Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 348 (2020) 23–41

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Deciding Quantifier-free Definability in Finite Algebraic Structures

Miguel Campercholi[1] Mauricio Tellechea[2] Pablo Ventura[3]

FaMAF
Universidad Nacional de Córdoba
Córdoba, Argentina

Abstract

This work deals with the definability problem by quantifier-free first-order formulas over a finite algebraic structure. We show the problem to be coNP-complete and present a decision algorithm based on a semantical characterization of definable relations as those preserved by isomorphisms of substructures. Our approach also includes the design of an algorithm that computes the isomorphism type of a tuple in a finite algebraic structure. Proofs of soundness and completeness of the algorithms are presented, as well as empirical tests assessing their performances.

Keywords: Definability, logic, decision algorithm, complexity

1 Introduction

Given a logic $\mathcal{L}$, a model-theoretic semantics for $\mathcal{L}$ is a function that assigns to each formula φ of $\mathcal{L}$ and each structure $\mathbf{A}$ for $\mathcal{L}$ a set of tuples from the domain of $\mathbf{A}$. This set of tuples is called the *extension* of φ in $\mathbf{A}$, and we usually think of these tuples as the interpretations that make φ true in $\mathbf{A}$, though it also makes sense to think of them as the ones singled out, or named, by the properties expressed in φ. Various fundamental computational problems arise from this setting, of which surely the most prominent one is the *satisfiability* problem for $\mathcal{L}$, that for a given formula φ from $\mathcal{L}$ asks whether there is model $\mathbf{A}$ such that the extension of φ in $\mathbf{A}$ is non-empty. The *model-checking* problem for $\mathcal{L}$, which consists in computing, given a model $\mathbf{A}$ and a formula φ, the extension of φ in $\mathbf{A}$, has also played an important role in computational logic. A third fundamental problem, the one that concerns us in this article, is the *definability* problem for $\mathcal{L}$, which given a finite model $\mathbf{A}$ and a

[1] Email:camper@famaf.unc.edu.ar
[2] Email:mauriciotellechea@gmail.com
[3] Email:pablogventura@gmail.com

https://doi.org/10.1016/j.entcs.2020.02.003
1571-0661/

set R of tuples from $\mathbf{A}$ asks whether there is a formula φ such that the extension of φ in $\mathbf{A}$ agrees with R. This problem has been studied for several logics, due to its fundamental nature and its applications, for example to the Theory of Databases [11] and in the Generation of Referral Expressions [10,4,3]. The computational complexity of the definability problem has also been investigated for various logics. The article [9] investigates the definability problem for classical propositional logic under the name of *inverse satisfiability*, proving it is complete for coNP in the general case. They also characterize for which special syntactical cases the problem lies in P. It is proved in [5] that the definability problem is GI-complete (under Turing reductions) for first-order logic. For the primitive positive fragment of first-order logic the definability problem is coNEXPTIME-complete [12], and for basic modal logic [4] and some of its fragments [3] it is in P. In the article [1], it is shown that for the quantifier-free fragment of first-order logic with a purely relational vocabulary the definability problem is coNP-complete, and in the same article a parameterized version of the problem is proved to be complete for the complexity class W[1].

To illustrate the concept of definability let us look at an example. Let $\mathbf{D} = \langle \{\bot, \top, u, u'\}, \wedge, \vee \rangle$ be the lattice depicted below, and let $R = \{(a, b) \in D^2 : a \leq b\}$.

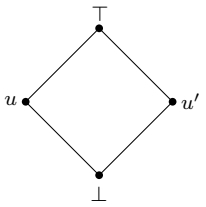

Figure 1. The lattice $\mathbf{D}$

We want to know if R is quantifier-free definable in $\mathbf{D}$. That is, if there exists a quantifier-free formula $\varphi(x, y)$, in the vocabulary of $\mathbf{D}$, such that $R = \{(a, b) \in D^2 : \mathbf{D} \vDash \varphi(a, b)\}$. It is easy to see that the formula $\varphi(x, y) := (x \vee y) = y$ works. Next, consider the relation $R' = \{(\bot, u), (\bot, u'), (\bot, \top)\}$; is it quantifier-free definable in $\mathbf{D}$? The answer is no, and here is why. Note that the map $\gamma : \{\bot, \top\} \to \{u, \top\}$ is an isomorphism between sublattices of $\mathbf{D}$, and quantifier free formulas are *preserved* by such maps. That is, if $\varphi(x, y)$ is quantifier free and $\mathbf{D} \vDash \varphi(a, b)$, then $\mathbf{D} \vDash \varphi(\gamma a, \gamma b)$. Thus, for any structure, every relation definable by a quantifier-free formula is preserved by isomorphisms between substructures (we call these maps subisomorphisms). Returning to our example, we can see that $(\bot, \top) \in R'$ but $(\gamma\bot, \gamma\top) \notin R'$, so R' is not preseved by γ, and thus it is not quantifier-free definable in $\mathbf{D}$. Interestingly, the converse of this criterion holds for finite structures; i.e., if a $\mathbf{A}$ is a finite structure, then a relation $R \subseteq A^k$ is quantifier-free definable in $\mathbf{A}$ if and only if R is preserved by all subisomorphisms of $\mathbf{A}$ (see Theorem 2.1 below).

This characterization of quantifier-free definable relations is applied in [2] to obtain an algorithm that decides the definability problem for the quantifier-free fragment of first-order logic with a purely relational vocabulary (called the *open-*

definability problem in [2]). In the current note, which continues and complements the work in [2], we develop an algorithm based on the same semantical characterization but for algebraic vocabularies (i.e., without relation symbols). It is worth noting that in fragments of first-order logic that allow quantification, the relational and algebraic definability problems collapse, since, at the expense of introducing quantifiers, every formula can be effectively transformed into an equivalent one which is unnested. However, in the quantifier-free case the problems are not the same and require different algorithmic strategies. The main difference lies in the fact that computing the isomorphisms between substructures of a structure in the algebraic case, requires computing the subuniverses of the structure. The approach of our algorithm to decide definability in this article is based on the observation that computing the subuniverse generated by (the elements of) a tuple $\bar{a}$ in a structure $\mathbf{A}$ is not far off from computing the isomorphism type of $\bar{a}$ in a $\mathbf{A}$ (see Section 4).

In addition to presenting the above mentioned algorithm, we prove that the computational complexity of the quantifier-free definability problem for algebraic vocabularies is complete for coNP.

The paper is organized as follows. In the next section we fix notation and provide the basic definitions. In Section 3 is the exposition of our complexity result. In Section 4 we introduce the algorithm that computes the isomorphism type of a tuple in an algebra. The algorithm to decide quantifier-free-definability for algebraic structures is presented in Section 5. In Section 6 some empirical tests are performed, showing that our algorithm performs better on models with a large amount of symmetries. Finally, Section 7 provides a summary of the results obtained as well as further research directions which arise from the developments we present here.

2 Preliminaries

In this section we provide some basic definitions and fix notation. We assume basic knowledge of first-order logic. For a detailed account see, e.g., [7]. Given a first order vocabulary τ and $k \in \omega$ let $\mathcal{T}_k$ denote the set of τ-terms of depth k with variables from $\mathsf{VAR} := \{x_1, x_2, \ldots, x_n, \ldots\}$. We write $\mathcal{T} := \bigcup_{k\in\omega} \mathcal{T}_k$ for the set of all τ-terms over VAR. For a function or relation symbol s we write $\mathrm{ar}(s)$ denote the arity of s. In this note we are concerned with *algebraic* vocabularies, that is, without relation symbols. In this context, *atomic formulas* are of the form $t = t'$ where t and t' are terms. A *quantifier-free* formula is a boolean combination of atomic formulas. We write $\varphi(v_1, \ldots, v_k)$ for a formula φ whose variables are all included in $\{v_1, \ldots, v_k\}$. An *algebra* is a structure of an algebraic vocabulary; algebras are denoted by boldface letters (e.g., $\mathbf{A}, \mathbf{B}, \mathbf{C}$,...) and their universes by the corresponding non-bold letter. Given an algebra $\mathbf{A}$, a formula $\varphi(v_1, \ldots, v_k)$, and a sequence of elements $\bar{a} = (a_1, \ldots, a_k) \in A^k$ we write $\mathbf{A} \models \varphi(\bar{a})$ if φ is true in $\mathbf{A}$ under an assignment that maps v_i to a_i. We say that a subset $R \subseteq A^k$ is *quantifier-free definable* in $\mathbf{A}$ (*qf-definable* for short) provided there is a quantifier-free first-order formula $\varphi(x_1, \ldots, x_k)$ in the vocabulary of $\mathbf{A}$ such that

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

In this article we study the following computational decision problem:

QfDefAlg
Input: A finite algebra $\mathbf{A}$ and a target relation $R \subseteq A^k$.
Question: Is R qf-definable in $\mathbf{A}$?

Let $\mathbf{A}$ be an algebra. A subset $S \subseteq A$ is a *subuniverse* of $\mathbf{A}$ if S is nonempty and it is closed under the fundamental operations of $\mathbf{A}$. For a tuple $\bar{a} = (a_1, \ldots, a_k) \in A^k$ we write $\mathrm{Sg}(\bar{a})$ to denote the subuniverse of $\mathbf{A}$ generated by $\{a_1, \ldots, a_k\}$. Let $\gamma : \mathrm{dom}\,\gamma \subseteq A \to A$ be a function. we say that γ *preserves* $R \subseteq A^k$ if for all $(a_1, \ldots, a_k) \in R \cap (\mathrm{dom}\,\gamma)^k$ we have $(\gamma a_1, \ldots, \gamma a_k) \in R$. The function γ is a *subisomorphism* of $\mathbf{A}$ provided that γ is injective, $\mathrm{dom}\,\gamma$ is a subuniverse of $\mathbf{A}$, and γ preserves the graph of $f^{\mathbf{A}}$ for each $f \in \tau$. (Note that a subisomorphism of $\mathbf{A}$ is precisely an isomorphism between two substructures of $\mathbf{A}$.) The following result characterizing qf-definability in terms of subisomorphisms plays a central role in this article.

Theorem 2.1 *[6, Thm 3.1]*
Let $\mathbf{A}$ be a finite model of an arbitrary first order signature and let $R \subseteq A^k$. The following are equivalent:

- *R is definable in $\mathbf{A}$ by a quantifier-free first-order formula.*
- *R is preserved by all isomorphisms between k-generated substructures of $\mathbf{A}$.*

3 QfDefAlg is coNP-complete

In what follows a *graph* is a model $\mathbf{G}$ of the language with a single binary relation E, such that $E^{\mathbf{G}}$ is symmetric and irreflexive. In [1] it is proved that the following problem is complete for coNP.

QfDef[Graphs]
Input: A finite graph $\mathbf{G}$ and a target relation $R \subseteq G^k$.
Question: Is R qf-definable in $\mathbf{G}$?

A *subisomorphism* of a graph $\mathbf{G} = (G, E)$ is an injective function $\gamma : \mathrm{dom}\,\gamma \subseteq G \to G$ such that $(a, b) \in E$ if and only if $(\gamma a, \gamma b) \in E$.

Theorem 3.1 QfDefAlg *is* coNP*-complete.*

Proof Given a finite graph $\mathbf{G} = (G, E)$ let $\hat{0}$ and $\hat{1}$ denote the first two positive integers not in G. We define the algebra $\mathbf{G}_* := (G \cup \{\hat{0}, \hat{1}\}, f)$ where f is the binary operation

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

It is straightforward to check that:

(i) If γ is a subisomorphism of $\mathbf{G}$, then the extension γ_* of γ given by $\gamma_*(\hat{0}) = \hat{0}$ and $\gamma_*(\hat{1}) = \hat{1}$ is a subisomorphism of $\mathbf{G}_*$.

(ii) If δ is a subisomorphism of $\mathbf{G}_*$, then $\delta[G] \subseteq G$ and $\delta|_G$ is a subisomorphism of $\mathbf{G}$.

We prove next that $\langle \mathbf{G}, R\rangle \longmapsto \langle \mathbf{G}_*, R\rangle$ is a polynomial time (Karp) reduction from QfDef[Graphs] to QfDefAlg. Clearly $\mathbf{G}_*$ can be computed from $\mathbf{G}$ in polynomial time, so it remains to show that R is qf-definable in $\mathbf{G}$ if and only if R is qf-definable in $\mathbf{G}_*$. Fix a finite graph $\mathbf{G}$ and $R \subseteq G^k$. Suppose R is not qf-definable in $\mathbf{G}$. Then, by Theorem 2.1, there is γ, a subisomorphism of $\mathbf{G}$, that does not preserve R. Now (1) says that γ_* is a subisomorphism of $\mathbf{G}_*$, and since it does not preserve R, it follows that R is not qf-definable in $\mathbf{G}_*$. For the remaining direction, suppose R is not qf-definable in $\mathbf{G}_*$. Once again Theorem 2.1 produces a subisomorphism δ of $\mathbf{G}_*$ that does not preserve R. It follows from (2), that $\delta|_G$ is a subisomorphism of $\mathbf{G}$ that does not preserve R; hence R is not qf-definable in $\mathbf{G}$.

Showing that QfDefAlg is in coNP is a straightforward application of Theorem 2.1. In fact, each negative instance $\langle \mathbf{A}, R\rangle$ of QfDefAlg is witnessed by a bijection γ between subsets of A satisfying conditions easily checked in poly-time w.r.t. the size of $\langle \mathbf{A}, R\rangle$. □

It follows from the proof of Theorem 3.1 that the restriction of QfDefAlg to structures with a single binary commutative operation is already complete for coNP.

4 Computing the isomorphism type of a tuple

In this section we present an algorithm to compute the isomorphism type of a tuple $\bar{a}$ in a finite algebraic structure $\mathbf{A}$, and prove it to be correct. We start with some needed definitions and preliminary results.

Let $\mathbf{A}$ be a finite algebra, for each $k \in \omega$ we define the equivalence relation $\approx_k$ over A^n by $\bar{a} \approx_k \bar{b}$ if and only if for all terms $t, s \in \mathcal{T}_k(x_0, \dots, x_{n-1})$ we have that

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

Define the equivalence relation $\approx$ over A^n by $\bar{a} \approx \bar{b}$ if and only if $\bar{a} \approx_k \bar{b}$ for all $k \in \omega$. For $\bar{a} \in A^n$ let

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

Note that, since $\mathbf{A}$ is finite, this minimum always exists.

Lemma 4.1 *Let $\mathbf{A}$ be a finite algebra and $\bar{a}, \bar{b} \in A^n$.*

(i) *If $\bar{a} \approx_{\mathrm{K}_{\bar{a}}} \bar{b}$, then for any term $t(\bar{x})$ there is a term $\hat{t}(\bar{x}) \in \mathcal{T}_l$ with $l < \mathrm{K}_{\bar{a}}$ such that $t^{\mathbf{A}}(\bar{a}) = \hat{t}^{\mathbf{A}}(\bar{a})$ and $t^{\mathbf{A}}(\bar{b}) = \hat{t}^{\mathbf{A}}(\bar{b})$.*

(ii) *If $\bar{a} \approx_{\mathrm{K}_{\bar{a}}} \bar{b}$, then $\bar{a} \approx \bar{b}$.*

Proof (i) Observe first that from the definition it follows that $\mathrm{K}_{\bar{a}} = \mathrm{K}_{\bar{b}}$; we write K for $\mathrm{K}_{\bar{a}}$. Fix a term $t(\bar{x})$. If $t(\bar{x}) \in \mathcal{T}_{K-1}$ we can take $t = \hat{t}$, so suppose $t \in \mathcal{T}_{K+j}$

with $j \geq 0$. We show that $\hat{t}$ exists by induction on j. Suppose $j = 0$; by the definition of $\mathrm{K}_{\bar{a}}$ there is $\hat{t} \in \mathcal{T}_{K-1}$ such that $t^{\mathbf{A}}(\bar{a}) = \hat{t}^{\mathbf{A}}(\bar{a})$, and since $\bar{a} \approx_{\mathrm{K}} \bar{b}$ it follows that $t^{\mathbf{A}}(\bar{b}) = \hat{t}^{\mathbf{A}}(\bar{b})$. Assume next that $j > 0$ and suppose $t = f(t_1, ..., t_r)$, with each $t_i \in \mathcal{T}_{K+j-1}$. By the inductive hypothesis, there are terms $\hat{t}_1, ..., \hat{t}_r \in \mathcal{T}_u$ with $u < K$, each satisfying $t_i^{\mathbf{A}}(\bar{a}) = \hat{t}_i{}^{\mathbf{A}}(\bar{a})$ and $t_i^{\mathbf{A}}(\bar{b}) = \hat{t}_i{}^{\mathbf{A}}(\bar{b})$. That means $\tilde{t} := f(\hat{t}_1, ..., \hat{t}_r) \in \mathcal{T}_{u+1}$ satisfies $\tilde{t}^{\mathbf{A}}(\bar{a}) = t^{\mathbf{A}}(\bar{a})$ and $\tilde{t}^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b})$. Since $u + 1 \leq K$, by the case $j = 0$ we have a term $\hat{t} \in \mathcal{T}_l$ with $l < K$ for which $\hat{t}^{\mathbf{A}}(\bar{a}) = \tilde{t}^{\mathbf{A}}(\bar{a}) = t^{\mathbf{A}}(\bar{a})$ and $\hat{t}^{\mathbf{A}}(\bar{b}) = \tilde{t}^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b})$.

(ii) Let $t, s \in \mathcal{T}$ such that $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$. Take $\hat{t}, \hat{s}$ the terms given by (i). Since $\hat{t}^{\mathbf{A}}(\bar{a}) = \hat{s}^{\mathbf{A}}(\bar{a})$, by the hypothesis we have $\hat{t}^{\mathbf{A}}(\bar{b}) = \hat{s}^{\mathbf{A}}(\bar{b})$, and therefore $t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b})$. □

Next we introduce, in the form of pseudocode, the algorithm that computes the function isoType (see Algorithm 1 below). This function takes a finite algebra $\mathbf{A}$ and a tuple $\bar{a}$ from A and returns a representation of the isomorphism type of $\bar{a}$, and the subuniverse of $\mathbf{A}$ generated by $\bar{a}$ listed in a specific order. The isomorphism type is obtained by traversing the terms in a specified order and evaluating them on $\bar{a}$. As we shall see (Corollary 4.3), it suffices to record which terms return the same value to capture the isomorphism type of $\bar{a}$.

Let us describe how Algorithm 1 works. Fix an input algebra $\mathbf{A}$ with vocabulary τ (assume the function symbols in τ are listed in a specific order). Given a tuple $\bar{a} \in A^n$, at the start of the algorithm, variable `V` is set to $[a_0, \ldots, a_{n-1}]$. Then the while loop starts, and the elements added to `V` are obtained by applying fundamental operations of $\mathbf{A}$ to earlier values in `V`. Thus, all elements in `V` belong to $\mathrm{Sg}(\bar{a})$. Variable `P` records the repetitions in appearances of such elements as they are computed, while variable `H` stores the indexes where each element of $\mathrm{Sg}(\bar{a})$ appears for the first time in `V`. At the end of each pass through the while loop, variable `N` is assigned with the indexes that were added to `H` on that pass. Since A is finite, eventually no new elements are produced, and `N` is assigned with the empty list. This guarantees termination and the fact that all elements of $\mathrm{Sg}(\bar{a})$ are listed in `V` when the algorithm halts. Observe that the while loop is executed exactly $\mathrm{K}_{\bar{a}}$ times on input $\bar{a}$. We remark that lines 3 and 16 - 17 in Algorithm 1, whose purpose is to initialize and update the variable `V'`, are not actually needed to compute the function isoType, but are included to make our proofs easier to follow.

To compute elements in `V` on each pass, for each arity of some fundamental operation we construct a list `T` of tuples of indexes of elements where the fundamental operations of that arity will be applied. We require these tuples have at least one element that has not appeared before to avoid unnecessary computations.

To illustrate how this algorithm functions let us work out an example. Let $\mathbf{D} = \langle \{\top, \bot, u, u'\}, \wedge, \vee \rangle$ be the lattice from the introduction (see Figure 1). The fundamental operations are assumed to be ordered: $[\wedge, \vee]$. On input $\mathbf{D}$, $\bar{a} = (u, u', \bot)$, the variables are initialized as follows:

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

After the first pass through the while loop the state is:

$$\mathtt{V} = [u, u', \bot, \underbrace{u}_{u\wedge u}, \underbrace{\bot}_{u\wedge u'}, \underbrace{\bot}_{u\wedge\bot}, \underbrace{\bot}_{u'\wedge u}, \underbrace{u'}_{u'\wedge u'}, \underbrace{\bot}_{u'\wedge\bot}, \underbrace{\bot}_{\bot\wedge u}, \underbrace{\bot}_{\bot\wedge u'},$$
$$\underbrace{\bot}_{\bot\wedge\bot}, \underbrace{u}_{u\vee u}, \underbrace{\top}_{u\vee u'}, \underbrace{u}_{u\vee\bot}, \underbrace{\top}_{u'\vee u}, \underbrace{u'}_{u'\vee u'}, \underbrace{u'}_{u'\vee\bot}, \underbrace{u}_{\bot\vee u}, \underbrace{u'}_{\bot\vee u'}, \underbrace{\bot}_{\bot\vee\bot}]$$
$$\mathtt{P} = [[0, 3, 12, 14, 18], [1, 7, 16, 17, 19], [2, 4, 5, 6, 8, 9, 10, 11, 20], [13, 15]]$$
$$\mathtt{H} = [0, 1, 2, 13]$$
$$\mathtt{N} = [13].$$

Now the element $\top$, which was not in $\mathtt{V}$ before, makes its first appearance in position 13. Thus, on the next pass the fundamental operations will be applied to all pairs with elements from $[u, u', \top, \bot]$ in which $\top$ appears at least once. So, the next round produces:

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

The variable $\mathtt{N}$ is now empty due to the fact that no new elements have been produced, so Algorithm 1 halts returning the partition $\mathtt{P}$ and $\mathtt{U} = [u, u', \bot, \top] = \mathrm{Sg}(\bar{a})$. Running the algorithm for the tuple $\bar{b} = (u', u, \bot)$ we get:

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

Since the final partitions are the same for $\bar{a}$ and $\bar{b}$, by Corollary 4.3 below, the map γ from $[u, u', \bot, \top]$ to $[u', u, \bot, \top]$ is a subisomorphism satisfying $\gamma : \bar{a} \mapsto \bar{b}$.

4.1 Soundness and completeness

For $\mathtt{X}$ a variable name in $\{\mathtt{V}, \mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$ and $k \in \{0, \dots, \mathrm{K}_{\bar{a}}\}$, let

- $\mathtt{X}_{\bar{a},k}$ be the value of variable $\mathtt{X}$ in a run of Algorithm 1 with input $\mathbf{A}, \bar{a}$, after k executions of the while loop.

Also, given an arity r of a function symbol of $\mathbf{A}$, let

- $\mathtt{T}^r_{\bar{a},k}$ be the value assigned to variable $\mathtt{T}$ in a run of Algorithm 1 with input $\mathbf{A}, \bar{a}$, once an arity r has been fixed in the for loop (line 10), during the kth execution of the while loop.

We write $\mathtt{X}_{\bar{a}}$ for $\mathtt{X}_{\bar{a},\mathrm{K}_{\bar{a}}}$, and for $k > \mathrm{K}_{\bar{a}}$ we define $\mathtt{X}_{\bar{a},k}$ to be $\mathtt{X}_{\bar{a}}$. The subindex $\bar{a}$ is omitted when no confusion may arise.

Algorithm 1 Tuple isomorphism type

function IsoType($\mathbf{A}$, $\bar{a}$)
▷ $\mathbf{A}$ is an algebra and $\bar{a}$ is a tuple from A
$\mathtt{V} = [a_0, \dots, a_{n-1}]$
$\mathtt{V}' = [\text{“}x_0\text{”}, \dots, \text{“}x_{n-1}\text{”}]$ ▷ this is a list of variables (terms are represented as strings)
$\mathtt{P} = [B_1, \dots, B_k]$ the partition of $\{0, \dots, n-1\}$ where i, j are in the same block iff $a_i = a_j$
$\mathtt{H} = \mathtt{sorted}([\min(B_j) : B_j \in \mathtt{P}])$ ▷ sorted increasingly
$\mathtt{N} = \mathtt{H}$
$\mathtt{arities} = [r_1, ..., r_k]$ ▷ where $r_1 < ... < r_k$ are all the arities of operations in $\mathbf{A}$
while $\mathtt{N} \neq \emptyset$ **do**
  $\mathtt{H\_old} = \mathtt{copy}(\mathtt{H})$
  for $r \in \mathtt{arities}$ **do**
    $\mathtt{T} = \mathtt{sorted}([\bar{l} \in (\mathtt{H\_old})^r : l_j \in \mathtt{N} \text{ for some } j])$ ▷ sorted lexicographically
    for $f \in \mathtt{op}(r)$ **do** ▷ $\mathtt{op}(r)$ is the list of operation symbols of arity r in the order given by τ
      for $\bar{l} \in \mathtt{T}$ **do**
        $\mathtt{value} = f^{\mathbf{A}}(\mathtt{V}[l_0], ..., \mathtt{V}[l_{r-1}])$
        append value to V
        $\mathtt{term} = f \mathbin{+\!\!+} \text{“(“} \mathbin{+\!\!+} \mathtt{V}'[l_0] \mathbin{+\!\!+} \text{“,”} \mathbin{+\!\!+} ... \mathbin{+\!\!+} \text{“,”} \mathbin{+\!\!+} \mathtt{V}'[l_{r-1}] \mathbin{+\!\!+} \text{“)”}$
        append term to V'
        ▷ update of the partition P
        if there is $j \in \mathtt{H}$ such that $\mathtt{value} = \mathtt{V}[j]$ **then** ▷ value was already in V
          add $|\mathtt{V}| - 1$ to the block of P that contains j
        else ▷ value is new
          add the block $\{|\mathtt{V}| - 1\}$ to P
          add $|\mathtt{V}| - 1$ to H
  ▷ update of N
  $\mathtt{N} = [l : l \in \mathtt{H} \text{ and } l \notin \mathtt{H\_old}]$
$\mathtt{U} = [\mathtt{V}[j] : j \in \mathtt{H}]$ ▷ U lists the elements of $\mathrm{Sg}(\bar{a})$ in the order they appeared in V
return P, U

Given a tuple $\bar{a} \in A^n$ and $k \in \omega$ let $\rho_{\bar{a},k}$ be the equivalence relation over $\mathcal{T}_k(x_0, \dots, x_{n-1})$ defined by $t \ \rho_{\bar{a},k} \ s \iff t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$. Also, $Eq(Y)$ denotes the set of all equivalence relations over the set Y. The following lemma provides a series of technical results necessary to establish the soundness of Algorithm 1, which is done in Corollary 4.3 below.

Lemma 4.2 *For any $\bar{a} \in A^n$ and $k \in \omega$ we have:*

(i) *Each member of $\mathtt{V}'_k$ is term in the variables $x_0, \dots, x_{n-1}$ of degree at most k.*

(ii) *The invariant $\mathtt{V}_k[i] = \mathtt{V}'_k[i]^{\mathbf{A}}(\bar{a})$ holds throughout the for loop in line 13.*

(iii) *If $\bar{a} \approx_k \bar{b}$ then $\mathtt{X}_{\bar{a},k} = \mathtt{X}_{\bar{b},k}$ for $\mathtt{X}$ any variable name in $\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$.*

(iv) *For all $1 \leq k \leq \mathrm{K}_{\bar{a}}$ there is a function*

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

such that for each $t \in \mathcal{T}_k(x_0, \dots, x_{n-1})$ the index $i = F_k(t, \rho_{\bar{a},k-1})$ satisfies $i \leq |\mathtt{V}_k| - 1$ and $t^{\mathbf{A}}(\bar{a}) = \mathtt{V}_k[i]$.

(v) *For all $k \leq \mathrm{K}_{\bar{a}}$ we have that $\mathtt{P}_k$ determines $\mathtt{X}_j$ for all $j < k$ and $\mathtt{X}$ any variable name in $\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$.*

(vi) *If $\mathtt{P}_{\bar{a}} = \mathtt{P}_{\bar{b}}$, then $\bar{a} \approx \bar{b}$.*

Proof (i) is easily proved by induction in k. (ii) is clear from lines 14 to 17.
(iii) We proceed by induction in k. The case $k = 0$ is straightforward. Assume now $k+1 \leq \min(\mathrm{K}_{\bar{a}}, \mathrm{K}_{\bar{b}})$, and suppose $\bar{a} \approx_{k+1} \bar{b}$. Since $\bar{a} \approx_k \bar{b}$, by our inductive hypothesis we have that

- $\mathtt{X}_{\bar{a},k} = \mathtt{X}_{\bar{b},k}$ for $\mathtt{X}$ any variable name in $\{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$.

Note that the first line executed when starting the $k+1$th run of the while loop sets the value of $\mathtt{H\_old}$ to the value of $\mathtt{H}_k$. Now, observe that the value assigned to $\mathtt{T}$ for each arity r only depends on $\mathtt{H\_old}$ and $\mathtt{N}_k$. Thus $\mathtt{T}^r_{\bar{a},k+1} = \mathtt{T}^r_{\bar{b},k+1}$ for all arities r. This, along with the fact that the function symbols at the for loop starting at Line 12 appear always in the same order, implies that $\mathtt{V}'_{\bar{a},k+1} = \mathtt{V}'_{\bar{b},k+1}$. We prove next that $\mathtt{P}_{\bar{a},k+1} = \mathtt{P}_{\bar{b},k+1}$; suppose the indexes i and j are in the same block of $\mathtt{P}_{\bar{a},k+1}$. That is, $\mathtt{V}_{k+1}(\bar{a})[i] = \mathtt{V}_{k+1}(\bar{a})[j]$, so, by (ii), we have $\mathtt{V}'_{\bar{a},k+1}[i]^{\mathbf{A}}(\bar{a}) = \mathtt{V}'_{\bar{a},k+1}[j]^{\mathbf{A}}(\bar{a})$. Now, (i) says that $\mathtt{V}'_{k+1}(\bar{a})[i]$ and $\mathtt{V}'_{k+1}(\bar{a})[j]$ are terms of degree at most $k+1$, and they agree on $\bar{a}$. Hence they agree on $\bar{b}$, since $\bar{a} \approx_{k+1} \bar{b}$. Invoking (ii) once again yields that i and j are in the same block of $\mathtt{P}_{\bar{b},k+1}$. From the fact that $\mathtt{P}_{\bar{a},k+1} = \mathtt{P}_{\bar{b},k+1}$ it easy to see that $\mathtt{H}_{\bar{a},k+1} = \mathtt{H}_{\bar{b},k+1}$ and $\mathtt{N}_{\bar{a},k+1} = \mathtt{N}_{\bar{b},k+1}$. In particular, for any $l \leq \min(\mathrm{K}_{\bar{a}}, \mathrm{K}_{\bar{b}})$ we have that $\mathtt{N}_{\bar{a},l} = \emptyset$ if and only if $\mathtt{N}_{\bar{b},l} = \emptyset$. So $\mathrm{K}_{\bar{a}} = \mathrm{K}_{\bar{b}}$, from which it follows at once that (iii) holds for $k+1 > \min(\mathrm{K}_{\bar{a}}, \mathrm{K}_{\bar{b}})$.
(iv) Take $t = f(t_0, \ldots, t_{r-1}) \in \mathcal{T}_{k+1}$ and suppose $k+1 \leq \mathrm{K}_{\bar{a}}$. For $k = 0$, define $F_0(x_j) = j$. If $k \geq 1$, by inductive hypothesis we have the function F_k, and thus we can obtain indexes $i_0 := F_k(t_0, \rho_{k-1}), \ldots, i_{r-1} := F_k(t_{r-1}, \rho_{k-1})$ functionally from t and ρ_k (obviously $t_0, \ldots, t_{r-1}$ can be obtained as functions of t and $\rho_{k-1} = \rho_k \cap \mathcal{T}^2_{k-1}$). If $k = 0$, just take $i_0 := F_0(t_0), \ldots, i_{r-1} := F_0(t_{r-1})$. Next, define

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

and once again, note that these $\tilde{t}_l$ can be obtained functionally since $\mathtt{V}'_k$ can be obtained functionally from ρ_k. Also, by our inductive hypothesis, we have $t^{\mathbf{A}}(\bar{a}) = f(\tilde{t}_0, \ldots, \tilde{t}_{r-1})^{\mathbf{A}}(\bar{a})$.

Assume first that there is $l \in \{0, \ldots, r-1\}$ such that $i_l \in \mathtt{H}_k \setminus \mathtt{H}_{k-1}$. Then $i_l \in \mathtt{N}_k$, so $\langle i_0, \ldots, i_{r-1} \rangle \in \mathtt{T}^r_{k+1}$, and we have that the term $f(\tilde{t}_0, \ldots, \tilde{t}_{r-1})$ is appended to $\mathtt{V}'$, say with index j, during the $(k+1)$th pass of the while loop. Observe that j only depends on the signature of $\mathbf{A}$ and the numbers $\left|\mathtt{T}^u_{k+1}\right|$ for u an arity less or equal than r. But $\mathtt{T}^u_{k+1}$ depends only on $\mathtt{H}_k$ and $\mathtt{N}_k$, which in turn are determined by ρ_k. Thus j can be obtained as a function of t and ρ_k, and we can define $F_{k+1}(t, \rho_k) := j$ in the case that there is $l \in \{0, \ldots, r-1\}$ such that $i_l \in \mathtt{H}_k \setminus \mathtt{H}_{k-1}$. Suppose on the other hand that $i_0, \ldots, i_{r-1} \in \mathtt{H}_{k-1}$. Then, by (i), we have that $\tilde{t}_0, \ldots, \tilde{t}_{r-1} \in \mathcal{T}_{k-1}$, and thus $f(\tilde{t}_0, \ldots, \tilde{t}_{r-1}) \in \mathcal{T}_k$. Hence, in this case we can define $F_{k+1}(t, \rho_k) := F_k(f(\tilde{t}_0, \ldots, \tilde{t}_{r-1}), \rho_{k-1})$.
(v) Notice first that, for $j \leq k$, we have $\mathtt{P}_j$ is the restriction of $\mathtt{P}_k$ to the set

$\{0, ..., |\mathtt{V}_j| - 1\}$. Also, $\mathtt{H}_j = [\min(B) : B \in \mathtt{P}_j]$, so that $\mathtt{P}_j$ determines $\mathtt{H}_j$. On the other hand, $\mathtt{N}_0$ is defined as $\mathtt{H}_0$, while $\mathtt{N}_{j+1} = \mathtt{H}_{j+1} \setminus \mathtt{H}_j$. Observe also that $\mathtt{V}'_0$ is also determined just by the length of $\bar{a}$, and that $\mathtt{V}'_{j+1}$ is obtained from $\mathtt{H}_j$ and $\mathtt{N}_j$, so it is determined by $\mathtt{H}_j$ and $\mathtt{H}_{j-1}$. Using these observations, the conclusion follows now from an easy inductive argument.

(vi) Let $K = \mathrm{K}_{\bar{a}}$. By (v), for every $k \leq K$ we have $\mathtt{P}_{\bar{a},k} = \mathtt{P}_{\bar{b},k}$ and $\mathtt{V}'_{\bar{a},k} = \mathtt{V}'_{\bar{b},k}$. Let $t, s \in \mathcal{T}_K$ such that $t^{\mathbf{A}}(\bar{a}) = s^{\mathbf{A}}(\bar{a})$. Define $i = F_K(t, \rho_{\bar{a},K-1})$ and $j = F_K(s, \rho_{\bar{a},K-1})$. Define also $\hat{t} = \mathtt{V}'_{\bar{a},k}[i]$ and $\hat{s} = \mathtt{V}'_{\bar{a},k}[j]$. This means $\hat{t}^{\mathbf{A}}(\bar{a}) = \hat{s}^{\mathbf{A}}(\bar{a})$ and, since $\mathtt{V}'_{\bar{a},K} = \mathtt{V}'_{\bar{b},K}$, we have $\hat{t}^{\mathbf{A}}(\bar{b}) = \hat{s}^{\mathbf{A}}(\bar{b})$, which in turn implies $t^{\mathbf{A}}(\bar{b}) = s^{\mathbf{A}}(\bar{b})$. This shows $\bar{a} \approx_K \bar{b}$, so by Lemma 4.1.(ii) we have $\bar{a} \approx \bar{b}$. □

Given a finite algebra $\mathbf{A}$ and $\bar{a} \in A^n$, define $\mathrm{iType}(\bar{a})$ and $\mathrm{iUniv}(\bar{a})$ by $\mathrm{isoType}(\mathbf{A}, \bar{a}) = (\mathrm{iType}(\bar{a}), \mathrm{iUniv}(\bar{a}))$. Notice that $\mathrm{iUniv}(\bar{a})$ is a list without repetitions of all the elements in $\mathrm{Sg}(\bar{a})$.

The next corollary summarizes the results collected in Lemma 4.2 using the terminology just introduced. It shows that the output of our algorithm captures the isomorphism type of a tuple $\bar{a}$ in a finite algebraic structure.

Corollary 4.3 *Let $\mathbf{A}$ be a finite algebra. Then for any $\bar{a}, \bar{b} \in A^n$ the following are equivalent:*

(i) $\bar{a} \approx \bar{b}$

(ii) $\mathrm{iType}(\bar{a}) = \mathrm{iType}(\bar{b})$.

Moreover, if either of these conditions hold, then the map $\gamma : \mathrm{iUniv}(\bar{a})[j] \mapsto \mathrm{iUniv}(\bar{b})[j]$ for $j \in \{0, ..., |\mathrm{iUniv}(\bar{a})| - 1\}$ is an isomorphism from $\mathbf{Sg}(\bar{a})$ into $\mathbf{Sg}(\bar{b})$ mapping $\bar{a}$ to $\bar{b}$.

Proof If $\bar{a} \approx \bar{b}$, in view of Lemma 4.2.(iii) we have $\mathrm{iType}(\bar{a}) = \mathrm{iType}(\bar{b})$. Conversely, if $\mathrm{iType}(\bar{a}) = \mathrm{iType}(\bar{b})$, then by lemma (4.2).(vi) we have $\bar{a} \approx \bar{b}$. In either case, by lemma (4.2).(iii), we get $\mathtt{X}_{\bar{a}} = \mathtt{X}_{\bar{b}}$ for $\mathtt{X} \in \{\mathtt{V}', \mathtt{P}, \mathtt{H}, \mathtt{N}\}$. In particular, $\mathtt{H}_{\bar{a}} = \mathtt{H}_{\bar{b}}$, from which it follows that $|\mathrm{iUniv}(\bar{a})| = |\mathrm{iUniv}(\bar{b})|$. Since all the elements in $\mathrm{iUniv}(\bar{a})$ (and also those in $\mathrm{iUniv}(\bar{b})$) are different, we have that γ is a well-defined bijection. Also, since $\mathtt{P}_{\bar{a},0} = \mathtt{P}_{\bar{b},0}$, we have that γ sends each a_j into b_j. To prove γ is an homomorphism, let t be a term. Let $i \in \mathtt{H}_{\bar{a}}$ be such that $t^{\mathbf{A}}(\bar{a}) = \mathtt{V}_{\bar{a}}[i]$, and define $\hat{t} = \mathtt{V}'_{\bar{a}}[i]$. By (ii), t and $\hat{t}$ agree on $\bar{a}$, hence they also agree on $\bar{b}$. So we have $\gamma(t^{\mathbf{A}}(\bar{a})) = \gamma(\hat{t}^{\mathbf{A}}(\bar{a})) = \mathtt{V}_{\bar{b}}[i] = \mathtt{V}'_{\bar{b}}[i]^{\mathbf{A}}(\bar{b})$, and since $\mathtt{V}'_{\bar{a}} = \mathtt{V}'_{\bar{b}}$, this equals $\hat{t}^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\bar{b}) = t^{\mathbf{A}}(\gamma(\bar{a}))$. □

5 Computing QfDefAlg

In this section we present our algorithm to decide qf-definability in a finite algebraic structure. The algorithm is based on a slight improvement of Theorem 2.1, stated below as Corollary 5.1.

Let $\bar{a} = (a_1, \ldots, a_k)$ be a tuple. We define the *pattern* of $\bar{a}$, denoted by $\mathrm{pattern}\,\bar{a}$, to be the partition of $\{1, \ldots, n\}$ such that i, j are in the same block if and only if

$a_i = a_j$. For example, $\text{pattern}(a, a, b, c, b, c) = \{\{1, 2\}, \{3, 5\}, \{4, 6\}\}$. We write $\lfloor \bar{a} \rfloor$ to denote the tuple obtained from $\bar{a}$ by deleting every entry equal to a prior entry. E.g., $\lfloor (a, a, b, c, b, c) \rfloor = (a, b, c)$. Let R be a relation and θ a pattern, we define:

- $\lfloor R \rfloor := \{\lfloor \bar{a} \rfloor : \bar{a} \in R\}$,
- $\text{spec}(R) := \{|\lfloor \bar{a} \rfloor| : \bar{a} \in R\}$,
- $R_\theta := \{\bar{a} \in R : \text{pattern}\, \bar{a} = \theta\}$

Given a tuple $\bar{a}$ from A and relations $R_1, \dots, R_l$ over A, let $\text{relType}(\bar{a}, R_1, \dots, R_l) \in \{\text{True}, \text{False}\}^l$ be the tuple $(R_1(\bar{a}), \dots, R_l(\bar{a}))$. For a positive integer k let $A^{(k)} := \{(a_1, \dots, a_k) \in A^k : a_i = a_j \Leftrightarrow i = j\}$.

Corollary 5.1 *Let* $\mathbf{A}$ *be a finite algebra,* $R \subseteq A^m$ *and let* $R_1, \dots, R_l$ *be all the relations of the form* $\lfloor R_{\text{pattern}\, \bar{a}} \rfloor$ *for* $\bar{a} \in R$*. The following are equivalent:*

(i) R *is qf-definable in* $\mathbf{A}$*.*

(ii) $R_1, \dots, R_l$ *are qf-definable in* $\mathbf{A}$*.*

(iii) *For all* $k \in \text{spec}(R)$ *and for all* $\bar{a}, \bar{b} \in A^{(k)}$ *we have that* $\bar{a} \approx \bar{b}$ *implies* $\text{relType}(\bar{a}, R_1, \dots, R_l) = \text{relType}(\bar{b}, R_1, \dots, R_l)$*.*

Proof The equivalence between (i)⇔(ii) is an easy exercise, and (ii)⇔(iii) is just a restatement of Theorem 2.1. □

5.1 Breakdown of Algorithm 2

Let $\mathbf{A}$ be an finite algebra and let $R \subseteq A^n$. Our strategy to decide if R is qf-definable in $\mathbf{A}$ can be summarized as follows.

(i) Compute all the relations of the form $\lfloor R_{\text{pattern}\, \bar{a}} \rfloor$ for $\bar{a} \in R$. Suppose these are $R_1, \dots, R_l$.

(ii) For each $k \in \text{spec}(R)$ compute the partition induced by the equivalence relation $\approx$ on $A^{(k)}$.

(iii) If for some $k \in \text{spec}(R)$ there are $\bar{a}, \bar{b} \in A^{(k)}$ such that $\text{relType}(\bar{a}, R_1, \dots, R_l) \neq \text{relType}(\bar{b}, R_1, \dots, R_l)$ and $\bar{a} \approx \bar{b}$ return False. Otherwise, return True.

Step (i) strips down the target relation into relations without superfluous information. This can have a serious impact on performance, given that the search space depends exponentially on the arity of the target.

An obvious method for carrying out steps (ii) and (iii) is to use the function isoType to tag each tuple in $\bigcup_{k \in \text{spec}(R)} A^{(k)}$ with its isomorphism type, and every time two tuples have the same iType check that they have the same relType with respect to $R_1, \dots, R_l$. Clearly, it is convenient to carry out this test as soon as two tuples with the same isomorphism type are discovered, because if it fails the algorithm can stop (and return False). Another observation to improve on this first approach is that, whenever we find tuples $\bar{a}, \bar{b}$ with the same isomorphism type, we have access (essentially without any further computational cost) to an isomorphism γ between $\text{Sg}(\bar{a})$ and $\text{Sg}(\bar{b})$. If two tuples are connected by γ, they have the same

isomorphism type, and thus it suffices to compute the isomorphism type of one of them. These observations together with a deliberate strategy to cycle through the tuples comprise the main ideas behind our algorithm. We explain next how they are implemented.

At every point during an execution, the data stored in $\texttt{orbits}_k$ is a partition of $A^{(k)}$ where each block is annotated with some additional information. We call these annotated blocks *orbits*, and they have the form (B, RT, T, U) where $B \subseteq A^{(k)}$ is the actual block, RT is the relType of all tuples in B, T shall store the iType of the tuples in B (when known), and U shall store the iUniv of a tuple in B (when known). When the algorithm starts, every block is a singleton annotated with its relType, and T,U are set to Null. The algorithm traverses and processes the tuples in $\bigcup_{k\in \mathrm{spec}(R)} A^{(k)}$ in a DFS-like fashion. The *tree* that is traversed is comprised by the subuniverses of $\mathbf{A}$ and has A as its root. At every point of an execution, the algorithm is working on a node S of this tree by processing the tuples in $\bigcup_{k\in \mathrm{spec}(R)} S^{(k)}$. To keep track of the current and previously visited nodes, and the tuples already processed at each node, a stack is used. The entries in the stack have the form (S, L, G) where S is a subuniverse, L is the list of tuples from S still to be processed, and $G \subseteq \bigcup_{k\in \mathrm{spec}(R)} S^{(k)}$ is a set of tuples such that: every tuple in G generates S, and no two tuples in G have the same isomorphism type. (The point of keeping track of such a G shall become clear below.) The stack is initialized with $(A, L_0, \emptyset)$ where L_0 is a list of the tuples in $\bigcup_{k\in \mathrm{spec}(R)} A^{(k)}$ ordered increasingly by arity. The first tuple processed during an execution is the first element in L_0, and every time the algorithm is ready to process a new tuple, there is a triplet (S, L, G) on the top of the stack and the first element of L is popped and processed. To see how a tuple is processed, suppose the top of the stack is (S, L, G) and a tuple $\bar{a}$ has been popped from L. If the isomorphism type of $\bar{a}$ is known (i.e., if $\bar{a}$ belongs to an orbit with known type), we pop the next tuple in L (if there are no more tuples in L, the entry (S, L, G) is removed from the stack). If, on the other hand, the type of $\bar{a}$ is unknown, the function isoType is called to compute it, and the next step is to search through the orbits to see if there is one tagged with $\mathrm{iType}(\bar{a})$. However, which orbits we inspect (and the behaviour of the algorithm afterwards) depends on whether $\mathrm{Sg}(\bar{a})$ is smaller than S.

Case $|\mathrm{Sg}(\bar{a})| = |S|$. Here we only inspect the orbits of tuples in G for one tagged with $\mathrm{iType}(\bar{a})$ (to see why this suffices see Lemma 5.3). If none of these orbits is tagged with $\mathrm{iType}(\bar{a})$, we call the function `tag_orbit` to tag $\bar{a}$'s orbit with $\mathrm{iType}(\bar{a})$ and $\mathrm{iUniv}(\bar{a})$, and add $\bar{a}$ to G. If, on the other hand, there is an orbit (B, RT, T, U) of an element in G with $T = \mathrm{iType}(\bar{a})$, then, by Corollary 4.3, the function $\gamma : \mathrm{iUniv}(\bar{a})[i] \mapsto U[i]$ is a subisomorphism of $\mathbf{A}$. So, we proceed to merge orbits containing tuples connected by γ. In particular, the orbit containing $\bar{a}$ is merged with (B, RT, T, U), and thus $\bar{a}$ ends up in an orbit tagged with $\mathrm{iType}(\bar{a})$. The merging is done by the function `try_merge_orbits` (see Algorithm 3), which also checks that any two orbits to be merged have the same relType (if this test fails, Algorithm 2 stops and returns False).

Case $|\mathrm{Sg}(\bar{a})| < |S|$. In this case we inspect all orbits in $\texttt{orbits}_{|\bar{a}|}$ in search for one

Algorithm 2 Quantifier-free Definability Algorithm

function isQfDefAlg($\mathbf{A}$, R)
▷ $\mathbf{A}$ is an algebra and R is a relation over A
  targets $= (R_1, \dots, R_l)$ where the R_j's are all the distinct relations of the form $\lfloor R_{\text{pattern}\,\bar{a}} \rfloor$ for $\bar{a} \in R$.
  spec $=$ sorted(spec(R)) ▷ sorted increasingly
  for $k \in$ spec **do**
    orbits$_k = \{(\{\bar{a}\}, \text{relType}(\bar{a}, \text{targets}), \text{Null}, \text{Null}) : \bar{a} \in A^{(k)}\}$ ▷ initialization of the orbits
  orbits $= \{\text{orbits}_k : k \in \text{spec}\}$
  tuples_to_process $=$ sorted$(\bigcup_{k \in \text{spec}} A^{(k)})$ ▷ sorted decreasingly by arity
  stack $= [(A, \text{tuples\_to\_process}, \emptyset)]$ ▷ initialization of the stack
  while stack is not empty **do**
    Let (current_sub, tuples_to_process, generators) be a reference to the top of stack
    while tuples_to_process $\neq []$ **do**
      $\bar{a} =$ pop first element of tuples_to_process
      if Type$(\bar{a}) =$ Null **then** ▷ $\bar{a}$'s type is unknown
        (type_a, universe_a) $=$ IsoType$(\boldsymbol{A}, \bar{a})$
        if |universe_a| $=$ |current_sub| **then** ▷ Sg$(\bar{a})$ is not smaller
          if there is (B, RT, T, U) $\in$ {orbits of tuples in generators} such that type_a $=$ T **then**
            $\gamma =$ the isomorphism from universe_a to U
            if not try_merge_orbits$(\gamma,$ orbits$)$ **then** ▷ see Algorithm 3
              return False
          else ▷ $\bar{a}$'s type is new
            tag_orbit$(\bar{a},$ type_a, universe_a$)$
            add $\bar{a}$ to generators
        else ▷ Sg$(\bar{a})$ is smaller
          if there is (B, RT, T, U) $\in$ orbits$_{|\bar{a}|}$ such that type_a $=$ T **then** ▷ $\bar{a}$'s type is not new
            $\gamma =$ the isomorphism from universe_a to U
            if not try_merge_orbits$(\gamma,$ orbits$)$ **then** ▷ see Algorithm 3
              return False
          else ▷ $\bar{a}$'s type is new
            tag_orbit$(\bar{a},$ type_a, universe_a$)$
            new_current_sub $=$ universe_a
            new_tuples_to_process $=$ sorted$(\bigcup_{k \in \text{spec}}$ universe_a$^{(k)})$
            new_generators $= \{\bar{a}\}$
            ▷ push new node on the stack
            push (new_current_sub, new_tuples_to_process, new_generators) onto stack
            tag_orbit$(\bar{a},$ type, universe$)$
            break
    if tuples_to_process $= []$ **then**
      delete top of the stack
  return True

with isomorphism type iType$(\bar{a})$. If there is no such orbit, then orbit containing $\bar{a}$ is annotated with iType$(\bar{a})$ and iUniv$(\bar{a})$. We also push $(\text{Sg}(\bar{a}), L', \{\bar{a}\})$ on top of the stack (i.e. we move to a new node of the subuniverse tree). If there is an orbit in orbits$_{|\bar{a}|}$ tagged with iType$(\bar{a})$, we proceed in the same way as above by calling try_merge_orbits.

Eventually, if no call to try_merge_orbits returns False, every tuple in $\bigcup_{k \in \text{spec}(R)} A^{(k)}$ is processed, and when the algorithm halts we have computed for each $k \in \text{spec}(R)$

the partition induced by $\approx$ on $A^{(k)}$. Since we ensure that all tuples in the same orbit have the same relType, Corollary 5.1 guarantees that R is qf-definable in $\mathbf{A}$.

Algorithm 3 Merging orbits

```
1  function try_merge_orbits(γ, orbits)
2      for k ∈ spec do
3          for ā ∈ (Dom(γ))^(k) do
4              if orbit(ā) ≠ orbit(γ(ā)) then
5                  (B, RT, T, U) = orbit(ā)
6                  (B', RT', T', U') = orbit(γ(ā))
7                  if RT ≠ RT' then
8                      return False                                  ▷ RelTypes differ
9                  B_merge = B ∪ B'                                  ▷ blocks are merged
10                 RT_merge = RT
11                 if T ≠ Null then                                  ▷ first block was tagged
12                     T_merge = T
13                     U_merge = U
14                 else if T' ≠ Null then                            ▷ second block was tagged
15                     T_merge = T'
16                     U_merge = U'
17                 else                                              ▷ no block is tagged
18                     T_merge = Null
19                     U_merge = Null
20                 delete (B, RT, T, U) and (B', RT', T', U') from orbits_k
21                 add (B_merge, RT_merge, T_merge, U_merge) to orbits_k]
22     return True
```

5.2 Proof of soundness and completeness

To prove soundness and completeness we need some auxiliary lemmas.

Lemma 5.2 *The following properties hold throughout an execution of Algorithm 2 for each $k \in \mathrm{spec}(R)$:*

(i) *The set $\{B : (B, RT, T, U) \in \mathtt{orbits}_k\}$ is a partition of $A^{(k)}$.*

(ii) *For each $(B, RT, T, U) \in \mathtt{orbits}_k$ we have that:*
   (a) *For all $\bar{a}$ in B we have* $\mathrm{relType}(\bar{a}, R_1, \dots, R_l) = RT$.
   (b) *For all $\bar{a}, \bar{b}$ in B we have $\bar{a} \approx \bar{b}$.*
   (c) *If $U \neq \mathtt{Null}$, then there is $\bar{a}_0 \in B$ such that* $\mathrm{iUniv}(\bar{a}_0) = U$.
   (d) *If $T \neq \mathtt{Null}$, then $T = \mathrm{iType}(\bar{a})$ for every $\bar{a}$ in B.*

Proof The properties are obviously true at the initialization of $\mathtt{orbits}_k$. Note that the only instructions of Algorithm 2 that may change $\mathtt{orbits}_k$ are calls to the functions `try_merge_orbits` and `tag_orbit`, and it is easy to see that these functions preserve (i) and (ii). □

In order to make the proofs below more succinct, we say that (at a certain point of an execution of Algorithm 2):

- an orbit (B, RT, T, U) is *tagged* if $T \neq \mathtt{Null}$.

- the tuple $\bar{a}$ has *known type* if the (unique) orbit containing $\bar{a}$ is tagged.

Lemma 5.3 *The following properties hold throughout an execution of Algorithm 2:*

(i) *If the stack is* $[(S_m, L_m, G_m), (S_{m-1}, L_{m-1}, G_{m-1}), \ldots, (S_0, L_0, G_0)]$ *then* $|S_m| < \cdots < |S_0|$.

(ii) *If an entry* (S, L, G) *is removed from the stack, then every tuple in* $\bigcup_{k\in \mathrm{spec}(R)} S^{(k)}$ *has known type.*

(iii) *Suppose the top of the stack is* (S, L, G). *If* $\bar{a}$ *is a tuple with known type such that* $|\mathrm{Sg}(\bar{a})| < |S|$, *then every tuple in* $\bigcup_{k\in \mathrm{spec}(R)} \mathrm{Sg}(\bar{a})^{(k)}$ *has known type.*

(iv) *Suppose the top of the stack is* (S, L, G). *If* $\bar{a}$ *is a tuple with known type such that* $|\mathrm{Sg}(\bar{a})| = |S|$, *then either:*
   (a) *there is* $\bar{g} \in G$ *and* $\bar{g}$ *is in the same orbit as* $\bar{a}$, *or*
   (b) *every tuple in* $\bigcup_{k\in \mathrm{spec}(R)} \mathrm{Sg}(\bar{a})^{(k)}$ *has known type.*

(v) *For all* $k \in \mathrm{spec}(R)$, *and for any* $O = (B, RT, T, U)$ *and* $O' = (B', RT', T', U')$ *in* $\texttt{orbits}_k$ *such that* $T \neq \texttt{Null}$, *we have that*

$$T = T' \iff O = O'.$$

Proof Item (i) is easily seen to hold. We prove (ii). When an element (S, L, G) is pushed onto the stack then L is a list containing all tuples in $\bigcup_{k\in \mathrm{spec}(R)} S^{(k)}$, and (S, L, G) is only removed from the stack when all tuples in T have been processed. It is not hard to see that when a tuple $\bar{a}$ is processed, we find that either:

- the orbit containing it is already tagged , or
- it is tagged by a call to `tag_orbit` or `try_merge_orbits`, unless the call to `try_merge_orbits` returns False (making Algorithm 2 halt and return False).

Let us prove (iii). Suppose the top of the stack is (S, L, G), and let $\bar{a}$ be as in the statement. Note that there are two ways in which the orbit containing $\bar{a}$ could have been tagged: either by a call to `tag_orbit` or by the merging of $\bar{a}$'s orbit with another orbit that was already tagged. Suppose first that at some point the algorithm executed the call $\texttt{tag\_orbit}(\bar{a}, \mathrm{iType}(\bar{a}), \mathrm{iUniv}(\bar{a}))$. The key observation is that then $(\mathrm{Sg}(\bar{a}), L', G')$ has been on the stack. Now, since $|\mathrm{Sg}(\bar{a})| < |S|$, item (i) says that $(\mathrm{Sg}(\bar{a}), L', G')$ is no longer on the stack, and, by (ii), it follows that every tuple from $\mathrm{Sg}(\bar{a})$ has known type. Next, assume that $\bar{a}$'s orbit was not tagged by such a call to `tag_orbit`.Since the only way to introduce a new type in `orbits` is with a call to `tag_orbit`, there must exist a tuple $\bar{b}$ such that:

- $\mathrm{iType}(\bar{b}) = \mathrm{iType}(\bar{a})$, and $\texttt{tag\_orbit}(\bar{b}, \mathrm{iType}(\bar{a}), \mathrm{iUniv}(\bar{b}))$ has been executed at an earlier time, and
- $\bar{a}$'s orbit was tagged by eventually being merged with $\bar{b}$'s orbit.

Using the same reasoning as above we can conclude that every tuple from $\mathrm{Sg}(\bar{b})$ has known type. Finally, observe that through the merging process every tuple from $\mathrm{Sg}(\bar{a})$ ends up in an orbit with a tuple from $\mathrm{Sg}(\bar{b})$, and thus has known type.

We leave (iv) to the reader as the proof is similar to the one of (iii).

To conclude we prove (v). Note that it suffices to check that the calls to `tag_orbit` and `try_merge_orbits` preserve (v). This is clear for `try_merge_orbits`, since this function does not tag orbits with new types. So we focus on the calls to `tag_orbit`; let us begin with the one on line 34 of Algorithm 2. Notice that immediately before this call is executed we check that the type to be introduced does not occur anywhere in `orbits`, and thus (v) is preserved. Finally, assume a call `tag_orbit`$(\bar{a}, \mathrm{iType}(\bar{a}), \mathrm{iUniv}(\bar{a}))$ on line 21. Due to the tests carried out before this call we know that (at the time of this call) the top of the stack is $(\mathrm{Sg}(\bar{a}), L, G)$ and that no tuple in G has the same type as $\bar{a}$. We want to show that no orbit is tagged with $\mathrm{iType}(\bar{a})$. For the sake of contradiction assume there is $\bar{b}$ such that its orbit, say O', has type $\mathrm{iType}(\bar{a})$. So, since no tuple from G is in O', item (iv) says that every tuple from $\mathrm{Sg}(\bar{b})$ has known type. It follows that $\mathrm{Sg}(\bar{b}) \neq A$, since otherwise $\bar{a}$ would have had known type. Hence $(\mathrm{Sg}(\bar{a}), L, G)$ was pushed on the stack by processing a tuple $\bar{g}$ which was found to have a new type. But this is a contradiction, since $\mathrm{Sg}(\bar{b})$ contains an isomorphic copy of $\bar{g}$, and thus would have had known type. □

Proposition 5.4 *Algorithm 2 is sound and complete.*

Proof Note that the algorithm terminates if and only if either:

- a call to `try_merge_orbits` returns false and Algorithm 2 returns false, or
- the stack is empty and Algorithm 2 returns true.

In the former case, it is clear from an inspection of Algorithm 3, that `try_merge_orbits` returns false if and only if there are two orbits tagged with the same isomorphism type and different relTypes. Thus, by Lemma 5.2, there are tuples $\bar{a}, \bar{b}$ such that $\bar{a} \approx \bar{b}$ and $\mathrm{relType}(\bar{a}, R_1, \dots, R_l) \neq \mathrm{relType}(\bar{b}, R_1, \dots, R_l)$. So, Corollary 5.1 says that R is not qf-definable in $\mathbf{A}$.

Suppose next that the algorithm ends due to the stack being empty. Recall that the stack is initialized with $(A, L_0, \emptyset)$, so, by Lemma 5.3.(ii), every tuple in $\bigcup_{k \in \mathrm{spec}(R)} A^{(k)}$ has known type when the algorithm halts. We prove that (3) of Corollary 5.1 holds. Fix $k \in \mathrm{spec}(R)$ and take $\bar{a}, \bar{b} \in A^{(k)}$ such that $\bar{a} \approx \bar{b}$. Now Lemma 5.2 together with Lemma 5.3.(v) guarantee that $\bar{a}$ and $\bar{b}$ are in the same orbit, and thus item (a) of Lemma 5.2 says that $\mathrm{relType}(\bar{a}, R_1, \dots, R_l) = \mathrm{relType}(\bar{b}, R_1, \dots, R_l)$. □

6 Empirical tests

In this section we present an empirical testing of the performance of algorithm IsoType (Algorithm 1) versus the isomorphisms search via CSP, as well as an empirical testing of algorithm QfDefAlg for different input algebras. Our implementations are written in Python 3 under GPL 3 license, source code available in https://github.com/pablogventura/QfDefAlg. All tests were performed using an Intel Xeon E5-2620v3 processor with 12 cores (however, our algorithm does not make use of parallelization) at 2.40GHz, 128 GiB DDR4 RAM 2133MHz. Memory was

never an issue in the tests we ran.

6.1 *Comparing* isoType *to a CSP solver*

In our previous work [2], we used the CSP solver Minion [8] to compute isomorphisms between substructures in our algorithm to decide qf-definability in a relational structure. In this new approach to the problem we use the function IsoType to compute those isomorphisms. In order to compare our previous strategy with the one we present here, we designed the following test. Given an algebra $\boldsymbol{A}$, the test consists in partitioning a random subset R of A^k, first using IsoType, and then via CSP to find relational isomorphisms in the following way: for each $\bar{a} \in R$, we compute the subuniverse it generates $\mathrm{Sg}(a)$ and then construct a relational structure $\boldsymbol{A}^{\mathrm{Rel}}_{\mathrm{Sg}(a)}$ with universe $\mathrm{Sg}(a)$ and, for each operation symbol f in $\boldsymbol{A}$, the relation $F = \mathrm{graph}(f^{\boldsymbol{A}}|_{\mathrm{Sg}(a)})$ (the graph of the restriction of $f^{\boldsymbol{A}}$ to $\mathrm{Sg}(a)$). Note that $\boldsymbol{A} \simeq \boldsymbol{B}$ if and only if $\boldsymbol{A}^{\mathrm{Rel}} \simeq \boldsymbol{B}^{\mathrm{Rel}}$. This allows us to use CSP to search for isomorphisms between $\boldsymbol{A}^{\mathrm{Rel}}_{\mathrm{Sg}(a)}$ and $\boldsymbol{A}^{\mathrm{Rel}}_{\mathrm{Sg}(b)}$ that extend the map $\bar{a} \to \bar{b}$. We compare running times of this last strategy to one of the first one.

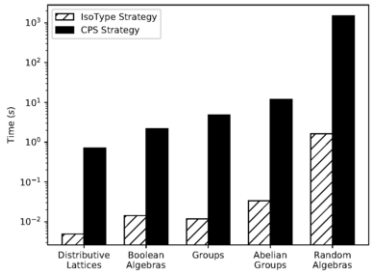

Figure 2. Time to compute the isomorphism types

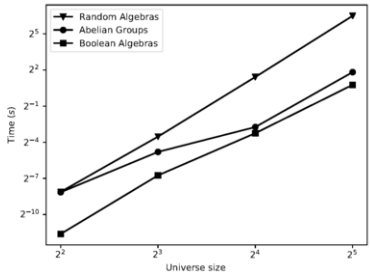

Figure 3. Time to decide definability

Since there is no standard set of tests for these problems, we used five families of algebras: distributive lattices, boolean algebras, groups, abelian groups and finally random algebras. These latter were generated by choosing random operations over a fixed universe. For each family we ran the test on 300 algebras and we took the median of the wall time samples. In figure 2 we can see how the IsoType strategy is faster; the reason behind this is that IsoType stores the isomorphism type instead of computing it every time, as CSP does. It is interesting to note how the amount of time increases as the number of subisomorphisms decreases, where our strategy based on inner symmetries is less efficient.

6.2 *Assessing the performance of QfDefAlg*

Now we present testings of algorithm 2 to decide definability, varying both the size of universes and the families of algebras. In these tests we used universes of

cardinality 4, 8, 16 and 32. In all cases the target relation was the binary relation consisting of all pairs from the domain of the algebra. Random algebras are endowed with a binary and a ternary operation. Abelian groups are obtained as products of cyclic groups $\mathbb{Z}_k$. In Figure 3 we can see how the algorithm takes advantage of inner symmetries of the algebras; the phenomenon of increasing time for decreasing number of subisomorphisms is again observed, as in Figure 2. It is interesting to observe the behaviour of the abelian groups curve. Since all groups in the test are direct products with factors in $\{\mathbb{Z}_2, \mathbb{Z}_4\}$ (e.g., the 16-element one is $\mathbb{Z}_2 \times \mathbb{Z}_2 \times \mathbb{Z}_4$ and the 32-element one is $\mathbb{Z}_2 \times \mathbb{Z}_4 \times \mathbb{Z}_4$), the fact that the larger groups have repeated factors increases the number of subisomorphisms, and thus our algorithm has a better performance.

7 Conclusions and future work

We have presented an algorithm that decides the definability problem by quantifier-free first-order formulas over a purely functional language. This algorithm relies on the semantic characterization of quantifier-free definability given in [6]. We showed that this problem is coNP-complete, which is the same complexity we obtained on the relational case [1]. Prior to this we developed an algorithm that gives a characterization of the isomorphism type of a tuple $\bar{a}$ in the given structure (in particular, the subuniverse it generates is obtained), and showed, by means of empirical tests, that this task is performed very efficiently.

Our definability-decision algorithm partitions the set of all k-tuples from the universe with $k \in \text{spec}(R)$, and has to exhaust the search space to give a positive answer. Therefore, the execution time of our algorithm depends exponentially on the parameter k. Our empirical tests confirm the exponential dependence on k, and give us an idea of the impact of the cardinality of the input structure and the family of algebras on which we performed the tests. They also confirm the hypothesis that our algorithm, which is based on the existence of symmetries, would perform better on models with a large amount of subisomorphisms, as we observed in the case of relational models; see [2]. We may add that the strategy used in our previous work [2] on quantifier-free definability for relational structures, based on the search of isomorphisms using CSP, proves much less efficient when applied to purely functional structures.

There are several possible lines for future research. Due to the lack of standard testings to decide definability, the empirical tests we presented are preliminary, and more evaluation is needed. We hope to extend the design of models to reach a broader class of algebras, and to be able to introduce control variables, as we extend our approach and tool to other fragments of first-order logic. It would also be natural to study definability over finite classes of models instead of a single structure. Lastly, it would be useful to find classes of algebras in which the problem is well-conditioned and for which polynomial time algorithms exist.

References

[1] C. Areces, M. Campercholi, D. Penazzi, and P. Ventura. The complexity of definability by open first-order formulas. *Submitted, arXiv:1904.04637*, 2019.

[2] C. Areces, M. Campercholi, and P. Ventura. Deciding open definability via subisomorphisms. In *Logic, Language, Information, and Computation - 25th International Workshop, WoLLIC 2018, Bogota, Colombia, July 24-27, 2018, Proceedings*, pages 91–105, 2018.

[3] C. Areces, S. Figueira, and D. Gorín. Using logic in the generation of referring expressions. In S. Pogodalla and J. Prost, editors, *Proceedings of the 6th International Conference on Logical Aspects of Computational Linguistics (LACL 2011)*, volume 6736 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2011.

[4] C. Areces, A. Koller, and K. Striegnitz. Referring expressions as formulas of description logic. In *Proceedings of the Fifth International Natural Language Generation Conference (INLG'08)*, pages 42–49. Association for Computational Linguistics, 2008.

[5] M. Arenas and G. Diaz. The exact complexity of the first-order logic definability problem. *ACM Transactions on Database Systems*, 41(2):13:1–13:14, 2016.

[6] M. Campercholi and D. Vaggione. Semantical conditions for the definability of functions and relations. *Algebra universalis*, 76(1):71–98, Sep 2016.

[7] H. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Springer-Verlag, 1994.

[8] I. Gent, Ch. Jefferson, and I. Miguel. MINION: a fast, scalable, constraint solver. In *Proceedings of the 17th European Conference on Artificial Intelligence (ECAI 2006)*, pages 98–102. IOS Press, 2006.

[9] D. Kavvadias and M. Sideri. The inverse satisfiability problem. *SIAM Journal on Computing*, 28(1):152–163, January 1998.

[10] E. Krahmer and K. van Deemter. Computational generation of referring expressions: A survey. *Computational Linguistics*, 38(1):173–218, 2012.

[11] Q. T. Tran, C.Y. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the 35th SIGMOD international conference on Management of data - SIGMOD '09*. ACM Press, 2009.

[12] R. Willard. Testing expressibility is hard. In D. Cohen, editor, *Principles and Practice of Constraint Programming – CP 2010*, pages 9–23, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.