arXiv:2303.17017v1 [cs.LO] 29 Mar 2023

有限代数结构中无量词可定义性的两个判定算法

Miguel Campercholi* Mauricio Tellechea† Pablo Ventura‡

科尔多瓦国立大学，阿根廷科尔多瓦

摘要

本文研究有限代数结构上由无量词一阶公式定义的可定义性问题。我们证明该问题是 coNP 完全，并提出两种判定算法，其基础是对可定义关系的语义刻画：可定义关系正是那些在子结构同构下保持不变的关系。第二种算法还可在肯定情形下给出一个公式。我们的方法还包括设计一种算法，用于计算有限代数结构中元组的同构类型。文中给出了这些算法的可靠性与完备性证明，并通过实证测试评估其性能。

1 引言

给定一种逻辑 $\mathcal{L}$， $\mathcal{L}$ 的模型论语义是一种函数，它为 $\mathcal{L}$ 中的每个公式φ 以及 $\mathcal{L}$ 的每个结构$\mathbf{A}$，指定一个由$\mathbf{A}$的定义域中的元组构成的集合。这个元组集合称为 φ 在$\mathbf{A}$ 中的扩张，我们通常将这些元组视为使 φ 在$\mathbf{A}$中为真的那些解释，不过，将它们视为由 φ所表达的性质挑选出来或命名的对象，也同样说得通。由此产生了各种基本的计算问题，其中最重要的无疑是 $\mathcal{L}$的可满足性 问题：给定 $\mathcal{L}$ 中的一个公式 φ，该问题询问是否存在模型$\mathbf{A}$，使得 φ 在$\mathbf{A}$ 中的扩张非空。 $\mathcal{L}$的模型检验 问题则是计算

*camper@famaf.unc.edu.元数
†mauriciotellechea@gmail.com
‡pablogventura@gmail.com

给定一个模型$\mathbf{A}$ 和一个公式 φ，计算 φ 在$\mathbf{A}$中的扩张；这一问题在计算逻辑中也发挥了重要作用。第三个基本问题，也是本文关注的问题，是 $\mathcal{L}$的可定义性 问题：给定一个有限模型$\mathbf{A}$ 以及$\mathbf{A}$ 中的一个元组集合 R，该问题询问是否存在一个公式 φ，使得 φ 在$\mathbf{A}$ 中的扩张与R一致。鉴于其根本性质及其应用，例如在数据库理论 [10] 和指代表达式生成 [9, 4, 3]中的应用，这一问题已在多种逻辑中得到研究。多种逻辑的可定义性问题的计算复杂度也已得到研究。文献 [8] 以“逆可满足性”之名研究了经典命题逻辑的可定义性问题，并证明该问题在一般情形下是 coNP 完全的。该文还刻画了问题在何种特殊句法情形下属于 P。文献 [5] 证明，一阶逻辑的可定义性问题（在图灵归约意义下）是 GI 完全的。一阶逻辑的原始正片段的可定义性问题是 coNEXPTIME 完全的 [11]，；而基本模态逻辑 [4] 及其某些片段 [3] 中的该问题属于 P。文献 [1], 指出，对于具有纯关系词汇的一阶逻辑无量词片段，可定义性问题是 coNP 完全的；同一文献还证明，该问题的一个参数化版本对于复杂度类 W[1]是完备的。

为说明可定义性的概念，我们来看一个例子。设$\mathbf{D} = \langle\{\bot, \top, u, u'\}, \wedge, \vee\rangle$是下图所示的格，并令 $R = \{(a, b) \in D^2 : a \leq b\}$。

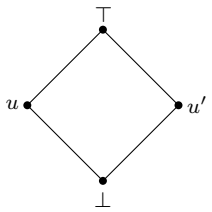


图 1：格$\mathbf{D}$

我们希望知道 R 在 D 中是否是无量词可定义的。也就是说，是否存在一个 D 的词汇中的无量词公式 $\varphi(x, y)$，使得 $R = \{(a, b) \in D^2 : \mathrm{D} \vDash \varphi(a, b)\}$。不难看出，公式 $\varphi(x, y) := (x \vee y) = y$ 是可行的。接下来，考虑关系 $R' = \{(\bot, u), (\bot, u'), (\bot, \top)\}$；它在 D 中是否是无量词可定义的？答案是否定的，原因如下。注意，映射 $\gamma\colon \{\bot, \top\} \to \{u, \top\}$ 是 D 的子格之间的一个同构，而无量词公式在此类映射下保持不变。也就是说，如果 $\varphi(x, y)$ 是无量词公式且 $\mathrm{D} \vDash \varphi(a, b)$，那么 $\mathrm{D} \vDash \varphi(\gamma a, \gamma b)$。因此，对于任意结构，每个由无量词公式可定义的关系

都在子结构之间的同构下保持不变（我们将这些映射称为子同构）。回到我们的例子，可以看到 $(\bot, \top) \in R'$，但 $(\gamma\bot, \gamma\top) \notin R'$，因此 R' 不被 γ保持，从而它在 $\mathbf{D}$ 中不是无量词可定义的。有趣的是，对于有限结构，这一判据的逆命题也成立；即，如果$\mathbf{A}$ 是一个有限结构，那么关系 $R \subseteq A^k$ 在$\mathbf{A}$ 中是无量词可定义的，当且仅当 R 在$\mathbf{A}$ 的所有子同构下都保持不变（见下文的定理 1）。

[2] 中应用了这一无量词可定义关系的刻画，以获得一种算法，用于判定纯关系词汇下一阶逻辑无量词片段的可定义性问题（在 [2]中称为开放可定义性 问题）。本短文延续并补充了 [2], 中的工作，基于同一语义刻画，为代数词汇（即不含关系符号）开发了两种算法。值得注意的是，在允许量化的一阶逻辑片段中，关系可定义性问题与代数可定义性问题会合为同一问题，因为只需引入量词，就可以将每个公式有效地转换为与之等价且不含嵌套的公式。然而，在无量词情形下，这两个问题并不相同，需要采用不同的算法策略。主要区别在于：在代数情形下，计算一个结构的子结构之间的同构，需要计算该结构的子宇宙。本文的第一种方法，即用于判定可定义性的合并策略，基于这样一种观察：在结构$\mathbf{A}$ 中，计算由元组 $\bar{a}$ （中的元素）生成的子宇宙，与计算 $\bar{a}$ 在一个$\mathbf{A}$ 中的同构类型相差不远（见第 4 节）。第二种方法，即分裂策略，同样基于这一观察；但这次我们会考虑目标，以便剪除部分搜索空间，并在肯定情形下生成一个定义公式。

除了介绍上述算法之外，我们还证明了代数词汇的无量词可定义性问题的计算复杂度对于 coNP 而言是完备的。

本文组织如下。下一节将固定记号并给出基本定义。第 3 节专门阐述我们的复杂度结果。第 4 节介绍用于计算代数中元组的同构类型的算法。第 5 节介绍用于判定代数结构的无量词可定义性的合并策略算法；第 6 节介绍分裂策略算法，该算法还能够计算公式。第 7 节进行了一些实证测试，用于比较我们的两种算法策略的性能。这些测试还表明，在具有大量对称性的模型上，我们的算法表现更好。最后，第 8 节总结所得结果，并给出由本文研究进展引出的进一步研究方向。

2 预备知识

本节给出一些基本定义并固定记号。我们假设读者具备一阶逻辑的基本知识。详细介绍参见例如 [7]。给定一阶词汇τ 和 $k \in \omega$，令 $\mathcal{T}_k$ 表示深度为 k 、变量来自 VAR 的所有 τ- 项的集合：$=\{x_0, x_2, \ldots, x_n, \ldots\}$。当变量属于 $\{v_0, \ldots, v_k\}$ 时，我们还使用 $\mathcal{T}_k(v_0, \ldots, v_k)$。我们记 $\mathcal{T} := \bigcup_{k\in\omega} \mathcal{T}_k$ 为 VAR 上所有 τ-项的集合。对于函数或关系符号 s，我们写作 ar(s)，表示 s的元数。在本文中，我们关注代数词汇，即不含关系符号的词汇。在此语境下，原子公式具有 $t = t'$ 的形式，其中t 和 t' 是项。无量词公式是原子公式的布尔组合。对于变量全部包含于 $\{v_0, \ldots, v_k\}$的公式 φ，我们记作 $\varphi(v_0, \ldots, v_k)$；对于项 $t(v_0, \ldots, v_k)$，也使用类似记号。代数是代数词汇的一个结构；代数用粗体字母表示（例如 A 、B、C.....），其宇宙则用相应的不加粗字母表示。给定代数 A、公式 $\varphi(v_0, \ldots, v_k)$以及元素序列 $\bar{a} = (a_0, \ldots, a_{k-1}) \in A^k$，如果 φ 在 A 中、在将 v_i 映射到 a_i的赋值下为真，则记作 $\mathbf{A} \models \varphi(\bar{a})$。我们使用 $[\varphi(\bar{v})]^{\mathbf{A}}$ （在不会引起混淆时简称为 $[\varphi]^{\mathbf{A}}$ ）表示 φ 在 A 中的扩张。若子集 $R \subseteq A^k$ 满足：存在 A 的词汇中的无量词一阶公式$\varphi(x_0, \ldots, x_{k-1})$，使得

$$R = [\varphi]^{\mathbf{A}} = \{\bar{a} \in A^k : \boldsymbol{A} \models \varphi(\bar{a})\}.$$

本文研究以下计算判定问题：

QfDefAlg输入：一个有限代数**A** 和一个目标关系 $R \subseteq A^k$。问题： R 在**A**中是否量词自由可定义?

设 τ 为一个代数词汇表，A 为一个代数。若子集 $S \subseteq A$ 非空且在 A 的基本运算下封闭，则它是 A 的一个子宇宙。对于元组 $\bar{a} = (a_0, \ldots, a_{k-1}) \in A^k$，我们用 $\mathrm{Sg}^{\mathbf{A}}(\bar{a})$（或在上下文明确时简写为 Sg($\bar{a}$)）表示由 $\{a_0, \ldots, a_{k-1}\}$生成的 A 的子宇宙。设 γ：dom $\gamma \subseteq A \to A$为一个函数。若对于所有 $(a_0, \ldots, a_{k-1}) \in R \cap (\mathrm{dom}\ \gamma)^k$，都有 $(\gamma a_0, \ldots, \gamma a_{k-1}) \in R$，则称 γ保持 $R \subseteq A^k$。若 γ为单射、dom γ是 A 的子宇宙，且对每个 $f \in \tau$， γ都保持 $f^{\mathbf{A}}$的图，则称函数 γ为 A 的子同构。（注意，A 的子同构恰好是 A 的两个子结构之间的同构。）下面这个用子同构刻画量词自由可定义性的结果，在本文中起着核心作用。

定理 1。 *[6,* 定理 *3*。*1]* 设 **A** 是任意一阶签名的有限模型，且 $R \subseteq A^k$。以下命题等价：

- R 可在 $\mathbf{A}$ 中由无量词一阶公式定义。
- R 在 $\mathbf{A}$ 的 k-生成子结构之间的所有同构下保持不变。

3 QfDefAlg 是 coNP 完全

下文中，图 是语言的一个模型 $\mathbf{G}$ ，该语言仅含一个二元关系 E，且 $E^{\mathbf{G}}$ 是对称且反自反的。在 [1]中证明了以下问题是 coNP 完备的。

QfDef[Graphs]输入：一个有限图 $\mathbf{G}$ 和一个目标关系 $R \subseteq G^k$。问题： R 在 $\mathbf{G}$ 中是否量词自由可定义?

图 $\mathbf{G} = (G, E)$ 的一个子同构 是一个单射 γ：定义域 $\gamma \subseteq G \to G$，满足 $(a, b) \in E$ 当且仅当 $(\gamma a, \gamma b) \in E$。

定理 2。 QfDefAlg 是coNP完全的。

证明。 给定有限图 $\mathbf{G} = (G, E)$，令 ˆ0 和 ˆ1 表示不属于 G 的前两个正整数。我们定义代数 $\mathbf{G}_* := (G \cup \{\hat{0}, \hat{1}\}, f)$，其中 f 是二元运算

$$f(a,b) = \begin{cases} \hat{1} & \text{if } (a,b) \in E \text{ or } a = b \in \{\hat{0}, \hat{1}\} \\ \hat{0} & \text{otherwise.} \end{cases}$$

不难验证:

1. 若 γ 是 $\mathbf{G}$ 的子同构，则将 γ 扩张为 γ_*，并令 $\gamma_*(\hat{0}) = \hat{0}$ 且 $\gamma_*(\hat{1}) = \hat{1}$，可得 $\mathbf{G}_*$ 的子同构。
2. 若 δ 是 $\mathbf{G}_*$ 的子同构，则 $\delta[G] \subseteq G$ 和 $\delta|_G$ 是 $\mathbf{G}$ 的子同构。

接下来我们证明， $\langle \mathbf{G}\ R\rangle \longmapsto \langle \mathbf{G}_*, R\rangle$ 是从 QfDef[图]到 QfDefAlg 的一个多项式时间（Karp）归约。显然，可以在多项式时间内根据 $\mathbf{G}$ 计算出 $\mathbf{G}_*$，因此只需证明， R 在 $\mathbf{G}$ 中量词自由可定义，当且仅当 R 在 $\mathbf{G}_*$中量词自由可定义。固定一个有限图 $\mathbf{G}$ 和 $R \subseteq G^k$。假设 R 在 $\mathbf{G}$ 中不可量词自由定义。于是，根据定理 1，存在 γ，它是 $\mathbf{G}$ 的一个子同构，但不保持 R。现在，由（1）可知， γ_* 是 $\mathbf{G}_*$的一个子同构；由于它不保持 R，因此 R 在 $\mathbf{G}_*$中不可量词自由定义。对于反方向，假设 R 在 $\mathbf{G}_*$中不可量词自由定义。再次由定理 1 可得，存在 $\mathbf{G}_*$ 的一个子同构 δ，它不保持 R。由（2）可知， $\delta|_G$ 是 $\mathbf{G}$ 的一个不保持 R的子同构；因此 R 在 $\mathbf{G}$ 中不可量词自由定义。

证明 QfDefAlg 属于 coNP 是定理 1 的一个直接应用。事实上，QfDefAlg 的每个否定实例 $\langle \mathbf{A}\ R\rangle$ 都有一个双射 γ 作为见证，该双射定义在满足相关条件的子集之间，而这些条件均可在相对于 $\langle \mathbf{A}\ R\rangle$大小的多项式时间内轻松检查。

□

由定理 2 的证明可知，将 QfDefAlg 限制在具有单个二元交换运算的结构上时，对于 coNP 已经是完备的。

4 计算元组的同构类型

本节介绍一种计算有限代数结构**A**中元组 $\bar{a}$ 的同构类型的算法，并证明该算法的正确性。我们先给出一些必要的定义和预备结果。

设 **A** 是一个有限代数，对于每个 $k \in \omega$，我们定义 $\approx_k$ 上的等价关系 A^n：当且仅当对于所有项 $t, s \in \mathcal{T}_k(x_0, \dots, x_{n-1})$，都有

$$t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a}) \Longleftrightarrow t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b}).$$

如果两个元组 $\bar{a}, b \in A^n$ 满足 $\bar{a} \approx_k b$ 对所有 $k \in \omega$均成立，则称它们是同构的（记作 $\bar{a} \approx b$）。注意，$\approx$ 是 A^n上的等价关系。对于 $\bar{a} \in A^n$，令

$$\mathrm{K}_{\bar{a}} := \min\{k \in \omega : \text{ for all } t \in \mathcal{T}_k(\bar{x}) \text{ there is } \hat{t} \in \mathcal{T}_{k-1}(\bar{x}) \text{ such that } t^{\mathbf{A}}(\bar{a}) = \hat{t}^{\mathbf{A}}(\bar{a})\}.$$

请注意，由于 **A** 是有限的，该最小值始终存在。

引理 1。设 **A** 是一个有限代数，且 $\bar{a}, b \in A^n$。

1. 如果 $\bar{a} \approx_{\mathrm{K}_{\bar{a}}} \bar{b}$，那么对于任意项 $t(\bar{x})$，存在一个项 $\hat{}t(\bar{x}) \in \mathcal{T}_l$，满足 $l < \mathrm{K}_{\bar{a}}$，使得 $t^{\mathbf{A}}(\bar{a}) = \hat{}t^{\mathbf{A}}(\bar{a})$ 且 $t^{\mathbf{A}}(\bar{b}) =$ <style id='43'>$\hat{}t^{\mathbf{A}}(\bar{b})$。

2. 如果 $\bar{a} \approx_{\mathrm{K}_{\bar{a}}} b$，那么 $\bar{a} \approx b$。

证明。（1）首先注意，由定义可知 $\mathrm{K}_{\bar{a}} = \mathrm{K}_{\bar{b}}$；我们将 K 记作 $\mathrm{K}_{\bar{a}}$。固定一个项 $t(\bar{x})$。若 $t(\bar{x}) \in \mathcal{T}_{K-1}$，则可取 $t = \hat{}t$，因此不妨设 $t \in \mathcal{T}_{K+j}$ 且 $j \geq 0$。我们对 j 使用归纳法，证明 $\hat{}t$ 存在。设 $j = 0$；根据 $\mathrm{K}_{\bar{a}}$ 的定义，存在 $\hat{}t \in \mathcal{T}_{K-1}$ 使得 $t^{\mathbf{A}}(\bar{a}) = \hat{}t^{\mathbf{A}}(\bar{a})$，又由于 $\bar{a} \approx_{\mathrm{K}} \bar{b}$，可得 $t^{\mathbf{A}}(\bar{b}) = \hat{}t^{\mathbf{A}}(\bar{b})$。接下来假设 $j > 0$，并设 $t = f(t_0, \dots, t_{r-1})$，其中每个 $t_i \in \mathcal{T}_{K+j-1}$。根据归纳假设，存在项 $\hat{}t_0, \dots, \hat{}t_{r-1} \in \mathcal{T}_u$，满足 $u < K$，且每个项都满足 $t_i^{\mathbf{A}}(\bar{a}) = \hat{}t_i{}^{\mathbf{A}}(\bar{a})$ 和 $t_i^{\mathbf{A}}(\bar{b}) = \hat{}t_i{}^{\mathbf{A}}(\bar{b})$。这意味着 $\tilde{t} := f(\hat{}t_0, \dots, \hat{}t_{r-1}) \in \mathcal{T}_{u+1}$ 满足 $\tilde{t}^{\mathbf{A}}(\bar{a}) = t^{\mathbf{A}}(\bar{a})$ 以及 $\tilde{t}^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b})$。由于 $u + 1 \leq K$，

根据情形 $j = 0$，我们有一个项 $\hat{}t \in \mathcal{T}_l$，满足 $l < K$，且 $\hat{}t^{\mathbf{A}}(\bar{a}) = \widetilde{t}^{\mathbf{A}}(\bar{a}) = t^{\mathbf{A}}(\bar{a})$，以及 $\hat{}t^{\mathbf{A}}(\bar{b}) = \widetilde{t}^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b})$。

(2) 令 $t,\ s \in \mathcal{T}$ 使得 $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$。取 (1) 给出的项 $\hat{}t,\ \hat{}s$。由于 $\hat{}t^{\mathbf{A}}(\bar{a}) = \hat{}s^{\mathbf{A}}(\bar{a})$，根据假设，我们有 $\hat{}t^{\mathbf{A}}(\bar{b}) = \hat{}s^{\mathbf{A}}(\bar{b})$，因此 $t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b})$。

□

接下来，我们以伪代码的形式介绍计算函数 isoType 的算法（见下文算法 1）。该函数以有限代数 $\mathbf{A}$ 和来自 A 的元组 $\bar{a}$ 作为输入，并返回 $\bar{a}$ 的同构类型表示，以及按典范序排列的由 $\bar{a}$ 生成的$\mathbf{A}$ 的子宇宙。同构类型通过按指定顺序遍历各项并在 $\bar{a}$ 上对它们求值来获得。正如我们将看到的（推论 1），只需记录哪些项返回相同的值，就足以刻画 $\bar{a}$ 的同构类型。

下面说明算法 1 的工作原理。固定一个输入代数$\mathbf{A}$，其词汇为 τ（假设 τ 中的函数符号按特定顺序排列）。给定一个元组 $\bar{a} \in A^n$，在算法开始时，变量 `V` 被设为 $[a_0, \ldots, a_{n-1}]$。随后开始执行 while 循环，添加到 `V` 中的元素，是通过对 `V` 中较早出现的值应用$\mathbf{A}$ 的基本运算得到的。因此，`V` 中的所有元素都属于 Sg($\bar{a}$)。变量 `P` 记录这些元素在计算过程中重复出现的次数，而变量 `H` 则存储 Sg($\bar{a}$) 中每个元素首次出现在 `V` 中的索引。在每次执行 while 循环结束时，变量 `N` 被赋值为该轮添加到 `H` 中的索引。由于 A 是有限的，最终不会再产生新元素，`N` 也就被赋值为空列表。这保证了算法会终止，并且算法停止时，Sg($\bar{a}$) 中的所有元素都已列在 `V` 中。注意，输入 $\bar{a}$ 时，while 循环恰好执行 $\mathbf{K}_{\bar{a}}$ 次。我们指出，算法 1 中第 3 行和第 16–17 行的目的，是初始化并更新变量 `V′`；实际上，计算函数 isoType 并不需要这些代码，但将其保留有助于读者理解我们的证明。

为了在每一轮计算 `V` 中的元素，对于某个基本运算的每个元数，我们构造一个列表 `T`，其中包含元素索引的元组，且该元数的基本运算将应用于这些元组。我们要求这些元组中至少有一个元素此前未出现过，以避免不必要的计算。

无结尾 1:

```
1  函数 IsoType(A,  ā)
                                          ▷ A 是一个代数，且 ā 是来自 A的一个元组
2      V = [a_0, ..., a_{n-1}]
3      V′ = [“x_0” ..., “x_{n-1}”]   ▷ 这是一个变量列表（项以字符串表示）
4      P = [B_1, ..., B_k]  {0, ..., n − 1} 的划分，其中 i, j属于同一个块当且仅当
   a_i = a_j
5      H = sorted([最小值(B_j) : B_j ∈ P])                          ▷ 按升序排序
6      N = H
7      arities =[r_1, ..., r_k]        ▷ 其中，  r_1 < ... < r_k 是 A 中所有运算的元数
8      当 N ≠ ∅ 时执行
9          H_old = copy(H)
10         对 r ∈ arities 执行
11             T = sorted([ l̄ ∈(H_old)^r :  l_j ∈ N 对于某些 j])        ▷ 按字典序排序
12             对于 f ∈ op(r) 执行   ▷ op(r)给定顺序中元数为 r 的操作符号列表
   by τ
13                 for l̄ ∈ T 执行
14                     value = f^A(V[l_0], ..., V[l_{r-1}])
15                     将值追加到 V
16                     term = f ++ “(” ++ V′[l_0] ++ “,  ” ++ ... ++ “,  ” ++ V′[l_{r-1}] ++ “)”
17                     将项追加到 V′
                                                          ▷ 划分 P 的更新
18                     如果存在  j ∈ H 使得  value = V[j] 那么 ▷ 值已在 V 中
19                         将  |V| − 1 添加到包含  j 的 P 块中
20                     else                                          ▷ 值是新的
21                         将块  {|V| − 1} 添加到 P
22                         将  |V| − 1 添加到 H
23         N = [l :  l ∈ H 和  l ∉ H_old]                          ▷对 N 的更新
24     U = [V[j] : j ∈ H]        ▷ U 按元素在 V 中出现的顺序列出 Sg(ā) 的元素
25     返回 P   , U
```

为了说明该算法的运行方式，我们来演示一个例子。设 $\mathbf{D} = \langle\{\top, \bot, u, u'\}, \wedge, \vee\rangle$ 为引言中的格（见图 1）。假设基本运算按以下顺序排列： $[\wedge, \vee]$。对于输入 $\mathbf{D}$， $\bar{a} = (u, u', \bot)$，变量初始化如下：

$$\mathtt{V} = [u, u', \bot] \qquad \mathtt{P} = [[0], [1], [2]] \qquad \mathtt{H} = [0, 1, 2] \qquad \mathtt{N} = [0, 1, 2].$$

第一次执行 while 循环后，状态为：

$$\mathtt{V} = [u, u', \bot, \underbrace{u}_{u\wedge u}, \underbrace{\bot}_{u\wedge u'}, \underbrace{\bot}_{u\wedge\bot}, \underbrace{\bot}_{u'\wedge u}, \underbrace{u'}_{u'\wedge u'}, \underbrace{\bot}_{u'\wedge\bot}, \underbrace{\bot}_{\bot\wedge u}, \underbrace{\bot}_{\bot\wedge u'}, \underbrace{\bot}_{\bot\wedge\bot}, \underbrace{u}_{u\vee u}, \underbrace{\top}_{u\vee u'}, \underbrace{u}_{u\vee\bot}, \underbrace{\top}_{u'\vee u}, \underbrace{u'}_{u'\vee u'}, \underbrace{u'}_{u'\vee\bot}, \underbrace{u}_{\bot\vee u}, \underbrace{u'}_{\bot\vee u'}, \underbrace{\bot}_{\bot\vee\bot}]$$

$\mathtt{P} = [[0, 3, 12, 14, 18], [1, 7, 16, 17, 19], [2, 4, 5, 6, 8, 9, 10, 11, 20], [13, 15]]$

$\mathtt{H} = [0, 1, 2, 13]$，$\mathtt{N} = [13]$。

现在，元素 $\top$ 首次出现在位置 13，而此前它不在 $\mathtt{V}$ 中。因此，下一轮将对来自 $[u, u', \top, \bot]$ 且至少包含一次 $\top$ 的所有元素对应用基本运算。于是，下一轮的结果为：

$$\mathtt{V} = \mathtt{V} + [\underbrace{u}_{u\wedge\top}, \underbrace{u'}_{u'\wedge\top}, \underbrace{\bot}_{\bot\wedge\top}, \underbrace{u}_{\top\wedge u}, \underbrace{u'}_{\top\wedge u'}, \underbrace{\bot}_{\top\wedge\bot}, \underbrace{\top}_{\top\wedge\top}, \underbrace{\top}_{u\vee\top}, \underbrace{\top}_{u'\vee\top}, \underbrace{\top}_{\bot\vee\top}, \underbrace{\top}_{\top\vee u}, \underbrace{\top}_{\top\vee u'}, \underbrace{\top}_{\top\vee\bot}, \underbrace{\top}_{\top\vee\top}]$$

$\mathtt{P} = [[0, 3, 12, 14, 18, 21, 24], [1, 7, 16, 17, 19, 22, 25], [2, 4, 5, 6, 8, 9, 10, 11, 20, 23, 26,],$
$[13, 15, 27, 28, 29, 30, 31, 32, 33, 34]]$，$\mathtt{H} = [0, 1, 2, 13]$，$\mathtt{N} = [\,]$。

变量 $\mathtt{N}$ 现在为空，因为没有生成新的元素，因此算法 1 停止，并返回划分 $\mathtt{P}$ 和 $\mathtt{U} = [u, u', \bot, \top] = \mathrm{Sg}(\bar{a})$。对元组 $b = (u',\ u,\ \bot)$ 运行 $^-$算法，我们得到：

$$\mathtt{V} = [u', u, \bot, u', \bot, \bot, \bot, u, \bot, \bot, \bot, \bot, u', \top, u', \top, u, u, u', u, \bot, u', u, \bot, u', u, \bot, \top, \top, \top, \top, \top, \top, \top, \top\ ]$$

$\mathtt{H} = [0, 1, 2, 13]$

$\mathtt{P} = [[0, 3, 12, 14, 18, 21, 24], [1, 7, 16, 17, 19, 22, 25], [2, 4, 5, 6, 8, 9, 10, 11, 20, 23, 26,],$
$[13, 15, 27, 28, 29, 30, 31, 32, 33, 34]]$，$\mathtt{U} = [u', u, \bot, \top]$。

$^-$由于 $\bar{a}$ 和 b的最终划分相同，根据下面的推论 1，γ 从$[u, u', \bot, \top]$到 $[u', u, \bot, \top]$的映射是满足 $\gamma\colon\ \bar{a} \mapsto b$ 的子同构。

4.1 可靠性与完备性

对于 $\{\mathtt{V}, \mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$ and $k \in \{0, \ldots, \mathtt{K}_{\bar{a}}\}$ 中的变量名 $\mathtt{X}\,\mathtt{a}$，设

- $\mathtt{X}_{\bar{a},\,k}$ 为算法 1 在输入 $\mathbf{A}$ $\bar{a}$时，执行 k 次 while 循环后变量 $\mathtt{X}$ 的值。

此外，给定 $\mathbf{A}$ 的函数符号的元数 r，令

- $\mathtt{T}^{r}_{\bar{a},\,k}$ 表示在输入为 $\mathbf{A}$ $\bar{a}$ 的算法 1 的一次运行中，在 for 循环（第 10 行）中确定元数 r 后，while 循环的第 k 次执行期间，赋给变量 $\mathtt{T}$ 的值。

我们将 $\mathtt{X}_{\bar{a}}$ 记作 $\mathtt{X}_{\bar{a},\,\mathrm{K}_{\bar{a}}}$，并且对于 $k > \mathbf{K}_{\bar{a}}$，定义 $\mathtt{X}_{\bar{a},\,k}$ 为 $\mathtt{X}_{\bar{a}}$。当不会引起混淆时，省略下标 $\bar{a}$。

给定元组 $\bar{a} \in A^n$ 和 $k \in \omega$，令 $\rho_{\bar{a},\,k}$ 表示定义在 $\mathcal{T}_k(x_0, \ldots, x_{n-1})$ 上的等价关系，其中 $t\ \rho_{\bar{a},\,k}\ s \iff t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$。此外，$Eq(Y)$ 表示集合 Y 上所有等价关系的集合。下面的引理给出了一系列用于证明算法 1 的可靠性所必需的技术性结果，该证明将在下文的推论 1 中完成。

引理 2。对于任意 $\bar{a} \in A^n$ 和 $k \in \omega$，我们都有：

1. $\mathtt{V}'_k$ 中的每个成员都是变量 x_0、…、x_{n-1} 的一个项，其次数至多为 k。*2.* 不变式 $\mathtt{V}_k[i] = \mathtt{V}'_k[i]^{\mathbf{A}}(\bar{a})$ 在第 *13* 行的 *for* 循环中始终成立。*3.* 如果 $\bar{a} \approx_k b$，那么对于 $\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$中的任意变量名 $\mathtt{X}$，$\mathtt{X}_{\bar{a},k} = \mathtt{X}_{\bar{b},k}$。

4. 对于所有 $1 \leq k \leq \mathrm{K}_{\bar{a}}$，存在一个函数

$$F_k : \mathcal{T}_k(x_0, \ldots, x_{n-1}) \times Eq(\mathcal{T}_{k-1}(x_0, \ldots, x_{n-1})) \to \omega,$$

使得对于每个 $t \in \mathcal{T}_k(x_0, \ldots, x_{n-1})$，索引 $i = F_k(t, \rho_{\bar{a},k-1})$ 满足 $i \leq |\mathtt{V}_k| - 1$和 $t^{\mathbf{A}}(\bar{a}) = \mathtt{V}_k[i]$。

5. 对所有 $k \leq \mathrm{K}_{\bar{a}}$，我们有 $\mathtt{P}_k$ 决定了对于所有 $j < k$ 和 $\mathtt{X}\,\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$中的任意变量名的 $\mathtt{X}_j$。

6. 如果 $\mathtt{P}_{\bar{a}} = \mathtt{P}_{\bar{b}}$，那么 $\bar{a} \approx b$。

*Pr*证明。(1) 可通过在 k 中归纳法轻松证明。(2) 可由第 14 行到第 17 行显然得出。

(3) 我们对 k 进行归纳法。情形 $k = 0$ 很直接。现在假设 $k + 1 \leq \min(\mathrm{K}_{\bar{a}}, \mathrm{K}_{\bar{b}})$，并假设 $\bar{a} \approx_{k+1} b$。由于 $\bar{a} \approx_k b$，根据归纳假设，我们有

- $\mathtt{X}_{\bar{a},\,k} = \mathtt{X}_{\bar{b},\,k}$ 对 $\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$ 中的任意变量名 $\mathtt{X}$。

注意，在 while 循环第 $k+1$ 次运行开始时执行的第一行，会将 $\mathtt{H_old}$ 的值设为 $\mathtt{H}_k$ 的值。现在观察到，对于每个元数 r，赋给 $\mathtt{T}$ 的值只取决于 $\mathtt{H_old}$ 和 $\mathtt{N}_k$。因此，对于所有元数 r，$\mathtt{T}^r_{\bar{a}、\,k+1} = \mathtt{T}^r_{\bar{b}、\,k+1}$。再结合从第 12 行开始的 for 循环中的函数符号总是以相同顺序出现这一事实，可知 $\mathtt{V}'_{\bar{a}、\,k+1} = \mathtt{V}'_{\bar{b}、\,k+1}$。接下来证明 $\mathtt{P}_{\bar{a}、\,k+1} = \mathtt{P}_{\bar{b}、\,k+1}$；假设索引 i 和 j 位于 $\mathtt{P}_{\bar{a}、\,k+1}$ 的同一个块中。也就是说，$\mathtt{V}_{k+1}(\bar{a})[i] = \mathtt{V}_{k+1}(\bar{a})[j]$，因此由 (2) 可得 $\mathtt{V}'_{\bar{a}、\,k+1}[i]^{\mathbf{A}}(\bar{a}) = \mathtt{V}'_{\bar{a}、\,k+1}[j]^{\mathbf{A}}(\bar{a})$。现在，(1) 表明 $\mathtt{V}'_{k+1}(\bar{a})[i]$ 和 $\mathtt{V}'_{k+1}(\bar{a})[j]$ 是次数至多为 $k+1$ 的项，且它们在 $\bar{a}$ 上相同。因此，由于 $\bar{a} \approx_{k+1} b$，它们在 b 上也相同。再次应用 (2) 可得，i 和 j 位于 $\mathtt{P}_{\bar{b}、\,k+1}$ 的同一个块中。由 $\mathtt{P}_{\bar{a}、\,k+1} = \mathtt{P}_{\bar{b}、\,k+1}$ 这一事实，很容易看出 $\mathtt{H}_{\bar{a}、\,k+1} = \mathtt{H}_{\bar{b}、\,k+1}$ 以及 $\mathtt{N}_{\bar{a}、\,k+1} = \mathtt{N}_{\bar{b}、\,k+1}$。特别地，对于任意 $l \leq \min(\mathsf{K}_{\bar{a}}, \mathsf{K}_{\bar{b}})$，都有 $\mathtt{N}_{\bar{a}、\,l} = \emptyset$ 当且仅当 $\mathtt{N}_{\bar{b}、\,l} = \emptyset$。因此 $\mathsf{K}_{\bar{a}} = \mathsf{K}_{\bar{b}}$，由此立即可知，对于 $k+1 > \min(\mathsf{K}_{\bar{a}}, \mathsf{K}_{\bar{b}})$，(3) 成立。

(4) 取 $t = f(t_0, \ldots\ldots, t_{r-1}) \in \mathcal{T}_{k+1}$ 并假设 $k+1 \leq \mathsf{K}_{\bar{a}}$。对于 $k=0$，定义 $F_0(x_j) = j$。如果 $k \geq 1$，根据归纳假设，我们有函数 F_k，因此可以从 t 和 ρ_k 函数地得到索引 $i_0 := F_k(t_0, \rho_{k-1}) \ldots\ldots, i_{r-1} := F_k(t_{r-1}, \rho_{k-1})$（显然，$t_0, \ldots\ldots, t_{r-1}$ 可以作为 t 和 $\rho_{k-1} = \rho_k \cap \mathcal{T}^2_{k-1}$ 的函数得到）。如果 $k=0$，只需取 $i_0 := F_0(t_0) \ldots\ldots, i_{r-1} := F_0(t_{r-1})$。接下来，定义

$$\begin{aligned} \tilde{t}_0 &:= \mathtt{V}'_k[i_0] \\ &\vdots \\ \tilde{t}_{r-1} &:= \mathtt{V}'_k[i_{r-1}], \end{aligned}$$

再次注意，由于 $\mathtt{V}'_k$ 可以从 ρ_k 函数地得到，因此这些 $\tilde{t}_l$ 也可以函数地得到。此外，根据我们的归纳假设，我们有 $t^{\mathbf{A}}(\bar{a}) = f(\tilde{t}_0, \ldots\ldots, \tilde{t}_{r-1})^{\mathbf{A}}(\bar{a})$。

首先假设存在 $l \in \{0, \ldots, r-1\}$ 使得 $i_l \in \mathtt{H}_k \setminus \mathtt{H}_{k-1}$。于是 $i_l \in \mathtt{N}_k$，因此 $\langle i_0, \ldots, i_{r-1} \rangle \in \mathtt{T}^r_{k+1}$，并且我们有项 $f(\tilde{t}_0, \ldots, \tilde{t}_{r-1})$ 在 while 循环的第 $(k+1)$ 次遍历期间被追加到 $\mathtt{V}'$ 中，设其索引为 j。注意，j 只取决于 $\mathbf{A}$ 的签名以及 u 的元数小于或等于 r 时的数值 $\left|\mathtt{T}^u_{k+1}\right|$。但是，$\mathtt{T}^u_{k+1}$ 只取决于 $\mathtt{H}_k$ 和 $\mathtt{N}_k$，而后两者又由 ρ_k 决定。因此，j 可以作为 t 和 ρ_k 的函数得到；在存在 $l \in \{0, \ldots, r-1\}$ 使得 $i_l \in \mathtt{H}_k \setminus \mathtt{H}_{k-1}$ 的情形下，我们可以定义 $F_{k+1}(t, \rho_k) := j$。另一方面，假设 $i_0, \ldots, i_{r-1} \in \mathtt{H}_{k-1}$。那么，根据 (1)，我们有 $\tilde{t}_0, \ldots, \tilde{t}_{r-1} \in \mathcal{T}_{k-1}$，从而 $f(\tilde{t}_0, \ldots, \tilde{t}_{r-1}) \in \mathcal{T}_k$。因此，在这种情形下，我们可以定义 $F_{k+1}(t, \rho_k) := F_k(f(\tilde{t}_0, \ldots, \tilde{t}_{r-1})\ \rho_{k-1})$。

(5) 首先注意，对于 $j \leq k$，我们有 $\mathtt{P}_j$ 是 $\mathtt{P}_k$ 在集合 $\{0, \ldots, |\mathtt{V}_j| - 1\}$ 上的限制。此外，$\mathtt{H}_j = [\min(B) : B \in \mathtt{P}_j]$，因此 $\mathtt{P}_j$ 决定了 $\mathtt{H}_j$。另一方面，$\mathtt{N}_0$ 定义为

$\mathtt{H}_0$，而 $\mathtt{N}_{j+1} = \mathtt{H}_{j+1} \setminus \mathtt{H}_j$。还要注意，$\mathtt{V}'_0$ 也仅由 $\bar{a}$ 的长度决定，并且 $\mathtt{V}'_{j+1}$ 是由 $\mathtt{H}_j$ 和 $\mathtt{N}_j$ 得到的，因此它由 $\mathtt{H}_j$ 和 $\mathtt{H}_{j-1}$ 决定。利用这些观察，结论现在可由一个简单的归纳论证得出。

(6) 设 $K = \mathrm{K}_{\bar{a}}$。由 (5) 可知，对每个 $k \leq K$，都有 $\mathtt{P}_{\bar{a},k} = \mathtt{P}_{\bar{b},k}$ 和 $\mathtt{V}'_{\bar{a},k} = \mathtt{V}'_{\bar{b},k}$。设 $t, s \in \mathcal{T}_K$ 满足 $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$。定义 $i = F_K(t, \rho_{\bar{a},K-1})$ 和 $j = F_K(s, \rho_{\bar{a},K-1})$。再定义 $\hat{}t = \mathtt{V}'_{\bar{a},k}[i]$ 和 $\hat{}s = \mathtt{V}'_{\bar{a},k}[j]$。这意味着 $\hat{}t^{\mathbf{A}}(\bar{a}) = \hat{}s^{\mathbf{A}}(\bar{a})$，并且由于 $\mathtt{V}'_{\bar{a},K} = \mathtt{V}'_{\bar{b},K}$，我们有 $\hat{}t^{\mathbf{A}}(\bar{b}) = \hat{}s^{\mathbf{A}}(\bar{b})$，这进而意味着 $t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b})$。这表明 $\bar{a} \approx_K \bar{b}$，因此由引理 1.(2)¯ 可得 $\bar{a} \approx b$。

□

给定有限代数 **A** 和 $\bar{a} \in A^n$，通过 isoType(**A** $\bar{a}$) =(iType($\bar{a}$),iUniv($\bar{a}$)) 定义 iType($\bar{a}$) 和 iUniv($\bar{a}$)。注意，iUniv($\bar{a}$) 是一个列表，其中包含 Sg($\bar{a}$) 中所有不重复的元素。

下一个推论总结了引理 2 中汇集的结果，并使用了刚刚引入的术语。它表明，我们的算法的输出能够捕捉有限代数结构中元组 $\bar{a}$ 的同构类型。

¯**推论 1.** 设 **A** 为有限代数。那么对于任意 $\bar{a}, b \in A^n$，以下条件等价：

¯*1.* $\bar{a} \approx b$ ¯*2.* 类型($\bar{a}$) =类型(b)。

¯此外，如果这两个条件中任一条件成立，那么映射 $\gamma : \mathrm{iUniv}(\bar{a})[j] \mapsto \mathrm{iUniv}(b)[j]$ *for* – $j \in \{0, ..., |\mathrm{iUniv}(\bar{a})| - 1\}$ 是从 **生成子代数**($\bar{a}$) 到**生成子代数**(b) 的同构，它将 $\bar{a}$ 映射到 b。

– ¯证明。如果 $\bar{a} \approx b$，根据引理2.(3)，我们有 iType($\bar{a}$) = iType(b)。反之，如果¯ ¯iType($\bar{a}$) = iType(b)，那么根据引理(2).(6)，我们有 $\bar{a} \approx b$。无论哪种情形，根据引理(2).(3)，我们得到 $\mathtt{X}_{\bar{a}} = \mathtt{X}_{\bar{b}}$ for $\mathtt{X} \in \{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$。特别地，$\mathtt{H}_{\bar{a}} = \mathtt{H}_{\bar{b}}$，由此可知

– ¬| iUniv($\bar{a}$)| = |iUniv(b)|。由于 iUniv($\bar{a}$) 中的所有元素（以及 iUniv(b) 中的所有元素）都各不相同，因此 γ 是一个定义良好的双射。此外，由于 $\mathtt{P}_{\bar{a},0} = \mathtt{P}_{\bar{b},0}$，我们有 γ将每个 a_j 映射到 b_j。为证明 γ 是一个同态，令 t 为一个项。令 $i \in \mathtt{H}_{\bar{a}}$ 满足 $t^{\mathbf{A}}(\bar{a}) = \mathtt{V}_{\bar{a}}[i]$，并定义 $\hat{}t = \mathtt{V}'_{\bar{a}}[i]$。根据(2)，$t$ 和$\hat{}t$ 在 $\bar{a}$上一致，因此它们在 $\bar{b}$上也一致。因此我们有 $\gamma(t^{\mathbf{A}}(\bar{a})) = \gamma(\hat{}t^{\mathbf{A}}(\bar{a})) = \mathtt{V}_{\bar{b}}[i] = \mathtt{V}'_{\bar{b}}[i]^{\mathbf{A}}(\bar{b})$，并且由于 $\mathtt{V}'_{\bar{a}} = \mathtt{V}'_{\bar{b}}$，这等于 $\hat{}t^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\gamma(\bar{a}))$。

□

5 使用合并策略计算 QfDefAlg

本节介绍我们开发的第一个算法，以下简称合并算法，用于判定有限代数结构中的量词自由可定义性。该算法基于对定理 1的略微改进，其表述如下，称为推论 2。在执行该算法之前，按照下一小节中的说明对其输入进行预处理是很方便的。

5.1 预处理输入关系

设 $\bar{a} = (a_0, \ldots, a_{k-1})$ 是一个元组。我们将 $\bar{a}$ 的模式（记作 pattern $\bar{a}$）定义为 $\{0, \ldots, n-1\}$ 的划分，使得 i, j 属于同一个块，当且仅当 $a_i = a_j$。例如，pattern(a, a, b, c, b, c) $= \{\{0, 1\}, \{2, 4\}, \{3, 5\}\}$。我们用 $\lfloor \bar{a} \rfloor$ 表示从 $\bar{a}$ 中删除所有与先前元素相等的元素后得到的元组。例如，$\lfloor (a, a, b, c, b, c) \rfloor = (a, b, c)$。设 R 是一个关系，θ 是一个模式，我们定义：

- $\lfloor R \rfloor := \{\lfloor \bar{a} \rfloor : \ \bar{a} \in R\}$,
- spec(R): $= \{|\lfloor \bar{a} \rfloor| : \ \bar{a} \in R\}$,
- R_θ : $= \{\bar{a} \in R :$ 模式 $\bar{a} = \theta\}$

给定来自 A 的元组 $\bar{a}$ 以及 R_0、.....、R_{l-1} 上的关系 A，令 relType$(\bar{a}$、R_0、.....、$R_{l-1})$ $\in$ {True、False}^{l} 表示元组（R_0（$\bar{a}$）.....、R_{l-1}（$\bar{a}$））。对于正整数 k，令 $A^{(k)}$: $= \{$（a_0、.....、a_{k-1}）$\in A^k$: $a_i = a_j \Leftrightarrow i = j\}$。

推论 2。 设**A** 为有限代数，$R \subseteq A^m$ 并设 R_0、.....、R_{l-1} 为所有形如 $\lfloor R_{\text{pattern}\,\bar{a}} \rfloor$ 的关系，其中 $\bar{a} \in R$。以下各项等价：

1. R 在**A**中是量词自由可定义的。
2. $R_0, \ldots, R_{l-1}$ 在**A**中是量词自由可定义的。
3. 对所有 $k \in \text{spec}(R)$ 以及所有 $\bar{a}, \bar{b} \in A^{(k)}$，若$\bar{a} \approx \bar{b}$，则$\text{relType}(\bar{a}, R_0, \ldots, R_{l-1}) = \text{relType}(\bar{b}, R_0, \ldots, R_{l-1})$。

证明。 等价关系 (1) $\Leftrightarrow$ (2) 是一个简单的练习，而 (2) $\Leftrightarrow$ (3) 只是定理 1 的另一种表述。 □

因此，在执行算法之前，我们可以将目标输入关系精简为不含冗余信息的关系。鉴于搜索空间取决于目标的元数且呈指数增长，这可能会对性能产生重大影响。为此，根据上述推论，我们只需计算所有形如 $\lfloor R_{\text{pattern}\,\bar{a}} \rfloor$for $\bar{a} \in R$ 的关系。

5.2 合并算法的剖析

设 **A** 是一个有限代数，并设 $R \subseteq A^n$。用于判定 R 在 **A** 中是否为量词自由可定义的合并策略可概括如下。

1. 按照第 5.1 小节中的说明预处理输入关系，得到关系 $R_0, \ldots, R_{l-}$ 1·

2. 对每个 $k \in$spec(R)，计算由等价关系 $\approx$ on$A^{(k)}$ 诱导的划分。3. 如果存在某个 $k \in$ spec(R)，使得存在 $\bar{a}, \bar{b} \in A^{(k)}$ 满足 relType($\bar{a}, R_0, \ldots, R_{l-1}$) $\neq$ relType($b, R_0, \ldots, R_{l-1}$)，且 $\bar{a} \approx b$，则返回 **False**。否则，返回 **True**。

执行步骤 (2) 和 (3) 的一种显而易见的方法，是使用函数 isoType 为 $\bigcup_{k \in \mathrm{spec}(R)} A^{(k)}$ 中的每个元组标记其同构类型，并在每次发现两个元组具有相同的 iType 时，检查它们相对于 $R_0, \ldots, R_{l-1}$ 是否具有相同的 relType。显然，一旦发现两个元组具有相同的同构类型，就立即进行这项检查会更方便，因为如果检查未通过，算法便可停止（并返回 **False**）。另一项改进是，每当我们找到具有相同同构类型的元组 $\bar{a}, b$ 时，便可以（基本上无需额外计算成本）获得一个同构 γ，它连接 Sg($\bar{a}$) 与 Sg(b)。如果两个元组由 γ 连接，那么它们具有相同的同构类型，因此只需计算其中一个元组的同构类型即可。这些观察与一种有意安排的元组轮转遍历策略相结合，构成了我们算法的核心思想。接下来我们将说明如何实现这些思想。合并算法的伪代码见算法 2，其关键子程序见算法 3。

在执行过程中的任意时刻，存储在 $\texttt{orbits}_k$ 中的数据都是 $A^{(k)}$ 的一个划分，其中每个块都附带一些额外信息。我们将这些带注释的块称为轨道，其形式为 (B, RT, T, U)，其中 $B \subseteq A^{(k)}$ 是实际的块，RT 是所有 B 中元组的 relType，T 用于存储 B 中元组的 iType（若已知），而 U 用于存储 B 中某个元组的 iUniv（若已知）。算法启动时，每个块都是一个单元素块，并附有其 relType，而 T 和 U 则设为空值。算法以类似深度优先搜索的方式遍历并处理 $\bigcup_{k \in}$spec(R)$A^{(k)}$ 中的元组。遍历的树由 **A** 的子宇宙构成，且以 A 为根。在执行过程中的任意时刻，算法都会在这棵树的一个节点 S 上处理 $\bigcup_{k \in}$spec(R) $S^{(k)}$中的元组。为跟踪当前节点和此前访问过的节点，以及每个节点上已处理的元组，算法使用一个栈。栈中的条目形式为 (S, L, G)，其中 S 是一个子宇宙，L 是尚待处理的 S 中元组的列表，而 $G \subseteq \bigcup_{k \in}$spec($R$)$S^{(k)}$ 是一个元组集合，满足：G 中的每个元组都生成 S，且 G 中任意两个元组的同构类型都不同。（保留这样一个 G 的用途将在下文说明。）

栈初始化为 $(A, L_0, \emptyset)$，其中 L_0 是 $\bigcup_{k\in \mathrm{spec}(R)} A^{(k)}$ 中按元数递增排序的元组列表。执行过程中处理的第一个元组是 L_0的第一个元素；每当算法准备处理新元组时，栈顶都有一个三元组 (S, L, G)，并且会弹出 L 的第一个元素进行处理。为说明元组的处理方式，假设栈顶为 (S, L, G)，且元组 $\bar{a}$ 已从L中弹出。如果 $\bar{a}$的同构类型已知（即，如果 $\bar{a}$属于一个类型已知的轨道），我们就弹出 L 中的下一个元组（如果 L中没有剩余元组，则将条目 (S, L, G)从栈中移除）。另一方面，如果 $\bar{a}$的类型未知，就调用函数 isoType 来计算该类型；接下来搜索各个轨道，查看是否有轨道标记为 iType($\bar{a}$)。不过，我们检查哪些轨道（以及算法之后的行为）取决于 Sg($\bar{a}$)是否小于 S。

情形 |生成子代数($\bar{a}$)| $=$ $|S|$。这里我们只检查 G 中带有 iType($\bar{a}$) 标记的元组轨道（关于为何这已足够，见引理 4）。如果这些轨道中没有任何一个带有 iType($\bar{a}$) 标记，我们就调用函数 tag_orbit，为 $\bar{a}$ 的轨道标记 iType($\bar{a}$) 和 iUniv($\bar{a}$)，并将 $\bar{a}$ 添加到 G 中。另一方面，如果 G 中某个元素的轨道为 (B, RT, T, U)，且带有$T =$ iType($\bar{a}$) 标记，那么根据推论 1，函数 γ : iUniv($\bar{a}$)$[i] \mapsto U[i]$ 是 A 的子同构。因此，我们继续合并包含由 γ连接的元组的轨道。具体而言，包含 $\bar{a}$ 的轨道与 (B, RT, T, U) 合并，因此 $\bar{a}$ 最终会进入一个带有 iType($\bar{a}$) 标记的轨道。合并由函数 try_merge_orbits 执行（见算法 3），该函数还会检查待合并的任意两个轨道是否具有相同的关系类型（如果此项检查失败，算法 2 将停止并返回 False）。

情形 |生成子代数($\bar{a}$)| $<$ $|S|$。在这种情形下，我们检查 $\texttt{orbits}_{|\bar{a}|}$ 中的所有轨道，寻找同构类型为 iType($\bar{a}$) 的轨道。如果不存在这样的轨道，则将包含 $\bar{a}$ 的轨道标记为 iType($\bar{a}$) 和 iUniv($\bar{a}$)。我们还将 (生成子代数($\bar{a}$) L', $\{\bar{a}\}$) 压入栈顶（也就是说，我们移动到子宇宙树的一个新节点）。如果 $\texttt{orbits}_{|\bar{a}|}$ 中存在标记为 iType($\bar{a}$) 的轨道，我们就像上文一样，调用 try_merge_orbits。

最终，如果 try_merge_orbits 的所有调用都没有返回 **False**，那么 $\bigcup_{k\in \mathrm{spec}(R)} A^{(k)}$ 中的每个元组都已处理完毕；算法停止时，我们已为每个 $k \in$ spec(R) 计算出由 $\approx$ 在 $A^{(k)}$ 上诱导的划分。由于我们确保同一轨道中的所有元组具有相同的 relType，推论 2 保证 R 在 **A** 中是量词自由可定义的。

无终结符 2:

元数 MergingAlgorithm($\mathbf{A}$, R) ▷ $\mathbf{A}$ 是一个代数，且 R 是定义在 A上的一个关系
    targets $=(R_1, \ldots, R_l)$ 中的 R_j 都是形如 $\lfloor R_{\text{pattern}\,\bar{a}} \rfloor$ 的不同关系，对应于 $\bar{a} \in R_\circ$
    spec $=$ sorted(spec(R)) ▷ 按递增顺序排序
    对 $k \in$ spec **执行**
        orbits$_k = \{(\{\bar{a}\},$ 关系类型($\bar{a}$, targets) Null, Null): $\bar{a} \in A^{(k)}\}$ ▷ 初始化
    orbits $= \{$orbits$_k$: $k \in$ spec$\}$
    tuples_to_process $=$ sorted($\bigcup_{k \in \text{spec}} A^{(k)}$) ▷ 按元数递减排序
    stack $=$ [(A, tuples_to_process, $\emptyset$)] ▷ 栈的初始化
    当 stack 不为空时 **执行**
        令 (current_sub, 待_处理的元组_to,生成元) 引用栈顶
        当 元组_至_待处理时 $\neq$ [ ] **执行**
            $\bar{a}$ =取出 tuples_to_process的第一个元素
            如果 类型($\bar{a}$) = Null **那么** ▷ $\bar{a}$的类型未知
                (type_a, universe_a) = IsoType($\boldsymbol{A}, \bar{a}$)
                如果 |universe_a| = |current_sub| **那么** ▷ 生成子代数($\bar{a}$)并不更小
                    如果 $\exists$(B, RT, T, U) $\in$ {在 generators} 中存在轨道，使得 type_a = T **那么**
                        $\gamma =$ 从 universe_a 到 U 的同构
                        如果 不 try_merge_orbits(γ, orbits) **那么** ▷ 参见算法 3
                            返回 False
                    else ▷ $\bar{a}$的类型是新的
                        tag_orbit($\bar{a}$, type_a, universe_a)
                        将 $\bar{a}$ 添加到 generators
                else ▷ 生成子代数($\bar{a}$)更小
                    如果 存在 (B, RT, T, U) $\in$ orbits$_{|\bar{a}|}$ 使得 type_a = T **那么**
                        $\gamma =$ 从 universe_a 到 U的同构
                        如果不 try_merge_orbits(γ, orbits) **则** ▷ 参见算法 3
                            返回 False
                    else ▷ $\bar{a}$的类型是新的
                        tag_orbit($\bar{a}$, type_a, universe_a) ▷ 然后将新节点压入栈
                        压入（universe_a, sorted（$\bigcup_{k \in \text{spec}}$ universe_a$^{(k)}$） $\{\bar{a}\}$）
                        tag_orbit($\bar{a}$, type, universe)
                        中断
        如果 tuples_to_process = [ ] **则**
            删除栈顶元素
    返回 True

无结束 3:

```
函数 try_merge_orbits(γ, 轨道)
    对 k ∈ spec 执行
        对 ā ∈(Dom(γ))^(k) 执行
            如果 orbit(ā) ≠ orbit(γ(ā)) 则
                (B, RT, T, U) = orbit(ā)
                (B′, RT′, T′, U′) = orbit(γ(ā))
                如果 RT ≠ RT′ 那么
                    返回 False                                  ▷ 关系类型不同
                B_merge = B ∪ B′                                ▷ 块已合并
                RT_merge = RT
                如果 T ≠ Null 那么                               ▷ 第一个块已标记
                    T_merge = T
                    U_merge = U
                否则, 如果 T′ ≠ Null 则                          ▷ 第二个块已标记
                    T_merge = T′
                    U_merge = U′
                else                                            ▷ 没有块被标记
                    T_merge = Null
                    U_merge = Null
                从 orbits_k 中删除 (B, RT, T, U) 和 (B′, RT′, T′, U′)
                将 (B_merge, RT_merge, T_merge, U_merge) 添加到 orbits_k]
    返回 True
```

5.3 可靠性与完备性的证明

为了证明可靠性与完备性，我们需要一些辅助引理。

引理 3. 在算法 *2* 的执行过程中，对每个$k \in \mathrm{spec}(R)$，以下性质始终成立：

1. 集合 $\{B : (B, RT, T, U) \in \mathtt{orbits}_k\}$ 是 $A^{(k)}$ 的一个划分。
2. 对每个 $(B, RT, T, U) \in \mathtt{orbits}_k$，我们有：
   - *(a)* 对所有 $\bar{a}$，在 B 中，我们有 关系类型$(\bar{a}, R_0, \ldots, R_{l-1}) = RT$.
   - *(b)* 对所有 $\bar{a}, \bar{b}$，在 B 中，我们有 $\bar{a} \approx \bar{b}$。

(c) 如果 $U \neq \texttt{Null}$，那么存在 $\bar{a}_0 \in B$，使得 $\mathrm{iUniv}(\bar{a}_0) = U$.*(*
d) 如果 $T \neq \texttt{Null}$，那么对于每个 $\bar{a}$，在 B中，$T = \mathrm{iType}(\bar{a})$。

证明。在 $\texttt{orbits}_k$ 初始化时，这些性质显然成立。注意，算法 2 中唯一可能改变 $\texttt{orbits}_k$ 的指令，是调用函数 try_merge_orbits 和 tag_orbit；很容易看出，这些函数保持性质 (1) 和 (2)。 □

为了使下面的证明更加简洁，我们规定，在算法 2 的某次执行过程中的某一时刻:

- 若轨道 (B, RT, T, U) 满足 $T \neq \texttt{Null}$，则称其为已标记。
- 若包含 $\bar{a}$ 的（唯一）轨道已标记，则称元组 $\bar{a}$ 具有已知类型。

Lemma 4. 在算法 *2* 的执行过程中，以下性质始终成立 :

1. 若栈为 $[(S_m, L_m, G_m), (S_{m-1}, L_{m-1}, G_{m-1}) \ldots, (S_0, L_0, G_0)]$，则 $|S_m| < \cdots \lhd |S_0|$。*2.* 若从栈中移除条目 (S, L, G)，则 $\bigcup_{k \in spec(R)}$ 中的每个元组都具有已知类型。*3.* 假设栈顶为 (S, L, G)。若 $\bar{a}$ 是一个具有已知类型的元组，且满足 $|Sg(\bar{a})| < |S|$，则 $\bigcup_{k \in spec(R)} Sg(\bar{a})^{(k)}$ 中的每个元组都具有已知类型。*4.* 假设栈顶为 (S, L, G)。若 $\bar{a}$ 是一个具有已知类型的元组，且满足 $|Sg(\bar{a})| = |S|$，则以下情况之一成立：*(a)* 存在 $\bar{g} \in G$，且 $\bar{g}$ 与 $\bar{a}$ 属于同一轨道；或 *(b)* $\bigcup_{k \in spec(R)} Sg(\bar{a})^{(k)}$ 中的每个元组都具有已知类型。

5. 对于所有 $k \in \mathrm{spec}(R)$，以及满足 $T \neq \texttt{Null}$ 的任意 $O = (B, RT, T, U)$ 和 $O' = (B', RT', T', U')$ *in* $\texttt{orbits}_k$，我们有

$$T = T' \Longleftrightarrow O = O'.$$

证明。容易看出，项目（1）成立。我们证明（2）。当将一个元素（S， L， G）压入栈时，L 是一个包含 $\bigcup_{k \in \mathrm{spec}(R)} S^{(k)}$ 中所有元组的列表，并且只有在处理完 T 中的所有元组后，才会从栈中移除（S， L， G）。不难看出，当处理元组 $\bar{a}$ 时，我们会发现以下两种情况之一:

- 包含它的轨道已经被标记，或者
- 它通过调用 tag_轨道或 try_合并_轨道来标记，除非调用 try_合并_轨道返回 **False** （此时算法 2 停止并返回 **False**）。

我们来证明 (3)。假设栈顶为 (Sˉ， Lˉ， Gˉ)，并令 $\bar{a}$ 如陈述所述。注意，包含 $\bar{a}$ 的轨道可能通过两种方式被标记：要么通过调用 tag_轨道，要么通过将 $\bar{a}$ 的轨道与另一个已标记的轨道合并。首先假设算法在某个时刻执行了调用 `tag_orbit`($\bar{a}$，iType($\bar{a}$)，iUniv($\bar{a}$))。关键观察是，此时 (Sg($\bar{a}$) L'， G') 已经在栈上。现在，由于 |Sg($\bar{a}$)| $\lhd |S|$，条目 (1) 表明 (Sg($\bar{a}$) L'， G') 已不在栈上，因此根据 (2)，Sg($\bar{a}$) 中的每个元组都具有已知类型。接下来，假设 $\bar{a}$ 的轨道并非通过这种 tag_轨道调用而被标记。由于在 `orbits` 中引入新类型的唯一方式是调用 ¯tag_轨道，因此必定存在一个元组 b，使得：

- iType($\bar{b}$) = iType($\bar{a}$)，并且 `tag_orbit`($\bar{b}$、iType($\bar{a}$)、iUniv($\bar{b}$)) 已在更早的时候执行，且_
- $\bar{a}$的轨道通过最终与 b的轨道合并而被标记。

¯运用与上述相同的推理，我们可以得出结论：来自 Sg(b) 的每个元组都具有已知类型。最后注意到，在合并过程中，来自 Sg($\bar{a}$) 的每个元组最终都会进入¯一个与来自 Sg(b) 的元组所在的轨道，因此具有已知类型。

我们将（4）留给读者，因为其证明与（3）的证明相似。

最后，我们证明 (5)。注意，只需验证对 tag_轨道 和 try_合并_轨道 的调用会保持 (5)。对于 try_合并_轨道，这显然成立，因为该函数不会给轨道标记新类型。因此，我们重点考察对 tag_轨道 的调用；先看算法 2 第 31 行的调用。注意，在执行该调用之前，我们会检查要引入的类型是否尚未出现在轨道中的任何位置，因此 (5) 得以保持。最后，假设在第 21 行调用 `tag_orbit`($\bar{a}$，iType($\bar{a}$)，iUniv($\bar{a}$))。根据此次调用之前进行的检查，我们知道（在此次调用时）栈顶为 (Sg($\bar{a}$) L， G)，并且 G 中没有元组与 $\bar{a}$ 具有相同的类型。我们要证明没有任何¯轨道被标记为 iType($\bar{a}$)。为导出矛盾，假设存在 b，其轨道（记作 O'）的类型为 iType($\bar{a}$)。由于 G 中没有元组属于 O'，根据条目 (4)，Sg(b) 中的每个¯元组都具有已知类型。因此，Sg(b) $\neq A$，否则 $\bar{a}$ 就会具有已知类型。于是，(Sg($\bar{a}$) L， G) 是在处理元组 $\bar{g}$ 时被压入栈中的，而该元组¯被发现具有新类型。但这就产生了矛盾，因为 Sg(b) 包含 $\bar{g}$ 的一个同构副本，因此它本应具有已知类型。

□

命题 1。 算法 *2* 是可靠且完备的。

证明。 注意，当且仅当以下任一情况发生时，算法终止：

- 调用 try_merge_orbits 返回 **False** ，且算法 2 返回 **False**；或者
- 栈为空，且算法 2 返回 **True**。

在前一种情形中，检查算法 3 可清楚看出，try_merge_orbits 返回 **False** ，当且仅当存在两个轨道，它们被标记为相同的同构类型$^{-}$ $^{-}$，但关系类型不同。因此，根据引理 3，存在元组 $\bar{a}, b$ ，使得 $\bar{a} \approx b$ 且 $^{-}$relType($\bar{a}, R_0, \ldots, R_{l-1}$) $\neq$ relType($b, R_0, \ldots, R_{l-1}$)。于是，推论 2 表明， R 在 **A** 中不可由无量词公式定义。

接下来，假设算法因栈为空而结束。回忆一下，栈初始化为 $(A, L_0, \emptyset)$，因此，根据引理 4.(2)，算法终止时，$\bigcup_{k \in \mathrm{spec}(R)} A^{(k)}$ 中的每个元组都具有已知类型。我们证明推论 2 的 (3) 成立。固定 $k \in$spec(R)，并取$\bar{a}, \bar{b} \in A^{(k)}$ ，使得 $\bar{a} \approx \bar{b}$。现在，引理 3 与引理 4.(5) 保证 $\bar{a}$ 和 $_b$ 属于同一轨道，因此引理 3 的第 (2a) 项表明，relType($\bar{a}, R_0, \ldots, R_{l-1}$) $=^{-}$ relType($b, R_0, \ldots, R_{l-1}$)。

□

6 使用分裂策略计算 QfDefAlg

现在我们介绍第二种用于判定无量词可定义性的算法，它同样基于推论 2。我们称之为分裂算法。该算法首先与合并算法相同，对目标关系进行预处理（见第 5.1 小节），并将其拆分为若干新的目标，形式为 $\lfloor R_{\mathrm{pattern}\,\bar{a}} \rfloor$ ，其中 $\bar{a}$ 属于原始目标关系（尤其是其中的元组不含重复项）。下一步是按元数对这些新目标分组并进行处理。如果其中任何一部分不可定义，算法就返回**False** （并附上一个反例）。另一方面，如果所有元数下的所有部分都可定义，算法便返回**True**，并附上一个定义公式。

为简化起见，下面的介绍将描述算法如何处理原始目标中的单个部分（而非所有具有相同给定元数的部分）。算法 4 展示了相应的伪代码。读者应该不难理解如何修改该算法，使其适用于一般情形。或许唯一需要注意的是如何生成最终公式，详情见下文。

无结束 4:

```
函数 SplittingAlgorithm(A，目标)
    formula = ⊥                                          ▷ 候选公式的初始化
    full_blocks = []
    k =目标的元数
    元组 ={t for t in A^(k)}                              ▷ 不含重复元素的元组
    项_至_处理 = [x_i 对于 i 属于 {0, . . . , 元数(target) − 1}]
    见证 = []
    新_见证 = []
    局部_公式 = ⊤
    步骤 = 0
                                          ▷ 现在，我们创建包含块字段的结构
    B_0 = 块(tuples，witnesses，new_witnesses，terms_to_process，
              local_公式，步骤)
    块_to_process = [B_0]                                ▷ 栈中的第一个块
    while 块_to_process ≠ [] do
        B =弹出要处理的块_到_
        如果 B。tuples ⊆ 目标那么                          ▷ B 是一个完整块
            formula = formula ∨ B。formula
            full_blocks。追加（B）
            继续
        否则，如果 B.tuples ∩ 目标 = ∅ 则                  ▷ B 是一个可丢弃块
            继续
        else                                             ▷ B 是一个混合块
            如果 B.terms_要处理_ == [] 和 B.new_见证 == [] 则
                返回 (False, B) ▷ B 是一个混合终端块，因此 B 是反例
            块_至_处理 = 处理_混合_块(B) + 块_至_处理
    返回 （True，formula）
```

6.1 分裂算法的剖析

与合并策略不同，合并策略会先完整计算每个元组的同构类型，然后才据此划分元组集合；这个新算法则从整个 k-元组集合（不含重复项）开始。该集合中的 k 是目标关系的元数，起初构成单个块，随着算法推进，这个划分会不断细化。核心思想是：在任意时刻，同一块中的元组尚未被判定为彼此不同构。算法的一步包括：对块中的所有元组计算一个项，并将所得值与先前已求值项的值进行比较。然后，根据元组所求得的值，将块划分为若干后继块。为此，每个块都保存已求值项的信息，以及生成接下来要计算的项所需的信息。为了存储这些带注释的块，算法使用一种名为 Block 的数据类型，可以将其视为包含以下五个字段的字典：。

- 一个集合 `tuples`，其中包含该块的元组，
- 三个项列表 `witnesses`, `new_witnesses`, `terms_to_process`,
- 以及 `formula`，其中包含一个无量词公式。

项见证列表和新_见证列表分别发挥着与算法 1 中变量 H 和 N 类似的作用。要计算的项会针对给定的每个深度 d，在当前块及其后继块中一次性生成。为避免生成冗余项，我们只构造至少包含一个深度为 $d-1$且在上一轮产生了新值的项的项。这些项存储在 `new_witnesses` 中，而避免生成冗余项正是该字段的唯一用途。为了理解字段 witnesses 的作用，我们来看一下处理块 B 时会发生什么。若一个块要么包含于目标中，要么与目标不相交，则称其为 纯块。由于算法 4 本质上是在寻找反例，即寻找一对被目标关系分离的同构元组，因此无需进一步检查纯块来寻找该反例。与目标不相交的块称为可弃置块，无需再作任何处理。然而，包含于目标中的块称为完全块，它们会将自身的公式作为输出公式的一个析取项。因此，算法 4（第 15 至 20 行）处理块时，首先检查该块是完全块还是可弃置块。接下来，我们分析非纯块（称为混合块）的处理方式。设 B 是一个混合块。我们首先考虑 B.`terms_to_process` 和 B.`new_witnesses` 为空的情形。这种块称为终止块。当且仅当（引理 6，（2））该块中的所有元组都同构时，才会发生这种情况；由于 B 是混合的，我们已经找到了一个反例，于是算法

停止（第 23 行）。接下来假设 B.terms_to_process 或 B.new_witnesses 至少有一个非空。此时，算法 4 调用函数 process_mixed_block 处理 B。执行时，该函数首先检查 B 中是否有待处理的项。如果没有，则生成一个新的项列表（深度增加），并将其赋给 B.terms_to_process。另一方面，如果 B.terms_to_process 非空，则取出其第一个元素 t。接着，根据元组通过 t 得到的值，将 B 中的元组分配到各个后继块中；对于每个 $s \in B$.witnesses，创建一个新块，其中包含所有满足 $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$ 的元组 $\bar{a} \in B$，前提是至少存在一个这样的元组。此外，还会创建另一个块，收集所有通过 t 得到的值与见证产生的任何值都不一致的元组。从伪代码中应当可以清楚看出，后继块中的其余字段是如何初始化的。该函数返回 B 的后继块列表；随后，主算法将该列表压入待处理的块栈中。最终，我们要么遇到一个终端混合块，要么没有可处理的块；在后一种情形下，算法停止并返回定义公式。

我们通过简要说明在正向情形下如何构造原始目标的公式，来结束对分裂算法的分析。回想一下，对于输入 $(\mathbf{A}\ R)$，我们首先将 R 分解为形如 $\lfloor R_\theta \rfloor$ 的关系，其中某个模式 θ 出现在 R 中。在 R 可定义于 $\mathbf{A}$ 的情形下，运行上述算法会为每个 R_θ 生成一个公式 φ_θ。现在，φ_θ 使用变量 $x_0, \ldots, x_{|\theta|-1}$。注意，除非 $\theta = \{\{0\}, \ldots, \{k-1\}\}$，否则 $|\theta|$ 小于 k：$= 元数(R)$，因此我们需要将 φ_θ 中的变量替换为 $\{x_0, \ldots, x_{k-1}\}$ 中对应的变量，并添加用于描述模式 θ 的变量间（不）等式。通过一个例子可以更好地理解这一点。假设 $k = 5$ 且 $\theta = \{\{0,1\},\{2,3\},\{4\}\}$。在这种情况下，得到的公式为

$$\tilde{\varphi}_\theta(x_0, x_1, x_2, x_3, x_4) := \varphi_\theta(x_0, x_2, x_4) \wedge x_0 = x_1 \wedge x_2 = x_3 \wedge x_0 \neq x_2 \wedge x_0 \neq x_4 \wedge x_2 \neq x_4.$$

现在，若 Θ 是 R 中出现的所有模式的集合，则在 $\mathbf{A}$ 中定义 R 的无量词公式为

$$\varphi(x_0, \ldots, x_{k-1}) := \bigvee_{\theta \in \Theta} \tilde{\varphi}_\theta(x_0, \ldots, x_{k-1}),$$

其中，对于每个 $\theta \in \Theta$，$\tilde{\varphi}_\theta$ 是针对 φ_θ 修改后的公式。

变量 step 和 full_blocks 对算法本身并非必需，但将它们纳入是为了让证明更易于理解。

无结束 5:

```
函数 process_mixed_block(B)
    successors = []
    如果 B.terms_to_process == [] 那么          ▷ 需要计算一个新术语组
        生成 项 使用 B.witnesses and B.new_witnesses in B.terms_to_process
        B.new_witnesses = []
    t =pop(B.terms_to_process)
    B.step =B.step + 1
    complement_block =B.tuples                          ▷ 使用所有元组初始化
    complement_formula = ⊤
    对于s ∈ B.witnesses 执行
        eq_tuples = [v ∈ B.tuples : t(v) == s(v)]              ▷ 新块的元组
        如果 eq_tuples ! = [] 那么
            successor :=块(eq_tuples, B.witnesses, B.new_witnesses,
                           B.terms_to_process, B.formula ∧ t = s, B.step)
            successors.append(successor)
                                          ▷ 删除新块中出现的元组
            complement_block = complement_block - eq_tuples
                                                    ▷ 附加否定公式
            complement_formula = B.formula ∧ complement_formula ∧ (t ≠ s)
    如果 complement_block != [] 则
        successor =块(complement_block, B.witnesses + [t],
                      B.new_witnesses + [t], B.terms_to_process,
                      complement_formula, B.step)
        successors.append(successor)
    如果 successors 那么                          ▷ formula 不需要增加
        successors[0].formula =B.formula
    返回 successors
```

6.2 可靠性与完备性

本节旨在证明分裂算法的正确性。我们先介绍一些初步性质。

我们的证明基于定理 1 的以下推论。回顾一下，给定 $\bar{a}$，$\bar{b} \in A^k$，当对于每一对项 t，s，$t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a}) \iff t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b})$ 时，我们称 $\bar{a} \approx \bar{b}$。

推论 3。设$\mathbf{A}$ 是一个有限代数，且 R 是$\mathbf{A}$上的一个 k 元关系。那么以下命题等价：

1. R 在$\mathbf{A}$中不可定义。
2. 存在元组 $\bar{a}$、$\bar{b} \in \mathbf{A}^k$，使得 $\bar{a} \in R$、$\bar{b} \notin R$ 且 $\bar{a} \approx \bar{b}$

给定一个项 $t(x_0, \ldots\ldots, x_{k-1})$，定义 depth($t$) 为满足 $t \in \mathcal{T}_i(x_0, \ldots\ldots, x_{k-1})$ 的最小自然数 i。此外，对于一个块 B，定义 depth(B) $= \max\{\text{depth}(t) : t \in B.\texttt{witnesses} +\!\!+ B.\texttt{terms_to_process}\}$，其中 $+\!\!+$ 是列表连接运算符。回顾一下，给定一个公式 φ 和一个模型 $\mathbf{A}$，φ 在 $\mathbf{A}$ 中的扩张记为 $[\varphi]^{\mathbf{A}}$。

引理 5。设 $\mathbf{A}$ 是一个有限代数，R 是定义在 $\mathbf{A}$ 上的 k 元关系。那么，在以 $(\mathbf{A}\ R)$ 为输入执行算法 *4* 的整个过程中，以下内容是不变的：

1. 对于任意块 B、t、$s \in B.\texttt{witnesses}$ 和 $\bar{a} \in B.\texttt{tuples}$，有 $t(\bar{a}) = s(\bar{a})$ 蕴含 $t = s$。
2. $\texttt{blocks_to_process} +\!\!+ \texttt{full_blocks}$ 中各块的元组集合两两不相交，且它们的并集包含 R。*3.* 对于任意块 B，有 $[B.\texttt{formula}]^{\mathbf{A}} = B.\texttt{tuples}$*4.* 对于任意块 B，以及 $\mathcal{T}_d(x_0, \ldots, x_{k-1})$ 中的每个项 t，其中 $d = \textit{depth}(B)$，存在 $B.\texttt{witnesses} +\!\!+ B.\texttt{terms_to_process}$ 中的一个项 $\hat{t}$，使得对于 $B.\texttt{tuples}$ 中的每个 $\bar{a}$，$t(\bar{a}) = \hat{t}(\bar{a})$。*5.* 对于任意块 B，以及 $\mathcal{T}_d(x_0, \ldots, x_{k-1}) \setminus \mathcal{T}_{d-1}(x_0, \ldots, x_{k-1})$ 中的每个项 t，其中 $d = \textit{depth}(B)$，存在 $B.\texttt{new_witnesses} +\!\!+ B.\texttt{terms_to_process}$ 中的一个项 $\hat{t}$，使得对于 $B.\texttt{tuples}$ 中的每个 $\bar{a}$，$t(\bar{a}) = \hat{t}(\bar{a})$。*6.* 对于任意满足 $\bar{a} \approx \bar{b}$ 的 $\bar{a}$、$\bar{b} \in A^{(k)}$，存在一个块 B，使得 $\bar{a}$、$\bar{b} \in B.\texttt{tuples}$。

证明。(1) 只需观察伪代码即可得出：弹出的项 t 仅当 $t(\bar{a}) \neq s(\bar{a})$ 对所有 $s \in B.\texttt{witnesses}$、$\bar{a} \in B.\texttt{tuples}$ 都成立时，才会被追加到正在构造的块 B 的项列表中。

(2) 注意，过程_mixed_块在将正在处理的块从 `blocks_to_process` 移至 `full_blocks` 时（算法 4，第 15 至 18 行），或在增加 depth(B) 时（算法5，第 3 至 5 行），不会拆分该块。因此，我们只需证明在增加 `step` 时不变式仍然成立（算法5，第 7 至 22 行）。由于只有可丢弃块会被丢弃，R 始终包含于 `blocks_to_process` $+\!\!+$ `full_blocks`中的块的并集中；根据算法5第 15 行可知，这些块仍然两两不相交。

(3) 对于初始块以及以下情况，该陈述显然成立：

- 将一个完全块从块_移至_过程中的完整_块时（算法 4，第 15 至 18 行）
- 从块_中移除一个可丢弃块并将其移至_过程时（算法 4，第 20 至 20 行）
- 增加 depth(B) 时（算法5，第 4 行）
- 从 t 项_中弹出一个项后，转到_处理，增加步骤数，但保留未 B划分的部分（如果算法5第20行中的条件成立）

现在考虑块 B 在弹出项 t后被划分的情形（算法5第10至19行）。归纳地假设 $[B.\texttt{formula}]^A = B.\texttt{tuples}$。根据构造，对于每个新块，该陈述变为 $[\ B.\texttt{formula} \wedge t = s\ ]^A =$ ( ) ( )，其中某 $[t = s]^A \cap B.\texttt{tuples}$ s $B.\texttt{witnesses}$ 个属于，而这由归纳假设立即得到。

（4）和（5）我们对 d进行归纳。在 $d = 0$ 和 `step` $= 0$ 的情形下，我们有 $B.\texttt{witnesses} = [\,]$ 和 $B.\texttt{terms_to_process} = [x_0, \ldots, x_{k-1}]$。注意，我们可以假设语言中没有常元符号，因为它们可以由常值一元函数替代，这意味着该情形下陈述成立。

现在假设 $d > 0$ 和 `step` > 0。我们注意到，增加 `step`后的所有指令都不会破坏不变式：从 $B.\texttt{terms_to_process}$ 中弹出一个项后，随该过程产生的块继承了 $B.\texttt{witnesses}$ 和 $B.\texttt{terms_to_process}$的内容，唯一的例外可能是被弹出的项 t，前提是它在待创建块的所有元组上都与另一个项一致。

因此，我们实际上可以假设 `step` $= 0$。此时，算法在从先前的混合块中构造出一系列 $B.\texttt{terms_to_process}$ 中的项后，达到了当前状态

在此过程中， terms_to_process 被清空，使 d 增加了 1。令 $t = f(t_0, \ldots\ldots, t_{r-1}) \in \mathcal{T}_{d+1}$。根据归纳假设，存在项 $\hat{}\, t_0, \ldots\ldots, \hat{}\, t_{r-1} \in \mathcal{T}_d$，每一项都满足 $t_i^{\mathbf{A}}(\bar{a}) = \hat{}\, t_i{}^{\mathbf{A}}(\bar{a})$，并且我们可以假设存在 i_0，使得 $t_{i_0} \in \mathcal{T}_d(x_0, \ldots\ldots, x_{k-1}) \setminus \mathcal{T}_{d-1}(x_0, \ldots\ldots, x_{k-1})$。但这意味着 $\hat{}\, t = f(\hat{}\, t_0, \ldots\ldots, \hat{}\, t_{r-1}) \in B.\texttt{terms_to_process}$。

(6) 算法开始时，每一对这样的元组都位于包含所有元组的原始块中；当一个块被划分时，第 10 行至第 19 行显然会将该对元组放入同一个块中。

□

引理 6. 设 B 是一个满足 $B.\texttt{new_witnesses} = B.\texttt{terms_to_process} = [\,]$ 的块。则：

1. 对于任意项 t，存在一个项 $\hat{}\, t \in B.\texttt{witnesses}$，使得对于每个 $\bar{a} \in B.\texttt{tuples}$，$t(\bar{a}) = \hat{}\, t(\bar{a})$。

2. 如果 $\bar{a}$、$\bar{b} \in B.\texttt{tuples}$ 和 t、s 是任意项，那么 $t(\bar{a}) = s(\bar{a})$ 蕴含 $t(\bar{b}) = s(\bar{b})$。

证明。 (1) 定义 $d = \mathrm{depth}(B)$，并令 $t \in \mathcal{T}_d$ 为一个项。如果 $\hat{}\, t$ 是由 (4) 给出的项，那么 (5) 以及假设表明 $\hat{}\, t \in \mathcal{T}_{d-1}$。由这一观察通过简单的归纳论证即可得到结果。

（2）取 $\bar{a}$，$\bar{b} \in B.\texttt{tuples}$。对于任意项 t，s，令 $\hat{}\, t$，$\hat{}\, s$ 为（1）给出的项。那么 $\hat{}\, t(\bar{a}) = \hat{}\, s(\bar{a})$，由引理 5（1）可知，这意味着 $\hat{}\, t = \hat{}\, s$，因此 $t(\bar{b}) = \hat{}\, t(\bar{b}) = \hat{}\, s(\bar{b}) = s(\bar{b})$。

□

命题 2。 算法 *4* 是可靠且完备的。更准确地说，对于输入 $(\mathbf{A}\ R)$，算法 *4* 最终会停止，并且要么返回一个满足 $R = [\varphi]^{\mathbf{A}}$ 的公式 φ，要么返回一个反例。当且仅当 R 不是量词自由可定义时，返回反例。

证明。 我们先考虑 R 是量词自由可定义的情形。这意味着不会有任何混合块 B成为终止块，从而每个混合块最终都会被分裂为（纯）完整块或可丢弃块。由于这些块会从 blocks_to_process 和A 中移除，且二者均为有限集，这保证了终止。还应注意，full_blocks 仅包含纯满块；因此，由于算法通过清空 blocks_to_process而停止，引理（5）中（2）的并集恰好与 R 一致，算法返回一个公式 φ，该公式按构造写成析取范式。其每个析取项对应于添加到 full_blocks的一个纯满块，并且根据引理（5）中的（3），每个此类块上的元组构成相应析取项的扩张。由上述考虑可得 $R = [\varphi]^{\mathbf{A}}$。

如果算法没有返回公式 φ，这意味着它在发现反例后终止；根据推论 3，当且仅当 R 不是量词自由可定义时才会出现这种情况。

□

7 合并算法与分裂算法的实证比较

既然有两种不同的算法方法来解决无量词可定义性问题，自然会想知道哪一种表现更好。然而，要在理论层面进行这种比较极其困难。因此，我们采取折中的办法，在一系列测试场景上运行（并实现）这些算法。作为这些测试的副产品，我们对这些实现的输入规模方面的实际局限有了更深入的认识。

由于无量词可定义性问题没有标准测试集，我们在三类代数结构上运行了算法：布尔代数、阿贝尔群和随机生成的代数。后者通过在固定全集上选择随机运算生成。对于每个类别，我们在 300 个代数上运行测试，并取墙钟时间样本的中位数。

考察这两种算法策略后，不难得出结论：最坏情形是目标关系可定义的情形。测试过程中进一步证实了这一点：我们发现，两种算法都只需极少时间即可判定给定目标关系不是无量词可定义的。因此，本文 प्रस्तुत的所有测试均针对目标可定义的输入。

我们的实现使用 Python 3 编写，遵循 GPL 3 许可证。合并算法的源代码可在 https://github.com/pablogventura/QfDefAlg 获取，分裂算法的源代码可在 https://github.com/pablogventura/QfDefAlgSplitting 获取。所有测试均使用配备 12 个核心的 Intel Xeon E5-2620v3 处理器完成（不过，我们的算法不使用并行化），处理器频率为 2.40GHz，内存为 128 GiB DDR4 2133MHz。内存从未成为问题。

7.1 评估合并算法的性能

我们的第一组实证数据涉及算法 2 在各种输入下的性能。代数的基数依次为 4、8、16 和 32。在所有情形中，目标都是由代数论域中的所有有序对组成的二元关系。随机代数配备一个二元运算和一个三元运算。阿贝尔群是由循环群 $\mathbb{Z}_k$ 的乘积构成的。

结果如图 2 所示。我们可以看到，算法如何利用代数的内部对称性：随着子同构数量的减少，运行时间会增加。特别值得注意的是阿贝尔群曲线的表现。由于测试中的所有群都是因子属于 $\{\mathbb{Z}_2, \mathbb{Z}_4\}$ 的直积（例如，16 元群是 $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_4$，而 32 元群是 $\mathbb{Z}_2 \times \mathbb{Z}_4 \times \mathbb{Z}_4$），较大的群

具有重复因子会增加子同构的数量，因此我们的算法性能更佳。

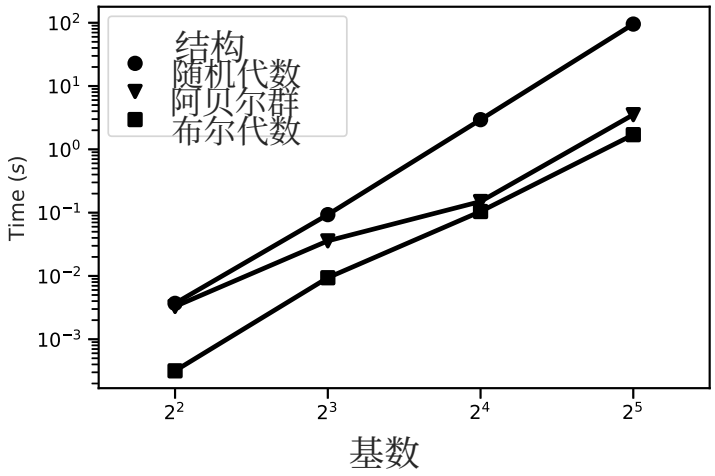


图 2：使用合并算法判定可定义性的时间

7.2 评估分裂算法的性能

现在我们转而分析算法 4 的性能。事实证明，该算法能够处理规模高达 128 个元素的输入代数。更令人瞩目的是，这些测试中的目标关系都是三元关系，而目标的元数会使搜索空间呈指数级增长（关于这一点的更多内容，请参见 [1], ，其中研究了 QfDefAlg 的参数化复杂度）。还值得一提的是，由于分裂算法在目标关系包含某一给定长度的所有元组时会立即给出答案，因此这些测试的输入关系是通过对随机构造的公式取扩张得到的。

从图 3 可以看出，尽管内部对称性更多的结构仍然需要更少的处理时间，但现在差异并不那么明显。这可以解释为，分裂算法无需完全确定搜索空间中每个元组的同构类型。

7.3 比较两种方法

我们最后一组实证数据展示了两个算法在同一组输入样本上的性能。此处，与分裂算法的测试相同，输入关系是

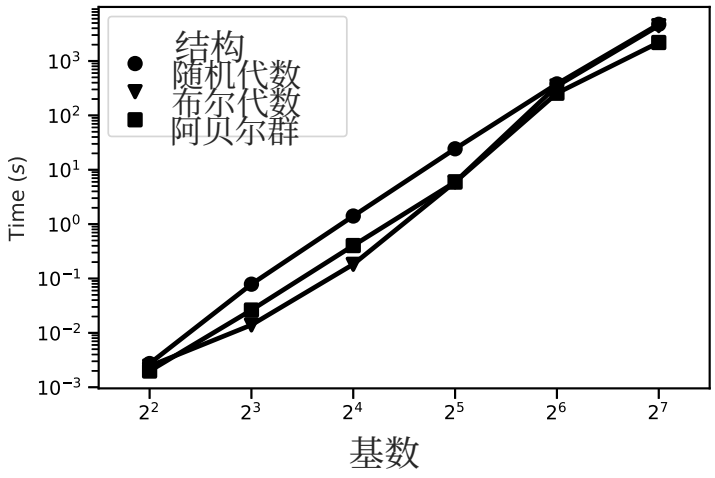


图 3：使用分裂算法判定可定义性的时间

通过一个随机生成的无量词公式的扩张得到的。结果如图 4 所示。正如先前的测试所预期的那样，分裂算法的性能远胜于合并算法。事实上，可以看到，对于规模为 32 及以上的模型，合并算法的执行时间超出了考察的时间范围（推算的执行时间超过 25 小时）。造成这种显著性能差异的一个明显原因是，与合并算法不同，分裂算法不会计算搜索空间中每个元组的同构类型。此外，对于分裂算法处理的许多元组，其同构类型只会被部分计算。考虑到在正例情形下，分裂算法还会给出一个公式，我们可以推断，分裂算法显然是一种更高效的策略。

8 结论与未来工作

我们介绍了两种算法，用于判定纯函数语言中由无量词一阶公式定义的可定义性问题。这些算法基于 [6] 中给出的无量词可定义性的语义刻画。我们证明了该问题是 coNP 完全的，这与我们在关系情形 [1] 中得到的复杂度相同。此前，我们开发了一种算法，用于刻画给定结构中元组 $\bar{a}$ 的同构类型（特别是可以求得它所生成的子宇宙）。

我们的第一个可定义性判定过程（合并算法）使用 $k \in \mathrm{spec}(R)$ 对宇宙中所有 k-元组的集合进行划分，并且必须穷尽搜索空间才能给出

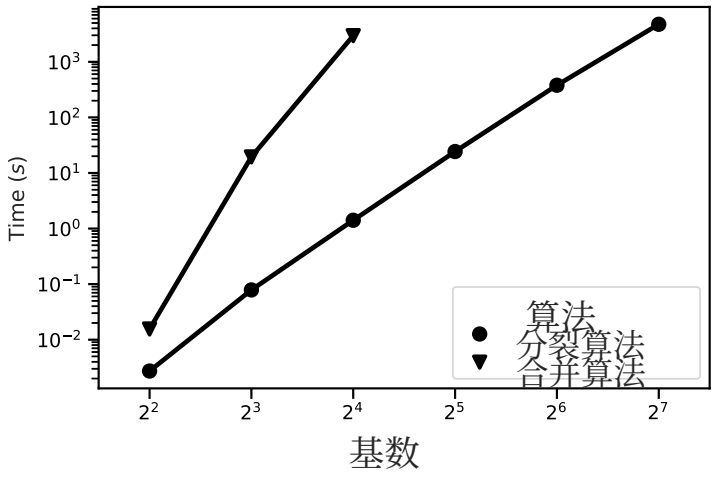


图 4：使用合并算法与分裂算法的耗时

肯定答案。因此，我们的算法的执行时间随参数 k 呈指数级增长。实证测试证实了这种对 k 的指数级依赖，并让我们了解了输入结构的基数以及我们进行测试的代数族所产生的影响。测试还证实了这一猜想：这种基于对称性存在的策略，在子同构数量较多的模型上表现会更好，正如我们

分裂算法从一个元组块开始，并随着对其中元组的同构类型的判定逐步进行，对各块进行划分。这与先前的策略不同：先前我们从孤立元组开始，并将具有相同同构类型的块合并。正如预期的那样，这种新策略效率高得多，因为无需完全确定块中元组的同构类型，就可以将整个块丢弃。实证测试清楚地证实了其性能优势。此外，这种方法还能在肯定情形下得到定义公式，这构成了一项根本优势。

未来研究有几个可能的方向。由于缺乏用于判定可定义性的标准测试，我们所呈现的实证测试只是初步的，还需要进一步评估。测试结果表明，分裂算法生成的公式（在肯定情形下）非常庞大。因此，一个由此引出的有趣问题是，如何简化生成的公式（如果可能的话）。这似乎是一个难题，而且我们知道，在某些情形下，定义公式必然很大（见 [1]）。我们计划进行的一项看似直接的改进，是将本文中的算法扩展为

判定有限模型类上的可定义性，而不是仅针对单个结构。最后，找出问题条件良好且存在多项式时间算法的代数类，也将是一个有趣的研究方向。

参考文献

[1] C. Areces、M. Campercholi、D. Penazzi 和 P. Ventura。开放一阶公式的可定义性复杂度。已提交，*arXiv:1904.04637*，2019。[2]C. Areces、M. Campercholi 和 P. Ventura。通过子同构判定开放可定义性。载于逻辑、语言、信息与计算——第*25*届国际研讨会，*WoLLIC 2018*，哥伦比亚波哥大，*2018*年*7*月*24*日至*27*日，论文集，第91–105页，2018。[3]C. Areces、S. Figueira 和 D. Gorín。在指称表达式生成中使用逻辑。载于 S. Pogodalla 和 J. Prost（编），第六届国际计算语言学逻辑方面会议论文集（*LACL 2011*），计算机科学讲义第6736卷，第17–32页。施普林格，2011。[4] C. Areces、A. Koller 和 K. Striegnitz。作为描述逻辑公式的指称表达式。载于第五届国际自然语言生成会议论文集（*INLG'08*），第42–49页。计算语言学协会，2008。[5] M. Arenas 和 G. Diaz。一阶逻辑可定义性问题的精确复杂度。*ACM*数据库系统汇刊，41(2)：13:1–13:14，2016。[6] M. Campercholi 和 D. Vaggione。函数和关系可定义性的语义条件。普遍代数，76(1)：71–98，2016年9月。[7] H. Ebbinghaus、J. Flum 和 W. Thomas。数理逻辑。施普林格出版社，1994。[8] D. Kavvadias 和 M. Sideri。逆可满足性问题。*SIAM*计算杂志，28(1)：152–163，1998年1月。[9] E. Krahmer 和 K. van Deemter。指称表达式的计算生成：综述。计算语言学，38(1)：173–218，2012。[10] Q. T. Tran、C.Y. Chan 和 S. Parthasarathy。按输出查询。载于第*35*届*SIGMOD*国际数据管理会议论文集——*SIGMOD '09*。ACM出版社，2009。

[11] R. Willard. 表达性测试很困难。载于 D. Cohen（编），约束编程的原理与实践 – *CP 2010*，第 9–23 页，柏林，海德堡，2010 年。Springer Berlin Heidelberg。