An infinite hierarchy in a class of polynomial-time program schemes

Richard L. Gault*,
Oxford University Computing Laboratory,
Wolfson Building, Parks Road, Oxford OX1 3QD, U.K.

Iain A. Stewart*,
Department of Computer Science, University of Durham,
Science Labs, South Road, Durham DH1 3LE, U.K.

Abstract

We define a class of program schemes RFDPS constructed around notions of forall-loops, repeat-loops, arrays and if-then-else instructions, and which take finite structures as inputs, and we examine the class of problems, denoted RFDPS also, accepted by such program schemes. The class of program schemes RFDPS is a logic, in Gurevich's sense, in that: every program scheme accepts an isomorphism-closed class of finite structures; we can recursively check whether a given finite structure is accepted by a given program scheme; and we can recursively enumerate the program schemes of RFDPS. We show that the class of problems RFDPS properly contains the class of problems definable in inflationary fixed-point logic (for example, the well-known problem *Parity* is in RFDPS) and that there is a strict, infinite hierarchy of classes of problems within RFDPS (the union of which is RFDPS) parameterized by the depth of nesting of forall-loops in our program schemes. This is the first strict, infinite hierarchy in any polynomial-time logic properly extending inflationary fixed-point logic (with the property that the union of the classes in the hierarchy consists of all problems definable in the logic). The fact that there are problems (like *Parity*) in RFDPS which cannot be defined in many of the more traditional logics of finite model theory (which often have zero-one laws) essentially means that existing tools, techniques and logical hierarchy results are of limited use to us.

1 Introduction

One of the central open problems of finite model theory is whether there is a logic capturing **P**. That is, does there exist a logic whose sentences define exactly the problems (the encodings of which are) recognizable by polynomial-time Turing machines? Of course, in order to make sense of this question, one needs to say exactly what one

*Supported by EPSRC Grant GR/M 12933. Most of the research in this paper was completed whilst the authors were at the University of Leicester.

means by a 'logic'. This has been done by Gurevich [10] who formulated a very liberal definition which encompasses many different 'traditional' logics as well as a variety of computational models. Over the years, a number of contenders have been suggested as logics which might capture **P** but so far all such logics, whilst only defining problems in **P**, have subsequently been shown not to have the expressive power to define every problem in **P**. Perhaps the best-known contender was *inflationary fixed-point logic with counting* (see, for example, [5]). Immerman had suggested that this logic might capture **P**, but this was later shown not to be the case by Cai, Fürer and Immerman [4].

In this paper we introduce a new logic (in the sense of Gurevich) which defines only problems in **P**. Whilst our logic is strictly more expressive than, for example, inflationary fixed-point logic and can define problems such as *Parity* (the problem consisting of all finite structures of even size over the empty signature), there are, however, problems in **P** which are not definable in our logic. This comes as no surprise to us as (intuitively) our logic lacks the sophistication we feel any such logic must have were it to capture **P**. In any case, it is not our real aim here to develop a logic which might capture **P** (though we feel that our logic might provide a stepping-off point in the search for such a logic, as we mention in the Conclusion): our primary motivation for introducing our logic is because such a logic arises naturally within our ongoing systematic study of the expressive power of classes of program schemes. Broadly speaking, *program schemes* are models of computation which are amenable to logical analysis yet closer to the general notion of programs than logical formulae are. Program schemes were extensively studied in the seventies, without much regard being paid to an analysis of resources, before a closer complexity analysis was undertaken in, mainly, the eighties. There are connections between program schemes and logics of programs, especially dynamic logic. (The reader is referred to [2] for references relating to the research mentioned in the preceding two sentences.)

We define our program schemes around 'high-level' programming constructs such as arrays, while-loops, assignments, non-determinism, and so on, but so that the input comes in the form of a finite structure and, in general, there is no access to a linear ordering of the elements of the input structure. The program schemes defined in [16, 18] all involve arrays, while-loops and non-determinism. Allowing unrestricted access to arrays enables one to accept **PSPACE**-complete problems, whilst by restricting access to arrays (to be, in a sense, 'write-once'), one can limit oneself to accept only problems in **NP** (although there is still sufficient power to accept **NP**-complete problems). Furthermore, classes of program schemes were defined in [2, 17] whereby every problem accepted by such a program scheme is in **P** and, additionally, there are such program schemes accepting **P**-complete problems. These program schemes involve while-loops, a stack and non-determinism. So as to emphasise that these models of computation are not given an ordering on the elements of an input structure, amongst the results in the aforementioned papers are that the class of problems accepted by any of the above classes of program schemes has a zero-one law (but, interestingly, not necessarily because the problems can be defined in bounded-variable infinitary logic as is often the case in finite model theory). On ordered structures, our classes of program schemes capture the complexity classes **P**, **NP** and **PSPACE** as appropriate (see also [15]).

In this paper, we introduce a class of program schemes RFDPS based on ar-

rays, if-then-else instructions and forall-loops, where our forall-loops result in parallel executions of a portion of code, with one execution for each element of the input structure. So as to provide a means for iteration, we allow portions of code to be repeatedly executed n times, where n is the size of the input structure. The class of program schemes RFDPS arose through our efforts to replace the notion of a while-loop in earlier classes of program schemes with one of a forall-loop. Note that unlike the program schemes mentioned in the previous paragraph, our program schemes are deterministic (and every problem accepted by such a program scheme is in **P**).

Another 'polynomial-time' model of computation has recently been examined by Blass, Gurevich and Shelah[1]. In [3], Blass, Gurevich and Shelah introduced a model of computation $\tilde{\text{C}}$PTime, *Choiceless Polynomial-Time*, a program $(\rho, p(n), q(n))$ of which is an adapted *Abstract State Machine* ρ (see [11, 12]) augmented with two polynomial bounds, $p(n)$ and $q(n)$, with $p(n)$ bounding the length of any run of the machine on any input and $q(n)$ bounding the number of 'parallel executions' in one of their forall-loops, where these polynomial bounds are in terms of the size n of the finite input structure upon which the program works. Although such a program takes a finite structure (over some relational signature) as input, it treats the elements of this finite structure as atoms and has the potential to build certain sets over these atoms and use these sets as new 'elements' in its 'computational domain'. Consequently, without restricting the run-time and the number of parallel executions, the program would have the capacity to build a computational domain of arbitrary size; indeed, it is not difficult to show that such an unrestricted program can simulate an arbitrary Turing machine. The instructions, or rules in the terminology of [3], of the programs of $\tilde{\text{C}}$PTime have similarities with those of the program schemes of this paper. For example: there are dynamic function symbols and assignments via update rules, whereas our program schemes have arrays and assignment-blocks; there are conditional rules, whereas our program schemes have if-then-fi-blocks; and there are do-forall rules, whereas our program schemes have forall-do-od-blocks. However, there are a number of important differences between the computational model of Blass, Gurevich and Shelah and ours, including the following. Their computational domain fluctuates, whereas ours is fixed and is always the domain of the input structure. Viewed as a logic, $\tilde{\text{C}}$PTime is three-valued (a program may accept, reject or neither accept nor reject), whereas our program schemes always either accept or reject. A program of $\tilde{\text{C}}$PTime has no access to the cardinality of the input structure, and the problem *Parity* cannot be accepted by a program of $\tilde{\text{C}}$PTime (furthermore, it has been reported in [3] that Shelah has shown that $\tilde{\text{C}}$PTime has a zero-one law), whereas our program schemes have access to the size of the input structure and there is such a program scheme accepting *Parity*. In order to force the abstract state machine to accept polynomial-time solvable problems, the polynomial bounds $p(n)$ and $q(n)$ must be imposed from without, whereas no such bounds need be imposed upon our program schemes: our program schemes naturally accept only polynomial-time solvable problems.

The motivation for the research in [3] was the search for an answer to the question, stated earlier, of whether there is a logic capturing **P**. In turn, this question

[1]Actually, although their work was published before this paper was written, the actual research was undertaken simultaneously and independently. The writing of our paper was delayed due to the writing of the Ph.D. thesis of Gault within which the research presented here is included.

has motivated a search for logics capturing an increasing sub-class of the class of polynomial-time solvable problems. The main results of [3] are that the class of problems accepted by the progams of $\tilde{\text{C}}$PTime properly contains the class of problems accepted by Abiteboul and Vianu's class of *polynomial-time relational machines* [1] but that there are polynomial-time solvable problems, in particular *Parity* and the problem *Bipartite Matching* (consisting of those bipartite undirected graphs whose two sets in the partition have equal size for which there exists a perfect matching), that are not accepted by any program of $\tilde{\text{C}}$PTime. In fact, it is also shown in [3] that *Bipartite Matching* is not accepted by any program of $\tilde{\text{C}}\text{PTime}^+$, an extension of $\tilde{\text{C}}$PTime which allows access to a constant fixed at the size of the input structure (however, *Parity* is accepted by a program of $\tilde{\text{C}}\text{PTime}^+$). It is also claimed in [3] that the class of problems accepted by the programs of $\tilde{\text{C}}$PTime includes any problem definable in any other 'polynomial-time logic' in the literature (the authors presumably mean only 'natural polynomial-time logics' and not augmentations of such by, for example, counting quantifiers or Lindström quantifiers).

Our results are of a somewhat different flavour to those of [3] and, in a sense, are more refined. We obtain a strong result which provides limitations on the problems accepted by our program schemes, and we use this result to obtain a strict, infinite hierarchy of classes of problems within the class of problems accepted by the program schemes of RFDPS. These classes are parameterized by the depth of nesting of forall-loops allowed in the defining program schemes, and the union of these classes is the class of problems accepted by the program schemes of RFDPS. Consequently, each class of problems in the hierarchy is definable by a logic in Gurevich's sense. To our knowledge, this is the first strict, infinite hierarchy in a polynomial-time logic properly extending inflationary fixed-point logic (with the property that the union of the classes of the hierarchy consists of the class of problems definable in the polynomial-time logic). Our results are obtained by a direct analysis of computations of our program schemes. Note that the existing hierarchy theorems of finite model theory, such as those in [7, 8, 9], are of no use to us here given that all of these hierarchy results are for explicit fragments of bounded-variable infinitary logic (which has a zero-one law), whereas our computational model is, first, not defined in terms of traditional logics and, second, is complicated by its ability to accept problems not having a zero-one law.

Like the Choiceless Polynomial-Time model of Blass, Gurevich and Shelah, our program schemes are different from other (polynomial-time) models of computation more prevalent in database theory, such as the relational machines of Abiteboul and Vianu [1], the extension of inflationary fixed-point logic with a symmetry-based choice operator proposed by Gire and Hoang [6] and the extension of first-order logic with for-loops proposed by Neven, Otto, Tyszkiewicz and Van den Bussche [14]. The models of computation proposed by these researchers (and others) allow the construction of whole relations as an atomic operation, whereas our construction of relations (stored in arrays) is, in a sense, 'one element at a time'. Some of these models are more expressive than our class of program schemes but, unlike our class of program schemes, no hierarchy results have been established. Hence, we do not discuss these models further here (although the reader is referred to our comments in the Conclusion).

This paper is organized as follows. In Section 2, we detail the basic definitions from finite model theory required throughout, and in Section 3 we define our class of

program schemes RFDPS and prove some lower bound results for RFDPS. In Section 4, we give a more formal semantics for our class of program schemes before proving some limitations of the program schemes of RFDPS in Section 5. We present our conclusions in Section 6.

2 Basic notions

Throughout, a *signature* σ is a tuple $\langle R_1, \ldots, R_r, C_1, \ldots, C_c \rangle$, where each R_i is a relation symbol, of arity a_i, and each C_j is a constant symbol: in the case that σ consists only of relation symbols, we say that σ is *relational*. *First-order logic over some signature* σ, $\mathrm{FO}(\sigma)$, consists of those formulae built from atomic formulae over σ using $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$; and $\mathrm{FO} = \cup\{\mathrm{FO}(\sigma) : \sigma \text{ is some signature}\}$.

A *finite structure* $\mathcal{A}$ *over the signature* σ, or σ*-structure*, consists of a finite *universe* or *domain* $|\mathcal{A}|$, together with a relation $R_i^{\mathcal{A}}$ of arity a_i for every relation symbol R_i of σ, and a constant $C_j^{\mathcal{A}} \in |\mathcal{A}|$ for every constant symbol C_j (by an abuse of notation, we often do not distinguish between constants or relations and constant or relation symbols). A finite structure $\mathcal{A}$ whose domain has size n is said to have *size* n, and we denote the size of $\mathcal{A}$ by $|\mathcal{A}|$ also (this does not cause confusion). The class of all finite structures over the signature σ is denoted $\mathrm{STRUCT}(\sigma)$. A *problem* over some signature σ consists of a subset of $\mathrm{STRUCT}(\sigma)$ which is closed under isomorphism. Clearly, we can consider the classes of problems definable by the sentences of FO. We denote this class of problems by FO also and do likewise with other logics and classes of problems.

Let $\varphi(\mathbf{x}, \mathbf{y}) \in \mathrm{FO}(\sigma \cup \langle R \rangle)$, for some signature σ and some relation symbol R, of arity k, say, not in σ, be such that the free variables of φ are those of the k-tuple $\mathbf{x}$ and the m-tuple $\mathbf{y}$, where $m \geq 0$ (with all variables distinct). Then

$$\mathrm{IFP}[\lambda \mathbf{x}, R, \varphi(\mathbf{x}, \mathbf{y}, R)]$$

denotes the inflationary fixed-point relation of $\varphi(\mathbf{x}, \mathbf{y}, R)$ with respect to R and $\mathbf{x}$. That is, for any σ-structure $\mathcal{A}$ and for any $\mathbf{v} \in |\mathcal{A}|^m$,

$$\mathrm{IFP}[\lambda \mathbf{x}, R, \varphi^{\mathcal{A}}(\mathbf{x}, \mathbf{v}, R)] = \bigcup_{i=0}^{\infty} R_i^{\mathcal{A}},$$

where $R_0^{\mathcal{A}}$ is the empty relation and where for each $i \geq 0$,

$$R_{i+1}^{\mathcal{A}} = \{\mathbf{u} \in |\mathcal{A}|^k : \mathcal{A} \models R_i^{\mathcal{A}}(\mathbf{u}) \vee \varphi(\mathbf{u}, \mathbf{v}, R_i^{\mathcal{A}})\}.$$

Inflationary fixed-point logic, denoted $(\pm\mathrm{IFP})^*[\mathrm{FO}]$, is the closure of first-order logic with the operator IFP.

We denote by $\mathcal{L}_{\infty\omega}$ the infinitary logic built as is first-order logic except that we allow conjunctions and disjunctions of arbitrary (and not just finite) sets of formulae. Obviously, any problem can be defined in $\mathcal{L}_{\infty\omega}$. So, let us turn to the bounded-variable fragment. Let $\mathcal{L}_{\infty\omega}^d$ be the fragment of $\mathcal{L}_{\infty\omega}$ where the only variables we allow, free or bound, are $x_1, x_2, \ldots, x_d$; and define *bounded-variable infinitary logic*, $\mathcal{L}_{\infty\omega}^{\omega}$, as $\cup_{d=1}^{\infty} \mathcal{L}_{\infty\omega}^d$. Let $\mathcal{A}$ and $\mathcal{B}$ be σ-structures, for some σ; let e be such that

$0 \leq e \leq d$; and let $\mathbf{u} \in |\mathcal{A}|^e$ and $\mathbf{v} \in |\mathcal{B}|^e$. If for all $\varphi \in \mathcal{L}^d_{\infty\omega}$ with free variables $x_1, x_2, \ldots, x_e$,

$$\mathcal{A} \models \varphi(u_1, u_2, \ldots, u_e) \Leftrightarrow \mathcal{B} \models \varphi(v_1, v_2, \ldots, v_e)$$

then we write

$$(\mathcal{A}, u_1, u_2, \ldots, u_e) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, v_1, v_2, \ldots, v_e).$$

There is a well-known game-theoretic characterization of definability in $\mathcal{L}^d_{\infty\omega}$. Two players, *Spoiler* (who is male) and *Duplicator* (who is female), play the following game on two structures $\mathcal{A}$ and $\mathcal{B}$, over the same signature, where the game involves d pairs of pebbles $\{(p_1, q_1), (p_2, q_2), \ldots, (p_d, q_d)\}$. Each move of the game consists of Spoiler placing some pebble p_i or q_i on some element of $|\mathcal{A}|$ or $|\mathcal{B}|$, respectively, with Duplicator replying by placing the other pebble of the pair on some element of the other domain as appropriate (note that a pebble can be removed from one element of a structure and placed on another element of that structure in a move of the game). Consider some play of the game which is a (possibly infinite) set of moves. If after some move the map $|\mathcal{A}| \to |\mathcal{B}|$ given by $\{p_i \mapsto q_i : \text{ pebble } p_i \text{ is in play}\}$ does not induce a partial isomorphism from $\mathcal{A}$ to $\mathcal{B}$ then Spoiler wins the play of the game. If there is no such move then Duplicator wins (when the play is necessarily of infinite length). Duplicator has a *winning strategy* on $\mathcal{A}$ and $\mathcal{B}$ if she has a strategy by which she can win every play of the game, i.e., a strategy by which she can continually maintain a partial isomorphism between the pebbled elements no matter what Spoiler does. Let $\mathbf{u} \in |\mathcal{A}|^e$ and $\mathbf{v} \in |\mathcal{B}|^e$, where $0 \leq e \leq d$. If Duplicator has a winning strategy in the above d-pebble game when the pebbles $p_1, p_2, \ldots, p_e$ start on $u_1, u_2, \ldots, u_e$ and the pebbles $q_1, q_2, \ldots, q_e$ start on $v_1, v_2, \ldots, v_e$ then we say that Duplicator *wins the d-pebble game on* $(\mathcal{A}, \mathbf{u})$ *and* $(\mathcal{B}, \mathbf{v})$. The game-theoretic characterization of definability in $\mathcal{L}^d_{\infty\omega}$, due to Barwise, Immerman and Poizat (see [5]), can be stated as follows.

Theorem 1 *Let $\mathcal{A}$ and $\mathcal{B}$ be two structures over the same signature and let $\mathbf{u} \in |\mathcal{A}|^e$ and $\mathbf{v} \in |\mathcal{B}|^e$, where $0 \leq e \leq d$. The following are equivalent.*

(i) $(\mathcal{A}, u_1, u_2, \ldots, u_e) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, v_1, v_2, \ldots, v_e)$.

(ii) Duplicator wins the d-pebble game on $(\mathcal{A}, \mathbf{u})$ and $(\mathcal{B}, \mathbf{v})$. □

We do not actually use the above characterization result in its full generality, only the notion of Duplicator winning the d-pebble game on $(\mathcal{A}, \mathbf{u})$ and $(\mathcal{B}, \mathbf{v})$. However, it is useful for the reader to know the logical significance of the Duplicator winning this game. We refer the reader to [5] for more details on the above definitions and results, and for more results involving the above logics and notions.

3 A class of program schemes

We now define a class of program schemes based around a notion of a forall-loop, rather than a while-loop (as was the case in [2, 16, 17, 18]), and where we disallow non-determinism. We compensate for this lack of non-determinism by interpreting our for-loops as the parallel execution(s) of a portion of code with one execution for each element of the input structure.

Definition 2 (*The syntax of our program schemes.*) A program scheme ρ of RFDPS: is over a signature σ; involves a finite set of *variables* $\{x_1, x_2, \ldots, x_k\}$, for some $k \geq 1$; and involves a finite set of *array symbols* $\{A_1, A_2, \ldots, A_g\}$, for some $g \geq 0$, where the array symbol A_i has an associated *arity* $a_i \geq 1$.

The set of *terms* consists of: the variables $\{x_1, x_2, \ldots, x_k\}$; the constant symbols of σ and the constant symbols 0 and max, which never appear in any signature; and the *array terms* $\{A_i[\tau_1, \tau_2, \ldots, \tau_{a_i}] :$ each τ_j is a variable or a constant symbol, and $1 \leq i \leq g\}$ (note that we do not allow array terms to be nested).

A *program scheme* ρ consists of a finite sequence of *blocks* of instructions, the *constituent blocks*, sandwiched between the *input-instruction*, INPUT$(x_1, x_2, \ldots, x_k)$, and the *output-instruction*, OUTPUT$(x_1, x_2, \ldots, x_k)$, where each block is as follows.

- An *assignment-block* α is simply an instruction of the form

  $\tau := \tau'$ *assignment-instruction*

  where τ is a variable or an array term and τ' is a variable, a constant symbol or an array term. The *scope* of α is the actual assignment-instruction constituting the block.

- An *if-then-fi-block* α is a sequence of instructions of the form

IF φ THEN	*if-instruction*
α_1	*block of instructions*
α_2	*block of instructions*
$\ldots$	
α_l	*block of instructions*
FI	*fi-instruction*

  for some $l \geq 1$, where φ is a quantifier-free first-order formula over $\sigma \cup \{0, max\}$ whose free variables come from $\{x_1, x_2, \ldots, x_k\}$ (note that we do not allow array terms in φ). The *scope* of α is the union of the if-instruction, the fi-instruction and the scopes of the blocks $\alpha_1, \alpha_2, \ldots, \alpha_l$.

- A *repeat-do-od-block* α is a sequence of instructions of the form

REPEAT DO	*repeat-do-instruction*
α_1	*block of instructions*
α_2	*block of instructions*
$\ldots$	
α_l	*block of instructions*
OD	*repeat-od-instruction*

  for some $l \geq 1$. The *scope* of α is the union of the repeat-do-instruction, the repeat-od-instruction and the scopes of the blocks $\alpha_1, \alpha_2, \ldots, \alpha_l$.

- A *forall-do-od-block* α is a sequence of instructions of the form

FORALL x_p WITH A_i^j DO	*forall-do-instruction*
α_1	*block of instructions*
α_2	*block of instructions*
...	
α_l	*block of instructions*
OD	*forall-od-instruction*

for some $l \geq 1$, where $1 \leq p \leq k$, $1 \leq i \leq g$ and $1 \leq j \leq a_i$. The variable x_p is the *control variable* and the array symbol A_i is the *control array symbol* of the forall-do-od-block. We say that the control variable x_p is *active* in A_i in α at index j (we shall define forall-do-od-blocks where the control variable is inactive in a moment). The *scope* of α is the union of the forall-do-instruction, the forall-od-instruction and the scopes of the blocks $\alpha_1, \alpha_2, \ldots, \alpha_l$. Furthermore, there are some additional constraints on α.

- There must exist at least one assignment-instruction in the scope of α where the term on the left-hand side of the assignment is an array term involving A_i.
- Any array term $A_i[\tau_1, \tau_2, \ldots, \tau_{a_i}]$ appearing in any assignment-instruction (on the left or on the right) in the scope of α must be such that the term τ_j is x_p (the control variable).
- No array symbol apart from the array control symbol may appear in a term on the left-hand side of any assignment-instruction in the scope of α.
- The control variable x_p must not appear (as a solitary variable) on the left-hand side of any assignment-instruction in the scope of α.
- Any other forall-do-od-block (with either an active or an inactive control variable) whose instructions are in the scope of α must not have x_p as its control variable.

- We also allow forall-do-od-blocks α of the form

FORALL x_p DO	*forall-do-instruction*
α_1	*block of instructions*
α_2	*block of instructions*
...	
α_l	*block of instructions*
OD	*forall-od-instruction*

for some $l \geq 1$, where $1 \leq p \leq k$. The control variable x_p is said to be *inactive* in α. The *scope* of α is the union of the forall-do-instruction, the forall-od-instruction and the scopes of the blocks $\alpha_1, \alpha_2, \ldots, \alpha_l$. The constraints on α are as follows.

- No array term appears on the left-hand side of any assignment-instruction in the scope of α.
- The control variable x_p must not appear (as a solitary variable) on the left-hand side of any assignment-instruction in the scope of α.

– Any other forall-do-od-block (with either an active or an inactive control variable) whose instructions are in the scope of α must not have x_p as its control variable.

The *scope* of the program scheme ρ consists of the union of the input-instruction, the output-instruction and the scopes of the blocks forming ρ. A block *appears* in ρ if it is used somewhere in the iterative process of building ρ; and we say that a block α *appears* in the scope of another block α' if the instructions in the scope of α are in the scope of α'. The *depth of nesting* of an instruction in the scope of ρ or of a block appearing in ρ is the number of forall-do-od-blocks in whose scope the instruction or the block appears; and the *depth of nesting* of ρ is the maximum of the depth of nesting of all instructions of ρ. □

We can clearly write any program scheme as a sequence of instructions, starting from the input-instruction and ending in the output-instruction, so that these instructions are numbered consecutively.

Our name for our class of program schemes, RFDPS, is an acronym for 'Repeat Forall Deterministic Program Schemes'.

Now is an apposite time to make some observations as regards control variables and control array symbols. Suppose that a forall-do-od-block α appears in the scope of a forall-do-od-block β where the control variable of α is x_i and the control variable of β is x_j (note that necessarily $i \neq j$).

- If x_i is active in α then x_j is active in β and the control array symbols of α and β are identical.
- If x_j is inactive in β then x_i is inactive in α.

Two forall-do-od-blocks can have the same control variable but if they do then neither will appear in the scope of the other. As to why we impose the restrictions that we do in Definition 2 will become clearer when we define the semantics of our program schemes (essentially, our restrictions mean that we can treat control array variables as 'partitioned shared memory').

Let us now explain how our program schemes compute (a more rigorous semantics will be forthcoming in the next section).

Definition 3 (*How our program schemes compute.*) Any program scheme ρ, as in Definition 2, takes a σ-structure $\mathcal{A}$, of size n, say, as input. The variables and array elements all take values from $|\mathcal{A}|$ with array elements indexed by tuples of elements of the input structure (where the length of the tuple is the arity of the array symbol). The program scheme ρ computes on input $\mathcal{A}$ in the obvious way except with the following provisos.

- Prior to the computation, the constant symbols 0 and max are given arbitrary distinct values from $|\mathcal{A}|$. Initially, all variables and array elements are made equal to 0.
- A repeat-do-od-block α of the form

REPEAT DO	*repeat-do-instruction*
α_1	*block of instructions*
α_2	*block of instructions*
$\ldots$	
α_l	*block of instructions*
OD	*repeat-od-instruction*

iteratively executes the blocks of instructions $\alpha_1; \alpha_2; \ldots; \alpha_l$ in this order exactly n times. The effect is precisely that of writing

$$\alpha_1; \alpha_2; \ldots; \alpha_l; \alpha_1; \alpha_2; \ldots; \alpha_l; \ldots; \alpha_1; \alpha_2; \ldots; \alpha_l \text{ (n repetitions)}$$

which, of course, is impossible within our syntax as the value of n depends upon the input structure.

- A forall-do-od-block α of the form

FORALL x_p WITH A_i^j DO	*forall-do-instruction*
α_1	*block of instructions*
α_2	*block of instructions*
$\ldots$	
α_l	*block of instructions*
OD	*forall-od-instruction*

causes a 'multi-way split' in the computation so that n 'child processes' are set off in parallel, each executing the blocks of instructions $\alpha_1; \alpha_2; \ldots; \alpha_l$ and each with its own 'local copy' of the variables of ρ (thus, the processes cannot use these variables to somehow communicate with each other). The only difference between the processes is that the variable x_p takes a different value in each; that is, for each $u \in |\mathcal{A}|$, there is exactly one process in which the variable x_p has the value u. Whilst the value of x_p does not change throughout any process (as a consequence of our syntactic constraints), the values of other variables might change within a process (so that the same variable has a different value in two different processes).

However, the arrays are *not* local to the individual processes: each process makes use of exactly the same arrays. Our syntactic conditions on forall-do-od-blocks ensure that two different processes never have access to the same array element of the array A_i (which is the only array whose values may change in any process). Exclusive access is ensured because the only array terms involving A_i allowed in assignment-instructions in the scope of α are those where the control variable x_p is the jth array index. Hence, in the child process corresponding to the control variable x_p having the value $u \in |\mathcal{A}|$, the only elements of A_i to which this process has (either read or write) access are elements of the form $A_i[u_1, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_{a_i}]$, where $u_1, \ldots, u_{j-1}, u_{j+1}, \ldots, u_{a_i} \in |\mathcal{A}|$. Note that two processes might have read-access to the same array elements of other arrays but no process has write-access to these arrays. The arrays can be considered as a form of 'partitioned shared-memory' where access is controlled so that non-deterministic behaviour, such as races, does not occur (a *race* occurs

when two computational processes have access to the same variable and different relative execution speeds of the processes can result in different values for the variable).

When all child processes have reached the forall-od-instruction, they terminate and the results of these child processes are registered as follows. Call those variables appearing on the left-hand side of an assignment-instruction in the scope of α the *local variables* (note that there may be no local variables). The main computation resumes at the instruction of ρ following the forall-od-instruction. At this point, the value of x_p is set to max if, in every child process, the values of the local variables of α are all set at max on termination of the process; otherwise the value of x_p is set at 0. If α has no local variables then x_p is always set to max. The values of all variables apart from x_p now take their original values held immediately prior to the execution of the forall-do-od-block. The values of the arrays stay as they are when all child processes have finished (as we have described above, these array values are consistent). So, execution of a forall-do-od-block: leaves all the values of variables, except possibly the control variable, unchanged; has an effect which is signalled by the resulting value of the control variable; and might change the values of some of the control array elements but no other array elements. A forall-do-od-block in which the control variable is inactive computes in exactly the same way as we have just described (obviously, the situation is more straight-forward as no value of any array element is changed due to the execution of such a forall-do-od-block).

The structure $\mathcal{A}$ is *accepted* by ρ if, and only if, there exist distinct values for 0 and max for which the computation of ρ on input $\mathcal{A}$ reaches the output-instruction with all variables set at max. □

Essentially, what we are doing is 'building in' two distinct constants 0 and max to our program schemes so that we might use these constants for initialization and acceptance purposes. However, our definition of how we build the constants 0 and max into our program schemes is slightly different to how we usually build constants and relations into logics in finite model theory. For example, any problem in $\mathbf{P}$ can be defined in inflationary fixed-point logic in the presence of a built-in successor relation (a result due to Immerman and Vardi: see [5]). That is, for any problem Ω in $\mathbf{P}$, over some signature σ, there is a sentence φ of inflationary fixed-point logic over the signature $\sigma \cup \{succ, 0, max\}$, where $succ$ is a binary relation and 0 and max are constant symbols not in σ, such that for every σ-structure $\mathcal{A}$:

(a) if $\mathcal{A} \in \Omega$ then $(\mathcal{A}, succ^{\mathcal{A}}, 0^{\mathcal{A}}, max^{\mathcal{A}}) \models \varphi$, for all successor relations $succ^{\mathcal{A}}$ over $\mathcal{A}$ with minimum $0^{\mathcal{A}}$ and maximum $max^{\mathcal{A}}$; and

(b) if $\mathcal{A} \notin \Omega$ then $(\mathcal{A}, succ^{\mathcal{A}}, 0^{\mathcal{A}}, max^{\mathcal{A}}) \not\models \varphi$, for all successor relations $succ^{\mathcal{A}}$ over $\mathcal{A}$ with minimum $0^{\mathcal{A}}$ and maximum $max^{\mathcal{A}}$,

where a *successor relation over* $\mathcal{A}$ is a binary relation of the form $\{(u_0, u_1), (u_1, u_2), \ldots, (u_{n-2}, u_{n-1})\}$, where $|\mathcal{A}| = \{u_0, u_1, \ldots, u_{n-1}\}$, and the minimum element is $u_0 = 0^{\mathcal{A}}$ and the maximum element is $u_{n-1} = max^{\mathcal{A}}$.

One might be tempted to regard inflationary fixed-point logic with a built-in successor relation as a logic. However, the problem of deciding whether a sentence φ

of inflationary fixed-point logic (and even first-order logic) with a built-in successor relation is well-formed, in that it obeys (a) and (b), above, is undecidable (this follows from Trakhtenbrot's Theorem: see [5]), and it is reasonable to insist that any logic should have a recursive syntax.

This motivated Gurevich to define what it means to be a logic (in the context of capturing complexity classes). A *logic* L is given by a pair of functions (Sen, Sat) satisfying the following conditions. The function Sen associates with every signature σ a recursive set $Sen(\sigma)$ whose elements are called *L-sentences over σ*. The function Sat associates with every signature a recursive relation $Sat_\sigma(\mathcal{A}, \varphi)$, where $\mathcal{A}$ is a σ-structure and φ is a sentence of L. We say that $\mathcal{A}$ *satisfies* φ (and write $\mathcal{A} \models \varphi$) if $Sat_\sigma(\mathcal{A}, \varphi)$ holds. Furthermore, we require that $Sat_\sigma(\mathcal{A}, \varphi)$ if, and only if, $Sat_\sigma(\mathcal{B}, \varphi)$ when $\mathcal{A}$ and $\mathcal{B}$ are isomorphic.

If we were to build our constant symbols 0 and max into the program schemes of RFDPS as is usually done in finite model theory then we would only be interested in program schemes for which acceptance of an input structure is independent of the particular pair of (distinct) values chosen for 0 and max. That is, not every program scheme of RFDPS would accept an isomorphism-closed class of structures, and so not every program scheme of RFDPS would be well-formed. However, with our notion of acceptance, every program scheme of RFDPS is automatically well-formed (in fact, the classes of problems accepted by the program schemes of RFDPS under the two different notions of acceptance are identical but we do not prove this result here) and this obviates the need to check that RFDPS is a logic. Also, with our notion of acceptance, some fragments of RFDPS, to be defined later, are also logics (this fact is not so clear if we adopt the alternative semantics). Note that whichever semantics one adopts, we obtain a class of problems solvable in polynomial-time. However, if we were to build a successor relation into a polynomial-time logic and adopt our semantics then (assuming that the polynomial-time logic is rich enough) we could accept **NP**-complete problems (such as the problem of deciding whether a given digraph has a Hamiltonian cycle).

Remark 4 We make the following two comments.

- (i) We can clearly simulate if-then-else-fi-blocks (with the obvious semantics) within the program schemes of RFDPS.
- (ii) We refer to 'the' computation of a program scheme on some input structure as we assume that 0 and max have been fixed as two arbitrary but distinct elements. □

We phrase the following trivial observations in the form of a lemma.

Lemma 5 *The computation of a program scheme $\rho \in$ RFDPS on some input structure terminates*; *and every problem in RFDPS can be solved in polynomial-time.* □

In the next section, we provide a more formal semantics for our program schemes. However, our definitions above are sufficiently detailed for us to now give examples of some program schemes and the problems they accept, and also to obtain some lower bounds on the computational power of the program schemes of RFDPS. To aid readability: we allow our variables and array symbols to have different names from

x_i and A_i; we assume that we are allowed to use if-then-else-blocks; and we indent in typical programming style.

Example 6 Let $\sigma_2 = \langle E \rangle$, where E is a binary relation symbol. Consider the following program scheme ρ of RFDPS where A and B are array symbols of arity 2.

```
1  INPUT(x, y, z, u, v)
2  FORALL x WITH A^1 DO                 make A the input digraph adjacency
3     FORALL y WITH A^2 DO              matrix with 1's on the leading diagonal
4        IF E(x,y) ∨ x = y THEN
5           A[x,y] := max
6        FI
7     OD
8  OD                                   compute the transitive closure of the
9  REPEAT DO                            input digraph by iteratively multiplying
10    FORALL x WITH B^1 DO              the adjacency matrix by itself
11       FORALL y WITH B^2 DO
12          B[x,y] := A[x,y]            copy the array A into B
13       OD
14    OD
15    FORALL x WITH A^1 DO
16       FORALL y WITH A^2 DO
17          FORALL z DO
18             u := B[x,z]
19             v := B[z,y]
20             IF u = max ∧ v = max THEN
21                u := 0
22                v := 0
23             ELSE
24                u := max
25                v := max
26             FI
27          OD
28          IF z = 0 THEN
29             A[x,y] := max
30          FI
31       OD
32    OD
33 OD
34 FORALL x WITH A^1 DO                 check that every pair appears in the
35    FORALL y WITH A^2 DO              transitive closure of the input digraph
36       z := A[x,y]
37    OD
38    z := y
39 OD
40 IF x = max THEN                      signal acceptance or rejection
41    y := max
42    z := max
```

43 $u := max$
44 $v := max$
45 FI
46 OUTPUT(x, y, z, u, v)

It is worthwhile noting how we simulate existential quantification in the above program scheme. Essentially, we have a forall-do-od-block with inactive control variable z (in line 17). Consequently, we have a child process for every possible value of z. An affirmative answer to our check as to whether there are edges (x, z) and (z, y) results in the local variables u and v being set to 0, otherwise they are set to max. Hence, at the corresponding forall-od-instruction (at line 27), the value of z is set to 0 if, and only if, two edges (x, z) and (z, y) exist, for some z. This is noted in lines 28-30.

This program scheme is such that acceptance and rejection is actually independent of the particular distinct values chosen for 0 and max (a stronger condition than we require); and a σ_2-structure is accepted by ρ if, and only if, when considered as a digraph, it is strongly connected. □

Example 7 Let σ be any signature. Consider the following program scheme ρ of RFDPS.

1 INPUT(x)
2 REPEAT DO
3 IF $x = 0$ THEN
4 $x := max$
5 ELSE
6 $x := 0$
7 FI
8 OD
9 OUTPUT(x)

This program scheme is such that acceptance and rejection is actually independent of the particular distinct values chosen for 0 and max; and a σ-structure is accepted by ρ if, and only if, it has odd size. □

We now exhibit some lower bounds on the class of problems accepted by the program schemes of RFDPS.

Theorem 8 *There is a program scheme of RFDPS accepting any first-order definable problem.*

Proof We shall prove the result by induction on the quantifier-rank d of any first-order formula where our induction hypothesis is: '*Let σ be some signature and σ' be the expansion of σ with m additional constant symbols. For any first-order formula ψ of quantifier-rank r less than $d \geq 1$ over σ and with free variables $x_1, x_2, \ldots, x_m$, say, there exists a program scheme $\rho' \in$ RFDPS over σ' such that if ψ is considered as a sentence over σ' then for every σ'-structure $\mathcal{A}'$*:

- *if $\mathcal{A}' \models \psi$ then $\mathcal{A}' \models \rho'$; and*

- *if* $\mathcal{A}' \not\models \psi$ *then the computation of* ρ' *on input* $\mathcal{A}'$ *(no matter what the distinct values given to* 0 *and* max *are) is such that the output-instruction is reached with all variables involved in* ρ' *having the value* 0.

Moreover, ρ' *does not involve any array symbols and has depth of nesting* r.'

First, the base case of the induction. Let ψ be quantifier-free. The following program scheme suffices.

INPUT(y)
IF ψ THEN
  $y := max$
ELSE
  $y := 0$
FI
OUTPUT(x)

Now, suppose that the inductive hypothesis holds for all formulae of quantifier-rank less than d. Let φ be a first-order formula of quantifier-rank $d \geq 1$ of the form $\exists x_m \psi(x_1, x_2, \ldots, x_m)$, where $x_1, x_2, \ldots, x_m$ are the free variables of ψ. By the induction hypothesis, there exists a program scheme ρ' over σ', the expansion of σ with m additional constant symbols, such that for every σ'-structure $\mathcal{A}'$:

- if $\mathcal{A}' \models \psi$ then $\mathcal{A}' \models \rho'$; and
- if $\mathcal{A}' \not\models \psi$ then the computation of ρ' on input $\mathcal{A}'$ is such that the output-instruction is reached with all variables involved in ρ' having the value 0.

Moreover, ρ' does not involve any array symbols and has depth of nesting $d-1$. Let ρ denote the program scheme ρ' with the input- and output-instructions stripped away; and suppose that the variables involved in ρ' are those of the tuple $\mathbf{y}$. Also, regard x_m now as a variable (not in $\mathbf{y}$) as opposed to a constant symbol (we have assumed that the name of the constant symbol of σ' corresponding to the variable x_m is x_m also). Define the program scheme ρ'' over σ'', the expansion of σ with a constant symbol for each of the variables $x_1, x_2, \ldots, x_{m-1}$, as follows.

INPUT($\mathbf{y}$, x_m)
FORALL x_m DO
  ρ'
  IF $\mathbf{y} = \mathbf{0}$ THEN
    $\mathbf{y} := \mathbf{max}$
  ELSE
    $\mathbf{y} := \mathbf{0}$
  FI
OD
IF $x_m = max$ THEN
  $(\mathbf{y}, x_m) := (\mathbf{0}, 0)$
ELSE
  $(\mathbf{y}, x_m) := (\mathbf{max}, max)$
FI
OUTPUT($\mathbf{y}$, x_m)

The shorthand used above should be obvious (except that **0** and **max** denote tuples of the constant symbols 0 and max, respectively, of the appropriate lengths). The program scheme ρ'' is clearly as required. The case where φ is of the form $\forall x_m \psi(x_1, x_2, \ldots, x_m)$ is similar. The result follows by induction. □

Theorem 9 *There is a program scheme of RFDPS accepting any problem definable in inflationary fixed-point logic.*

Proof Let $\varphi(\mathbf{y}, \mathbf{z})$ be a formula of inflationary fixed-point logic of the form

$$\mathrm{IFP}[\lambda \mathbf{x}, R, \psi(\mathbf{x}, \mathbf{y}, R)](\mathbf{z}),$$

where: $|\mathbf{x}| = |\mathbf{z}| = k$; R is a relation symbol of arity k, not in the underlying signature σ; and ψ is a formula with free variables those of the k-tuple $\mathbf{x}$ and the m-tuple $\mathbf{y}$ such that there exists a program scheme ρ' over σ', the extension of σ with $k + m$ additional constant symbols called $x_1, x_2, \ldots, x_k, y_1, y_2, \ldots, y_m$, with the following properties. Involved in ρ' is an array symbol B of arity k such that B does not appear on the left-hand side of an assignment-instruction. We shall regard the array symbol B as being 'free' in the sense that we shall set the values of its elements from without; and we shall only be interested in valuations of B for which every element is either 0 or max. In this way, B models a k-ary relation over the elements of the input structure. Furthermore, the program scheme ρ' is such that for every σ-structure $\mathcal{A}$, for every k-tuple $\mathbf{u}$ and m-tuple $\mathbf{v}$ over $|\mathcal{A}|$ and for every array valuation $\mathrm{val}(B)$ modelling the k-ary relation R, as above,

$$((\mathcal{A}, \mathbf{u}, \mathbf{v}), \mathrm{val}(B)) \models \rho' \text{ if, and only if, } \psi^{\mathcal{A}}(\mathbf{u}, \mathbf{v}, R)$$

(of course, $(\mathcal{A}, \mathbf{u}, \mathbf{v})$ is the σ'-structure obtained from $\mathcal{A}$ by augmenting $\mathcal{A}$ with the $k + m$ constants $(\mathbf{u}, \mathbf{v})$).

Suppose that the variables involved in the program scheme ρ' are those of the tuple $\mathbf{w}$, and regard the additional constant symbols $x_1, x_2, \ldots, x_k$ now as variables. Consider the program scheme ρ over σ'', the extension of σ with m additional constant symbols $y_1, y_2, \ldots, y_m$, built as follows (where A is another array symbol of arity k).

```
INPUT(w, x)
FORALL x_1 WITH A^1 DO                          initialize A to 0
   FORALL x_2 WITH A^2 DO
      ...
      FORALL x_k WITH A^k DO
         A[x_1, x_2, ..., x_k] := 0
      OD
      ...
   OD
OD
REPEAT DO                                       k nested repeat-do-od-
   REPEAT DO                                    blocks
      ...
      REPEAT DO
         FORALL x_1 WITH B^1 DO                 copy A to B
```

```
                FORALL x_2 WITH B^2 DO
                    ...
                    FORALL x_k WITH B^k DO
                        B[x_1, x_2, ..., x_k] := A[x_1, x_2, ..., x_k]
                    OD
                    ...
                OD
            OD
            FORALL x_1 WITH A^1 DO
                FORALL x_2 WITH A^2 DO
                    ...
                    FORALL x_k WITH A^k DO
                        w := 0                          ρ' has its input- and
                        ρ'                              output-instructions
                        IF w = max THEN                 stripped away
                            A[x_1, x_2, ..., x_k] := max
                        FI
                    OD
                    ...
                OD
            OD
        OD
        ...
    OD
OD
(w, x) := (max, max)
OUTPUT(w, x)
```

The program scheme ρ is such that for every σ-structure $\mathcal{A}$ and for every m-tuple $\mathbf{u}$ over $|\mathcal{A}|$, $(\mathcal{A}, \mathbf{u}) \models \rho$ and on termination, the array element $A[\mathbf{z}]$ encodes $\varphi^{\mathcal{A}}(\mathbf{u}, \mathbf{z})$, the inflationary fixed point of $\psi^{\mathcal{A}}(\mathbf{x}, \mathbf{u}, R)$.

Immerman [13] proved that every problem definable in inflationary fixed-point logic can be defined by a sentence of the form

$$\exists z_1 \exists z_2 \ldots \exists z_k \mathrm{IFP}[\lambda \mathbf{x}, R, \psi(\mathbf{x}, R)](z_1, z_2, \ldots, z_k),$$

where ψ is first-order. A combination of the above construction and that of Theorem 8 yields the required result. □

4 A more formal semantics

Having introduced the program schemes of RFDPS, let us now give a more formal description of a computation of a program scheme on some finite structure. This more formal description will be necessary when we come to prove some (very refined) limitations of our program schemes.

Let the program scheme $\rho \in$ RFDPS be over the signature σ, and involve the variables of $\{x_1, x_2, \ldots, x_k\}$ and have h instructions. Let $\mathcal{A}$ be a σ-structure of size n. An *instantaneous description* (*ID*) of ρ on $\mathcal{A}$ is a tuple $(\mathbf{V}, \mathbf{A}, I, \mathbf{R})$ consisting of:

- a tuple $\mathbf{V}$ which contains a value of $|\mathcal{A}|$ for every variable of $\{x_1, x_2, \ldots, x_k\}$ (with the components ordered in some canonical fashion);
- a tuple $\mathbf{A}$ which contains a value of $|\mathcal{A}|$ for every array element of

$$\{A[u_1, u_2, \ldots, u_a] \quad : \quad A \text{ is an array symbol in } \rho, \text{ of arity } a, \text{ and } u_1, u_2, \ldots, u_a \in |\mathcal{A}|\}$$

  (with the components ordered in some canonical fashion);
- a number $I \in \{1, 2, \ldots, h\}$; and
- if instruction I of ρ is within the scope of r repeat-do-od-blocks then an r-tuple $\mathbf{R}$ of numbers from the set $\{1, 2, \ldots, n\}$.

We can represent a computation of ρ on $\mathcal{A}$ as a labelled acyclic digraph $\mathcal{G}(\rho^{\mathcal{A}})$, whose vertices are labelled with IDs of ρ on input $\mathcal{A}$, built as follows. Start with two vertices q_0 and q_1, and an edge (q_0, q_1). Label the vertex q_0 with the ID $(\mathbf{V}, \mathbf{A}, I, \mathbf{R}) = (\mathbf{0}, \mathbf{0}, 1, \epsilon)$ (this represents the values of the variables and the array elements and the instruction about to be executed in the computation of ρ on input $\mathcal{A}$ initially, where ϵ denotes the empty tuple). If the second instruction is not a repeat-do-instruction then label the vertex q_1 with the ID $(\mathbf{0}, \mathbf{0}, 2, \epsilon)$, otherwise label q_1 with the ID $(\mathbf{0}, \mathbf{0}, 2, (1))$ (this represents the values of the variables and the array elements and the instruction about to be executed after execution of the first instruction, the input-instruction, of ρ on input $\mathcal{A}$). Now apply the following rules until these rules can no longer be applied.

- If the instruction associated with (the ID labelling a) vertex q is an assignment-instruction of the form $\tau := \tau'$, where τ and τ' are terms, then create a new vertex q' and include the edge (q, q'). Label the vertex q' with the same ID as that labelling vertex q except:
  - with the value of the term τ altered so that it is made equal to the value of the term τ' (where value means according to the ID labelling vertex q);
  - with the value of I increased by 1; and
  - if the instruction whose number is the new value of I is a repeat-do-instruction then tag an extra component onto the tuple $\mathbf{R}$ and give this component the value 1.
- If the instruction associated with vertex q is an if-instruction, involving some test φ, then create a new vertex q' and include the edge (q, q'). Label the vertex q' with the same ID as that labelling vertex q except:
  - with the value of I increased by 1 if the test φ holds in $\mathcal{A}$ when the values of any terms in φ are taken according to the ID labelling vertex q;
  - with the value of I made equal to 1 plus the number of the fi-instruction corresponding to the if-instruction if the test φ does not hold; and
  - if the instruction whose number is the new value of I is a repeat-do-instruction then tag an extra component onto the tuple $\mathbf{R}$ and give this component the value 1.

- If the instruction associated with vertex q is a fi-instruction then create a new vertex q' and include the edge (q, q'). Label the vertex q' with the same ID as that labelling vertex q except:
  - with the value of I increased by 1; and
  - if the instruction whose number is the new value of I is a repeat-do-instruction then tag an extra component onto the tuple $\mathbf{R}$ and give this component the value 1.
- If the instruction associated with vertex q is a repeat-do-instruction then create a new vertex q' and include the edge (q, q'). Label the vertex q' with the same ID as that labelling vertex q except:
  - with the value of I increased by 1; and
  - if the instruction whose number is the new value of I is a repeat-do-instruction then tag an extra component onto the tuple $\mathbf{R}$ and give this component the value 1.
- If the instruction associated with vertex q is a repeat-od-instruction then create a new vertex q' and include the edge (q, q'). Label the vertex q' with the same ID as that labelling vertex q except:
  - if the value of the final component of $\mathbf{R}$ is not equal to n then increase this value by 1 and set the value of I to be the value of the corresponding repeat-do-instruction; or
  - if the value of the final component of $\mathbf{R}$ is equal to n then remove the final component from $\mathbf{R}$ and increase the value of I by 1, unless the instruction whose number is the new value of I is a repeat-do-instruction when we do not remove this final component but simply reset it to 1.
- If the instruction associated with vertex q is a forall-do-instruction, for which the control variable is x_p, then create n new vertices $q_0, q_1, \ldots, q_{n-1}$ and include edges $(q, q_0), (q, q_1), \ldots, (q, q_{n-1})$. Label the vertices of $\{q_0, q_1, \ldots, q_{n-1}\}$ with the same ID as that labelling vertex q except:
  - with the values of x_p (in $\mathbf{V}$) in each of the IDs set at a unique value of $|\mathcal{A}|$;
  - with the value of I increased by 1; and
  - if the instruction whose number is the new value of I is a repeat-do-instruction then tag an extra component onto the tuple $\mathbf{R}$ and give this component the value 1.
- If the instruction associated with vertex q is a forall-od-instruction of a forall-do-od-block α, where the control variable corresponding to this instruction is x_p, then find the (unique) first ancestor q'' of q (working backwards up the already constructed acyclic digraph) for which the instruction associated with q'' is the forall-do-instruction corresponding to our forall-od-instruction. Let Q be the set of leaves, i.e., vertices of out-degree 0, of the already constructed acyclic digraph that are descendants of q''. If the instruction associated with every vertex of Q is our forall-od-instruction then create a new vertex q' and include edges $\{(q, q') : q \in Q\}$. Label the vertex q' with the following ID.

- The value of $\mathbf{V}$ is the same as the value of $\mathbf{V}$ in the ID labelling vertex q'' except if the values of the local variables of α in the IDs labelling the vertices of Q are all max then the value of x_p is made equal to max; otherwise it is made equal to 0. If α has no local variables then the value of x_p is made equal to max.
- The values of the array elements of $\mathbf{A}$ are as they are in the ID labelling vertex q'' except that if any of these array elements has a different value in any of the IDs labelling vertices of Q then the value of the array element in the ID labelling the vertex q' is the new value at this vertex of Q (note that because of our syntactic restrictions on forall-do-od-blocks, all array elements in the ID labelling q' are well defined).
- The value of I is increased by 1.
- The values of $\mathbf{R}$ are the same as in the ID labelling vertex q'', unless the instruction whose number is the new value of I is a repeat-do-instruction when we tag an extra component onto the tuple $\mathbf{R}$ and give this component the value 1.

For any block α appearing in ρ, there might be a number of connected components of $\mathcal{G}(\rho^{\mathcal{A}})$ corresponding to α (where by 'connected' we mean with respect to the underlying undirected graph obtained from $\mathcal{G}(\rho^{\mathcal{A}})$ by replacing all directed edges with undirected ones): this is because the block α might appear in the scope of a repeat-do-od-block or a forall-do-od-block. We call the subgraphs of $\mathcal{G}(\rho^{\mathcal{A}})$ corresponding to these connected components *images* of α in $\mathcal{G}(\rho^{\mathcal{A}})$, and we denote an image by $\mathrm{Im}^{\mathcal{A}}(\alpha)$ (it is always clear as to which image of α we are referring). Note that every image has a *source*, the unique vertex of in-degree 0, and a *sink*, the unique vertex of out-degree 0. Note also that the sink of one image will generally be the source of another image, and that the digraph $\mathcal{G}(\rho^{\mathcal{A}})$ is formed by gluing together images of blocks by identifying sources and sinks. The *source* of $\mathcal{G}(\rho^{\mathcal{A}})$ is the unique vertex of in-degree 0, and the *sink* of $\mathcal{G}(\rho^{\mathcal{A}})$ is the unique vertex of out-degree 0. We can clearly talk of a child and a parent of a vertex of $\mathcal{G}(\rho^{\mathcal{A}})$ (indeed, we have already spoken of ancestors and descendants).

Let q be a vertex of $\mathcal{G}(\rho^{\mathcal{A}})$ and let τ be some term. We denote by $q(\tau)$ the value of the term τ in the ID labelling the vertex q (note that if τ is an array term then we must instantiate the appropriate values for the index terms). The input structure $\mathcal{A}$ is accepted by ρ if, and only if, the ID labelling the sink, s, of $\mathcal{G}(\rho^{\mathcal{A}})$ is such that $s(x_1) = max$, $s(x_2) = max$, ..., $s(x_k) = max$.

A *cut* in $\mathcal{G}(\rho^{\mathcal{A}})$ is a set U of vertices such that the source of $\mathcal{G}(\rho^{\mathcal{A}})$ is in U and the vertices of U form a connected component (in the above sense). A vertex q of $\mathcal{G}(\rho^{\mathcal{A}}) \setminus U$ is a *successor vertex* of the cut U if there exists an edge from a vertex of U to q. A *leaf* of U is a vertex of U from which there is no edge to another vertex of U.

5 Some limitations of our program schemes

We begin by proving some limitations on the actual values held by variables and array elements throughout a computation of a program scheme of RFDPS on some input structure. Essentially, Lemma 10's intuitive interpretation is as follows.

- A non-control variable can only ever assume a value equal to a constant or that of a control variable within whose associated block the non-control variable appears.

- An array value can only ever be equal to a constant or that of one of the variables used to index it.

We use the following shorthand in what follows: we denote the set of constant symbols from a signature σ in union with the set $\{0, max\}$ by κ_σ.

Lemma 10 *Let $\rho \in RFDPS$ involve the variables $x_1, x_2, \ldots, x_k$ (and no others) and be over the signature σ. Let $\mathcal{A}$ be some σ-structure and let q be some vertex of $\mathcal{G}(\rho^{\mathcal{A}})$ for which the associated instruction I is in the scope of forall-do-od-blocks in ρ whose control variables are (w.l.o.g.) $x_1, x_2, \ldots, x_m$, for some $m \geq 0$.*

(*i*) *If I is not a forall-do-instruction then*

$$\{q(x_{m+1}), q(x_{m+2}), \ldots, q(x_k)\} \subseteq \kappa_\sigma \cup \{q(x_1), q(x_2), \ldots, q(x_m)\};$$

and if I is a forall-do-instruction, with control variable x_m, say, then

$$\{q(x_m), q(x_{m+1}), \ldots, q(x_k)\} \subseteq \kappa_\sigma \cup \{q(x_1), q(x_2), \ldots, q(x_{m-1})\}.$$

(*ii*) *Let A be any array symbol, of arity a, say, and let $(u_1, u_2, \ldots, u_a) \in |\mathcal{A}|^a$. Then*

$$q(A[u_1, u_2, \ldots, u_a]) \in \kappa_\sigma \cup \{u_1, u_2, \ldots, u_a\}.$$

Proof We shall show that if (*i*) and (*ii*) hold for all vertices in a cut of $\mathcal{G}(\rho^{\mathcal{A}})$ then they hold for any successor vertex of this cut. As the statement trivially holds for the source of $\mathcal{G}(\rho^{\mathcal{A}})$, the result will follow by induction. There are a number of cases, depending upon the type of the instruction associated with a leaf or leaves of our cut.

Suppose that the instruction associated with a leaf q of our cut is a repeat-do-instruction, a repeat-od-instruction, an if-instruction, a fi-instruction or a forall-do-instruction. Then (*i*) and (*ii*) trivially hold for any successor vertex of q in $\mathcal{G}(\rho^{\mathcal{A}})$. (Let us remark that if the instruction associated with a successor vertex of q is a forall-do-instruction then we have another control variable to contend with. However, note that this control variable was not a control variable at q and so (*i*) still holds at a successor vertex of q . This remark applies throughout.)

Suppose that the instruction I associated with a leaf q of our cut is an assignment-instruction and let the successor vertex of q in $\mathcal{G}(\rho^{\mathcal{A}})$ be q'.

- If I is of the form $x_i := \tau$, for some variable or constant symbol τ, then (*i*) and (*ii*) can easily be seen to hold for q' (note that $i \notin \{1, 2, \ldots, m\}$).

- If I is of the form $x_i := B[\tau'_1, \tau'_2, \ldots, \tau'_b]$, for some array symbol B, of arity b, say, then $q(B[\tau'_1, \tau'_2, \ldots, \tau'_b]) \in \kappa_\sigma \cup \{q(\tau'_1), q(\tau'_2), \ldots, q(\tau'_b)\}$ and $q(\tau'_j) \in \kappa_\sigma \cup \{q(x_1), q(x_2), \ldots, q(x_m)\}$, for each $j \in \{1, 2, \ldots, b\}$. So, $q'(x_i) \in \kappa_\sigma \cup \{q(x_1), q(x_2), \ldots, q(x_m)\}$ and $q(x_j) = q'(x_j)$, for each $j \in \{1, 2, \ldots, m\}$ (again, note that $i \notin \{1, 2, \ldots, m\}$). Hence, (*i*) and (*ii*) hold for q'.

- If I is of the form $A[\tau_1, \tau_2, \ldots, \tau_a] := \tau$, where A is an array symbol, of arity a, say, and where τ is a variable or a constant symbol, then $q'(A[\tau_1, \tau_2, \ldots, \tau_a]) \in \kappa_\sigma \cup \{q(x_1), q(x_2), \ldots, q(x_m)\} = \kappa_\sigma \cup \{q'(x_1), q'(x_2), \ldots, q'(x_m)\}$. However, as I is in the scope of forall-do-od-blocks with control variables $x_1, x_2, \ldots, x_m$, we have that $\{x_1, x_2, \ldots, x_m\} \subseteq \{\tau_1, \tau_2, \ldots, \tau_a\}$. Hence, $q'(A[\tau_1, \tau_2, \ldots, \tau_a]) \in \kappa_\sigma \cup \{q'(\tau_1), q'(\tau_2), \ldots, q'(\tau_a)\}$, and (i) and (ii) hold for q'.

- If I is of the form $A[\tau_1, \tau_2, \ldots, \tau_a] := B[\tau'_1, \tau'_2, \ldots, \tau'_b]$, where A and B are array symbols, of arities a and b, say, then $q'(A[\tau_1, \tau_2, \ldots, \tau_a]) \in \kappa_\sigma \cup \{q(\tau'_1), q(\tau'_2), \ldots, q(\tau'_b)\}$. Also, $q(\tau'_j) \in \kappa_\sigma \cup \{q(x_1), q(x_2), \ldots, q(x_m)\}$, for each $j \in \{1, 2, \ldots, b\}$. However, because I is in the scope of forall-do-od-blocks with control variables $x_1, x_2, \ldots, x_m$, we have that $\{x_1, x_2, \ldots, x_m\} \subseteq \{\tau_1, \tau_2, \ldots, \tau_a\}$. Hence, as $q(x_j) = q'(x_j)$, for each $j \in \{1, 2, \ldots, m\}$, we have that (i) and (ii) hold for q'.

Let α be a forall-do-od-block, with control variable x_m, say, and let $\mathrm{Im}^{\mathcal{A}}(\alpha)$ be an image of α in $\mathcal{G}(\rho^{\mathcal{A}})$ so that every vertex of $\mathrm{Im}^{\mathcal{A}}(\alpha)$ apart from the sink is in our cut. Denote the sink of $\mathrm{Im}^{\mathcal{A}}(\alpha)$ by q' and the source by p. Consider $q'(A[u_1, u_2, \ldots, u_a])$, where A is some array symbol of arity a, say, and $u_1, u_2, \ldots, u_a \in |\mathcal{A}|$. As (i) and (ii) hold for p and every parent of q' in $\mathrm{Im}^{\mathcal{A}}(\alpha)$, we have that (ii) also holds for q'. It is also the case that $q'(x_m) \in \{0, max\}$ and $q'(x_j) = p(x_j) \in \kappa_\sigma \cup \{p(x_1), p(x_2), \ldots, p(x_{m-1})\} = \kappa_\sigma \cup \{q'(x_1), q'(x_2), \ldots, q'(x_{m-1})\}$, for each $j \in \{m+1, m+2, \ldots, k\}$. Hence, (i) holds for vertex q'. The result follows by induction. □

We now turn to our main result (note that, by Theorem 1, we think of $\mathcal{A} \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}$ in game-theoretic terms). We emphasise that the following theorem only holds for structures of equal size.

Theorem 11 *Let $\rho \in RFDPS$ be over the signature σ and have depth of nesting $d \geq 0$. Let $\mathcal{A}$ and $\mathcal{B}$ be $\sigma \cup \{0, max\}$-structures of equal size such that $0^{\mathcal{A}} \neq max^{\mathcal{A}}$, $0^{\mathcal{B}} \neq max^{\mathcal{B}}$ and $\mathcal{A} \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}$. Then*

$$\mathcal{A} \models \rho \text{ if, and only if, } \mathcal{B} \models \rho.$$

Proof We begin with some definitions and notation before we outline the structure of the proof.

Let the variables involved in ρ be $x_1, x_2, \ldots, x_k$ and let the constant symbols of σ be $C_1, C_2, \ldots, C_c$, where $c \geq 0$. Let α be any block of instructions appearing in ρ. Suppose that α is in the scope of forall-do-od-blocks $\beta_1, \beta_2, \ldots, \beta_m$ with control variables $x_1, x_2, \ldots, x_m$, respectively, for some $m \geq 0$, and suppose further that block β_{i+1} is in the scope of block β_i, for each $i \in \{1, 2, \ldots, m-1\}$.

Let $\mathrm{Im}^{\mathcal{A}}(\alpha)$ and $\mathrm{Im}^{\mathcal{B}}(\alpha)$ be images of α in $\mathcal{G}(\rho^{\mathcal{A}})$ and $\mathcal{G}(\rho^{\mathcal{B}})$, respectively, and let $s^{\mathcal{A}}$ and $t^{\mathcal{A}}$ be the source and the sink of $\mathrm{Im}^{\mathcal{A}}(\alpha)$, and $s^{\mathcal{B}}$ and $t^{\mathcal{B}}$ the source and sink of $\mathrm{Im}^{\mathcal{B}}(\alpha)$. Write $\kappa_\sigma(\mathcal{A})$ for $\{0^{\mathcal{A}}, max^{\mathcal{A}}, C_1^{\mathcal{A}}, C_2^{\mathcal{A}}, \ldots, C_c^{\mathcal{A}}\}$, with $\kappa_\sigma(\mathcal{B})$ defined similarly.

Definition 12 We write $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$ to denote that the following two conditions hold:

(i) $(\mathcal{A}, s^{\mathcal{A}}(x_1), s^{\mathcal{A}}(x_2), \ldots, s^{\mathcal{A}}(x_m)) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, s^{\mathcal{B}}(x_1), s^{\mathcal{B}}(x_2), \ldots, s^{\mathcal{B}}(x_m))$;

(*ii*) for each $i \in \{m+1, m+2, \ldots, k\}$, one of the following is true:

- $s^{\mathcal{A}}(x_i) = s^{\mathcal{A}}(x_j)$ and $s^{\mathcal{B}}(x_i) = s^{\mathcal{B}}(x_j)$, for some $j \in \{1, 2, \ldots, m\}$,

or

- $s^{\mathcal{A}}(x_i) = C^{\mathcal{A}}$ and $s^{\mathcal{B}}(x_i) = C^{\mathcal{B}}$, for some $C \in \kappa_\sigma$. □

Let us reiterate the comment made after Definition 2 but in the present context. Consider the forall-do-od-blocks $\beta_1, \beta_2, \ldots, \beta_m$. Note that if x_i is active in the array symbol A in β_i then x_j is active in (the same array symbol) A in β_j, for each $j \in \{1, 2, \ldots, i-1\}$; and if x_i is inactive in β_i then x_j is inactive in β_j, for each $j \in \{i+1, i+2, \ldots, m\}$. In particular: either there exists a unique array symbol A so that x_i is active in A in β_i, for at least one $i \in \{1, 2, \ldots, m\}$, when we say that A is the array symbol associated with α; or x_i is not active in β_i, for each $i \in \{1, 2, \ldots, m\}$, when we say that α has no associated array symbol. Whenever α has an associated array symbol, which we always take to be the array symbol A, of arity a, say, and $f \in \{1, 2, \ldots, m\}$ is the maximal such element for which x_f is active in A in β_f then w.l.o.g. we assume that x_i is active in A in β_i at index i, for every $i \in \{1, 2, \ldots, f\}$ (throughout, f always refers to this particular index if α has an associated array symbol).

Definition 13 Suppose that $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$. Let $u_{m+1}, u_{m+2}, \ldots, u_d \in |\mathcal{A}|$ and let $v_{m+1}, v_{m+2}, \ldots, v_d \in |\mathcal{B}|$ be such that

$$(\mathcal{A}, s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, s^{\mathcal{B}}(x_1), \ldots, s^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d).$$

Define π_s to be the natural map from $\kappa_\sigma(\mathcal{A}) \cup \{s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d\}$ to $\kappa_\sigma(\mathcal{B}) \cup \{s^{\mathcal{B}}(x_1), \ldots, s^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d\}$ (note that this map is well-defined and depends upon $u_{m+1}, u_{m+2}, \ldots, u_d$, but we have suppressed this fact in the notation). We say that $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are *array-consistent at* $(u_{m+1}, u_{m+2}, \ldots, u_d)$ *and* $(v_{m+1}, v_{m+2}, \ldots, v_d)$ if

$$\pi_s(s^{\mathcal{A}}(B[\mathbf{w}])) = s^{\mathcal{B}}(B[\pi_s(\mathbf{w})])$$

(with π_s applied point-wise) whenever $\mathbf{w} \in (\kappa_\sigma(\mathcal{A}) \cup \{s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d\})^b$ and either

- B is an array symbol, of arity b, say, and different from the associated array symbol of α, if there is one

or

- there is an associated array symbol A of α, $B = A$ and $w_i = s^{\mathcal{A}}(x_i)$, for each $i \in \{1, 2, \ldots, f\}$

(note that, by Lemma 10, $\pi_s(s^{\mathcal{A}}(B[\mathbf{w}]))$ is always well-defined). If $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are array-consistent at $(u_{m+1}, u_{m+2}, \ldots, u_d)$ and $(v_{m+1}, v_{m+2}, \ldots, v_d)$, for every u_{m+1}, $u_{m+2}, \ldots, u_d \in |\mathcal{A}|$ and $v_{m+1}, v_{m+2}, \ldots, v_d \in |\mathcal{B}|$ for which

$$(\mathcal{A}, s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, s^{\mathcal{B}}(x_1), \ldots, s^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d)$$

then we say that $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are *array-consistent* (note that the notion of array-consistency is symmetric). □

We shall proceed by induction on the building process for the constituent blocks of ρ. The following will be our induction hypothesis: '*For all images* $Im^{\mathcal{A}}(\alpha)$ *and* $Im^{\mathcal{B}}(\alpha)$*, if* $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$ *and* $s^{\mathcal{A}}$ *and* $s^{\mathcal{B}}$ *are array-consistent then* $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$ *and* $t^{\mathcal{A}}$ *and* $t^{\mathcal{B}}$ *are array-consistent*'. If we can prove our induction hypothesis then, as ρ is essentially a finite sequence of blocks of instructions, we can apply it to the program scheme ρ on input $\mathcal{A}$ and $\mathcal{B}$ as follows. If $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are the sources of the (unique) images of the first block of ρ and $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$ are the sinks of the (unique) images of the last block of ρ then the facts that $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$ and $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are array-consistent implies that, in particular, $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$, with all variables of ρ on input $\mathcal{A}$ being set at max on termination if, and only if, all variables of ρ on input $\mathcal{B}$ are set at max on termination.

Base Case The block α is an assignment-block.

Let $\mathrm{Im}^{\mathcal{A}}(\alpha)$ and $\mathrm{Im}^{\mathcal{B}}(\alpha)$ be such that $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$ and $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are array-consistent.

Base Case (a) Suppose that α consists of an instruction of the form $x_i := \tau$, for some term τ (note that $m+1 \leq i \leq k$).

We begin by proving that $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$. If τ is a variable or a constant symbol then trivially $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$. Hence, we suppose that τ is an array term of the form $B[\tau_1, \tau_2, \ldots, \tau_b]$. As $t^{\mathcal{A}}(x_j) = s^{\mathcal{A}}(x_j)$ and $t^{\mathcal{B}}(x_j) = s^{\mathcal{B}}(x_j)$, for each $j \in \{1, 2, \ldots, m\}$, we have that

$$(\mathcal{A}, t^{\mathcal{A}}(x_1), t^{\mathcal{A}}(x_2), \ldots, t^{\mathcal{A}}(x_m)) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, t^{\mathcal{B}}(x_1), t^{\mathcal{B}}(x_2), \ldots, t^{\mathcal{B}}(x_m)).$$

Hence, we need to show that

- $t^{\mathcal{A}}(x_i) = t^{\mathcal{A}}(x_j)$ and $t^{\mathcal{B}}(x_i) = t^{\mathcal{B}}(x_j)$, for some $j \in \{1, 2, \ldots, m\}$

or

- $t^{\mathcal{A}}(x_i) = C^{\mathcal{A}}$ and $t^{\mathcal{B}}(x_i) = C^{\mathcal{B}}$, for some $C \in \kappa_\sigma$.

As $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are array-consistent,

$$\begin{aligned} &\pi_s(s^{\mathcal{A}}(B[s^{\mathcal{A}}(\tau_1), s^{\mathcal{A}}(\tau_2), \ldots, s^{\mathcal{A}}(\tau_b)])) \\ &\quad = s^{\mathcal{B}}(B[\pi_s(s^{\mathcal{A}}(\tau_1)), \pi_s(s^{\mathcal{A}}(\tau_2)), \ldots, \pi_s(s^{\mathcal{A}}(\tau_b))]) \end{aligned}$$

(use Lemma 10 and the fact that $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$). Again, by Lemma 10 and the fact that $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$, we have that

$$\pi_s(s^{\mathcal{A}}(B[s^{\mathcal{A}}(\tau_1), s^{\mathcal{A}}(\tau_2), \ldots, s^{\mathcal{A}}(\tau_b)])) = s^{\mathcal{B}}(B[s^{\mathcal{B}}(\tau_1), s^{\mathcal{B}}(\tau_2), \ldots, s^{\mathcal{B}}(\tau_b)]),$$

thus

$$\pi_s(t^{\mathcal{A}}(x_i)) = t^{\mathcal{B}}(x_i)$$

as required (note that, as remarked in Definition 13, Lemma 10 results in our statements being well defined).

We now show that $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$ are array-consistent. Suppose that $u_{m+1}, u_{m+2}, \ldots, u_d \in |\mathcal{A}|$ and $v_{m+1}, v_{m+2}, \ldots, v_d \in |\mathcal{B}|$ are such that

$$(\mathcal{A}, t^{\mathcal{A}}(x_1), \ldots, t^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, t^{\mathcal{B}}(x_1), \ldots, t^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d),$$

and let π_t be the natural map from $\kappa_\sigma(\mathcal{A}) \cup \{t^{\mathcal{A}}(x_1), \ldots, t^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d\}$ to $\kappa_\sigma(\mathcal{B}) \cup \{t^{\mathcal{B}}(x_1), \ldots, t^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d\}$. As it is the case that $s^{\mathcal{A}}(x_j) = t^{\mathcal{A}}(x_j)$, for all $j \in \{1, 2, \ldots, m\}$, we have that

$$(\mathcal{A}, s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, s^{\mathcal{B}}(x_1), \ldots, s^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d)$$

and π_t is identical to π_s. The fact that no value of any array element changes in the transitions from $s^{\mathcal{A}}$ to $t^{\mathcal{A}}$ and from $s^{\mathcal{B}}$ to $t^{\mathcal{B}}$ makes it routine to check that $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$ are array-consistent.

Base Case (b) Suppose that α consists of an instruction of the form $A[\tau_1, \tau_2, \ldots, \tau_a] := \tau$, for some terms $\tau_1, \tau_2, \ldots, \tau_a, \tau$, and where A is the associated array symbol of α.

As no variable value has changed in the transitions from $s^{\mathcal{A}}$ to $t^{\mathcal{A}}$ and from $s^{\mathcal{B}}$ to $t^{\mathcal{B}}$, we trivially have that $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$.

We now show that $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$ are array-consistent. By arguing as in Base Case (a), above, the only array element we need to consider is $A[\tau_1, \tau_2, \ldots, \tau_a]$ in $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$. Let π_s be the natural map from $\kappa_\sigma(\mathcal{A}) \cup \{s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m)\}$ to $\kappa_\sigma(\mathcal{B}) \cup \{s^{\mathcal{B}}(x_1), \ldots, s^{\mathcal{B}}(x_m)\}$, with π_t defined similarly. We have that

$$\begin{aligned}
&\pi_t(t^{\mathcal{A}}(A[s^{\mathcal{A}}(\tau_1), s^{\mathcal{A}}(\tau_2), \ldots, s^{\mathcal{A}}(\tau_a)])) \\
&\quad = \quad \pi_t(s^{\mathcal{A}}(\tau)) \\
&\quad = \quad \pi_s(s^{\mathcal{A}}(\tau)) \\
&\quad = \quad s^{\mathcal{B}}(\tau) \text{ (as } \mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s \text{ and } s^{\mathcal{A}} \text{ and } s^{\mathcal{B}} \text{ are array-consistent)} \\
&\quad = \quad t^{\mathcal{B}}(A[s^{\mathcal{B}}(\tau_1), s^{\mathcal{B}}(\tau_2), \ldots, s^{\mathcal{B}}(\tau_a)]) \\
&\quad = \quad t^{\mathcal{B}}(A[\pi_s(s^{\mathcal{A}}(\tau_1)), \pi_s(s^{\mathcal{A}}(\tau_2)), \ldots, \pi_s(s^{\mathcal{A}}(\tau_a))]) \text{ (as } \mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s)
\end{aligned}$$

as required.

Thus, whatever the form of the assignment-block α, we have that $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$ and $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$ are array-consistent.

Inductive Case (i) Let α be a forall-do-od-block of the form

FORALL x_{m+1} WITH A^p DO	*forall-do-instruction*
α_1	*block of instructions*
α_2	*block of instructions*
$\ldots$	
α_l	*block of instructions*
OD	*forall-od-instruction*

where our induction hypothesis holds for the blocks $\alpha_1, \alpha_2, \ldots, \alpha_l$. Let $\mathrm{Im}^{\mathcal{A}}(\alpha)$ and $\mathrm{Im}^{\mathcal{B}}(\alpha)$ be images of α such that $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$ and $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are array-consistent. Note that A is the associated array symbol of α and that we may assume that x_i is active in A for β_i at index i, for each $i \in \{1, 2, \ldots, m+1\}$ (renaming α as β_{m+1}).

We begin by showing that $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$. Pick some $u_{m+1} \in |\mathcal{A}|$ and let $s^{\mathcal{A}}_{m+1}$ be the child of $s^{\mathcal{A}}$ in $\mathcal{G}(\rho^{\mathcal{A}})$ for which $s^{\mathcal{A}}_{m+1}(x_{m+1}) = u_{m+1}$. Let $v_{m+1} \in |\mathcal{B}|$ be such that

$$(A, s^{\mathcal{A}}(x_1), s^{\mathcal{A}}(x_2), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}) \equiv^{\mathcal{L}^d_{\infty\omega}} (B, s^{\mathcal{B}}(x_1), s^{\mathcal{B}}(x_2), \ldots, s^{\mathcal{B}}(x_m), v_{m+1})$$

(at least one such v_{m+1} exists since $m < d$) and let $s^{\mathcal{B}}_{m+1}$ be the son of $s^{\mathcal{B}}$ in $\mathcal{G}(\rho^{\mathcal{B}})$ for which $s^{\mathcal{B}}_{m+1}(x_{m+1}) = v_{m+1}$. Rewriting, we obtain that

$$\begin{aligned}&(A, s^{\mathcal{A}}_{m+1}(x_1), s^{\mathcal{A}}_{m+1}(x_2), \ldots, s^{\mathcal{A}}_{m+1}(x_{m+1}))\\ &\quad \equiv^{\mathcal{L}^d_{\infty\omega}} (B, s^{\mathcal{B}}_{m+1}(x_1), s^{\mathcal{B}}_{m+1}(x_2), \ldots, s^{\mathcal{B}}_{m+1}(x_{m+1}))\end{aligned}$$

and so $\mathcal{A}_{s_{m+1}} \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_{s_{m+1}}$.

Now we turn to the array-consistency of $s^{\mathcal{A}}_{m+1}$ and $s^{\mathcal{B}}_{m+1}$. Let $u_{m+2}, u_{m+3}, \ldots, u_d \in |\mathcal{A}|$ and $v_{m+2}, v_{m+3}, \ldots, v_d \in |\mathcal{B}|$ be such that

$$\begin{aligned}&(\mathcal{A}, s^{\mathcal{A}}_{m+1}(x_1), \ldots, s^{\mathcal{A}}_{m+1}(x_{m+1}), u_{m+2}, \ldots, u_d)\\ &\quad \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, s^{\mathcal{B}}_{m+1}(x_1), \ldots, s^{\mathcal{B}}_{m+1}(x_{m+1}), v_{m+2}, \ldots, v_d).\end{aligned}$$

Rewriting, we obtain that

$$(\mathcal{A}, s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, s^{\mathcal{B}}(x_1), \ldots, s^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d),$$

and the corresponding maps $\pi_{s_{m+1}}$ and π_s are identical. By assumption, $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are array-consistent at $(u_{m+1}, u_{m+2}, \ldots, u_d)$ and $(v_{m+1}, v_{m+2}, \ldots, v_d)$. As the transitions from $s^{\mathcal{A}}$ to $s^{\mathcal{A}}_{m+1}$ and from $s^{\mathcal{B}}$ to $s^{\mathcal{B}}_{m+1}$ cause no array element to change value, we must have that $s^{\mathcal{A}}_{m+1}$ and $s^{\mathcal{B}}_{m+1}$ are array-consistent at $(u_{m+2}, u_{m+3}, \ldots, u_d)$ and $(v_{m+2}, v_{m+3}, \ldots, v_d)$. That is, $s^{\mathcal{A}}_{m+1}$ and $s^{\mathcal{B}}_{m+1}$ are array-consistent.

By the induction hypothesis applied to $\alpha_1, \alpha_2, \ldots, \alpha_l$, we have that $\mathcal{A}_{t_{m+1}} \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_{t_{m+1}}$ and $t^{\mathcal{A}}_{m+1}$ and $t^{\mathcal{B}}_{m+1}$ are array-consistent, where $t^{\mathcal{A}}_{m+1}$ (resp. $t^{\mathcal{B}}_{m+1}$) is the parent of $t^{\mathcal{A}}$ (resp. $t^{\mathcal{B}}$) for which $t^{\mathcal{A}}_{m+1}(x_{m+1}) = u_{m+1}$ (resp. $t^{\mathcal{B}}_{m+1}(x_{m+1}) = v_{m+1}$).

We have just proved that for every 'child process' in the image $\mathrm{Im}^{\mathcal{A}}(\alpha)$, there is a 'similar' process in $\mathrm{Im}^{\mathcal{B}}(\alpha)$. Now we must show that the converse is true. Pick some $v_{m+1} \in |\mathcal{B}|$ and let $s^{\mathcal{B}}_{m+1}$ be the child of $s^{\mathcal{B}}$ in $\mathcal{G}(\rho^{\mathcal{B}})$ for which $s^{\mathcal{B}}_{m+1}(x_{m+1}) = v_{m+1}$. Let $u_{m+1} \in |\mathcal{A}|$ be such that

$$(\mathcal{B}, s^{\mathcal{B}}(x_1), \ldots, s^{\mathcal{B}}(x_m), v_{m+1}) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{A}, s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}).$$

An identical argument to the above yields that $B_{t_{m+1}} \equiv^{\mathcal{L}^d_{\infty\omega}} A_{t_{m+1}}$ and $t^{\mathcal{B}}_{m+1}$ and $t^{\mathcal{A}}_{m+1}$ are array-consistent (where the notation is as above). Consequently, either

- $t^{\mathcal{A}}(x_{m+1}) = 0$ and $t^{\mathcal{B}}(x_{m+1}) = 0$

or

- $t^{\mathcal{A}}(x_{m+1}) = max$ and $t^{\mathcal{B}}(x_{m+1}) = max$;

so $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$.

What remains to be shown is that $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$ are array-consistent. Let u_{m+1}, u_{m+2}, $\ldots, u_d \in |\mathcal{A}|$ and $v_{m+1}, v_{m+2}, \ldots, v_d \in |\mathcal{B}|$ be such that

$$(\mathcal{A}, t^{\mathcal{A}}(x_1), \ldots, t^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, t^{\mathcal{B}}(x_1), \ldots, t^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d),$$

with π_t defined as usual. There are two cases, according to the definition of array-consistency and the array symbol involved.

Inductive Case $(i)(a)$ Let B be an array symbol, of arity b, say, and different from A, and let $\mathbf{w} \in (\kappa_\sigma(\mathcal{A}) \cup \{t^{\mathcal{A}}(x_1), \ldots, t^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d\})^b$. Rewriting, we obtain that

$$(\mathcal{A}, s^{\mathcal{A}}(x_1), \ldots, s^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d) \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, s^{\mathcal{B}}(x_1), \ldots, s^{\mathcal{B}}(x_m), v_{m+1}, \ldots, v_d)$$

and, with π_s defined as usual, π_s is identical to π_t. Consequently, by assumption,

$$\pi_s(s^{\mathcal{A}}(B[\mathbf{w}])) = s^{\mathcal{B}}(B[\pi_s(\mathbf{w})]).$$

Note that no transition from $s^{\mathcal{A}}$ to $t^{\mathcal{A}}$ and from $s^{\mathcal{B}}$ to $t^{\mathcal{B}}$ causes an array element of B to change value and so

$$\pi_t(t^{\mathcal{A}}(B[\mathbf{w}])) = t^{\mathcal{B}}(B[\pi_t(\mathbf{w})]).$$

Inductive Case $(i)(b)$ Let $\mathbf{w} \in (\kappa_\sigma(\mathcal{A}) \cup \{t^{\mathcal{A}}(x_1), \ldots, t^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d\})^a$ with $w_i = t^{\mathcal{A}}(x_i)$, for each $i \in \{1, 2, \ldots, m\}$. Note that the value of $A[\mathbf{w}]$ at $t^{\mathcal{A}}$ (resp. $A[\pi_t(\mathbf{w})]$ at $t^{\mathcal{B}}$) is identical to the value of $A[\mathbf{w}]$ at $t^{\mathcal{A}}_{m+1}$ (resp. $A[\pi_t(\mathbf{w})]$ at $t^{\mathcal{B}}_{m+1}$), where $t^{\mathcal{A}}_{m+1}$ (resp. $t^{\mathcal{B}}_{m+1}$) is the parent of $t^{\mathcal{A}}$ (resp. $t^{\mathcal{B}}$) for which $t^{\mathcal{A}}_{m+1}(x_{m+1}) = w_{m+1}$ (resp. $t^{\mathcal{B}}_{m+1}(x_{m+1}) = \pi_t(w_{m+1})$).

In particular, as $w_{m+1} \in \kappa_\sigma(\mathcal{A}) \cup \{t^{\mathcal{A}}(x_1), \ldots, t^{\mathcal{A}}(x_m), u_{m+1}, \ldots, u_d\}$,

$$\begin{aligned}&(\mathcal{A}, t^{\mathcal{A}}(x_1), \ldots, t^{\mathcal{A}}(x_m), w_{m+1}, u_{m+2}, \ldots, u_d)\\ &\quad \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, t^{\mathcal{B}}(x_1), \ldots, t^{\mathcal{B}}(x_m), \pi_t(w_{m+1}), v_{m+2}, \ldots, v_d).\end{aligned}$$

Rewriting yields that

$$\begin{aligned}&(\mathcal{A}, t^{\mathcal{A}}_{m+1}(x_1), \ldots, t^{\mathcal{A}}_{m+1}(x_m), t^{\mathcal{A}}_{m+1}(x_{m+1}), u_{m+2}, \ldots, u_d)\\ &\quad \equiv^{\mathcal{L}^d_{\infty\omega}} (\mathcal{B}, t^{\mathcal{B}}_{m+1}(x_1), \ldots, t^{\mathcal{B}}_{m+1}(x_m), t^{\mathcal{B}}_{m+1}(x_{m+1}), v_{m+2}, \ldots, v_d).\end{aligned}$$

Furthermore, $\pi_{t_{m+1}}$ is either identical to π_t or a restriction of π_t; either way, $\pi_{t_{m+1}}$ is identical to π_t on the domain of $\pi_{t_{m+1}}$. Thus

$$\begin{aligned}&\pi_t(t^{\mathcal{A}}(A[\mathbf{w}]))\\ &\quad = \quad \pi_t(t^{\mathcal{A}}_{m+1}(A[\mathbf{w}]))\\ &\quad = \quad \pi_{t_{m+1}}(t^{\mathcal{A}}_{m+1}(A[\mathbf{w}])) \text{ (by Lemma 10)}\\ &\quad = \quad t^{\mathcal{B}}_{m+1}(A[\pi_{t_{m+1}}(\mathbf{w})])\\ &\quad = \quad t^{\mathcal{B}}_{m+1}(A[\pi_t(\mathbf{w})])\\ &\quad = \quad t^{\mathcal{B}}(A[\pi_t(\mathbf{w})]).\end{aligned}$$

Thus, we have that $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$ and $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$ are array-consistent.

Inductive Case (ii) Let α be a forall-do-od-block of the form

FORALL x_{m+1} DO	*forall-do-instruction*
α_1	*block of instructions*
α_2	*block of instructions*
$\ldots$	
α_l	*block of instructions*
OD	*forall-od-instruction*

where our induction hypothesis holds for the blocks $\alpha_1, \alpha_2, \ldots, \alpha_l$. Let $\mathrm{Im}^{\mathcal{A}}(\alpha)$ and $\mathrm{Im}^{\mathcal{B}}(\alpha)$ be images of α such that $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$ and $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are array-consistent. The proof of Case (i) can be applied when some β_i, for $i \in \{1, 2, \ldots, m\}$ has an active control variable and also when no such β_i has an active control variable. (Note that so far in the proof of the theorem we have not needed to use the fact that $|\mathcal{A}| = |\mathcal{B}|$.)

Inductive Case (iii) Let α be a repeat-do-od-block or an if-then-fi-block. In both cases, immediate applications of the induction hypothesis yield the required result. Note that we require for the case when α is a repeat-do-od-block that $|\mathcal{A}| = |\mathcal{B}|$ as in order for our reasoning to hold, the number of iterations of α in the corresponding computations must be identical.

Consequently, we have that the induction hypothesis holds for every constituent block of ρ. Let $s^{\mathcal{A}}$ and $t^{\mathcal{A}}$ be the source and the sink of $\mathcal{G}(\rho^{\mathcal{A}})$, with $s^{\mathcal{B}}$ and $t^{\mathcal{BS}}$ the source and the sink of $\mathcal{G}(\rho^{\mathcal{B}})$. Clearly, $\mathcal{A}_s \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_s$ and $s^{\mathcal{A}}$ and $s^{\mathcal{B}}$ are array-consistent. Hence, $\mathcal{A}_t \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_t$ and $t^{\mathcal{A}}$ and $t^{\mathcal{B}}$ are array-consistent, and our result follows. □

Let RFDPS_d be those program schemes of RFDPS with depth of nesting at most d (and also the class of problems definable by such program schemes). Note that RFDPS_d is a logic (in Gurevich's sense).

Corollary 14

$$\mathrm{RFDPS}_0 \subset \mathrm{RFDPS}_1 \subset \ldots \subset \mathrm{RFDPS}_d \subset \mathrm{RFDPS}_{d+1} \subset \ldots$$

Proof Let $\sigma = \langle E, C, D \rangle$, where E is a binary relation symbol and C and D are constant symbols. Hence, a σ-structure can be thought of as a directed graph with two distinguished vertices. Fix $d \geq 1$. Define the σ-structure $\mathcal{A}_{d+1}$ as follows. The vertices $C^{\mathcal{A}_{d+1}}$ and $D^{\mathcal{A}_{d+1}}$ are distinct vertices of in-degree 0 and out-degree $d+3$ so that they have no neighbour in common (this constitutes all vertices and edges of $\mathcal{A}_{d+1}$). Define the σ-structure $\mathcal{B}_{d+1}$ as follows. The vertices $C^{\mathcal{B}_{d+1}}$ and $D^{\mathcal{B}_{d+1}}$ are distinct vertices of in-degree 0 and out-degree $d+2$ and $d+4$, respectively, so that they have no neighbour in common (this constitutes all vertices and edges of $\mathcal{B}_{d+1}$).

Consider the following program scheme ρ_{d+1} of RFDPS_{d+1}.

INPUT(x_1, x_2, …, x_{d+1}, y)
FORALL x_1 DO
  FORALL x_2 DO
    …
    FORALL x_{d+1} DO
      $y := max$
      IF $\bigwedge_{i \neq j} x_i \neq x_j \wedge \bigwedge_i (x_i \neq 0 \wedge x_i \neq max) \wedge \bigwedge_i E(C, x_i)$
      $\wedge E(C, 0) \wedge E(C, max)$ THEN
        $y := 0$
      FI
    OD
    …
  OD

OD
IF $x_1 = 0$ THEN
  (**x**,y) = (**max**, max)
ELSE
  (**x**,y) = (**0**, 0)
FI
OUTPUT(x_1, x_2, ..., x_{d+1}, y)

Clearly, $\mathcal{A}_{d+1}$ is accepted by ρ_{d+1} but $\mathcal{B}_{d+1}$ is not. Suppose that the problem accepted by ρ_{d+1} is accepted by some program scheme ρ of RFDPS$_d$. As Duplicator clearly has a winning strategy in the d-pebble game on $\mathcal{A}_{d+1}$ and $\mathcal{B}_{d+1}$, by Theorem 1 $\mathcal{A}_{d+1} \equiv^{\mathcal{L}^d_{\infty\omega}} \mathcal{B}_{d+1}$. Hence, Theorem 11 yields a contradiction. The result follows (as clearly RFDPS$_0 \subset$ RFDPS$_1$). □

Note that the proof of Corollary 14 can be used to show that the problem consisting of all those digraphs for which every vertex has even out-degree is not in RFDPS.

Corollary 15 *There are problems in* **P** *which are not in RFDPS.* □

Let φ be a formula of inflationary fixed-point logic. The *quantifier-rank* q.r.(φ) of φ is defined inductively as follows.

- If φ is first-order quantifier-free then q.r.(φ) = 0.
- If φ is of the form $\neg\psi$ then q.r.(φ) = q.r.(ψ).
- If φ is of the form $\psi_1 \vee \psi_2$ or $\psi_1 \wedge \psi_2$ then q.r.(φ) = $\max\{\text{q.r.}(\psi_1), \text{q.r.}(\psi_2)\}$.
- If φ is of the form $\exists\psi$ or $\forall\psi$ then q.r.(φ) = $1 + \text{q.r.}(\psi)$.
- If φ is of the form $\text{IFP}[\lambda\mathbf{x}, R, \psi(\mathbf{x}, \mathbf{y}, R)](\mathbf{z})$ then q.r.(φ) = $|\mathbf{x}| + \text{q.r.}(\psi)$.

Let IFP$_d$ be those formulae of inflationary fixed-point logic with quantifier rank at most d (and also the class of problems definable by such sentences). The proof of Corollary 14 suffices to prove the following.

Corollary 16 IFP$_0 \subset$ IFP$_1 \subset \ldots \subset$ IFP$_d \subset$ IFP$_{d+1} \subset \ldots$ □

The above corollary has not been studied before but, as pointed out by Martin Grohe in a personal communication, it follows quite easily from known results. The fragment IFP$_d$ is contained in $L^d_{\infty\omega}$. This implies that the problem consisting of all those structures over the empty signature having at least $d + 1$ elements is not expressible in IFP$_d$; but it clearly is in IFP$_{d+1}$ (actually in FO$_{d+1}$). We remark that our proof of Corollary 16 relies on no existing results from finite model theory (and not even on an understanding and appreciation of bounded-variable infinitary logic).

6 Conclusion

Whilst our concerns in this paper have been the development of the class of program schemes RFDPS and an investigation of its refined structure, we feel that RFDPS will make a good stepping-off point in the quest for a logic for **P**, as we now explain. Throughout any computation by a program scheme of RFDPS, we construct arrays of values. It will be relatively straightforward to incorporate Lindström quantifiers (see [5]) into the program schemes of RFDPS by extending if-instructions so that the test can be an application of some Lindström quantifier to some arrays, the values of whose elements are either 0 or max (so that the arrays model relations as in the proof of Theorem 9). It will also be entirely natural to include variables of a different type. For instance, one might allow an additional universe $\{0, 1, \ldots, n-1\}$, when the input to some program scheme is a structure of size n, with some appropriate numeric relations and a mechanism for 'tying' the two universes together; for example, an instruction $x := \sharp\varphi(y)$, where x has numeric type and φ is first-order, whose semantics are such that the number of values of y for which $\varphi(y)$ holds is assigned to the variable x. We shall pursue such extensions in future work.

A natural question to consider is how the class of problems accepted by the program schemes of RFDPS (and any extensions we might developed, as in the preceding paragraph) compares with those accepted by the programs of $\tilde{\mathrm{C}}$PTime and by other models more prevalent in database theory. We have not so far considered this question: however, let us remark that the problem consisting of those digraphs for which every vertex has even out-degree is not accepted by any program scheme of RFDPS yet can be accepted by a program of [14].

Whereas we feel that it will be fruitful to extend the program schemes of RFDPS, as hinted above, and investigate the expressive power of any resulting class of program schemes, there are still questions to be asked of RFDPS. For example, as was the case for the program schemes NPS, NPSS and NPSA of [2, 16, 18], can the class of problems accepted by the program schemes of RFDPS be realized as a *vectorized Lindström logic*? Does this class of problems have a complete member (via some suitable logical translation)? Is this class of problems nothing other than an extension of inflationary fixed-point logic?

Acknowledgement The authors are extremely grateful to an anonymous referee for his or her excellent remarks as regards an earlier draft of this paper.

References

[1] S. Abiteboul and V. Vianu, Generic computation and its complexity, *Proceedings of ACM Symposium on Theory of Computing*, ACM Press (1991) 209–219.

[2] A.A. Arratia-Quesada, S.R. Chauhan and I.A. Stewart, Hierarchies in classes of program schemes, *Journal of Logic and Computation* **9** (1999) 915–957.

[3] A. Blass, Y. Gurevich and S. Shelah, Choiceless polynomial time, *Annals of Pure and Applied Logic* **100** (1999) 141–187.

[4] J. Cai, M. Fürer and N. Immerman, An optimal lower bound on the number of variables for graph identification, *Combinatorica* **12** (1992) 389–410.

[5] H.-D. Ebbinghaus and J. Flum, *Finite Model Theory*, Springer-Verlag (1995).

[6] F. Gire and H.K. Hoang, An extension of fixpoint logic with a symmetry-based choice construct, *Information and Computation* **144** (1998) 40–65.

[7] M. Grohe, Bounded-arity hierarchies in fixed-point logics, *Proceedings of Computer Science Logic* (ed. E. Börger, Y. Gurevich, K. Meinke), Lecture Notes in Computer Science Vol. 832, Springer-Verlag (1994) 150–164.

[8] M. Grohe, Arity hierarchies, *Annals of Pure and Applied Logic* **82** (1996) 103–163.

[9] M. Grohe and L. Hella, A double arity hierarchy theorem for transitive closure logic, *Archive for Mathematical Logic* **35** (1996) 157–171.

[10] Y. Gurevich, Logic and the challenge of computer science, *Current Trends in Theoretical Computer Science* (ed. E. Börger), Computer Science Press (1988) 1–57.

[11] Y. Gurevich, Evolving algebras 1993: Lipari guide, *Specification and Validation* (ed. E. Börger), Oxford University Press (1995) 9–36.

[12] Y. Gurevich, May 1997 Draft of the ASM Guide, Technical Report, EECS Department, University of Michigan (1997).

[13] N. Immerman, Relational queries computable in polynomial time, *Information and Control* **68** (1986) 86–104.

[14] F. Neven, M. Otto, J. Tyszkiewicz, and J. Van den Bussche, Adding for-loops to first-order logic, *Information and Computation* **168** (2001) 156–186.

[15] I.A. Stewart, Logical and schematic characterization of complexity classes, *Acta Informatica* **30** (1993) 61–87.

[16] I.A. Stewart, Program schemes, arrays, Lindström quantifiers and zero-one laws, *Theoretical Computer Science* **275** (2002) 283–310.

[17] I.A. Stewart, Using program schemes to logically capture polynomial-time on certain classes of structures, *London Mathematical Society Journal of Computation and Mathematics* **6** (2003) 40–67.

[18] I.A. Stewart, Program schemes with binary write-once arrays and the complexity classes they capture, *submitted for publication*.