arXiv:2302.05792v1 [cs.PL] 11 Feb 2023

带等式的 Datalog 求值算法

Martin E. Bidlingmaier

摘要

我们介绍一种用于 *relational Horn logic (RHL)* 的求值算法。RHL 在 Datalog 的基础上扩展了对种类的量化、结论中的存在量化，以及至关重要的推断 *equalities* 的能力。这些特性使 RHL 求值能够涵盖并扩展 Datalog 和同余闭包算法的应用。

我们阐述如何将快速同余闭包算法的某些方面融入 Datalog 求值，从而获得高效的 RHL 求值算法。随后，我们概述如何利用 RHL 求值实现 Steensgaard 指向分析和类型推断。RHL 及本文所述的求值算法是 *Eqlog* Datalog 引擎的基础。

1 引言

近期研究已指出，关系 Horn 逻辑（RHL）和部分霍恩逻辑（PHL）是 Datalog 在语义上表现良好的扩展（Bidlingmaier，2023）。本文介绍如何将 Datalog 求值算法推广至 RHL 求值。

RHL 理论包含以下数据的声明：

- 一个 S 类型集合。
- 一个 R 关系集合。
- 元数 r： $s_1 \times \cdots \times s_n$ 适用于所有关系 $r \in R$。
- 一个序列式集合（或规则集合、公理集合），其中每个元素的形式为 $\mathcal{F} \implies \mathcal{G}$，其中 $\mathcal{F}$ 和 $\mathcal{G}$ 是由原子式 ϕ_i 构成的合取 $\phi_1 \wedge \cdots \wedge \phi_n$。

Datalog 引擎通常接受以下形式的规则，而不是上述 RHL 序列式：

$$\phi \text{ :- } \phi_1, \ldots, \phi_n.$$

头部 ϕ 对应于一个 RHL 序列式的结论，并由单个原子式组成。体 $\phi_1, \ldots, \phi_n$ 对应于一个 RHL 序列式的前提，可以包含多个原子式，并按合取解释。因此，乍看之下，RHL 序列式的结构更为一般，因为 RHL 序列式的结论可以是原子式的合取。不过，一个包含 n 个结论原子式的序列式，等价于 n 个序列式，每个序列式都只有一个结论原子式。因此，仅仅允许结论为一般的合取，并不会带来更大的表达能力。

RHL 对 Datalog 的推广体现在它允许使用何种原子式，以及如何处理变量。在 Datalog 中，每个原子式的形式为 $r(v_1, \ldots, v_n)$，其中 r 是一个关系符号，而 v_i 是变量，其类型与 r的元数相匹配。除了这类关系原子式，RHL 还认可以下类型的原子式:

1. 一个等式原子式

$$u \equiv v$$

其中 u 和 v 具有相同的类型。我们将符号 $\equiv$ 保留用于 RHL 语法，而 $=$ 用于元理论等式。

2. 类型量化

$$v\downarrow$$

其中 v 是一个类型已知为 s的变量。如果序列式中的其他原子式无法确定 v 的类型，我们也使用语法 $v : s$ 作为 $v\downarrow$ 的同义形式，并断言 v 的类型为 s。

如果等式原子式 $u \equiv v$ 出现在序列式的前提中，那么只有当 u 和 v 被解释为同一个常量时，前提的匹配才有效。因此，可以通过在序列式中用其中一个变量替换另一个变量的所有出现位置，来消除前提中的等式原子式。

然而，序列式结论中的等式原子式 $u \equiv v$ 的语义并不简单：每当某个序列式的前提匹配成功，使得 u 被解释为常量 a，且 v 被解释为常量 b时，我们期望求值引擎此后在所有上下文中都将 a 和 b 视为相同。例如，如果先前已推导出等式 $b_1 = b_2$，那么传递性公理 $\mathsf{Le}(u, v) \wedge \mathsf{Le}(v, w) \implies \mathsf{Le}(u, w)$ 的前提应当匹配元组 (a, b_1)、$(b_2, c) \in \mathsf{Le}$。

部分函数可以在 RHL 中使用表示部分函数函数图的关系进行编码。因此，可以将部分函数 $f\colon s_1 \times \cdots \times s_n \to s$ 与关系 $f\colon s_1 \times \ldots s_n \times s$ 对应起来，其中每个条目的前 n 个分量表示函数定义域中的一个元素，最后一个分量表示函数值。函数性公理

$$f(v_1, \ldots, v_n, u) \wedge f(v_1, \ldots, v_n, w) \implies u \equiv w \tag{1}$$

确保关系 f 确实对应于一个定义良好的部分函数。

类型量化 $v : s$ 在前提中允许匹配给定类型 s 的元素，即使这些元素没有出现在任何关系中。在标准 Datalog 中，序列式头部中的所有变量也必须出现在体中。RHL 不再要求这一点。只出现在结论中的变量会被隐式地存在量化。如果序列式的前提匹配，则求值引擎必须通过寻找仅出现在结论中的变量的解释来扩展该匹配，使结论成立。如果不存在这样的扩展，求值引擎就必须创建新的标识符来解释仅出现在结论中的变量，并确保结论中的原子式在该解释下成立。我们希望求值引擎输出每种类型的标识符列表，其中包括求值过程中创建的标识符。所有结论变量

也出现在前提中的 RHL 序列式称为满射序列式。如果 RHL 理论包含非满射序列式，则求值不一定会终止。Souffle Datalog 引擎通过其选择构造实现了类似的机制（Hu 等，2021）。

非满射序列式的存在不仅可能导致非终止，还可能导致非确定性结果，也就是说，结果取决于匹配序列式的顺序。强 RHL 理论则不会出现这种情况，因为一旦所有序列式都得到满足，结论变量的解释便是唯一确定的。例如，由函数性序列式 (2.2) 和序列式

$$v_1 : s_1 \wedge \cdots \wedge v_n : s_n \implies f(v_1, \ldots, v_n, v) \tag{2}$$

构成的 RHL 理论是强的，因为如果函数性公理得到满足，那么序列式 (1) 中变量 v 的解释便是唯一确定的。遗憾的是，判断给定的 RHL 理论是否为强理论是不可判定的。

RHL 的另一个问题是，必须通过函数图来编码函数。这使得在表述涉及由函数符号构成的复杂表达式的公理时，内容会过于冗长。

部分霍恩逻辑（PHL）（Palmgren and Vickers，2007）是建立在 RHL 之上的一个语法层，用以弥补这些不足。在部分霍恩逻辑中，必须明确声明关系是谓词还是部分函数。谓词与 RHL 中的关系直接对应，而函数则被转换为一个关系，该关系对应于函数的函数图以及隐含函数性公理 (2.2)。

在 RHL 接受变量的位置，PHL 也允许使用复合项

$$t ::= v \mid f(t_1, \ldots, t_n)$$

复合项由变量以及将部分函数符号应用于类型与函数签名匹配的项而递归定义。将 PHL 降低为 RHL 时，复合项 t 会被递归地降低为表示该项 t 的结果变量，以及额外的 RHL 原子式。根据 t 出现的位置，这些额外的原子式会插入序列式的前提或结论中。要降低复合项 $t = f(t_1, \ldots, t_n)$，我们可以假设通过先递归降低参数 $t_1, \ldots, t_n$，得到 $t = f(v_1, \ldots, v_n)$，其中变量为 v_i。现在，我们选择一个新的变量 v 来表示 t，并添加 RHL 原子式 $f(v_1, \ldots, v_n, v)$。由于降低 PHL 公式会将嵌套表达式化为扁平的原子式序列，因此将 PHL 降低为 RHL 的过程也称为扁平化。

如果结论中的所有变量都已在前提中引入，则该 PHL 序列式是满射的。请注意，将满射 PHL 序列式降级后，可能会得到非满射的 RHL 序列式，因为降级复合项时可能会引入新变量。尽管如此，由满射 PHL 理论（即仅包含满射序列式的 PHL 理论）降级得到的 RHL 理论是强的。反过来，每个强 RHL 理论都等价于一个满射 PHL 理论；详情请参阅 Bidlingmaier（2023，第 4.3 节）。因此，满射 PHL 与强 RHL 具有相同的表达能力。另一方面，检查一个 PHL 序列式是否为满射是轻而易举的，而判断一个 RHL 理论是否为强理论则不可判定。与 RHL 相比，这使 PHL 更适合作为求值引擎的面向用户输入语言。

本文描述了 Eqlog 引擎底层的算法，该引擎接受满射 PHL 作为输入语言。Eqlog 会将用户提供的满射 PHL 理论降级为 RHL，再将其转译为 Rust 模块。这类似于 Souffle Datalog 引擎将 Datalog 转译为 C++。与 Souffle 不同，Eqlog 旨在作为更大型项目的一部分使用，而非作为独立工具。与 egg 等式饱和库（Willsey 等，2021）类似，Eqlog 支持在线使用场景：在此场景中，用户交替向模型中插入新的基底事实，并根据 PHL 序列式对模型进行闭包运算。详情请参阅 Eqlog 项目主页（Eqlog 作者）。

与本文介绍的 Eqlog 工作无关，Egg 社区的成员创建了一种非常相似的工具，将 e-图与 Datalog 相结合，相关成果将在即将发表的文章中报告。

概要。第 2 节介绍用于对 RHL 进行求值的基本算法，以及一种在许多情况下检测终止的技术。随后，第 3 节讨论 RHL 求值算法的优化。最后，第 4 节概述 PHL 和 RHL 求值在编程语言实现中的应用。第 5 节作总结。

致谢。Jakob Botsch Nielsen 对 Eqlog 工具实现的早期版本及其在依赖类型证明助手的实验性类型检查器中的应用作出了重要贡献。我本人关于 Eqlog 以及本文所述算法的工作始于在奥胡斯大学攻读博士期间；当时 Bas Spitters 担任我的导师，我还获得了空军科学研究办公室“Homo- topy Type Theory and Probabilistic Computation”项目的资助，项目编号为 12595060。

2 RHL 求值

本节将介绍一种对 RHL 理论进行求值的基本算法。求值算法的输入是一个 RHL 序列式列表，以及一个表示基底事实的关系结构。关系结构由每种类型的数值标识符集合和每个关系的元组集合给出，其中前者表示该类型的元素。从第 2.2 节开始，我们假设关系结构还包含每种类型对应的并查集数据结构，用于表示该类型元素之间的语义相等。

如果 RHL 求值终止，那么我们要求输出是一个相对于输入关系结构、关于 RHL 序列式列表的弱自由关系结构（Bidlingmaier，2023）。直观地说，这意味着输出必须满足 RHL 理论中的所有序列式，并且只能通过匹配序列式前提以及添加与结论相对应的数据，从输入关系结构中构造得到。

弱自由性并不能唯一刻画一个关系结构。一般而言，输出关系结构取决于 RHL 序列式前提的匹配顺序。然而，如果 RHL 理论是强的，那么输出关系结构就是输入关系结构上的（强）自由关系结构，并且除唯一同构（即重命名标识符）之外，它是唯一确定的。相关的强 RHL 理论类别包括仅包含满射序列式的理论，以及通过降低满射 PHL 理论得到的理论（Bidlingmaier，2023）。

在第 2.1 节中，我们讨论 Datalog 求值的朴素算法，并

为其添加对非满射序列式和类型量化的支持，但不支持相等。接着，在第 2.2 节中，我们考察一种众所周知的简单同余闭包算法，并将其理解为一种专用于求值函数性 RHL 序列式的算法。函数性序列式的结论是一个等式，而我们的朴素 Datalog 求值算法无法处理等式。并查集数据结构和规范化是该同余闭包算法中专门用于处理等式的部分。我们将这些机制整合到朴素 Datalog 算法中，从而得到朴素 RHL 求值算法。

在第 2.3 节中，我们讨论了一个例子：对于具有有限自由模型的非满射 RHL 理论，我们的 RHL 求值算法无法终止。基于这个例子，我们展示了如何修改求值算法，使其在这个特定例子以及其他多种 RHL 理论上都能终止。

2.1 朴素 Datalog 求值

朴素 Datalog 算法通过重复执行前提匹配和应用结论两个阶段，直至达到不动点。该算法的高层结构可以用类似 Rust 的伪代码表示如下：

```
fn datalog(structure,sequents){loop {// 1. 匹配前提。let matches = [];
for sequent in sequents{matches.push(find_matches(structure,
sequent.premise));}// 2. 应用结论。let has_changed = false;for
(sequent, matches)in sequents.zip(matches){for matchin matches
{ifapply_conclusion(structure, sequent.conclusion, match) {has_
changed = true;}}}// 如果应用结论没有产生任何影响，则终止。if !
changed{break;}}return structure;}
```

find matches 是一个子过程，用于返回给定公式在关系结构中的匹配列表。每个匹配都由一个映射给出，该映射将公式中出现的变量集合映射到关系结构中的元素。A

此函数的朴素实现使用嵌套循环连接枚举匹配。例如，公式 Le(u, v) ∧ Ge(w, v) ∧ Le(w, x) 的匹配可以按如下方式枚举：

```
for (u, v) in structure.rels[Le] { for (w,
v1) in structure.rels[Ge]{ if v1 != v {
continue; } for (w1, x) instructure.rels[Le
] { if w1 != w { continue; } 匹配.push({u,
v, w, x}); } } }
```

每个关系原子都会转换为针对关系结构中相应关系的嵌套循环，而每个类型量化都会转换为针对相应元素列表的循环。

应用结论是一个子过程，它根据给定的结论以及将变量替换为关系结构中元素的替换，将数据插入关系结构。它返回一个布尔值，表示该操作是否产生了效果，也就是说，所推导的数据中是否至少有一部分此前尚未存在于关系结构中。对于不含等式的满射序列式，其中结论中的每个变量都已由前提的匹配绑定，我们会替换每个关系原子中的变量，并将相应的元组插入关系结构。

对于非满射序列式，我们首先检查给定的前提变量替换能否扩展为结论变量的解释，使得结论成立。为此，可以使用查找匹配函数的一个变体，该函数接受一个列表，其中列出公式中部分变量已经固定的解释。如果不存在这样的扩展，那么我们就在关系结构中添加新元素，用来解释尚未绑定的结论变量，然后按照满射情形继续处理。

2.2 同余闭包与朴素 RHL 求值

可以通过为每种类型引入表示推导出的相等关系的二元等式关系，将 RHL 等式原子约化为 Datalog。然而，这种 setoid 约化（Bidlingmaier，2023，第 3.4 节）通常会导致效率低下的 Datalog 程序。从语义上看，推导出的相等关系往往会缩小关系结构，因为等式可以将某个关系中原先不同的两个元组合并为一个元组。相反，setoid 约化会因同余公理而造成大量重复；这些公理断言，所有关系在每个参数位置上都必须对推导出的相等关系封闭。本节的目标是弥补这一缺陷：每个推导出的相等关系都应当只会缩小关系结构。

请注意，RHL 尤其可以用于解决同余闭包问题：判断一组子表达式之间的等式能推出一列表达式 t_1、.....、t_n 中的哪些等式。该问题可以编码到 RHL 中，其理论由一个 $n+1$ 元关系符号 f 给出，用来表示一个在 t_i 中出现的 n 元函数符号的函数图，以及函数性

公理

$$f(v_1, \ldots, v_n, u) \wedge f(v_1, \ldots, v_n, w) \implies u \equiv w. \tag{3}$$

然后，将与 t_i 对应的数据插入关系结构中，在子表达式之间施加等式，并根据函数性公理对该结构进行闭包。因此，我们可以将同余闭包算法视为用于函数性 RHL 序列式的专用求值算法，并尝试将现有的同余闭包算法推广到一般的 RHL 求值。

我们的灵感来自 Downey 等人（1980）描述的同余闭包算法。考虑他们的朴素算法 2.1 的以下版本；他们将其归功于 Cocke 和 Schwartz（1970）。此处给出的版本针对单个二元函数进行了专门化。该算法的输入是一个三元组列表，表示函数图。

```
函数 同余_闭包(图) { 并查集 =并查集::新建(); 循环 {// 1. 匹配前提。令 等式 = []; 对于 graph 中的 (x0, x1, x2) { 对于 graph 中的 (y0, y1, y2) { 如果 x0 == y0 && x1 == y1 { 等式.push((x2, y2)); } } }// 2. 应用等式。令 有_变化 = 假; 对于等式中的 (lhs, rhs) { lhs = 并查集.find(lhs); rhs = 并查集.find(rhs); 如果 lhs != rhs { 并查集.union(lhs, rhs); 有_变化 = 真;}}// 如果没有任何变化，则终止。如果 !有_变化{ 中断; }// 3. 规范化。函数图0 = []; 对于 graph 中的 (x0, x1, x2) { graph0.push(并查集.find(x0), 并查集.find(x1), 并查
                                                
```

返回 uf; }

与第 2.1 节的 Datalog 求值算法类似，该同余闭包算法会重复执行若干步骤，直到达到不动点。步骤 1 对应于二元函数的函数性公理 (2.2) 的前提所用的查找匹配函数。

步骤 2 会对步骤 1 中找到的每个匹配应用结论 $u \equiv v$。该算法使用并查集数据结构来表示相等。并查集数据结构会为每个等价类指定一个规范代表元。当且仅当两个等价类具有相同的规范代表元时，它们才相等。

并查集数据结构支持快速的查找和合并操作，其运行时间接近常数。查找操作会计算给定元素所在等价类的规范代表元。合并操作会合并两个规范代表元各自所属的等价类。

步骤 3 会将函数图关系各项中的所有元素替换为规范代表元；这一步在 Datalog 求值中没有对应操作。由于使用了并查集数据结构，只有规范代表元之间的比较才能反映推断出的相等关系。请注意，除了执行规范化步骤，我们也可以在步骤 1 的前提匹配过程中比较元素时查询并查集数据结构。不过，单独执行规范化步骤使得多种优化成为可能，我们将在第 3 节讨论这些优化。

通过将同余闭包算法中处理等式的部分纳入第 2.1 节的朴素 Datalog 求值算法，我们现在得到朴素 RHL 求值算法：

1. 除了元素集合和关系之外，关系结构现在还为每种类型包含并查集数据结构，用于表示语义相等。我们保持以下不变式：在求值算法的每次迭代开始之前，关系结构中的关系只包含规范代表元。2. 应用结论通过调用 union 合并 u 和 v 的解释所对应的等价类，从而处理等式 $u \equiv v$。3. 在循环体结束之前，我们插入一个规范化步骤，通过调用 find，将任意关系中元组内的每个元素替换为其规范代表元。

由于关系结构将关系存储为不含重复元素的集合，规范化可能会减少关系中元素的数量。

Souffle Datalog 引擎提供了使用并查集数据结构实现等价关系的高效实现（Nappa 等人，2019），但它没有实现规范化。

2.3 检测终止

如果待求值的 RHL 理论中的所有序列式都是满射的，那么我们在第 2.2 节中描述的算法就保证会终止。然而，对于非满射序列式，有限结构上的（弱）自由模型

不一定是有限的。因此，在这些情况下，RHL 求值可能无法终止，必须在超时或推断出所需性质后中止。尽管如此，有些 RHL 理论即使包含非满射序列式，其在有限关系结构上的自由模型仍然是有限的。但即便在这些情况下，迄今为止讨论的 RHL 求值算法也不一定会终止。

例如，考虑以下公理化映射对 $f\colon A \rightleftarrows B\colon g$ 的 PHL 理论，使得对所有 $x \in A\colon\ g(f(x)) = x$

1. $x : A \implies f(x)\downarrow$
2. $y : B \implies g(y)\downarrow$
3. $y = f(x) \implies g(y) = x$

公理 3 被化为 RHL 公理 $f(x, y) \implies g(y, x)$，该公理是满射的。然而，公理 1 和 2 并非满射。尽管如此，这一理论在有限关系结构上的自由模型总是有限的，以下自由模型构造证明了这一点：给定集合 A, B 和关系 $f \subseteq A \times B$, $g \subseteq B \times A$，首先根据 f 和 g 的函数性公理以及公理 3，合并 A 和 B 中的元素。对于每个 g 未定义的元素 $b \in B$，我们将新元素添入 A，并根据公理 2 对 g 作相应扩展，使其成为全函数。类似地，对于每个 f 未定义的 $a \in A$，我们将新元素添入 B，并根据公理 1 对 f 作相应扩展，使其成为全函数。现在，B 中所有 g 未定义的元素都具有 $g(a)$ 的形式，其中 $a \in A$ 唯一，因此根据公理 3，我们可以通过令 $g(f(a)) = a$，将 g 扩展为全函数。现在，f 和 g 都是全函数，而 $g \circ f$ 是 A 上的恒等函数。

乍一看，我们可能会希望该理论的 RHL 求值最终达到不动点并终止。遗憾的是，第 2.2 节所述的 RHL 求值算法并非如此。考虑以下求值迭代：初始关系结构中，$A = \{a_0\}$、$B = \emptyset$、f 和 g 均未定义：

1. 公理 1 在 $a_0 \in A$ 上匹配，生成一个新元素 $b_0 \in B$ 和元组 $(a_0, b_0) \in f$。2. 公理 2 在 $b_0 \in B$ 上匹配，公理 3 在 (a_0, b_0) 上匹配。前者生成一个新元素 $a_1 \in A$ 和元组 $(b_0, a_1) \in g$，后者则生成元组 $(b_0, a_0) \in g$。3. 公理 1 在 $a_1 \in A$ 上匹配，而针对 g 的隐含函数性公理在 (b_0, a_0)、(b_0, a_1) 上匹配。前者生成一个新元素 $b_1 \in B$ 和元组 $(a_1, b_1) \in f$，后者则产生相等关系 $a_0 = a_1$。4. 公理 2 在 $b_1 \in B$ 上匹配，而针对 f 的隐含函数性公理在 (a_0, b_0)、(a_1, b_1) 上匹配。前者生成一个新元素 $a_2 \in A$ 和元组 $(b_1, a_2) \in g$，后者则产生相等关系 $b_0 = b_1$。5. 此后的所有迭代都在迭代 3 和 4 的不同变体之间交替进行。

因此，该理论无法终止的原因在于，求值算法会一次性匹配所有序列式：请注意，我们关于自由模型是有限的证明

依赖于按照某种特定顺序应用序列式。对于这一特定理论，可以通过谨慎地表述公理来避免交替状态，从而防止非终止；例如，用 $x : A \implies g(f(x)) = x$ 替换公理 1 和 3。

然而，Section 2.2 中描述的 RHL 求值算法的以下变体，在广泛的 RHL 理论上都能终止，包括上述理论。我们将顶层求值循环拆分为一个负责满射序列式的内层循环，以及一个负责非满射序列式的外层循环。因此，该算法交替执行以下操作：将关系结构在所有满射序列式下闭包化（这总会终止），以及执行一步匹配并添加非满射序列式的结论。如果最终某一步非满射处理没有改变关系结构，那么所有序列式都已满足，求值便会终止。

不过，我预计存在这样的理论：其终止性取决于应用非满射序列式的特定顺序；在这种情况下，这种简单的方法并不足够。

3 优化

本节将讨论我们在 Section 2.2 中介绍的朴素 RHL 算法的优化。这些技术大多改编自适用于 Datalog 求值、同余闭包问题或两者的优化。将这些优化结合实现后，我们便能将 Downey 等人（1980）提出的快速同余闭包算法，作为函数性公理的 RHL 求值的一个特例重新实现。

3.1 半朴素求值

半朴素求值是一种常见的 Datalog 求值优化。它利用了这样一个观察：在前一次迭代中找到的前提匹配无需再次考虑，因为这些匹配对应的结论已经被加入。若一个前提中的至少一个原子式与新数据匹配，即数据仅在上一次迭代中被添加，则该匹配在之前的迭代中尚未被找到。为了区分旧数据和新数据，我们针对每个关系和每个类型，分别存储在上一次迭代中添加的元组或元素列表。现在，可以将用于匹配序列式前提的 n 重嵌套循环替换为 n 个副本，其中在第 i 个副本中，第 i 个循环仅遍历新数据。例如，第 2.1 节中描述的、枚举 Le(u, v) $\wedge$ Ge(w, v) $\wedge$ Le(w, x) 前提的嵌套循环，可以替换为以下三个嵌套循环：

```
for (u, v) in structure.rels_new[Le] { for (w,
v1) in structure.rels_all[Ge] { if v1 != v {
continue; } for (w1, x) instructure.rels_all[Le]
{ if w1 != w { continue; } 匹配.push([u, v, w, x]);
} } }
```

```
for (u, v) in structure.rels_all[Le] { for (w, v1)
in structure.rels_new[Ge] { if v1 != v {
continue; } for (w1, x) instructure.rels_all[Le]
{ if w1 != w { continue; } 匹配.push([u, v, w, x]);
} } } for (u, v) in structure.rels_all[Le] { for (
w, v1) in structure.rels_all[Ge] { if v1 != v {
continue; } for (w1, x) instructure.rels_new[
Le] { if w1 != w { continue; } 匹配.push([u, v,
w, x]); } } }
```

请注意，不仅结论应用阶段会产生新数据，规范化阶段也会产生新数据：如果某个关系的元组中的一个元素因规范化而发生变化，那么该元组必须视为新元组。

Downey 等人（1980）描述的优化的同余闭包算法也实现了半朴素求值。他们在算法 2.4 中的待处理列表，对应于我们在表示函数图的关系中的新元组集合。

半朴素求值非常适用于在线应用，因为在这类应用中，RHL 求值与临时操作交替进行。如果这种操作仅包含添加数据，那么这些数据可以添加到在 RHL 求值期间存放新数据的同一数据结构中。这样，后续 RHL 求值的第一次迭代只需针对这些新数据查找匹配，而不必在整个关系结构中查找匹配。

3.2 对称性

对（二元）函数的函数性公理的前提进行半朴素匹配，会产生两个循环：

```
对于 (x0, x1, x2) in structure.rels_new[f] { 对于
(y0, y1,y2) in structure.rels_all[f] { ... } } 对于
(x0, x1, x2) in structure.rels_all[f] { 对于 (y0,
y1,y2) in structure.rels_new[f] { ... } }
```

另一方面，Downey 等人（1980）描述的同余闭包算法只需要一个循环。事实上，第二个循环并非必需，因为

函数性公理 $f(v_0, v_1, u) \wedge f(v_0, v_1, w) \implies u \equiv w$ 中存在对称性。这种对称性通过交换 u 和 w 得到。这样会产生语义等价的前提，而交换变量的效果与交换两个前提原子式相同。在这种情况下，只需考虑第一个原子式以新数据进行解释的匹配。反对称公理 $\text{Le}(u, v) \wedge \text{Le}(v, u) \implies u \equiv v$ 是另一个可以利用对称性的例子。

3.3 索引和出现列表

索引旨在加速枚举前提匹配项的嵌套循环。其思路是使用固定投影进行高效的亚线性查找，以替代每个内层循环。例如，传递性公理的前提 $\text{Le}(u, v) \wedge \text{Le}(v, w)$ 可以通过在第一个投影上建立索引来加速匹配。因此，需要维护一个映射，以便快速查找所有首项固定为 v 的元组 (v, w)。然后，可以按如下方式枚举该前提：

```
for (u, v) instructure.rels[Le] { for (_, w) in
structure.rels[Le].索引[v] { 匹配.push((u, v,
w)); } }
```

索引通常通过有序搜索树或哈希映射的变体实现。索引可以在所有迭代期间持续维护，也可以在每次迭代的前提匹配阶段开始前重新创建，并在该阶段结束后立即释放。与维护索引相比，重新创建索引所需的内存更少，因为任何时候只需将匹配单个序列式所需的索引存储在内存中。

快速 Datalog 引擎通常会在所有迭代过程中维护索引，从而以增加内存使用量为代价提高执行速度。如果在所有迭代过程中都维护索引，那么在结论应用阶段也必须将新元组插入索引。然而，对于 RHL 求值而言，由于规范化阶段的存在，索引维护会带来问题：在该阶段，关系中的元素会在必要时替换为其规范代表元。使用索引时，索引也需要规范化。

我们再次采用 Downey 等人（1980年，第2.4节）描述的快速同余闭包算法，以高效实现索引规范化。他们的签名表是一种哈希映射索引，可加快函数性公理前提的匹配。该索引在同余闭包算法的整个迭代过程中都会得到维护。高效规范化通过使用更多数据结构来实现，我们将这些数据结构称为出现列表。对于每个等价类，出现列表都会包含至少有一个子节点属于该等价类的表达式节点。在规范化步骤中，只需规范化那些出现在已不再是规范代表元的元素的出现列表中的元组。

出现列表可以按如下方式适配 RHL 求值。我们为每个规范代表元关联一个列表，其中包含该规范代表元出现的任意关系中的元组。出现列表中的元组无需规范化。

在结论应用阶段，我们还会将元组插入元组中每个元素的出现列表中。当合并两个等价类时，我们会保存不再作为规范代表元的元素

及其出现列表，以便在接下来的规范化阶段使用。仍作为规范代表元的元素的出现列表被设置为两个原始出现列表的拼接。如果将出现列表实现为链表或绳索数据结构，则可以高效地实现拼接操作。在规范化阶段，我们会移除之前保存的某个出现列表中的每个元组，对该元组进行规范化，然后将其重新插入每个索引中。

在结论阶段强制应用等式时，Downey 等人（1980年，第 2.4 节）所描述的算法会选择保留为规范代表元的元素，以尽量减少所需的规范化量：因此，应保留出现列表较长的元素为规范代表元。这同样直接适用于 RHL 求值中的出现列表。

为避免对当前迭代中插入的元组进行规范化，可以将结论应用阶段拆分为等式应用阶段和关系应用阶段：在等式应用阶段，我们只考虑结论中的等式；在关系应用阶段，我们只考虑关系原子式。然后，在等式应用阶段与关系应用阶段之间进行规范化。这样做的好处是，新元组无需在插入后立即进行规范化。

3.4 函数投影

我们称 RHL 理论中关系 r 的第 i 个投影：$s_1 \times \cdots \times s_n$ 具有函数性，如果每个元组 $(x_1, \ldots, x_n) \in r$ 的第 i 个投影 x_i 由该元组的其他分量 $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ 唯一确定。更一般地，我们可以考虑一组投影 I，它们由互补投影唯一确定。顾名思义，n元函数的函数性公理断言，该函数的函数图的第 $(n+1)$ 个投影具有函数性。另一个例子是单射函数，其中函数图的前 n 个投影函数性地依赖于第 $(n+1)$ 个投影。

当在具有函数投影的关系上维护索引时，可以在插入索引期间就生成等式约束，而不必等到之后的匹配阶段再生成。例如，如果 r 是表示函数图的一个 $(n+1)$ 元关系，那么可以维护一个针对前 n 个参数的索引，以匹配函数性公理的前提。不考虑第 $(n+1)$ 个投影的函数性时，我们预计该索引能够查询出前 n 个投影取固定值时的元组列表。由于函数依赖，我们可以改为强制要求查询结果至多包含一个元组。每当要向索引中插入第二个元组，且该元组会违反这一性质时，我们就根据函数投影生成等式约束。随后，这些等式会在下一结论应用阶段得到强制执行。

Downey 等人（1980年，第2.4节）所描述的高效同余闭包算法中的签名表哈希映射，可以理解为一种带有函数投影优化的索引。

4 应用

本节讨论 RHL 和 PHL 求值在编程语言实现中的两个应用：Steensgaard 指向分析

（第 4.1 节）和类型推断（第4.2节）。

4.1 Steensgaard 的指向分析

指向分析旨在限定程序中的变量可能指向的堆对象集合。两种著名的方法分别由 Andersen 和 Steensgaard 提出。这两种算法都将堆对象与其分配点对应起来，也就是说，与分配对象的程序表达式对应起来。因此，Andersen 分析和 Steensgaard 分析会对如下问题给出一个过度近似的答案：给定变量 x 是否可能指向在表达式 e 中分配的对象。

两种分析都必须考虑变量 x 被赋值给变量 y 的情况。Andersen 风格分析会为每个变量 x 计算一个对象集合 e，使得 x 可以指向这些对象，并满足以下性质：

1. 如果存在一条语句，将分配表达式 e 赋给变量 x，那么 x 可以指向 e。
2. 如果变量 y 可以取变量 x 的值（例如，由赋值或函数调用产生），并且 x 可以指向 e，那么 y 可以指向 e。

使用 Datalog 实现 Andersen 风格分析的最简版本，会根据程序源代码推导出的数据填充输入关系 Alloc(x, e) 和 Assign(x, y)。随后，将上述用于定义 PointsTo(x, e) 关系的规则编码为 Datalog，如下所示：

1. $\text{Alloc}(x, e) \implies \text{PointsTo}(x, e)$
2. 赋值(x, y) $\wedge$ 指向(x, e) $\implies$ 指向(y, e)

总而言之，Andersen 算法施加子集约束：如果变量y 可以通过函数调用或直接赋值取得变量 x的值，那么 x 的指向集合是 y的指向集合的子集。

Steensgaard 算法是 Andersen 算法的一种精度较低但通常速度更快的变体。它基于等式约束。因此，如果变量y 可以取得变量 x的值，那么 Andersen 算法会使 x 和 y的指向集合相等。Steensgaard 算法的一种直接实现会在程序的变量上维护一个并查集数据结构，并维护一种映射，为每个规范代表元分配一个堆分配的指向集合。该算法扫描程序源代码并执行以下操作：

1. 对于每条将分配表达式 e 赋给变量 x的语句，将 e 加入 x的规范代表元的指向集合。2. 如果变量 y 可以取得变量 x的值，则合并 x 和 y的等价类。合并后的等价类的指向集合是 x 和 y所属等价类的指向集合的并集。

Steensgaard 算法的精确度严格低于 Andersen 的算法，但通常所需的内存较少，因为每个变量等价类只需维护一个指向集合。要在 RHL 中编码 Steensgaard 算法，只需将上文 Andersen 分析中的规则 2 替换为以下规则

$$\text{Assign}(x, y) \implies x \equiv y$$

以强制执行相等约束。

4.2 类型推断

类型推断（或类型重建）是根据程序片段中的使用情况，为变量和表达式赋予类型的任务。Pierce（2002，22.3、22.4）描述的简单类型 λ 演算约束式类型推断算法，为每个项分别赋予一个初始时不受限制的类型变量。然后，该算法根据相应项的使用和定义收集关于类型变量的约束。这通过考察类型规则及其逆规则来实现。例如，根据应用类型规则

$$\frac{k : \sigma \to \tau \qquad s : \sigma}{k\, s : \tau} \tag{4}$$

我们从项 $t_2 = t_0\, t_1$ 推断出以下约束：

1. 如果 t_0 的类型为 $S \to T$，则 t_1 的类型为 S。2. 反之，如果 t_1 的类型为 S，则对于某个 T，t_0 的类型为 $S \to T$。3. 如果 t_0 的类型为 $S \to T$，则 t_2 的类型为 T。4. 反之，如果 t_2 的类型为 T，则对于某个 S，t_0 的类型为 $S \to T$。

约束 2 和 4 中隐含的存在量词会生成新的类型变量 T 和 S，每次使用该约束时都必须为其选择新变量。在接下来的合一步骤中，会检查生成的约束是否兼容；如果兼容，则创建满足所有约束的最一般类型变量替换。例如，对于变量 x 和 y 对应的片段 $x\, y$，算法会输出替换 $[T_0 \mapsto (T_1 \to T_2)\ T_1 \mapsto T_1,$ $T_2 \mapsto T_2]$，其中 T_0、T_1 和 T_2 是最初分配给 x、y 和 $x\, y$ 的类型变量。

该推理过程可按如下方式在 PHL 中实现。我们引入项的类型 Tm 和类型的类型 Ty。每个 n 元类型构造子或项构造子都对应一个 n 元 PHL 函数。例如，函数类型编码为二元函数符号 $\mathsf{Fun} : \mathsf{Ty} \times \mathsf{Ty} \to \mathsf{Ty}$，而应用编码为函数 $\mathsf{Tm} \times \mathsf{Tm} \to \mathsf{Tm}$。为确保类型构造子的单射性，我们为每个类型构造子的参数引入逆函数。对于函数类型，我们添加函数 $\mathsf{Dom} : \mathsf{Ty} \to \mathsf{Ty}$ 和 $\mathsf{Cod} : \mathsf{Ty} \to \mathsf{Ty}$，并加入公理，规定 Dom 和 Cod 确实是 Ty 上的逆函数：

$$\mathrm{Dom}(\kappa)\downarrow \implies \mathrm{Cod}(\kappa)\downarrow \qquad\qquad \mathrm{Cod}(\kappa)\downarrow \implies \mathrm{Dom}(\kappa)\downarrow$$

$$\sigma = \mathrm{Dom}(\kappa) \wedge \tau = \mathrm{Dom}(\kappa) \implies \kappa = \mathrm{Fun}(\sigma, \tau)$$

$$\kappa = \mathrm{Fun}(\sigma, \tau) \implies \mathrm{Dom}(\kappa) = \sigma \wedge \mathrm{Cod}(\kappa) = \tau$$

因此，Dom 和 Cod 定义在同一组类型上：这些类型均形如 $\mathsf{Fun}(\sigma, \tau)$，其中类型为 σ 和 τ。

为了检测类型构造子的联合单射性是否遭到违反，我们引入一个零元谓词 $\perp$ 和如下规则

$$\mathrm{Fun}(-,-) \equiv \mathrm{List}(-) \implies \perp()$$

针对每一对不同的类型构造子，例如函数类型和列表类型。我们总是在进行 PHL 求值之前将 $\perp$ 设为空，因此，求值后 $\perp$有元素，当且仅当推导出了联合单射性遭到违反。

我们将类型关系 $t : \tau$ 编码为函数 TmTy : Tm $\to$ Ty，而不是关系，因为每个项都有唯一的类型。公理 t : Tm $\Longrightarrow$TmTy(t) $\downarrow$ 确保每个项都有类型。在求值过程中，如有必要，这条非满射规则会引入一个新的标识符作为每个项的类型。

最后，我们将项构造子编码为 PHL 函数，并根据推理规则及其逆规则添加公理。例如，函数应用的类型规则 (4.2) 会生成一个 PHL 函数 App : Tm $\times$ Tm $\to$ Tm，以及以下约束它的 PHL 公理，对应于上文的约束 1–4:

1. $\mathrm{App}(t_0, t_1)\downarrow \wedge \mathrm{TmTy}(t_0) = \mathrm{Fun}(\sigma, -) \implies \mathrm{TmTy}(t_1) = \sigma$
2. $\mathrm{App}(t_0, t_1)\downarrow \wedge \mathrm{TmTy}(t_1) = \sigma \implies \mathrm{Dom}(\mathrm{TmTy}(t_0)) = \sigma$
3. $t_2 = \mathrm{App}(t_0, t_1) \wedge \mathrm{TmTy}(t_0) = \mathrm{Fun}(-, \tau) \implies \mathrm{TmTy}(t_2) = \tau$
4. $t_2 = \mathrm{App}(t_0, t_1) \wedge \mathrm{TmTy}(t_2) = \tau \implies \mathrm{Cod}(\mathrm{TmTy}(t_0)) = \tau$

请注意，公理 2 和 4 的结论断言某些类型的定义域和陪域存在；结合我们关于 Dom 和 Cod 的公理可知，这些类型是函数类型。如有必要，在 PHL 求值过程中，非满射公理会引入新的类型标识符；这些公理断言 Dom 函数和 Cod 函数在同一组类型上有定义。

有了用于建模当前类型系统的 PHL 理论，现在可以通过以下三个步骤实现完整的类型推断算法:

1. 根据程序片段填充关系结构：为程序片段中的每个项添加一个唯一的项标识符，并在表示项构造子函数图的关系中添加条目。

2. 根据上述公理对关系结构进行闭包。

3. 如果 $\perp$ 关系中包含一个元素，则输出错误。否则，对于每个类型标识符 T，至多存在一个类型构造子 μ，以及一个形如 ($T_1, \ldots, T_n, T$) 的条目，其中 T 是最后一个分量，该条目位于与 μ 对应的关系中。根据这些条目，递归地将每个类型标识符展开为最大语法树。对于输入程序片段中的每个项 t，输出表示 TmTy(t) 的最大语法树。

5 结论

正如我们在第4.2节所概述的，使用 Datalog 实现类型检查和推断并非新想法。例如，Matsakis 的博客文章讨论了如何使用 Datalog 实现 Rust 的特质系统。其中提出的一个问题是，Rust 的关联类型需要对类型的等式进行推理。Haskell 的类型族也会遇到类似问题。该博客文章提出的解决方案是将 Datalog 与规范化算法结合起来。RHL 内置的相等机制或许可以提供一种声明式替代方案，取代规范化

这种方案也适用于不存在强正规化重写系统的情况。

类型规则通常采用自然演绎的记法来表示（例如应用类型规则（4.2））。除语法差异外，这类规则的结构及其语义几乎与 PHL 完全对应。事实上，人们普遍认为，许多类型系统都可以编码为本质代数理论（Adamek 和 Rosicky，1994，第 3.D 章），其表达能力与满射 PHL 相同。从这一视角看，程序片段可以视为公理化该类型系统的 PHL 理论初始模型中的元素，即空关系结构上的自由模型。

这些概念上的联系以及第4.2节中的示例表明，PHL 和 RHL 求值有望在编程语言实现中发挥与解析器生成器相似的作用：解析器生成器根据文法生成解析器，而关系霍恩逻辑求值引擎则根据类型系统生成类型检查器。

参考文献

J. Adamek 和 J. Rosicky。《局部可呈现且可达范畴》。剑桥大学出版社，1994年3月。doi：10.1017/cbo9780511600579。URL：https://doi.org/10.1017/cbo9780511600579。

M. E. Bidlingmaier。《带相等的 Datalog 的代数语义》，2023年。网址：https://arxiv.org/abs/2302.03167。

J. Cocke 和 J. T. Schwartz。《编程语言及其编译器》。库朗数学科学研究所，纽约，1970年。初步笔记，第二修订版。P. J. Downey、R. Sethi 和 R. E. Tarjan。常见子表达式问题的变体。《ACM学报》，27(4):758–771，1980年10月。doi：10.1145/322217.322228。网址：https://doi.org/10.1145/322217.322228。X. Hu、J. Karp、D. Zhao、A. Zreika、X. Wu 和 B. Scholz。Souffl´e 语言中的选择构造。载于《编程语言与系统》，第163–181页。Springer International Publishing，2021年。doi：10.1007/978-3-030-89051-3 10。网址：https://doi.org/10.1007/978-3-030-89051-3_10。N. Matsakis。将 Rust 特征降低为逻辑。http://smallcultfollowing.com/babysteps/blog/2017/01/26/lowering-rust-traits-to-logic/.

访问日期：2023年1月30日。P. Nappa、D. Zhao、P. Subotic 和 B. Scholz。《Datalog 编译器中的快速并行等价关系》。载于《2019年第28届并行体系结构与编译技术国际会议（PACT）论文集》。电气电子工程师学会，2019年9月。doi：10. 1109/pact.2019.00015。URL https://doi.org/10.1109/pact.2019.00015。E. Palmgren 和 S. Vickers。《部分霍恩逻辑与笛卡尔范畴》。《纯粹与应用逻辑年鉴》，145(3)：314–353，2007年3月。doi：10.1016/j.apal. 2006.10.001。URL https://doi.org/10.1016/j.apal.2006.10.001。

B. C. Pierce。《类型与程序设计语言》。麻省理工学院出版社。麻省理工学院出版社，英国伦敦，2002年1月。Eqlog 作者。Eqlog。https://github.com/eqlog/eqlog。访问日期：2023年2月7日。M. Willsey、C. Nandi、Y. R. Wang、O. Flatt、Z. Tatlock 和 P. Panchekha。Egg：快速且可扩展的等式饱和。《ACM 程序设计语言汇刊》，5(POPL)，2021年1月。doi：10.1145/3434304。URL https://doi.org/10.1145/3434304。