{x}$。该关系归纳定义如下：

$$
\begin{array}{lcl}
\theta, (s,\mu) \models E & \Leftrightarrow & [\![E]\!]_{\theta,s} = \mathsf{true} \ \wedge \ \mu = \emptyset \\
\theta, (s,\mu) \models E_1 \mapsto E_2 & \Leftrightarrow & \mu = \{[\![E_1]\!]_{\theta,s} \mapsto [\![E_2]\!]_{\theta,s}\} \\
\theta, (s,\mu) \models p(\vec{E}) & \Leftrightarrow & [\![\vec{E}]\!]_{\theta,s} = \vec{v} \ \wedge \ \{\vec{x} \mapsto \vec{v}\}, (\emptyset,\mu) \models P \text{ for } p(\vec{x}) = P \in \mathsf{PF} \\
\theta, (s,\mu) \models \mathsf{emp} & \Leftrightarrow & \mu = \emptyset \\
\theta, (s,\mu) \models \mathsf{True} & \Leftrightarrow & \mathsf{true} \\
\theta, (s,\mu) \models P \star Q & \Leftrightarrow & \exists \mu_1, \mu_2.\, \mu = \mu_1 \uplus \mu_2 \ \wedge \ \theta, (s,\mu_1) \models P \ \wedge \ \theta, (s,\mu_2) \models Q \\
\theta, (s,\mu) \models P \vee Q & \Leftrightarrow & \theta, (s,\mu) \models P \ \vee \ \theta, (s,\mu) \models Q \\
\theta, (s,\mu) \models P \Rightarrow Q & \Leftrightarrow & \theta, (s,\mu) \models P \ \Rightarrow \ \theta, (s,\mu) \models Q \\
\theta, (s,\mu) \models \exists x.\, P & \Leftrightarrow & \exists v.\, \theta[x \mapsto v], (s,\mu) \models P
\end{array}
$$

我们将 $\langle\!\langle P \rangle\!\rangle$ C语言 $\langle\!\langle ok:\ Q_{ok} \rangle\!\rangle \langle\!\langle err:\ Q_{err} \rangle\!\rangle$ 表示任意一种四元组，即分离逻辑或 Iris分离逻辑四元组。分离逻辑和 Iris分离逻辑四元组的 OX 和 UX 判断式（根据函数实现上下文 γ）定义为断言满足关系：

$$
\begin{array}{lcl}
\gamma \models \{P\}\ \mathsf{C}\ \{ok : Q_{ok}\}\ \{err : Q_{err}\} & \Leftrightarrow & \forall \theta, \sigma, \sigma', o.\ \theta, \sigma \models P \ \wedge \ \sigma, \mathsf{C} \Downarrow_\gamma o : \sigma' \ \Rightarrow \\
 & & o \neq \mathit{miss} \ \wedge \ \theta, \sigma' \models Q_o \\
\gamma \models [P]\ \mathsf{C}\ [ok : Q_{ok}]\ [err : Q_{err}] & \Leftrightarrow & \forall \theta, \sigma', o.\ \theta, \sigma' \models Q_o \ \wedge \ o \neq \mathit{miss} \ \Rightarrow \\
 & & \exists \sigma.\ \theta, \sigma \models P \ \wedge \ \sigma, \mathsf{C} \Downarrow_\gamma o : \sigma'
\end{array}
$$

一个函数四元组

$$S = \langle\!\langle \vec{\mathsf{x}} = \vec{x} \star P \rangle\!\rangle\ \mathsf{f}(\vec{\mathsf{x}})\ \langle\!\langle ok : Q_{ok} \rangle\!\rangle \langle\!\langle err : Q_{err} \rangle\!\rangle$$

具有程序参数 $\vec{\mathsf{x}}$ 和逻辑参数 $\vec{x}$。当且仅当逻辑参数互不相同，且数量与程序参数相同，P 不包含任何程序变量，并且 Q_{ok} 和 Q_{err} 分别不包含除专用变量 ret 和 err 以外的程序变量（它们分别表示返回值和错误描述）时，该四元组是良构的。当且仅当它是良构的，$\mathsf{f}(\vec{\mathsf{x}})\ \{\ \mathsf{C};\mathsf{return\ E}\ \} \in \gamma$，且

$$\gamma \models \langle\!\langle \vec{\mathsf{x}} = \vec{x} \star P' \rangle\!\rangle\ \mathsf{C}\ \langle\!\langle ok : Q'_{ok} \rangle\!\rangle \langle\!\langle err : Q'_{err} \rangle\!\rangle$$

存在一些内部断言 P'、Q' ok 和 Q'_{err}，其中 P' 通过对 C 的局部变量进行初始化来扩展 P，并且 Q' ok 和 Q'_{err}（将 C 的变量作存在量化闭包，并将 ret 绑定到返回表达式 E）对于 OX 规约蕴含 Q_{ok} 和 Q_{err}，而对于 UX 规约则由它们蕴含时，才满足上述条件。具体细节参见 Lööw 等人 [34]。函数规约上下文 $\Gamma \in 2^{Spec}$ 收集此类函数四元组；当且仅当对所有 $S \in \Gamma$，$\gamma \models S$ 时，它在实现上下文 γ 下有效。若满足此条件，我们记作 $\models (\gamma, \Gamma)$。

A.2 符号引擎

现在，我们将更详细地介绍引擎的 消耗 和 生成 操作。符号 $\hat{}$ 替换 θ：局部变量 $\rightharpoonup_{fin}$ SExp 在模式匹配断言时，会实例化自由逻辑变量（例如 a 和 α 在 前置 的前置条件中，如图 2f），这一过程发生在 消耗中。 消耗 和生成 的判断式如下：

$$\mathrm{consume}(m, P, \hat{\theta}, \hat{\sigma}) \rightsquigarrow (o, \hat{\theta}', \hat{\sigma}') \qquad \text{and} \qquad \mathrm{produce}(Q, \hat{\theta}, \hat{\sigma}) \rightsquigarrow \hat{\sigma}'$$

其中，消耗断言 P 从状态 $\hat{}\sigma$、在模式 $\hat{}m$ 下，并采用初始符号替换 θ，会得到结果 o、框架状态 $\hat{}\sigma'$，以及一个习得的 $\hat{}$ 符号替换 θ'，其中现在包含了 P 中的所有自由逻辑变量的绑定；而生成的判断式会将 Q（其中自由变量由 $\hat{}$ 绑定）施加的资源和约束 θ 添加到状态 $\hat{}\sigma$ 中，$\hat{}\sigma'$。

A.3 函数验证

函数验证过程由以下规则定义：

$$\frac{\forall\, \hat{}\sigma, o, \hat{}\sigma'.\ \text{生成}(P, \hat{}\theta_{\text{init}}, \hat{}\sigma_{\text{init}}) \rightsquigarrow \hat{}\sigma \text{ 且 } \hat{}\sigma, \text{C语言} \Downarrow_{\Gamma}^{\text{OX}} o : \hat{}\sigma' \text{ 且 } \text{checkSat}(\text{OX}, \hat{}\sigma') \neq \text{不可满足} \Rightarrow \text{=成功且 } (\forall\, \hat{}E_{ret}.\ [\![\text{E}]\!]_{\hat{}\sigma'}.\text{sto} \Downarrow \hat{}E_{ret} \Rightarrow \text{checkEntails}(\hat{}\sigma', \hat{}E_{ret} \in \text{Val}) \text{ 且 } \hat{}\hat{}\hat{}\hat{} (\forall o', \theta_1, \hat{}\sigma''.\ \text{消耗}(\text{OX } Q[r/\text{ret}], \theta_{\text{init}}[r \mapsto E_{ret}], \hat{}\sigma') \rightsquigarrow (o', \theta_1, \hat{}\sigma'') \text{ 且 } \text{checkSat}(\text{OX}, \hat{}\sigma'') \neq \text{不可满足} \Rightarrow o' \neq \text{中止 且 } \hat{}\sigma''.\text{mem} = \emptyset \text{ 且 } \hat{}\sigma''.\text{preds} = \emptyset))}{\Gamma \vdash \{P\}\ \mathtt{f}(\vec{\mathtt{x}})\ \{\, \mathtt{C}; \mathtt{return\ E}\,\}\ \{Q\}}$$

待验证的规约为 $\{\vec{\mathtt{x}} = \vec{\mathtt{x}} \star P\}\,\mathtt{f}(\vec{\mathtt{x}})\ \{Q\}$，其中逻辑参数 $\vec{\mathtt{x}}$ 是程序参数 $\vec{\mathtt{x}}$ 在逻辑中的对应项（在机械化中使用相同名称）。前置条件是通过将初始符号替换 $\hat{}\,\theta_{\text{init}}\ \text{def}= \{\, \mathtt{x} \mapsto \mathtt{x} \mid \mathtt{x} \in \vec{\mathtt{x}} \cup \text{lv}(P) \,\}$（即作用于逻辑参数和 P 的逻辑变量上的恒等替换）应用于初始符号状态 $\hat{\sigma}_{\text{init}}\ \text{def}= (\hat{}s_{\text{init}}, \emptyset, \emptyset, \text{true})$ 得到的。其中，初始存储 $\hat{s}_{\text{init}}\ \text{def}= \{\, \mathtt{x} \mapsto \mathtt{x} \mid \mathtt{x} \in \vec{\mathtt{x}} \,\} \cup \{\, \mathtt{y} \mapsto \text{null} \mid \mathtt{y} \in \text{pv}(\mathtt{C}) \setminus \vec{\mathtt{x}} \,\}$ 将每个参数映射到其逻辑对应项，并将函数体中的其他所有变量映射为空值；内存和谓词均为空，路径条件为真。经过 $\hat{}$ 符号执行后，在最终存储中对返回表达式 E 求值，并且必须能够推出其值 E_{ret} 是一个有效值。随后，通过将特定变量 ret 替换为逻辑变量 $r \notin \text{lv}(\hat{}\sigma')$ 来消耗后置条件；该替换将其绑定到 $\hat{}\,E_{ret}$。最后一步是进行泄漏检查：消耗后置条件后剩余的框架状态 $\hat{}\sigma''$ 必须具有空内存和空谓词，从而确保后置条件涵盖所有剩余内存和谓词，不会发生泄漏。

B 扩展前端定义

B.1 代数数据类型

逻辑表达式求值。 我们给出为代数数据类型新增的三种结构所对应的逻辑表达式求值函数 $[\![\cdot]\!]_{\theta,s}$ 的形式化定义。

构造器应用。 $c(E)$ 对其参数求值并返回由此得到的构造器值，前提是该值类型正确：

$$[\![c(\vec{E})]\!]_{\theta,s} = \begin{cases} c(\vec{v}) & \text{if } [\![\vec{E}]\!]_{\theta,s} = \vec{v},\ c(\vec{\tau}) : \delta \in \Delta, \text{ and } \vec{v} \in \vec{\tau} \\ \lightning & \text{otherwise.} \end{cases}$$

测试器。E 求值结果始终为布尔值，尤其绝不会得到 $\lightning$，即使被匹配表达式会得到该值：

$$[\![E \text{ is } c]\!]_{\theta,s} = \begin{cases} \text{true} & \text{if } [\![E]\!]_{\theta,s} = c(\vec{v}) \text{ for some } \vec{v} \\ \text{false} & \text{otherwise.} \end{cases}$$

匹配表达式。**模式匹配** $E:\ \delta$ **其中** $p_i \to E_i|_{i=1}^n$，其中每个模式 p_i 要么是变量 x，要么是构造器模式 $c(\vec{x})$，首先对被匹配表达式求值。除非它是代数数据类型 δ的构造器值 $c(\vec{v})$（也就是说，$c(\vec{\tau}) :\ \delta \in \Delta$），否则结果为 $\lightning$。否则，设 j为满足 p_j与 $c(\vec{v})$匹配的最小索引：变量模式 $p_j = x$匹配任意值，而构造器模式 $p_j = c'(\vec{x})$ 当且仅当 $c' = c$时匹配。随后在扩展后的解释下继续对相应分支求值：

$$[\![\textbf{match } E : \delta \textbf{ with } p_i \to E_i|_{i=1}^n]\!]_{\theta,s} = \begin{cases} [\![E_j]\!]_{\theta',s} & \text{if } [\![E]\!]_{\theta,s} = c(\vec{v}),\ c(\vec{\tau}) : \delta \in \Delta, \text{ and} \\ & \quad p_j \text{ is the first pattern matching } c(\vec{v}) \\ \lightning & \text{otherwise,} \end{cases}$$

其中 $\theta' \overset{\text{def}}{=} \theta[x \mapsto c(\vec{v})]$表示变量模式 $p_j = x$，而 $\theta' \overset{\text{def}}{=} \theta[\vec{x} \mapsto \vec{v}]$表示构造器模式 $p_j = c(\vec{x})$。

命令级匹配。 正文给出了处理构造器模式的 `match` 符号执行规则。这里给出其余规则。变量绑定规则与构造器规则相似，但会将一个新的逻辑变量 x 绑定到被匹配表达式，并额外记录被匹配表达式是所匹配代数数据类型的一个值 δ（而构造器模式的分支方程并不能像这里这样直接蕴含这一点）：

$$\frac{\begin{array}{c}[\![E]\!]_{\hat{\sigma}.\text{sto}} \Downarrow \hat{E} \\ p_j = x \text{ a variable binder, for } 1 \le j \le n \\ p_k \text{ not a variable binder for } 1 \le k < j \\ \hat{\pi} = (x = \hat{E}) \wedge (\hat{E} \in \delta) \wedge \bigwedge_{k=1}^{j-1} \neg(\hat{E} \text{ is } c_k) \\ \hat{\sigma}' = \hat{\sigma}[\text{pc} := \hat{\pi} \wedge \hat{\sigma}.\text{pc}] \\ \hat{\sigma}', \mathsf{C}_j \Downarrow_\Gamma^m o : \hat{\sigma}'' \end{array}}{\hat{\sigma}, \texttt{match}\ (E : \delta)\ \{\, p_i \to \mathsf{C}_i|_{i=1}^n \,\} \Downarrow_\Gamma^m o : \hat{\sigma}''}$$

三条错误规则都会对被匹配表达式求值，然后中止，并使用刻画错误的（互斥）条件强化路径条件，同时在错误存储变量中记录错误类型：被匹配表达式求值失败，未能得到一个值；它是一个值，但不是代数数据类型 δ；或者它是一个代数数据类型值，却没有匹配任何模式（这要求每个模式都是构造器模式，

因为变量绑定器可以匹配任何值）。

$$\frac{[\![E]\!]_{\hat{\sigma}.\mathrm{sto}} \Downarrow \hat{E} \quad \hat{\sigma}' = \hat{\sigma}[\mathrm{sto} := \hat{\sigma}.\mathrm{sto}[\mathtt{err} \mapsto \text{“ExprEval”}]][\mathrm{pc} := \neg(\hat{E} \in \mathrm{Val}) \wedge \hat{\sigma}.\mathrm{pc}]}{\hat{\sigma}, \mathtt{match}\ (E : \delta)\ \{\, p_i \to \mathsf{C}_i|_{i=1}^{n} \,\} \Downarrow_{\Gamma}^{m} \mathit{abort} : \hat{\sigma}'}$$

$$\frac{[\![E]\!]_{\hat{\sigma}.\mathrm{sto}} \Downarrow \hat{E} \quad \hat{\pi} = (\hat{E} \in \mathrm{Val}) \wedge \neg(\hat{E} \in \delta) \quad \hat{\sigma}' = \hat{\sigma}[\mathrm{sto} := \hat{\sigma}.\mathrm{sto}[\mathtt{err} \mapsto \text{“Type”}]][\mathrm{pc} := \hat{\pi} \wedge \hat{\sigma}.\mathrm{pc}]}{\hat{\sigma}, \mathtt{match}\ (E : \delta)\ \{\, p_i \to \mathsf{C}_i|_{i=1}^{n} \,\} \Downarrow_{\Gamma}^{m} \mathit{abort} : \hat{\sigma}'}$$

$$\frac{\begin{array}{c}[\![E]\!]_{\hat{\sigma}.\mathrm{sto}} \Downarrow \hat{E} \quad p_i = c_i(\vec{x}_i) \text{ for } 1 \le i \le n \\ \hat{\pi} = (\hat{E} \in \delta) \wedge \bigwedge_{i=1}^{n} \neg(\hat{E} \text{ is } c_i) \quad \hat{\sigma}' = \hat{\sigma}[\mathrm{sto} := \hat{\sigma}.\mathrm{sto}[\mathtt{err} \mapsto \text{“NoMatch”}]][\mathrm{pc} := \hat{\pi} \wedge \hat{\sigma}.\mathrm{pc}]\end{array}}{\hat{\sigma}, \mathtt{match}\ (E : \delta)\ \{\, p_i \to \mathsf{C}_i|_{i=1}^{n} \,\} \Downarrow_{\Gamma}^{m} \mathit{abort} : \hat{\sigma}'}$$

B.2 引理与全函数正确性

应用规则。 我们给出应用的成功规则，省略所有无趣的错误规则；这些规则都会中止：

$$\frac{\begin{array}{c}\mathrm{lv}(\vec{E}) \subseteq \mathrm{lv}(\hat{\sigma}) \quad (1 : \forall \vec{x}.\ P \rightleftarrows Q) \in \Lambda \quad \rightleftarrows \text{ is sound in mode } m \\ [\![\vec{E}]\!]_{\hat{\sigma}.\mathrm{sto}} \Downarrow \hat{\vec{E}} \text{ and } \hat{\theta} = \{\vec{x} \mapsto \hat{\vec{E}}\} \quad \hat{\sigma}_1 = \hat{\sigma}[\mathrm{pc} := (\hat{\vec{E}} \in \mathrm{Val} \wedge \hat{\sigma}.\mathrm{pc})] \\ \mathrm{consume}(m, P, \hat{\theta}, \hat{\sigma}_1) \rightsquigarrow (\mathit{ok}, \hat{\theta}', \hat{\sigma}_2) \quad \mathrm{produce}(Q, \hat{\theta}', \hat{\sigma}_2) \rightsquigarrow \hat{\sigma}'\end{array}}{\hat{\sigma}, \mathtt{apply}\ \mathtt{l}(\vec{E}) \Downarrow_{\Gamma\Lambda}^{m} \mathit{ok} : \hat{\sigma}'}$$

当所应用的引理指定了度量 M时， 应用 规则会增加一个额外前提 $\hat{}\,\mathrm{checkEntails}(\hat{}\sigma_2, \theta, (M) < M)$， 以确保每次归纳应用都会使度量减小。

引理验证。 引理验证判断由以下规则定义， 该规则仿照函数验证（附录 A）， 用证明 C替代函数体：

$$\frac{\begin{array}{l}\forall \hat{\sigma}, o, \hat{\sigma}'.\ \mathrm{produce}(P, \hat{\theta}_{\mathrm{init}}, \hat{\sigma}_{\mathrm{init}}) \rightsquigarrow \hat{\sigma} \text{ and } \hat{\sigma}, \mathsf{C} \Downarrow_{\Gamma\Lambda}^{\mathsf{OX}} o : \hat{\sigma}' \text{ and } \mathrm{checkSat}(\mathsf{OX}, \hat{\sigma}') \neq \mathtt{UNSAT} \Rightarrow \\ \quad o = \mathit{ok} \text{ and} \\ \quad (\forall o', \hat{\theta}_1, \hat{\sigma}''.\ \mathrm{consume}(\mathsf{OX}, Q, \hat{\theta}_{\mathrm{init}}, \hat{\sigma}') \rightsquigarrow (o', \hat{\theta}_1, \hat{\sigma}'') \text{ and } \mathrm{checkSat}(\mathsf{OX}, \hat{\sigma}'') \neq \mathtt{UNSAT} \Rightarrow \\ \quad\quad o' \neq \mathit{abort} \text{ and } \hat{\sigma}''.\mathrm{mem} = \emptyset \text{ and } \hat{\sigma}''.\mathrm{preds} = \emptyset)\end{array}}{\Lambda \vdash \mathtt{l} : \forall \vec{x}.\ P \Rightarrow Q\ \{\mathsf{C}\}\ (M \textbf{ decreases})}$$

其中 $\hat{}\,\theta_{\mathrm{init}}$ 定义= $\{\ x \mapsto x \mid x \in \vec{}x \cup \mathrm{lv}(P)\ \}$， 以及 $\hat{\sigma}_{\mathrm{init}}$ 定义= $(\emptyset,\ \emptyset,\ \emptyset,\ \mathrm{true}\ \emptyset)$， 其中 $\hat{}T$，和 $\hat{}T''$ 是 $\hat{}\sigma'$ 和 $\hat{}\sigma''$的类型环境：与函数体不同，证明不包含程序变量，因此初始存储为空。该规则以 $\Rightarrow$ 方向验证引理；$\Leftarrow$ 引理的验证方式是将 P和 Q 互换后，作为$\Rightarrow$ 引理进行验证；而 $\Leftrightarrow$ 引理则需双向验证。每种情况下， Λ 都包含正在验证的引理及其度量，因此在引理自身的证明中，每次应用该引理时都要接受上述度量检查。

全函数正确性。用于引理验证的基于度量的方法可自然地扩展到函数验证，从而能够可靠地验证总正确性。其构造几乎完全相同：

(1) 终止判断 $\sigma, \mathsf{C} \downarrow_\gamma$ 断言：在函数实现上下文 γ 下，C 从状态 σ 开始会终止。(2) 完全 OX 规范 $\{\!|P|\!\}\ \mathsf{f}(\vec{\mathsf{x}})\ \{\!|ok : Q_{ok}|\!\}\ \{\!|err : Q_{err}|\!\}$(*M* 递减) 将通常的有效性条件加强，并额外要求：对于 $\mathsf{f}(\vec{\mathsf{x}})\ \{\ \mathsf{C};\ \mathtt{return}\ \mathsf{E}\ \} \in \gamma$，有 $\forall \theta, \sigma.\ \theta, \sigma \models P \not\Rightarrow \mathsf{C} \downarrow_\gamma$。(3) 新的引擎模式 OXT 与 OX 的运行方式完全相同，但只查阅完全规范。调用带有度量的函数规范时，会检查度量是否递减，类似于上面的应用规则。(4) 函数规范上下文的带测度终止要求：每个带度量的完全规范的函数体，对于每一个满足其（内部化）前置条件且度量小于 n 的状态，都必须终止。

根据这 些定义，我们证明，在 OXT 模式下进行验证可以确立完全正确性 ：

定理 B.1（完全函数验证正确性）。设

$$\Gamma' \stackrel{\text{def}}{=} \Gamma \cup \{\{\!|P|\!\}\ \mathsf{f}(\vec{\mathsf{x}})\ \{\!|Q|\!\}\ (M\ \textbf{decreases})\} \quad \textit{and} \quad \Gamma'' \stackrel{\text{def}}{=} \Gamma \cup \{\{\!|P|\!\}\ \mathsf{f}(\vec{\mathsf{x}})\ \{\!|Q|\!\}\},$$

and 假设 Γ 中没有规约带有度量，$\mathsf{pv}(M) = \emptyset$，且 $\mathsf{lv}(M) \subseteq \vec{\mathsf{x}} \cup \mathsf{lv}(P$):

$$\begin{array}{ll} \textit{If} & \models (\gamma, \Gamma)\ \textit{and}\ \mathsf{f}(\vec{\mathsf{x}})\ \{\ \mathsf{C};\mathtt{return}\ \mathsf{E}\ \} \in \gamma\ \textit{and} \\ & \Gamma' \vdash \{P\}\ \mathsf{f}(\vec{\mathsf{x}})\ \{\ \mathsf{C};\mathtt{return}\ \mathsf{E}\ \}\ \{Q\}\ \textit{in mode}\ \mathsf{OXT} \\ \textit{then} & \models (\gamma, \Gamma'') \end{array}$$

其中，在 OXT 模式下的验证是附录*A*中的过程，只需将 OX 替换为 OXT。

证明概要。 证明的结构与定理 4.3相同，只是现在度量用于界定终止性，而非有效性。在度量终止性成立且上下文在通常的部分正确性意义下有效时，对于一个在 OXT 模式下的符号执行不会在可满足状态上中止的命令，从满足初始符号状态且度量等于 *n*的每个具体状态出发，该命令都会终止。然后，通过对度量进行归纳，并结合定理4.1中的部分正确性，即可得出完全正确性。 □

B.3 机械化说明

值。 在未扩展的 CSE 理论中，值的定义很简单：

```
1 归纳定义 值 : =空值 | 布尔值 （b : 布尔值） | 自然数 （n : 自然数） | (* ... *)。
```

然而，对于代数数据类型，值良构性取决于类型判断，而类型判断又取决于值。由于 Rocq 不支持这种互归纳-归纳定义，我们将值定义为依赖对，并假设证明无关性，以恢复标准相等性：

```
1 (* 预值：扩展了代数数据类型值的旧 val 类型 *) 2 Inductive preval := pVADT (adt ctor : string)
(args : list preval) | pVNull | (* ..... *)。 3 (* 类型正确的值 *) 4 Inductive preval_wf : preval ->
Prop :=(* ..... *)。 5 Definition val :={ v : preval |preval_wf v }.
```

逻辑表达式求值。 在扩展之前， $[\![E]\!]_{\theta,s}$ 的定义很简单，如下所示：

```
不动点局部表达式_求值(t:theta)(s:cstore)(e:局部表达式):
optionval:=模式匹配e with | LEValv => Somev (*...*)。
```

使用逻辑函数时，只有当 Ψ 中的所有函数都终止时，局部表达式_求值才是一个函数；然而，终止性最自然的表述方式是借助求值。我们分两个阶段打破这种循环性：

```
Notationlfun_impl_ctx:=(*...*). Inductivelexp_eval_rel: lfun_impl_ctx->theta->
cstore->局部表达式->optionval->Prop:=(*...*). Definitionlfun_ctx_
terminates(ctx:lfun_impl_ctx):=(*...*). Notationlfun_ctx:={ctx:lfun_impl_ctx |lfun_
ctx_terminatesctx}. Definitionlexp_eval(ctx:lfun_ctx)tse:optionval:=
epsiloninhabited_option_val(lexp_eval_rel(ctx)tse).
```

首先，lexp_eval_rel 以关系方式描述状态的求值；这使得 lfun_ctx_terminates 能够表达终止性。接着，lfun_ctx 将一个实现上下文及其终止性证明打包在一起，而希尔伯特 ε 算子则将该关系转换为 lexp_eval。

由于希尔伯特 ε 算子，我们无法用 lexp_eval 进行计算（例如，simpl 在证明中不起作用）。不过，我们可以通过以下定理进行重写来手动计算：

```
定理局部表达式_求值_求值: 对所有tse,
  局部表达式_求值 t s e = 模式匹配e 并
且 | LEValv => Somev (*...*)。
```

C 语言后端的 SMT-LIB 编码定义

C.1 背景理论

ˆ背景理论由符号类型环境参数化 T：局部变量 $\rightharpoonup_{fin}$ 类型。它由 Core、实数_整数、Seq、Strings 标准 SMT-LIB 理论组成，并扩展了第6.2节引入的代数数据类型和函数符号。

C.1.1 排序符号。排序符号扩展为

$$\mathsf{Null},\quad \mathsf{Val},\quad \mathsf{MaybeVal},\quad \delta \text{ for each } \delta \in \Delta.$$

映射源类型 τ 到该背景理论中各排序的转换排序(τ)，与正文中使用的转换排序相同。

C.1.2 函数符号。函数符号扩展后包括逻辑函数、前置条件函数、逻辑函数的无解释对应函数、可表示性谓词、代数数据类型构造器、选择器和测试器。

对于每个逻辑函数 $f(\vec{\tau})$：$\tau \in \Psi$，[12]，背景理论都会声明函数符号、其前置条件符号，以及在前置条件之外给出其值的未解释符号：

$$\begin{array}{rcl} f & : & \mathrm{sort}(\tau_1), \ldots, \mathrm{sort}(\tau_n); \mathrm{sort}(\tau) \quad \text{for } f(\vec{\tau}) : \tau \in \Psi \\ f_{pre} & : & \mathrm{sort}(\tau_1), \ldots, \mathrm{sort}(\tau_n); \mathsf{Bool} \\ f_{\perp} & : & \mathrm{sort}(\tau_1), \ldots, \mathrm{sort}(\tau_n); \mathrm{sort}(\tau). \end{array}$$

生成的可表示性谓词符号的秩为：

$$\begin{array}{rcl} \mathit{reprNat} & : & \text{整数}; \text{布尔值} \\ \mathit{reprRat} & : & \text{实数}; \text{布尔值} \\ \mathit{reprVal} & : & \mathsf{Val}; \text{布尔值} \\ \mathit{repr}_{\delta} & : & \delta; \text{布尔值}。 \end{array}$$

空值、Val和MaybeVal 代数数据类型的构造子为：

$$\begin{array}{rcllrcl} \mathit{null} & : & \mathsf{Null} & & \mathit{nullVal} & : & \mathsf{Null}; \mathsf{Val} \\ \mathit{bool} & : & \mathsf{Bool}; \mathsf{Val} & & \mathit{nat} & : & \mathsf{Int}; \mathsf{Val} \\ \mathit{rat} & : & \mathsf{Real}; \mathsf{Val} & & \mathit{string} & : & \mathsf{String}; \mathsf{Val} \\ \mathit{list} & : & \mathsf{Seq}(\mathsf{Val}); \mathsf{Val} & & \mathit{adt}_{\delta} & : & \delta; \mathsf{Val} \\ \mathit{some} & : & \mathsf{Val}; \mathsf{MaybeVal} & & \lightning & : & \mathsf{MaybeVal}. \end{array}$$

每个非空构造器都有对应的选择器：*getNull*、*getBool*、*getNat*、*getRat*、*getStr*、*getList*、getADT_{δ}，以及*getVal*。测试器函数符号的表示方式类似，例如，*isNat* 和*isVal*。

对于每个源构造函数声明 $c(\tau_1, \ldots, \tau_n)$：$\delta \in \Delta$，生成的代数数据类型排序 δ包含构造器 c，其秩为

$$\mathrm{sort}(\tau_1), \ldots, \mathrm{sort}(\tau_n); \delta.$$

其 i阶选择器 $\mathit{get}\text{-}c_i$ 的秩为 δ；排序(τ_i)，其测试器 $\mathit{is}\text{-}c$ 的秩为 δ；布尔值。

C.2 逻辑函数和可表示性谓词定义

对于逻辑函数 $f(\vec{x}: \vec{\tau}) : \tau = E$当 E^{pre}时，令$\hat{T} \stackrel{\text{def}}{=} \{\vec{x} \mapsto \vec{\tau}\}$。回顾第 5节，定义一个函数在语义上等价于断言其定义方程；对于我们的逻辑函数，生成的定义方程为：

$$\forall(\vec{x} : \mathrm{sort}(\vec{\tau})).\ f(\vec{x}) = \mathit{ite}(f_{pre}(\vec{x}), t, f_{\perp}(\vec{x})) \quad \text{where } \hat{T} \vdash E \rightsquigarrow_{\tau} t \mid \Phi.$$

[12]我们将 $f(\vec{\tau}) : \tau \in \Psi$当 $f(\vec{x}: \vec{\tau}) : \tau = E$当 $E_{pre} \in \Psi$对于某个 $\vec{x}$、E 和 E^{pre}。

该等式中并未断言主体编码的侧条件；可靠性依赖于逻辑函数在其前置条件下的语义有效性。前置条件函数确实包含这些侧条件，以及参数上的可表示性保护条件 repr_{τ_i} (definedbelow)：

$$\forall(\vec{x} : \mathrm{sort}(\vec{\tau})).\ f_{pre}(\vec{x}) = (\bigwedge_i \mathrm{repr}_{\tau_i}(x_i) \wedge \bigwedge \Phi \wedge \varphi), \quad \text{where } \hat{T} \vdash E_{pre} \rightsquigarrow_{\mathrm{Bool}} \varphi \mid \Phi.$$

所有此类方程的集合为 Ψ。

可表示性谓词由相同机制生成：下方每个展示的子句都是由 `define-funs-rec`发出的定义方程的缩写。我们写作

$$p(x : \tau) \triangleq \varphi \quad \text{for} \quad \forall(x : \tau).\ p(x) = \varphi.$$

对于生成的谓词：

$$\begin{aligned}
\mathit{reprNat}(x : \mathsf{Int}) &\triangleq x \geq 0 \\
\mathit{reprRat}(x : \mathsf{Real}) &\triangleq \exists(n : \mathsf{Int}).\ \exists(d : \mathsf{Int}).\ d \neq 0 \wedge x = \mathit{to_real}(n) / \mathit{to_real}(d) \wedge x > 0 \\
\mathit{reprVal}(x : \mathsf{Val}) &\triangleq \mathit{isNull}(x) \\
&\quad \vee \mathit{isBool}(x) \\
&\quad \vee (\mathit{isNat}(x) \wedge \mathit{reprNat}(\mathit{getNat}(x))) \\
&\quad \vee (\mathit{isRat}(x) \wedge \mathit{reprRat}(\mathit{getRat}(x))) \\
&\quad \vee \mathit{isString}(x) \\
&\quad \vee (\mathit{isList}(x) \wedge \forall(i : \mathsf{Int}).\ 0 \leq i < \mathit{seq.len}(\mathit{getList}(x)) \Rightarrow \\
&\qquad \mathit{reprVal}(\mathit{seq.nth}(\mathit{getList}(x), i))) \\
&\quad \vee \bigvee_{\delta \in \Delta} (\mathit{isADT}_\delta(x) \wedge \mathit{repr}_\delta(\mathit{getADT}_\delta(x))).
\end{aligned}$$

对于每个代数数据类型 δ，

$$\mathit{repr}_\delta(a : \delta) \triangleq \bigvee_{c(\vec{\tau}) : \delta \in \Delta} \mathit{is\text{-}c}(a) \wedge \bigwedge_i \mathrm{repr}_{\tau_i}(\mathit{get\text{-}c}_i(a)).$$

生成的 $\mathit{reprNat}$、$\mathit{reprRat}$、$\mathit{reprVal}$以及每个 repr_δ 的定义方程汇集于 Ψ_{repr}。类型索引守卫 repr_τ，用于 repr_δ 方程中，是一个元层缩写：

$$\begin{aligned}
\mathrm{repr}_{\mathsf{Val}}(t) &= \mathit{reprVal}(t) & \mathrm{repr}_{\mathsf{Null}}(t) &= \mathit{true} \\
\mathrm{repr}_{\mathsf{Bool}}(t) &= \mathit{true} & \mathrm{repr}_{\mathsf{Nat}}(t) &= \mathit{reprNat}(t) \\
\mathrm{repr}_{\mathsf{Rat}^+}(t) &= \mathit{reprRat}(t) & \mathrm{repr}_{\mathsf{Str}}(t) &= \mathit{true} \\
\mathrm{repr}_\delta(t) &= \mathit{repr}_\delta(t)
\end{aligned}$$

$$\mathrm{repr}_{\mathsf{List}(\tau)}(t) = \forall(i : \mathsf{Int}).\ 0 \leq i < \mathit{seq.len}(t) \Rightarrow \mathrm{repr}_\tau(\mathit{seq.nth}(t, i)).$$

C.3 表达式编码器与强制转换

编码状态是一个三元组（t、τ、Φ），由一个 SMT-LIB 项、其排序以及一个有限的布尔侧条件集合组成。

值强制转换 toVal 将项转换为 Val排序的项：

$$\begin{aligned}
\mathrm{toVal}(t, \mathrm{Val}, \Phi) &= (t, \mathrm{Val}, \Phi) \\
\mathrm{toVal}(t, \mathrm{MaybeVal}, \Phi) &= (\mathit{getVal}(t), \mathrm{Val}, \Phi \cup \{\mathit{isVal}(t)\}) \\
\mathrm{toVal}(t, \mathrm{Null}, \Phi) &= (\mathit{nullVal}(t), \mathrm{Val}, \Phi) \\
\mathrm{toVal}(t, \mathrm{Bool}, \Phi) &= (\mathit{bool}(t), \mathrm{Val}, \Phi) \\
\mathrm{toVal}(t, \mathrm{Int}, \Phi) &= (\mathit{nat}(t), \mathrm{Val}, \Phi) \\
\mathrm{toVal}(t, \mathrm{Real}, \Phi) &= (\mathit{rat}(t), \mathrm{Val}, \Phi) \\
\mathrm{toVal}(t, \mathrm{String}, \Phi) &= (\mathit{string}(t), \mathrm{Val}, \Phi) \\
\mathrm{toVal}(t, \mathrm{sort}(\delta), \Phi) &= (\mathit{adt}_\delta(t), \mathrm{Val}, \Phi).
\end{aligned}$$

对于序列排序 $\mathrm{Seq}(\tau)$，机械化会将每个元素映射为 toVal：

$$\mathrm{toVal}(t, \mathrm{Seq}(\tau), \Phi) = (\mathit{list}(\mathit{seq.map}(\lambda(x : \tau).\ t_x, t)), \mathrm{Val}, \Phi),$$

其中 $\mathrm{toVal}(x, \tau, \emptyset) = (t_x, \mathrm{Val}\ \emptyset)$。

原始强制转换 toNull、toBool、toNat、toRat、toString 和 toADT_δ 是在原生排序、Val以及 MaybeVal上的逆投影。例如：

$$\mathrm{toNat}(t, \mathrm{Int}\ \Phi) = (t, \mathrm{Int}\ \Phi)\ \mathrm{toNat}(t, \mathrm{Val}\ \Phi) = (\mathrm{getNat}\ (t), \mathrm{Int}\ \Phi \cup \{\mathrm{isNat}(t)\})$$
$$\mathrm{toNat}(t, \mathrm{MaybeVal}\ \Phi) = (\mathrm{getNat}\ (\mathrm{getVal}(t)), \mathrm{Int}\ \Phi \cup \{\mathrm{isVal}(t), \mathrm{isNat}\ (\mathrm{getVal}(t))\})\ \mathrm{toNat}(t, \tau, \Phi) = \mathrm{undefined},\ \text{对于其他任何}\ \tau.$$

类型引导的强制转换 to_τ 会分派到相应的原始强制转换。列表情形比较特殊。如果输入的排序已经是 Seq(sort(τ))，则原样返回。如果其排序是某个不同元素排序的 $\mathrm{Seq}(\tau)$，则输出为 $\mathrm{seq.map}(\lambda(x{:}\ \tau).\ t_x, t)$，其中 $\mathrm{to}_\tau(x, \tau, \emptyset) = (t_x, \mathrm{sort}(\tau)\ \Phi_x)$，并满足侧条件

$$\forall(i : \mathrm{Int}).\ 0 \le i < \mathit{seq.len}(t) \Rightarrow (\bigwedge \Phi_x)[\mathit{seq.nth}(t, i)/x],$$

其中 $\varphi[t'/x]$ 表示将 t' 替换为 x 于 φ中：t 中的每个元素都必须满足其强制转换的侧条件。Val 和 MaybeVal 列表分支首先选取包装后的列表负载，并添加相应的测试器侧条件。

$\hat{}\quad\hat{}$编码判断 $T \vdash E\quad t : \tau \mid \Phi$由以下语法导向规则定义，这些规则扩展了图 8中的规则。与该图一样，带有索引 i的规则对每个 i都有一个前提；对于包含多条规则的表达式形式，各规则的侧条件彼此不相交。在类型 $\hat{}\quad\hat{}T \vdash E\quad_\tau t \mid \Phi$处进行编码时，会将编码通过 to_τ强制转换为 $\rightsquigarrow$，其中：

$$\frac{\hat{T} \vdash \hat{E} \rightsquigarrow t : \tau \mid \Phi \quad \mathrm{to}_\tau(t, \tau, \Phi) = (t', \mathrm{sort}(\tau), \Phi')}{\hat{T} \vdash \hat{E} \rightsquigarrow_\tau t' \mid \Phi'}$$

基本值会转换为原生 SMT-LIB 字面量，且没有侧条件：

$$\frac{}{\hat{T} \vdash \mathtt{null} \rightsquigarrow \mathit{null} : \mathrm{Null} \mid \emptyset} \qquad \frac{}{\hat{T} \vdash b \rightsquigarrow b : \mathrm{Bool} \mid \emptyset} \qquad \frac{}{\hat{T} \vdash n \rightsquigarrow n : \mathrm{Int} \mid \emptyset}$$

$$\frac{}{\hat{T} \vdash q \rightsquigarrow q : \mathrm{Real} \mid \emptyset} \qquad \frac{}{\hat{T} \vdash s \rightsquigarrow s : \mathrm{String} \mid \emptyset}$$

变量、构造子、列表和逻辑函数调用编码如下：

$$\frac{x : \tau \in \hat{T}}{\hat{T} \vdash x \rightsquigarrow x : \mathrm{sort}(\tau) \mid \emptyset} \qquad \frac{x \notin \mathrm{dom}\, \hat{T}}{\hat{T} \vdash x \rightsquigarrow x : \mathrm{MaybeVal} \mid \emptyset}$$

$$\frac{c(\tau_1, \ldots, \tau_n) : \delta \in \Delta \quad \hat{T} \vdash \hat{E}_i \rightsquigarrow_{\tau_i} t_i \mid \Phi_i}{\hat{T} \vdash c(\hat{E}_1, \ldots, \hat{E}_n) \rightsquigarrow c(t_1, \ldots, t_n) : \delta \mid \bigcup_i \Phi_i} \qquad \frac{\hat{T} \vdash \hat{E}_i \rightsquigarrow_{\mathrm{Val}} t_i \mid \Phi_i}{\hat{T} \vdash [\hat{E}_1, \ldots, \hat{E}_n] \rightsquigarrow t_\ell : \mathrm{Seq}(\mathrm{Val}) \mid \bigcup_i \Phi_i}$$

$$\frac{f(\vec{x} : \vec{\tau}) : \tau = E \ \textbf{when}\ E_{pre} \in \Psi \quad \hat{T} \vdash \hat{E}_i \rightsquigarrow_{\tau_i} t_i \mid \Phi_i}{\hat{T} \vdash f(\hat{E}_1, \ldots, \hat{E}_n) \rightsquigarrow f(t_1, \ldots, t_n) : \mathrm{sort}(\tau) \mid \{f_{pre}(t_1, \ldots, t_n)\} \cup \bigcup_i \Phi_i}$$

其中，在列表规则中，t_ℓ定义=序列。单元 (t_1) ++⋯ ++序列。单元(t_n) ++(序列。空 :Seq(Val))。下面给出代表性操作情况。否定和长度分别在布尔值 和列表(Val)类型上编码其操作数，而自然数运算则在 自然数类型上编码两个操作数：

$$\frac{\hat{T} \vdash \hat{E} \rightsquigarrow_{\mathrm{Bool}} t \mid \Phi}{\hat{T} \vdash \neg \hat{E} \rightsquigarrow \neg t : \mathrm{Bool} \mid \Phi} \qquad \frac{\hat{T} \vdash \hat{E} \rightsquigarrow_{\mathrm{List(Val)}} t \mid \Phi}{\hat{T} \vdash |\hat{E}| \rightsquigarrow \mathit{seq.len}(t) : \mathrm{Int} \mid \Phi}$$

$$\frac{\hat{T} \vdash \hat{E}_i \rightsquigarrow_{\mathrm{Nat}} t_i \mid \Phi_i}{\hat{T} \vdash \hat{E}_1 + \hat{E}_2 \rightsquigarrow t_1 + t_2 : \mathrm{Int} \mid \Phi_1 \cup \Phi_2} \qquad \frac{\hat{T} \vdash \hat{E}_i \rightsquigarrow_{\mathrm{Nat}} t_i \mid \Phi_i}{\hat{T} \vdash \hat{E}_1 - \hat{E}_2 \rightsquigarrow \mathit{ite}(t_1 - t_2 < 0, 0, t_1 - t_2) : \mathrm{Int} \mid \Phi_1 \cup \Phi_2}$$

$$\frac{\hat{T} \vdash \hat{E}_i \rightsquigarrow_{\mathrm{Nat}} t_i \mid \Phi_i}{\hat{T} \vdash \hat{E}_1 / \hat{E}_2 \rightsquigarrow t_1 \mathit{\ div\ } t_2 : \mathrm{Int} \mid \Phi_1 \cup \Phi_2 \cup \{t_2 \neq 0\}}$$

$$\frac{\hat{T} \vdash \hat{E}_i \rightsquigarrow_{\mathrm{Nat}} t_i \mid \Phi_i}{\hat{T} \vdash \hat{E}_1 \bmod \hat{E}_2 \rightsquigarrow t_1 \mathit{\ mod\ } t_2 : \mathrm{Int} \mid \Phi_1 \cup \Phi_2 \cup \{t_2 \neq 0\}}$$

当两个编码项的类型相同且该类型不是 MaybeVal 时，相等性会直接比较它们；否则，就会用 toVal 对两者进行强制转换。进行强制转换是必要的，因为不同类型的项仍可能表示相等的值，因此类型不同并不能说明相等性为假。例如，若x: List(Val) $\in \hat{T}$且 y: List(Nat) $\in \hat{T}$，那么 x 和 y 分别编码在不同的类型 Seq(Val) 和 Seq(Int) 上，但只要二者表示同一个自然数列表，$x = y$ 的计算结果就为真。即使两侧的类型都是 MaybeVal，也会对它们进行强制转换。若直接比较，θ论域之外的两个变量都会是 $\lightning$，从而使相等性为真；但在源程序中，其计算结果为 $\lightning$。toVal 规则所添加的侧条件 isVal(t_i) 排除了这种情况：

$$\frac{\hat{T} \vdash \hat{E}_i \rightsquigarrow t_i : \tau_i \mid \Phi_i \quad \tau_1 = \tau_2 \neq \mathrm{MaybeVal}}{\hat{T} \vdash \hat{E}_1 = \hat{E}_2 \rightsquigarrow t_1 = t_2 : \mathrm{Bool} \mid \Phi_1 \cup \Phi_2}$$

$$\frac{\hat{T} \vdash \hat{E}_i \rightsquigarrow t_i : \tau_i \mid \Phi_i \quad \tau_1 \neq \tau_2 \text{ or } \tau_1 = \mathrm{MaybeVal} \quad \mathrm{toVal}(t_i, \tau_i, \Phi_i) = (t_i', \mathrm{Val}, \Phi_i')}{\hat{T} \vdash \hat{E}_1 = \hat{E}_2 \rightsquigarrow t_1' = t_2' : \mathrm{Bool} \mid \Phi_1' \cup \Phi_2'}$$

列表操作将其元素编码为 Val，将其列表编码为 List(Val)：

$$\frac{\hat{T} \vdash \hat{E}_1 \rightsquigarrow_{\mathsf{Val}} t_1 \mid \Phi_1 \quad \hat{T} \vdash \hat{E}_2 \rightsquigarrow_{\mathsf{List(Val)}} t_2 \mid \Phi_2}{\hat{T} \vdash \hat{E}_1 :: \hat{E}_2 \rightsquigarrow \mathit{seq.unit}(t_1) \mathbin{++} t_2 : \mathsf{Seq(Val)} \mid \Phi_1 \cup \Phi_2}$$

$$\frac{\hat{T} \vdash \hat{E}_1 \rightsquigarrow_{\mathsf{Val}} t_1 \mid \Phi_1 \quad \hat{T} \vdash \hat{E}_2 \rightsquigarrow_{\mathsf{List(Val)}} t_2 \mid \Phi_2}{\hat{T} \vdash \hat{E}_1 \in \hat{E}_2 \rightsquigarrow \mathit{seq.contains}(t_2, \mathit{seq.unit}(t_1)) : \mathsf{Bool} \mid \Phi_1 \cup \Phi_2}$$

有理数算术与自然数算术类似，在 Rat^+ 处对操作数进行编码，并使用 Real 及其运算符。减法是个例外：Rat^+ 仅包含正有理数，因此只有当结果为正数时，有理数减法才有定义；它不会像自然数减法那样饱和。其编码为 $t_1 - t_2$，侧条件为 $t_2 < t_1$。

类型测试情形会刻意将侧条件折叠到布尔项中：

$$\frac{\neg \mathcal{W}f(\tau)}{\hat{T} \vdash \hat{E} \in \tau \rightsquigarrow \mathit{false} : \mathsf{Bool} \mid \emptyset} \qquad \frac{\hat{T} \vdash \hat{E} \rightsquigarrow_{\mathsf{Val}} t \mid \Phi \quad \mathcal{W}f(\tau)}{\hat{T} \vdash \hat{E} \in \tau \rightsquigarrow \bigwedge \Phi \wedge \mathsf{is}_\tau(t) : \mathsf{Bool} \mid \emptyset}$$

这是$_\tau$ 包装器形状谓词：

$$\begin{array}{rclcrcl} \mathsf{is}_{\mathsf{Val}}(t) & = & \mathit{true} & & \mathsf{is}_{\mathsf{Null}}(t) & = & \mathit{isNull}(t) \\ \mathsf{is}_{\mathsf{Bool}}(t) & = & \mathit{isBool}(t) & & \mathsf{is}_{\mathsf{Nat}}(t) & = & \mathit{isNat}(t) \\ \mathsf{is}_{\mathsf{Rat}^+}(t) & = & \mathit{isRat}(t) & & \mathsf{is}_{\mathsf{Str}}(t) & = & \mathit{isString}(t) \\ \mathsf{is}_{\delta}(t) & = & \mathit{isADT}_{\delta}(t) & & & & \end{array}$$

$$\text{是}_{\mathsf{List}(\tau)}(t) = \mathit{isList}(t) \wedge \forall (i{:}\text{整数}).\ 0 \le i < \mathit{seq.len}(\mathit{getList}(t)) \Rightarrow \text{是}_\tau(\mathit{seq.nth}(\mathit{getList}(t)\ i))。$$

ˆ构造器测试情形类似。未声明的构造器会得到 *false*，且不进行编码 E：

$$\frac{c \notin \mathrm{dom}\,\Delta}{\hat{T} \vdash \hat{E}\ \mathsf{is}\ c \rightsquigarrow \mathit{false} : \mathsf{Bool} \mid \emptyset} \qquad \frac{c(\vec{\tau}) : \delta \in \Delta \quad \hat{T} \vdash \hat{E} \rightsquigarrow t : \tau \mid \Phi \quad \mathsf{toADT}_\delta(t, \tau, \Phi)\ \text{undefined}}{\hat{T} \vdash \hat{E}\ \mathsf{is}\ c \rightsquigarrow \mathit{false} : \mathsf{Bool} \mid \emptyset}$$

$$\frac{c(\vec{\tau}) : \delta \in \Delta \quad \hat{T} \vdash \hat{E} \rightsquigarrow_\delta t \mid \Phi}{\hat{T} \vdash \hat{E}\ \mathsf{is}\ c \rightsquigarrow \bigwedge \Phi \wedge \mathit{is\text{-}c}(t) : \mathsf{Bool} \mid \emptyset}$$

模式匹配的各个分支均在其模式所导出的类型环境下编码。SMT-LIB 匹配表达式是总函数，但求值源语言匹配表达式可能失败，因此第二个模式匹配会记录所选分支的侧条件：

$$\frac{\delta \in \Delta \quad n \ge 1 \quad \hat{T} \vdash \hat{E} \rightsquigarrow_\delta t \mid \Phi \quad \hat{T}_i \vdash \hat{E}_i \rightsquigarrow_{\mathsf{Val}} t_i \mid \Phi_i}{\hat{T} \vdash \mathbf{match}\ \hat{E} : \delta\ \mathbf{with}\ p_i \to \hat{E}_i|_{i=1}^n \rightsquigarrow t_m : \mathsf{Val} \mid \Phi \cup \{\varphi_m\}}$$

其中 y 是新鲜的，

$$t_m = \mathbf{match}\ t\ \mathbf{with}\ (p_i \to t_i)\,|_{i=1}^n \quad y \to \mathit{nullVal}(\mathit{null}),$$
$$\varphi_m = \mathbf{match}\ t\ \mathbf{with}\ (p_i \to \textstyle\bigwedge \Phi_i)\,|_{i=1}^n \quad y \to \mathit{false},$$

以及 $\hat{}\,T_{i\text{定义}=}\hat{}\,T[x \mapsto \delta]$若 $p_i = x$，以及 $\hat{}\,T_{i\text{定义}=}\hat{}\,T[\vec{x} \mapsto \vec{\tau}]$若 $p_i = c(\vec{x})$并 $c(\vec{\tau}) : \delta \in \Delta$。兜底分支使 t_m 保持总函数。它在 φ_m 中的对应项是 *false*，因此当没有模式匹配时，侧条件也会失败，正如求值源语言匹配表达式时一样。

C.4 可表示性保护条件

顶层查询断言：

$$\mathrm{guards}_{\hat{T}}(\hat{E}) \stackrel{\text{def}}{=} \{G_{\hat{T}}(x) \mid x \in \mathrm{lv}(\hat{E}) \cup \mathrm{dom}\,\hat{T}\},$$

其中

$$G_{\hat{T}}(x) = \begin{cases} \mathrm{repr}_{\tau}(x) & x : \tau \in \hat{T} \\ \mathit{isVal}(x) \Rightarrow \mathit{reprVal}(\mathit{getVal}(x)) & x \notin \mathrm{dom}\,\hat{T}. \end{cases}$$

C.5 语义等价

固定一个用于生成的背景理论的 SMT-LIB 结构 $\mathbf{A}$。与正文一样，我们记作$v \simeq u$，读作 v 在语义上等价于 u，当源结果 v 对应于 SMT-LIB 值 u，且该值在 $\mathbf{A}$中解释时。外围类型决定使用该关系的哪种情形。在原生排序下，具有代表性的情形如下：

$$\begin{array}{rcll} \mathtt{null} & \simeq & \mathit{null}^{\mathbf{A}} & \\ b & \simeq & u & b \in \mathrm{Bool},\ u = b \\ n & \simeq & u & n \in \mathrm{Nat},\ u = n,\ u \in \mathrm{Int} \\ q & \simeq & u & q \in \mathrm{Rat}^{+},\ u = q,\ u \in \mathrm{Real} \\ s & \simeq & u & s \in \mathrm{Str},\ u = s \\ [v_1, \ldots, v_n] & \simeq & [u_1, \ldots, u_n] & \text{if } v_i \simeq u_i \\ c(v_1, \ldots, v_n) & \simeq & c^{\mathbf{A}}(u_1, \ldots, u_n) & \text{if } v_i \simeq u_i. \end{array}$$

由于左侧取值范围是源值，因此不可表示的原生 SMT-LIB 值（例如负整数）与任何值都不相关联。

对于 Val，该关系会展开动态构造器，然后对负载使用相同的关系：

$$\begin{array}{rcll} \mathtt{null} & \simeq & \mathit{nullVal}^{\mathbf{A}}(u) & \text{if } \mathtt{null} \simeq u \\ b & \simeq & \mathit{bool}^{\mathbf{A}}(u) & \text{if } b \simeq u \\ n & \simeq & \mathit{nat}^{\mathbf{A}}(u) & \text{if } n \simeq u \\ q & \simeq & \mathit{rat}^{\mathbf{A}}(u) & \text{if } q \simeq u \\ s & \simeq & \mathit{string}^{\mathbf{A}}(u) & \text{if } s \simeq u \\ [v_1, \ldots, v_n] & \simeq & \mathit{list}^{\mathbf{A}}(u) & \text{if } [v_1, \ldots, v_n] \simeq u \\ c(v_1, \ldots, v_n) & \simeq & \mathit{adt}_{\delta}{}^{\mathbf{A}}(u) & \text{if } c(v_1, \ldots, v_n) \simeq u. \end{array}$$

对于 MaybeVal，$v \simeq \mathit{some}^{\mathbf{A}}(u)$ 当 $v \simeq u$时，而 $\lightning \simeq \lightning^{\mathbf{A}}$。

C.6 渐进类型与类型错误

渐进类型判断使用类型一致性：

$$\tau \sim \tau' \quad \Leftrightarrow \quad \tau = \tau' \text{ or } \tau = \mathrm{Val} \text{ or } \tau' = \mathrm{Val}.$$

关键规则如下：

$$\frac{v \in \tau \quad \tau \neq \mathsf{Val}}{\hat{T} \vdash v : \tau} \qquad \frac{x : \tau \in \hat{T} \quad \mathcal{W}f(\tau') \quad (\tau' = \tau \text{ or } \tau = \mathsf{Val})}{\hat{T} \vdash x : \tau'} \qquad \frac{x \notin \mathrm{dom}\, \hat{T} \quad \mathcal{W}f(\tau)}{\hat{T} \vdash x : \tau}$$

$$\frac{c(\vec{\tau}) : \delta \in \Delta \quad \hat{T} \vdash \vec{\hat{E}} : \vec{\tau}' \quad \vec{\tau} \sim \vec{\tau}'}{\hat{T} \vdash c(\vec{\hat{E}}) : \delta} \qquad \frac{f(\vec{\tau}) : \tau \in \Psi \quad \hat{T} \vdash \vec{\hat{E}} : \vec{\tau}' \quad \vec{\tau} \sim \vec{\tau}'}{\hat{T} \vdash f(\vec{\hat{E}}) : \tau}$$

$$\frac{\odot : \tau_{in} \to \tau_{out} \quad \mathcal{W}f(\tau_{in}) \quad \hat{T} \vdash \hat{E} : \tau'_{in} \quad \tau_{in} \sim \tau'_{in}}{\hat{T} \vdash \odot \hat{E} : \tau_{out}}$$

$$\frac{\oplus : \tau_1 \times \tau_2 \to \tau \quad \mathcal{W}f(\tau_i) \quad \hat{T} \vdash \hat{E}_i : \tau'_i \quad \tau_i \sim \tau'_i}{\hat{T} \vdash \hat{E}_1 \oplus \hat{E}_2 : \tau} \qquad \frac{\hat{T} \vdash \hat{E} : \tau'}{\hat{T} \vdash \hat{E} \in \tau : \mathsf{Bool}} \qquad \frac{\hat{T} \vdash \hat{E} : \tau}{\hat{T} \vdash \hat{E} \text{ is } c : \mathsf{Bool}}.$$

列表要求所有元素的类型都与所选元素类型一致。对 δ 进行匹配表达式求值要求 $\delta \in \Delta$、其类型与 δ 一致的被匹配表达式、良构的结果类型、在模式绑定变量下类型一致的分支主体，以及至少一个模式为变量或 δ 的构造器的分支。

否定判断 $\hat{T} \dashv \hat{E} : \tau$ 具有以下性质：每当 $\theta \models \hat{T}$、$\hat{T} \dashv \hat{E} : \tau$ 且 $[\![\hat{E}]\!]_\theta = v$ 时，就有 $v \notin \tau$。我们用 $\tau' \leq \tau$ 表示类型上的子类型关系：它是自反的，每种类型都低于顶类型 Val，并且列表类型是协变的，因此当 $\tau_1 \leq \tau_2$ 时，$\mathsf{List}(\tau_1) \leq \mathsf{List}(\tau_2)$。当两个类型不相交，即不存在同时属于这两种类型的值时，我们写作 $\tau \# \tau'$。其规则包括：

$$\frac{\hat{T} \dashv \hat{E} : \tau \quad \tau' \leq \tau}{\hat{T} \dashv \hat{E} : \tau'} \qquad \frac{v \in \tau \quad \tau \# \tau'}{\hat{T} \dashv v : \tau'} \qquad \frac{x : \tau \in \hat{T} \quad \tau \# \tau'}{\hat{T} \dashv x : \tau'} \qquad \frac{c \notin \mathrm{dom}\, \Delta}{\hat{T} \dashv c(\vec{\hat{E}}) : \mathsf{Val}}$$

$$\frac{c(\vec{\tau}) : \delta \in \Delta \quad |\vec{\hat{E}}| \neq |\vec{\tau}|}{\hat{T} \dashv c(\vec{\hat{E}}) : \mathsf{Val}} \qquad \frac{c(\vec{\tau}) : \delta \in \Delta \quad \hat{T} \dashv \hat{E}_i : \tau_i}{\hat{T} \dashv c(\vec{\hat{E}}) : \mathsf{Val}} \qquad \frac{c(\vec{\tau}') : \delta \in \Delta \quad \delta \# \tau}{\hat{T} \dashv c(\vec{\hat{E}}) : \tau} \qquad \frac{\hat{E} \in \vec{\hat{E}} \quad \hat{T} \dashv \hat{E} : \tau}{\hat{T} \dashv [\vec{\hat{E}}] : \mathsf{List}(\tau)}$$

$$\frac{f \notin \mathrm{dom}\, \Psi}{\hat{T} \dashv f(\vec{\hat{E}}) : \mathsf{Val}} \qquad \frac{f(\vec{\tau}) : \tau \in \Psi \quad |\vec{\hat{E}}| \neq |\vec{\tau}|}{\hat{T} \dashv f(\vec{\hat{E}}) : \mathsf{Val}} \qquad \frac{f(\vec{\tau}) : \tau \in \Psi \quad \hat{T} \dashv \hat{E}_i : \tau_i}{\hat{T} \dashv f(\vec{\hat{E}}) : \mathsf{Val}} \qquad \frac{f(\vec{\tau}) : \tau \in \Psi \quad \tau \# \tau'}{\hat{T} \dashv f(\vec{\hat{E}}) : \tau'}.$$

C.7 可满足性查询与符号命令

符号语义通过后端的可满足性命令检查 pc 和检查 state来扩展幽灵命令关系。对于 $\hat{\sigma} = (\hat{s}, \hat{\mu}, \hat{\mathcal{P}}, \hat{\pi}, \hat{T})$，仅当相应的查询没有返回不可满足时，规则才会继续：

$$\frac{\mathrm{checkSat}_{\hat{T}}(\hat{\pi}) \in \{\mathsf{SAT}, \mathsf{UNKNOWN}\}}{\hat{\sigma}, \mathsf{check\ pc} \Downarrow_\Gamma^m ok : \hat{\sigma}} \qquad \frac{\mathrm{checkSat}(m, \hat{\sigma}) \in \{\mathsf{SAT}, \mathsf{UNKNOWN}\}}{\hat{\sigma}, \mathsf{check\ state} \Downarrow_\Gamma^m ok : \hat{\sigma}}$$

当结果为 不可满足时不存在成功的规则；此类分支会被剪枝。因此，检查路径条件只检查路径条件，而检查状态 则调用完整状态检查 $\mathrm{checkSat}(m, \hat{\sigma})$，参见第 6.7节。

D 评估示例

我们在所有 VeriFast 实验中使用了 VeriFast 26.10。下面的两个 VeriFast 文件都没有主函数，因此 VeriFast 要求使用 -shared 标志来验证它们：例如，verifast -shared reverse_list.c。

D.1 VeriFast：非破坏性链表反转

```
// 为 malloc 引入（下文的 prepend 会用到）。stdlib.h 会间接引入
VeriFast'的链表<类型> 库，但此处不使用该链表库，以使本文件自包含。#
include "stdlib.h"


/* * 图 2 中的非破坏性链表反转。 * * 本文件有意不复用 VeriFast 的标准库（前奏库）
链表<类型> 或其 append/reverse 不动点函数。 * 相反，它从头定义了自己的链表抽
象数据类型（图 2b）以及自己的 ++ 和 dagger * 逻辑函数（图 2c），因此翻译是自
包含的，并直接对应图示。 * * 命名：论文中的 []，::、 ++、和 dagger 在此分别命
名为 LNil、LCons、concat * 和 dagger，因为 C/VeriFast 标识符不能包含这些符号。
同样，reverse' 命名为 reverseP（而 alpha'、a' * 命名为 alphaP、aP），因为 C 标
识符不能包含 '。 * * 注意：论文中的 Val 在此译为 int，因为 VeriFast 没有与 Val 对
应的类型。 */
```

```
结构体 节点 {整数 值;结
构体 节点 *下一个; };
```

```
// C语言 re q需要 prep在 reverseP 调用它之前声明结束。
结构体 节点 *前置(整数 a, 结构体 节点 *x);
```

```
// （a）函数及其规约 // {list(x, alpha) * list(y, beta)} // reverse'(x, y) // {list(x,
alpha) * list(ret, alpha^dagger ++beta)} 节点结构体 *reverseP(节点结构体 *x, 节点
结构体 *y) //@ requires list(x, ?alpha) &*& list(y, ?beta); //@ ensures list(x,
alpha) &*& list(result, concat(dagger(alpha),beta)); { 节点结构体 *r; //@ open
list(x, alpha); if (x == 0) { r = y; //@ close list(x, alpha); } else { 整数 h = x->
value; 节点结构体 *t =x->next; y = prepend(h, y); r = reverseP(t, y); /*@
switch (alpha) {
```

case LNil: case LCons(a, alphaP): concatr(dagger(alphaP), a, beta); } @*/ //@ close list(x, alpha); } return r; } /*@ // （b）抽象数据类型定义：列表 ::= [] | Val :: 列表 归纳定义 列表 =LNil | LCons(整数, 列表); // （c）逻辑函数定义：（alpha : 列表） ++ (beta : 列表) : 列表 不动点 列表 concat(列表 alpha, 列表 beta) { switch (alpha) { case LNil: return beta; case LCons(a, alphaP): return LCons(a, concat(alphaP, beta)); } } // （c）逻辑函数定义：（alpha : 列表)^dagger : 列表 不动点 列表 dagger(列表 alpha) { switch (alpha) { case LNil: return LNil; case LCons(a, alphaP): return concat(dagger(alphaP), LCons(a, LNil)); } } // （d）列表谓词定义 谓词 list(节点结构体 *x, 列表 alpha) =x == 0? alpha == LNil : x->value |-> ?a&*& x->next |-> ?y &*& malloc_block_node(x) &*& list(y, ?alphaP) &*& alpha == LCons(a, alphaP); // （e）引理陈述及证明 // concatr：对于所有 alpha、a、beta。 // alpha 属于列表 * beta 属于列表 ==>// alpha ++ (a :: beta) = (alpha ++ [a]) ++ beta // (|alpha| 递减) // // VeriFast 采用静态类型，因此论文中关于 alpha 和 // beta 的类型假设由构造保证，下面的前置条件实际上为真。 // // VeriFast 的引理没有 decreases/terminates 子句（与 // 不动点、循环以及带有 terminates 注解的 C 函数不同）。引理 // 递归会自动检查，因为它是结构性的： // 每次递归调用都针对 alphaP，它是通过模式匹配获得的 alpha 的严格子项。 lemma void concatr(List alpha, int a, List beta) requires true; ensures concat(alpha, LCons(a, beta)) == concat(concat(alpha, LCons(a, LNil)

), beta);

{ switch (alpha) {
case LNil:

case LCons(aP, alphaP): concatr(alphaP, a, beta); } } @*/ // （f）前置的规约 // {list(x, alpha)} // prepend(a, x) // {list(ret, a :: alpha)} struct node *prepend(int a, struct node *x) //@ requires list(x, ?alpha); //@ ensures list(result, LCons(a, alpha)); { // 与本文所采用的线性内存模型不同，在该模型中分配总是成功；在 VeriFast/C 中，// malloc 可能失败（返回 0），因此我们必须进行空指针检查。 struct node *n =malloc(sizeof(struct node)); if (n == 0) { 中止(); } n->value = a; n->next = x; //@ close list(n,LCons(a, alpha)); return n; } /*@ // 下面的 reverse(x) 需要此引理。在标准库中，// 借助自动的 append_nil 引理，可以免费获得对应的事实。 lemma void concat_nil(List alpha) requires true; ensures concat(alpha, LNil) == alpha; { switch (alpha) { case LNil: case LCons(a, alphaP): concat_nil(alphaP); } } @*/ // 我们正在验证的顶层函数（图 2 中未显示） // {list(x, alpha)} // reverse(x) // {list(x, alpha) * list(ret, alpha^dagger)} struct node *reverse(struct node *x) //@ requires list(x, ?alpha); //@ ensures list(x, alpha) &*& list(result, dagger(alpha)); { struct node *r; //@ close list(0,LNil); r = reverseP(x, 0); //@ concat_nil(dagger(alpha)); return r; }

D.2 VeriFast：有序二叉树插入

以下是 VeriFast 的 sorted_bintree.c 示例中的添加函数，包含所有必需的定义。我们只包含验证添加函数本身所需的定义，省略所有其他函数及其辅助引理。

```
包含 "stdlib.h"

/* * VeriFast 的 sorted_bintree.c 示例中的有序二叉树插入。 * * 我们只保留验证
add() 所需的内容，省略所有其他 * 函数及其辅助引理。除此之外，不更改任何标识符、
* 格式或证明提示；唯一的例外是，“Neededby Redux” * 提示现在作为行注释放在每
行之后，而不是作为块注释放在该行之前。* 我们之所以移动它们，是因为 C 语言中
的块注释不能嵌套：在 * 鬼注释块（本身就是一个块注释）中，原来的提示会在这里
使用的简单 C 语法高亮器中提前结束该块，* 导致高亮显示混乱。 */

树结构体 树{整数
value;树结构体 树
*left;树结构体 树
*right;};

/*@ 谓词 tree(struct tree *t,bintree b)  =switch(b){ case empty: return t==0; case
tree(a,bl,br): return t->value  |-> ?v &*& t->left  |-> ?l &*&t->right  |-> ?r &*& malloc_
block_tree(t) &*& tree(l,bl) &*& tree(r,br) &*& t!=0  &*& a==v ; }&*& inorder(b)==true; 归纳
定义 bintree = |empty  |tree(int,bintree,bintree); 不动点 bool t_contains(bintree b, int v) {
switch (b) { case empty: return false; case tree(a,l,r): return a==v  || (v  <  a? t_
contains(l,v):t_contains(r,v)); } } 不动点 bintree tree_添加(bintree b, int x) { switch (b) {
case empty: return tree(x,empty,empty); case tree(v,l,r): return x  <  v? tree(v,tree_添加
(l,x),r): (x==v? tree(v,l,r):tree(v,l,tree_添加(r,x))); } } 不动点 整数 最大值(bintree b){
switch(b){ case tree(a,bl,br): return (br==empty? a:max(br)); case empty: return 0; } } 不动
点 整数 min(bintree b){ switch(b){ case tree(a,bl,br): return (bl==empty? a:min(bl));
```

case 空堆: return 0; } } 不动点 布尔值 中序遍历(二叉树 b){ switch(b){ case 空堆: return true; case 树(a,bl,br): return (bl==空堆? true:最大值(bl)<a)&& (br==空堆? true: a < 最小值(br)) && 中序遍历(bl) && 中序遍历(br); } } 引理 void 最大值_conj_添加(二叉树 l,整数 v,整数 x) requires x <v &*& (最大值(l)<v || l==空堆) &*& 中序遍历 (l)==true; ensures 最大值 (树 _ 添加 (l,x))<v &*& 中序遍历 (l)==true; { switch(l){ case 空堆 : case 树(a,b,c):if(x < a){ if (b == 空堆) {} else {} // Redux 所需。 最大值_conj_添加(b,a,x); } if(a < x){ if (c == 空堆) {} else {} // Redux 所需。 最大值 _conj_ 添加 (c,v,x); } if ((x < a) == true) {} else { if (x == a) {} else {}} // Redux 所需。 if (c == 空堆) {} else {} // Redux 所需。 if (x == a) {} else {} // Redux 所需。 if (树_添加(c, x) == 空堆) {} else {} // Redux 所需。 } } 引理 void 最小值_conj_添加(二叉树 r,整数 v,整数 x) requires v <x &*& (v <最小值 (r)|| r== 空堆) &*& 中序遍历 (r)==true; ensures v < 最小值 (树 _ 添加 (r,x)) &*& 中序遍历 (r)== true; { switch(r){ case 空堆: case 树(a,b,c):if(a < x){ 最小值_conj_添加(c,a,x); } if(x < a){ 最小值_conj_添加(b,v,x); } } } 引理 void 树_添加_中序遍历(二叉树 b, 整数 x) requires 中序遍历(b)==true &*& t_contains(b,x)== false; ensures 中序遍历(树_添加(b,x))==true &*& t_contains(树_添加(b,x),x)==true; { switch (b) { case 空堆: case 树(v,l,r):if(x < v){ 最大值_conj_添加(l,v,x); 树_添加_中序遍历(l,x); } if(v < x){ 最小值_conj_添加(r,v,x); 树_添加_中序遍历(r,x);

```
    } } } @*/ 树结构体 *init_tree(int x) //@
requires true; //@ ensures
tree(result,tree(x,empty,empty)); { 树结构体 *t  =
malloc(sizeof(struct tree)); if(t!=0){ t->value=x; t->
left=0; t->right=0; //@ struct tree *l  =  t->left; //@
close tree(l,empty); //@ struct tree *r  =  t->rig
ht; //@ close tree(r,empty); //@ int v  =  t->value; //
@ close tree(t,tree(x,empty,empty)); return t; }
else{ 中止(); } }
```

```
void 添加(树结构体 树 *t, 整数 x)//@ requires tree(t,?b) &*& b!=empty &*& false==t_
contains(b,x) &*& inorder(b)==t                                          rue;
      //@ 确保 tree(t,tree_add(b,x)) &*& inorder(tree_add(b,x))==true; { //
 @ 打开 tree(t,b); int v=t->value; struct tree *l=t->left; //@ 打开 tree(l,?
 bl); //@ 关闭 tree(l,bl); struct tree *r=t->right; //@ 打开 tree(r,?br); //@ 关
 闭 tree(r,br); if(x  <  v){ if(l!=0){ add(l,x); //@ tree_add_中序遍历(b,x); //
 @ 关闭 tree(t,tree(v,tree_add(bl,x),br)); }else{ struct tree *temp=初始化_
 tree(x); t->left=temp; //@ 打开 tree(l,bl); //@ 关闭 tree(t,tree(v,tree(x,空堆,
 空堆),br)); //@ tree_add_中序遍历(b,x); } }else{ if(v  <  x){ if(r!=0){
 add(r,x); //@ tree_add_中序遍历(b,x); //@ 关闭 tree(t,tree(v,bl,tree_
 add(br,x))); }else{ struct tree *temp=初始化_tree(x); t->right=temp;
```

```
    //@ open 树(r,br); //@ close 树(t,树(v,bl,树
(x,empty,empty)));}}}}
```

D.3 我们的 CSE 理论：有序二叉树插入

我们按照添加（附录 D.2）中理论的记法，以图 2 的风格呈现2中的验证。不同于图 2，我们保留用于折叠和展开树谓词的幽灵命令，以展示其在实践中的使用方式。我们按照源文件中的相同顺序给出定义，并将 VeriFast 的 `int` 转换为我们的自然数 Nat。我们的理论还要求为每个递归引理（第 4.3节）提供一个度量，因此我们添加了逻辑函数 size；该函数不在源文件中，并将其用于度量。

谓词定义。

$\mathsf{tree}(x, b) \;=\; ((x = \text{空值} \;\bigstar\; b = \text{空堆}) \;\vee (\exists a, l, r, bl, br.\; x \mapsto a, l, r \bigstar\; b = \mathsf{tree}(a, bl, br) \bigstar\; x \neq \text{空值} \bigstar \mathsf{tree}(l, bl) \bigstar \mathsf{tree}(r, br)) \bigstar \mathsf{inorder}(b) = \mathsf{true}$

抽象数据类型定义。二叉树::= 空 | 树(自然数, 二叉树, 二叉树) 逻辑函数定义。t_包含(b : 二叉树 v : 自然数) : 布尔值 =模式匹配 b : 二叉树，匹配 空 →假 树$(a, l, r) \to a = v \vee (v < a ?$ t_包含(l, v) : t_包含$(r, v))$ 当 真 树_添加(b : 二叉树 x : 自然数) : 二叉树 =模式匹配 b : 二叉树，匹配 空 → 树 $(x,$ 空 , 空) 树 $(v, l, r) \to x < v ?$ 树 $(v,$ 树 _ 添加(l, x) $r)$: $(x = v ?$ 树(v, l, r) : 树$(v, l,$ 树_添加$(r, x)))$ 当 真 最大值(b : 二叉树) : 自然数 =模式匹配 b : 二叉树，匹配 树$(a, bl, br) \to br =$ 空 ? a: 最大值 (br) 空 → 0 当 真 最小值 (b : 二叉树) : 自然数 = 模式匹配 b : 二叉树，匹配 树$(a, bl, br) \to bl =$ 空 ? a: 最小值(bl) 空 → 0当真

中序遍历(*b* : 二叉树) : 布尔值 =模式匹配 *b* : 二叉树， 匹配 空 →真 树(*a*, *bl*, *br*) →(*bl* = 空？真 : 最大值(*bl*) < *a*)∧ (*br* =空？真： *a* < 最小值(*br*))∧中序遍历(*bl*) ∧ 中序遍历(*br*) 当 真 大小(*b* : 二叉树) : 自然数 = 模式匹配 *b* : 二叉树， 匹配 空 → 0 树 (*a*, *l*, *r*) → 1 + 大小 (*l*) + 大小 (*r*) 当 真 引理语句与证明。最大值 _ 合取_添加： ∀*l*, *v*, *x*.*x* < *v* ★ (最大值(*l*) < *v* ∨ *l* = 空) ★ 中序遍历(*l*) = 真 =⇒最大值 (树 _ 添加 (*l*, *x*)) < *v* ★ 中序遍历 (*l*) = 真 { 匹配 (*l* : 二叉树) { 空 → 空操作 树(*a*, *b*, *c*) →若 (*x* < *a*) {应用 最大值_合取_添加(*b*, *a*, *x*)} 否则 {空操作}; 若 (*a* < *x*) {应用 最大值_合取_添加(*c*, *v*, *x*)} 否则 {空操作} } } (大小(*l*) 递减) 最小值_合取_添加： ∀*r*, *v*, *x*.*v* < *x* ★ (*v* < 最小值(*r*) ∨ *r* = 空) ★ 中序遍历(*r*) = 真 =⇒< 最小值(树_添加(*r*, *x*)) ★ 中序遍历(*r*) = 真 { 匹配 (*r* : 二叉树) { 空 → 空操作 树(*a*, *b*, *c*) →若 (*a* < *x*) {应用 最小值_合取_添加(*c*, *a*, *x*)} 否则 {空操作}; 若 (*x* < *a*) {应用 最小值_合取_添加(*b*, *v*, *x*)} 否则 {空操作} } } (大小(*r*) 递减) 树_添加_中序遍历 : ∀*b*, *x*. 中序遍历 (*b*) = 真 ★ t_ 包含 (*b*, *x*) = 假 =⇒中序遍历 (树_添加(*b*, *x*)) = 真 ★ t_包含(树_添加(*b*, *x*) *x*) =

true

{模式匹配 (*b* :二叉树) {空 → 空操作树(*v*, *l*, *r*) →如果 (*x* < *v*) {
应用最大值_合取_添加(*l*, *v*, *x*); 应用树_添加_

中序遍历(*l*, *x*)} 否则 {空操作};
如果 (*v* < *x*) {应用最小值_合取_添加(*r*, *v*, *x*); 应用树_添加 _中序遍历(*r*, *x*)} 否则 {空操作}
}}(大小(*b*) **递减**)

辅助函数及其规约。

{x = x ★ 真}初始化_树(x) { t :=新建(3);[t] :=x;[t + 1] :=空值;[t + 2] :=空值; 折叠 树(空值, 空堆); 折叠 树(空值, 空堆); 折叠 树(t, 树(x,空堆, 空堆)); 返回 t }{树(ret, 树(x, 空堆, 空堆))}

函数及其规约。严格来说，Lööw 等人提出的 CSE 理论中谓词的折叠与展开 [34] 基于所谓的输入和输出参数。这里提及这一点是因为，下文带问号变量的展开命令严格来说并未完全转换为该理论。不过，由于本文不讨论输入和输出参数，我们忽略 VeriFast 与该理论之间这一细微差异，并在下文直接复用 VeriFast 的语法。

{t = t ★ x = x ★ 树(t, b) ★ $b \neq$ 空堆 ★ t_包含(b, x) = f alse ★ 中序遍历(b) =true}
添加(t, x) { 展开 tree(t b); v := [t]; l := [t + 1]; 展开 tree(l, ?bl); 折叠 tree(l bl); r := [t + 2]; 展开 tree(r, ?br); 折叠 tree(r br); if (x < v) { if (l ≠空值) { ignore := 添加(l, x); apply tree_添加_中序遍历(b, x); 折叠 tree(t, tree(v, tree_添加(bl, x) br)) } else { temp :=init_tree(x);[t + 1] :=temp; 展开 tree(l bl); 折叠 tree(t, tree(v, tree(x, empty, empty) br)); apply tree_添加_中序遍历(b, x) } } else { if (v < x) { if (r ≠ 空值) { ignore := 添加(r, x); apply tree_添加_中序遍历(b,x); 折叠 tree(t, tree(v bl, tree_添加(br, x))) } else { temp :=init_tree(x);[t + 2] :=temp; 展开 tree(r br);

折叠 tree(t, tree(v bl, tree(x, empty, empty))) } } }; return 空值 }{tree(t, tree_添加(b, x)) ★ 中序遍历(tree_添加(b, x)) = true ★ ret = 空值}

D.4 后端评估详情

表 2 给出了后端评估按语言划分的详细情况。每次运行的超时时间为 120 秒。其中一次运行是对 C AWS 案例研究的主函数进行的，触发了超时；由于比较是查询局部的，我们纳入了此前生成的 2,948 个查询。同样，有几次运行以分析错误结束，Gillian 在不使用经验证的后端时也会报告这些错误；我们纳入了这些运行所生成的查询。

表 2. Gillian 示例生成的 SMT 查询的后端比较。计时列报告两个后端均能回答的查询上的求解器时间中位数 / 第 95 百分位数，单位为毫秒；测量的仅为求解器时间。

语言	Runs	查询	已翻译	已回答	一致	Gillian 毫秒	已验证毫秒
JavaScript	16	1324	985	937	923	0.25 / 0.77	0.47 / 311.59
C	7	4073	2989	2988	2988	0.07 / 0.42	0.14 / 2.29
WISL	9	423	416	416	416	0.09 / 0.22	0.40 / 0.96
总计	32	5820	4390	4341	4327	0.09 / 0.59	0.16 / 2.68

未经验证的代码。仅提取了第6.3 节的表达式编码器。将 Gillian 表达式转换为编码表达式的桥接器，以及将编码后的表达式发送给 SMT 求解器的 OCaml 代码，均未经验证；后者还会输出背景声明。具体而言，该实验没有运行第6.6节的操作可满足性检查：它既不执行 -检查，也不应用可表示性保护条件；此外，它没有声明任何用户定义的代数数据类型或逻辑函数，因为 Gillian 的查询不会用到这些内容。省略保护条件只会弱化查询，因此，不带这些条件得出的不可满足结论，带上这些条件后仍然是不可满足。出于与保护条件相同的原因，桥接器会转换 Gillian 中的非正浮点数字面量（其中几乎全部为`0.`），编码器会像处理其他有理数一样处理它们。它不会转换负整数字面量，因为我们理论中的整数是自然数。提取出的代码还包含一个小型补丁，仅通过添加内容来支持三种我们的理论尚未涵盖的 Gillian 值。

未翻译查询。 在 1,430 个未翻译查询中，有 1,111 个包含负整数字面量，其中 1,068 个仅受此因素阻碍；1,111 个查询中有 988 个来自 C语言。此外，212 个包含量词，195 个包含集合，39 个包含列表切片，3 个包含列表反转；这些类别之间存在重叠。

求解器耗时尾部。 编码器可以通过两种方式之一对列表长度进行编码：精确编码为表示该列表的 SMT-LIB 序列的长度，或抽象编码为列表的未解释函数，其值可由求解器自由选择，而无需构造列表。经验证的编码器始终精确编码长度。Gillian 的编码器会根据参数，针对查询中每个长度应用分别选择编码方式。对于列表字面量（例如 $[1, 2, 3]$, ）、任何其他复合列表表达式，以及查询中还在其他地方使用的列表变量，它都会精确编码其长度。如果列表是一个只在查询中作为长度参数出现的变量，

然而，查询仅通过列表长度对列表施加约束，因此列表内容并不重要，Gillian 的编码器便会抽象编码该长度。抽象编码不会使可满足查询变得不可满足，因为精确查询的模型也是抽象查询的模型，所以它给出的不可满足答案总是正确的。当查询还要求长度非负，且列表变量未受量词约束时，它给出的可满足答案也是正确的，因为任意一个具有所选长度的列表都可以作为见证。我们这次运行中的每个查询都满足这两个条件，其中 3,170 个查询使用了抽象编码；不过，维持这些条件的是 Gillian 的符号引擎，而不是它的编码器。使用经验证的编码时，最慢的 67 个查询都要求求解器构造列表：最慢的查询中，约束$32 \le |l|$要求列表至少包含 32 个元素；仅对 l 的长度进行抽象编码，就能使该查询的速度与其他查询一样快。如果我们修改经验证的编码器，像 Gillian 的编码器那样对这些长度进行抽象编码，那么这 67 个查询都会给出相同答案，耗时也会与其他查询一样。