Using program schemes to logically capture polynomial-time on certain classes of structures

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

Abstract

We continue the study of the expressive power of certain classes of program schemes on finite structures, in relation to more mainstream logics studied in finite model theory and to computational complexity. We show that there exists a program scheme, whose constructs are assignments and while-loops with quantifier-free tests and which has access to a stack, that can accept a **P**-complete problem, the deterministic path system problem, even in the absence of non-determinism so long as problem instances are presented in a functional style. (Our proof leans heavily on Cook's proof that the classes of formal languages accepted by deterministic and non-deterministic logspace auxiliary pushdown machines coincide). However, whilst our result is of independent interest, as it leads to a deterministic model of computation capturing **P** whose non-deterministic variant also captures **P**, we then show how our constructed program scheme can be used to build a successor relation in certain classes of structures, namely: the class of strongly-connected locally-ordered digraphs; the class of connected planar embeddings; and the class of triangulations, with the consequence that on these classes of graphs, (a fragment of) path system logic (with no built-in relations) captures exactly the polynomial-time solvable problems.

1 Introduction

One of the central open problems in finite model theory is whether there is a logic for capturing the complexity class **P** (polynomial-time); that is, whether there is a logic such that the class of problems definable in this logic coincides with the class of polynomial-time solvable problems. Of course, one has to be precise about what one means by a 'logic' (the generally accepted definition is given in, for example, [9]) but one sensible property that any logic should have is that it should have a recursive syntax; that is, the well-formed formulae of any logic should be recursively enumerable.

*Most of this research was completed whilst the author was at the University of Leicester. Supported by EPSRC Grant GR/K 96564.

This property immediately rules out all existing 'logical' characterisations of **P** based around 'logics' with built-in relations, such as inflationary fixed-point logic with a built-in successor relation [4] and path system logic with a built-in successor relation [12]. (Throughout, for convenience, we try to use [4] as our main reference text for definitions and results in finite model theory and descriptive complexity, and the reader is referred to this text for more details on the proper attribution of results.)

Working on the assumption that there is a logic capturing **P**, one can approach this central problem in two directions. One can try and develop more and more expressive logics (but where the expressibility stays within polynomial-time) and hope that eventually a logic capturing **P** will emerge; or one can consider existing logics, such as inflationary fixed-point logic and path system logic (in the absence of built-in relations) and try to capture **P** on certain classes of finite structures, in the hope that eventually such characterisations will show exactly what has to be added to one of these logics (and still retain the property of being a logic) so as to capture **P**. Of course, it may be the case that there does not exist a logic capturing **P** (with the consequence that $\mathbf{P} \neq \mathbf{NP}$). If this is so then it is clearly worthwhile to discover on which classes of finite structures and for which logics **P** can be logically captured. It is essentially with this question that we are concerned here.

Existing results regarding capturing **P** on restricted classes of structures are all concerned with inflationary fixed-point logic. In particular: Immerman and Lander [8] proved that inflationary fixed-point logic with counting (that is, where there is an additional universe of numbers and a total ordering on this universe) captures **P** on the class of trees; and Grohe [6] and Grohe and Mariño [7] proved that this same logic does likewise on the class of planar graphs and the class of graphs of bounded tree-width, respectively. Grohe [6] also proved that inflationary fixed-point logic (without counting) captures **P** on the class of 3-connected planar graphs. In this paper, we show that a fragment of path system logic, which itself is a proper fragment of inflationary fixed-point logic (even on the class of trees) suffices to capture **P** on the following classes of structures: strongly-connected locally-ordered digraphs; connected planar embeddings; and triangulations. The class of triangulations (that is, the class of planar graphs having a planar embedding whose faces, including the outer face, are all cycles of length 3) forms a (significant) proper sub-class of the class of 3-connected planar graphs, and so one might interpret our result as a strengthening of Grohe's result for this class of graphs (we do not as yet know whether it is the case that path system logic captures **P** on the class of 3-connected planar graphs).

Our results, mentioned in the preceding paragraph, are applications of another result in this paper concerning program schemes. Program schemes are essentially a model of computation that is amenable to logical analysis yet is closer to the general notion of a program than a logical formula is. They 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. Program schemes have since been further developed to work on finite structures, mindful of advances in finite model theory (see, for example, [1, 13, 14] for more details). One appealing characteristic of program schemes is that they form a model of computation for computing on unordered data.

Our main result involving program schemes is that there is a deterministic program scheme ρ, whose constructs are assignments and while-loops with quantifier-free tests and which has access to a stack, that accepts a **P**-complete problem, the deterministic path system problem, if the instances of this problem are presented as finite structures over a signature consisting of a binary function symbol and two constant symbols. Our proof is very close in essence to Cook's proof [2] that the classes of formal languages accepted by deterministic and non-deterministic logspace auxiliary pushdown machines coincide (it is, however, much more rigorously presented than Cook's proof). Whilst our result is of independent interest, as it leads to a deterministic model of computation capturing **P** whose non-deterministic variant also captures **P**, the actual program scheme ρ, above, allied with results from [1] linking similar program schemes with path system logic, enables us to canonically build a successor relation in any graph from one of the classes mentioned above. Thus, we can logically capture **P** on these classes of graphs.

In the next section, we give the basic definitions pertaining to finite model theory and program schemes, before proving in Section 3 that we can solve the deterministic path system problem in the manner described above. Our applications are detailed in Section 4, and we present our conclusions in Section 5.

2 Preliminaries

Ordinarily, 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. However, we sometimes allow our signatures to also contain function symbols. When we do, we explicitly denote that this is the case by referring to the signature as σ'; that is, we use a superscript $'$ to denote signatures which might contain function symbols. Consequently, definitions, theorems and the like might apply only to signatures σ not involving function symbols or they might apply to signatures σ' where function symbols are allowed (though not necessarily present). For example, *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 $\mathrm{FO} = \cup\{\mathrm{FO}(\sigma) : \sigma \text{ is some signature}\}$. Thus, according to our notation, we have defined $\mathrm{FO}(\sigma)$ and FO only for signatures not containing function symbols. Of course, first-order logic can be defined over signatures containing function symbols; however, our definition suffices for our needs. The same can be said of other subsequent definitions.

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 of arity a_i, for every relation symbol R_i of σ, of arity a_i, and a constant $C_j \in |\mathcal{A}|$ for every constant symbol C_j (by an abuse of notation, we do not distinguish between constants or relations and constant or relation symbols). If $\mathcal{A}$ is a finite σ'-structure, for some signature σ' (possibly containing function symbols, note) then in addition to the above, for every function symbol F_i of arity b_i, there is a total function $F_i : |\mathcal{A}|^{b_i} \to |\mathcal{A}|$.

A finite structure $\mathcal{A}$ whose domain consists of n distinct elements has *size* n, and we denote the size of $\mathcal{A}$ by $|\mathcal{A}|$ also (this does not cause confusion). We only ever consider finite structures of size at least 2, and the set of all finite structures over the signature σ' of size at least 2 is denoted $\mathrm{STRUCT}(\sigma')$. A *problem* over some signature σ' consists of a subset of $\mathrm{STRUCT}(\sigma')$ that is closed under isomorphism; that is, if $\mathcal{A}$ is in the problem then so is every isomorphic copy of $\mathcal{A}$. Throughout, all our structures are finite.

We are now in a position to consider the class of problems defined by the sentences of FO: we denote this class of problems by FO also, and do likewise for other logics. It is widely acknowledged that, as a means for defining problems, first-order logic leaves a lot to be desired especially when we have in mind developing a relationship between computational complexity and logical definability. In particular, every first-order definable problem can be accepted by a logspace deterministic Turing machine yet there are problems in **L** (logspace) which can not be defined in first-order logic (one such being the problem consisting of all those structures over the empty signature that have even size). Consequently, we now illustrate one way of increasing the expressibility of FO: we augment FO with a uniform or vectorized sequence of Lindström quantifiers, or operator for short (the reader is referred to [4] for a fuller exposition on the limitations of FO and on a number of different methods, including this one, for increasing the expressibility of FO).

Our illustration uses an operator derived from a problem whose underlying instances can be regarded as path systems. A *path system* consists of a finite set of *vertices* and a finite set of *rules*, each of the form (x, y, z), where x, y and z are (not necessarily distinct) vertices. There is a unique distinguished vertex called the *source* and a unique distinguished vertex called the *sink*. The set of *accessible vertices* in any path system is built as follows. Initially, the source is deemed to be accessible and new vertices are shown to be accessible by *applying* the rules via: if x and y are accessible (with possibly $x = y$) and there is a rule (x, y, z) then z becomes accessible. The *path system problem* consists of all those path systems for which the sink is accessible from the source, and it was the first problem to be shown to be complete for **P** via logspace reductions [2].

We encode the path system problem as a problem over the signature σ_3 which consists of the relation symbol R of arity 3 and the constant symbols *source* and *sink*. A σ_3-structure $\mathcal{P}$ can be thought of as a path system where the vertices of the path system are given by $|\mathcal{P}|$, the source is given by *source*, the sink is given by *sink* and the rules of the path system are given by $\{(x, y, z) : R(x, y, z) \text{ holds in } \mathcal{P}\}$. Hence, we define the problem PS as

$$\begin{aligned}\{\mathcal{P} \in \mathrm{STRUCT}(\sigma_3) \quad : \quad & \text{the vertex } \textit{sink} \text{ is accessible from the vertex} \\ & \textit{source} \text{ in the path system } \mathcal{P}\}.\end{aligned}$$

Let us return to increasing the expressibility of FO. Corresponding to the problem PS is an operator of the same name. The logic $(\pm\mathrm{PS})^*[\mathrm{FO}]$, or *path system logic*, is the closure of FO under the usual first-order connectives and quantifiers and also the operator PS, with PS applied as follows.

Given a formula $\varphi(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in (\pm\mathrm{PS})^*[\mathrm{FO}]$ over the signature σ, where the variables of the k-tuples $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$, for some $k \geq 1$, are all distinct and free in φ, the formula Φ defined as $\mathrm{PS}[\lambda\mathbf{x}, \mathbf{y}, \mathbf{z}\varphi](\mathbf{u}, \mathbf{v})$, where $\mathbf{u}$ and $\mathbf{v}$ are k-tuples of (not necessarily distinct) constant symbols and variables, is also a formula of $(\pm\mathrm{PS})^*[\mathrm{FO}]$, with the free variables of Φ being those variables in $\mathbf{u}$ and $\mathbf{v}$ together with the free variables of φ different from those in the tuples $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$. If Φ is a sentence then it is interpreted in a structure $\mathcal{A} \in \mathrm{STRUCT}(\sigma)$ as follows. We build a path system with vertex set $|\mathcal{A}|^k$ and set of rules

$$\{(\mathbf{a}, \mathbf{b}, \mathbf{c}) \in |\mathcal{A}|^k \times |\mathcal{A}|^k \times |\mathcal{A}|^k : \varphi(\mathbf{a}, \mathbf{b}, \mathbf{c}) \text{ holds in } \mathcal{A}\},$$

and say that $\mathcal{A} \models \Phi$ if, and only if, the sink $\mathbf{v}$ is accessible in this path system from the source $\mathbf{u}$ (the semantics can easily be extended to arbitrary formulae of $(\pm\mathrm{PS})^*[\mathrm{FO}]$: see, for example, [4] for a more detailed semantic definition of operators such as PS). Note that $(\pm\mathrm{PS})^*[\mathrm{FO}]$ defines a class of problems over signatures not containing function symbols. Note also that there is nothing special about the problem PS: any problem can be converted into an operator and used to extend first-order logic. Syntactically, such logics are very similar although their semantics depend on the operator in hand.

It is indeed the case that we have increased expressibility as we can define problems in $(\pm\mathrm{PS})^*[\mathrm{FO}]$ which can not be defined in FO (a simple Ehrenfeucht-Fraïssé game shows that PS is not definable in FO: see [4] for more on such games). In the presence of a built-in successor relation, we can obtain a precise complexity-theoretic characterisation of the problems definable in $(\pm\mathrm{PS})^*[\mathrm{FO}]$. We say that we have a *built-in successor relation* if no matter over which signature we happen to be working, there is always a binary relation symbol *succ* and two constant symbols 0 and *max* available such that this relation symbol *succ* is always interpreted as a successor relation, of the form $\{(a_0, a_1), (a_1, a_2), \ldots, (a_{n-2}, a_{n-1})\}$, in a structure of size n, where all the a_i's are distinct and $a_0 = 0$ and $a_{n-1} = max$. Note that whether a structure satisfies a sentence in which the relation symbol *succ* or the constant symbols 0 or *max* appear might depend upon the particular successor relation chosen as the interpretation for *succ*. Consequently, we only consider those sentences of $(\pm\mathrm{PS})^*[\mathrm{FO}]$ with a built-in successor relation which define problems as being well-formed; that is, those sentences for which satisfaction is independent of the particular interpretation chosen for *succ*. We denote the logic $(\pm\mathrm{PS})^*[\mathrm{FO}]$ with a built-in successor relation by $(\pm\mathrm{PS})^*[\mathrm{FO}_s]$. As to whether $(\pm\mathrm{PS})^*[\mathrm{FO}_s]$ should really be called a logic is highly debatable (for example, it is undecidable as to whether a sentence of $(\pm\mathrm{PS})^*[\mathrm{FO}_s]$ is *order-invariant*, i.e., satisfies the property we want as regards *succ*, and so this 'logic' does not have a recursive syntax), and the reader is referred to [4] and [9] for a detailed discussion of this and related points.

Theorem 1 [12] *A problem over the signature σ is in* $\mathbf{P}$ *if, and only if, it can be defined in $(\pm PS)^*[FO_s]$. Moreover, any problem in $(\pm PS)^*[FO_s]$ can be defined by a sentence of the form*:

$$PS[\lambda\mathbf{x}, \mathbf{y}, \mathbf{z}\varphi(\mathbf{x}, \mathbf{y}, \mathbf{z})](\mathbf{0}, \mathbf{max}),$$

where: $|\mathbf{x}| = |\mathbf{y}| = |\mathbf{z}| = k$*, for some* $k \geq 1$; $\mathbf{0}$ (*resp.* $\mathbf{max}$) *is the constant symbol* 0 (*resp.* max) *repeated* k *times*; *and* φ *is a quantifier-free formula of* FO_s. □

Our notation for $(\pm\text{PS})^*[\text{FO}]$ is such that $\pm$ denotes the fact that applications of the operator PS can appear within the scope of negation signs and * denotes the fact that we are allowed to nest applications of PS as many times as we like. The fragment $(\pm\text{PS})^k[\text{FO}]$, for some $k \geq 1$, is obtained by allowing at most k nestings of applications of PS, and the fragment $\text{PS}^k[\text{FO}]$ is obtained by further disallowing any application of PS to appear within the scope of a negation sign. Hence, by Theorem 1, $\mathbf{P} = \text{PS}^1[\text{FO}_s]$.

The class of problems $(\pm\text{PS})^*[\text{FO}]$ is also intimately related with the class of problems accepted by certain program schemes which have access to a stack. A *program scheme* $\rho \in \text{NPSS}(1)$ involves a finite set $\{x_1, x_2, \ldots, x_k\}$ of *variables*, for some $k \geq 1$, and is over a signature σ'. It consists of a finite sequence of *instructions* where each instruction, apart from the first and the last, is one of the following:

- an *assignment instruction* of the form '$x_i := y$', where $i \in \{1, 2, \ldots, k\}$ and where y is a variable from $\{x_1, x_2, \ldots, x_k\}$, a constant symbol of σ' or one of the special constant symbols 0 and max which do not appear in any signature;

- an *assignment instruction* of the form '$x_i := F(y_1, y_2, \ldots, y_m)$', where: $i \in \{1, 2, \ldots, k\}$; each y_j is a variable from $\{x_1, x_2, \ldots, x_k\}$, a constant symbol of σ' or one of the special constant symbols 0 and max; and F is a function symbol of σ' of arity m;

- a *guess instruction* of the form 'GUESS x_i', where $i \in \{1, 2, \ldots, k\}$;

- a *while instruction* of the form 'WHILE t DO $\alpha_1; \alpha_2; \ldots; \alpha_q$ OD', where t is a quantifier-free formula of $\text{FO}(\sigma \cup \{0, max\})$, with σ the signature σ' minus any function symbols, whose free variables are from $\{x_1, x_2, \ldots, x_k\}$ and where each of $\alpha_1, \alpha_2, \ldots, \alpha_q$ is another instruction of a form given here (note that there may be nested while instructions); or

- a *stack instruction* of the form '$x_i :=$ POP' or 'PUSH x_i', where $i \in \{1, 2, \ldots, k\}$.

The first instruction of ρ is 'INPUT$(x_1, x_2, \ldots, x_l)$' and the last instruction is 'OUTPUT$(x_1, x_2, \ldots, x_l)$', for some l where $1 \leq l \leq k$. The variables $x_1, x_2, \ldots, x_l$ are the *input-output variables* of ρ, the variables $x_{l+1}, x_{l+2}, \ldots, x_k$ are the *free variables* of ρ and, further, any free variable of ρ never appears on the left-hand side of an assignment instruction, in a POP instruction nor in a guess instruction. Essentially, free variables appear in ρ as if they were constant symbols.

A program scheme $\rho \in \text{NPSS}(1)$ over σ' with s free variables, say, takes a σ'-structure $\mathcal{A}$ and s additional values from $|\mathcal{A}|$, one for each free variable of ρ, as input; that is, an expansion $\mathcal{A}'$ of $\mathcal{A}$ by adjoining s additional constants. The program scheme ρ computes on $\mathcal{A}'$ in the obvious way except that the POP and PUSH instructions provide access to a stack and:

- execution of the instruction 'GUESS x_i' non-deterministically assigns an element of $|\mathcal{A}|$ to the variable x_i;

- when the instruction 'PUSH x_i' is encountered in some program scheme, the value of the variable x_i is placed on the top of the stack (so increasing the height of the stack by 1) but so that x_i retains its value, and when the instruction '$x_i :=$ POP' is encountered, the value on the top of the stack is removed (so decreasing the height of the stack by 1) and the variable x_i assumes this value (if the stack is empty when the instruction '$x_i :=$ POP' is encountered then the computation halts);

- the constants 0 and max are interpreted as two arbitrary but distinct elements of $|\mathcal{A}|$; and

- initially, every input-output variable is assumed to have the value 0.

Note that throughout a computation of ρ, the value of any free variable does not change. The expansion $\mathcal{A}'$ of the structure $\mathcal{A}$ is *accepted* by ρ, and we write $\mathcal{A}' \models \rho$, if, and only if, there exists a computation of ρ on this expansion such that the output-instruction is reached with all input-output variables having the value max. (We can easily build the usual 'if' and 'if-then-else' instructions using while instructions: see, for example, [11]. Henceforth, we shall assume that these instructions are at our disposal.)

We want the sets of structures accepted by our program schemes to be problems, i.e., closed under isomorphism, and so we only ever consider program schemes ρ where a structure is accepted by ρ when 0 and max are given two distinct values from the universe of the structure if, and only if, it is accepted no matter which pair of distinct values is chosen for 0 and max. Let us reiterate: when we say that ρ is a program scheme of NPSS(1) we mean that ρ accepts a problem and the acceptance of any input structure does not depend upon the pair of distinct values we give to 0 and max. This is analogous to how we build a successor relation into a logic. Indeed, we can build a successor relation into our program schemes of NPSS(1), so as to obtain the class of program schemes $\text{NPSS}_s(1)$, or alternatively we can build two constants into our logics. As with our logics, we write NPSS(1) and $\text{NPSS}_s(1)$ to also denote the class of problems accepted by the program schemes of NPSS(1) and $\text{NPSS}_s(1)$, respectively. The reader is referred to [1] for more details on program schemes such as those of NPSS(1) and for some illustrative examples.

Theorem 2 [1]

(*a*) *A problem over some signature σ is in NPSS(1) if, and only if, it can be defined by a sentence of $(\pm PS)^*[FO]$ with two built-in constants, of the form*:

$$PS[\lambda \mathbf{x}, \mathbf{y}, \mathbf{z} \varphi(\mathbf{x}, \mathbf{y}, \mathbf{z})](\mathbf{0}, \mathbf{max}),$$

where: $|\mathbf{x}| = |\mathbf{y}| = |\mathbf{z}| = k$, *for some* $k \geq 1$; $\mathbf{0}$ (*resp.* $\mathbf{max}$) *is the constant symbol* 0 (*resp. max*) *repeated k times*; *and φ is quantifier-free first-order.*

(*b*) *A problem over some signature σ is in* **P** *if, and only if, it can be accepted by a program scheme of $NPSS_s(1)$.* □

It was also proven in [1] that the class of problems defined by the sentences of $(\pm\mathrm{PS})^*[\mathrm{FO}]$ with two built-in constants is identical to the class of problems accepted by a (proper infinite) hierarchy of classes of program schemes, the first level of which is NPSS(1) (hence our notation).

3 Deterministic path systems

Theorem 2 provides yet another characterisation of the complexity class **P**. However, this characterisation is different in flavour from most characterisations of **P** in that it equates **P** with the class of problems accepted by a 'non-deterministic' model of computation; namely the program schemes of $\mathrm{NPSS}_s(1)$. One question which immediately arises is: 'What can we say about the problems accepted by those program schemes of $\mathrm{NPSS}_s(1)$ in which the guess instruction does not appear?' The immediate response to this question is that without the ability to guess, no program scheme of $\mathrm{NPSS}_s(1)$ can accept any 'non-trivial' problem. However, by representing our built-in successor relation in a functional style we can make this question meaningful. Instead of having a built-in successor relation, let us assume that there is a built-in *successor function* and assignment instructions of the form '$x_i := succ(x_j)$' (of course, we still have 0 and max denoting the least and greatest elements of the ordering, respectively). Clearly, whether we have a built-in successor relation or a built-in successor function does not alter the class of problems accepted by the program schemes of $\mathrm{NPSS}_s(1)$.

Denote the class of program schemes of NPSS(1) in which the guess instruction does not appear by DPSS(1), with $\mathrm{DPSS}_s(1)$ defined likewise. Note that it makes no sense to consider program schemes of DPSS(1) over signatures involving only relation and constant symbols; as again no 'non-trivial' problems can be accepted by such program schemes. However, if the underlying signature σ' contains function symbols then we have assignment instructions of the form '$x_i := F(x_{j_1}, x_{j_2}, \ldots, x_{j_b})$', for every function symbol F of σ' of arity b. In such a situation, it does make sense to examine the class of problems accepted by the program schemes of DPSS(1).

In this section[1], we examine the classes of program schemes DPSS(1) and $\mathrm{DPSS}_s(1)$. In the subsequent section, we shall use results obtained in this section to give logical characterisations of **P** on certain classes of structures (where by 'logical' we mean not involving any sort of built-in relations; more precisely, 'logical' in the sense laid out in [4] and [9]).

We begin by defining a deterministic path system. A *deterministic path system* is a path system such that for every pair of vertices x and y (where possibly $x = y$), there is exactly one vertex z such that either (x, y, z) or (y, x, z) is a rule (this vertex z might be identical to either x or y). So, in a deterministic path system there is at most one new vertex which can be deduced as accessible from the known accessibility of any two vertices. The *deterministic path system problem* consists of all those deterministic path systems for which the sink is accessible from the source. Define the signature

[1] All results in this section were proven in collaboration with S.R. Chauhan: they are included here with her permission.

$\sigma_3' = \langle F, source, sink \rangle$, where F is a binary function symbol and *source* and *sink* are constant symbols. A σ_3'-structure $\mathcal{P}$ encodes a deterministic path system in a similar way to a σ_3-structure encoding a path system except that:

- there is a rule (x, y, z) if $F(x, y) = z = F(y, x)$ and $z \neq source$ (where possibly $z = x$ or $z = y$); and
- there is a rule $(x, y, source)$ otherwise.

The problem DetPS is defined as:

$$\{\mathcal{P} \in \text{STRUCT}(\sigma_3') \;:\; \text{the vertex } sink \text{ is accessible from the vertex } source \text{ in the deterministic path system } \mathcal{P}\}.$$

Intuitively, to decide whether a σ_3-structure is not in PS or whether a σ_3'-structure is not in DetPS, we need to know that at some point in the process of building the set of accessible vertices, every pair of *accessed* vertices, i.e., those vertices that have so far been shown to be accessible, has been checked so that no new vertices can be shown to be accessible from these accessed vertices, and that the sink has so far not been accessed. Hence, it appears to be necessary to dynamically build a set of accessed vertices and to keep a record of those pairs of accessed vertices which have already been checked. Later on in this section, we show that we can actually do this for deterministic path systems with a program scheme of DPSS(1) over σ_3'. We derive this program scheme by developing an algorithm (to be called *DFSearch*) to solve the deterministic path system problem and then by showing that this algorithm can be implemented in DPSS(1).

3.1 An informal algorithm

Consider the following (informally presented) algorithm, *DFSearch*, which takes a deterministic path system as input. In this algorithm, the order in which the vertices are accessed plays a critical role. During an execution of *DFSearch* on some input, at any time there is always exactly one accessed vertex which is described as *active*. An accessed vertex is the active vertex when it is the one currently being checked with each of the accessed vertices in turn in order to see whether a new vertex can be shown to be accessible (initially, *source* is the only accessed vertex, hence it is the active vertex). The main feature of this algorithm is that as soon as a new vertex, say x, is accessed, it becomes the active vertex and is checked with each accessed vertex in turn (including itself), not in any random order but *in the order in which these vertices were accessed* until: either a new vertex is accessed, say y, and y becomes the active vertex and we stop checking pairs involving x and start checking pairs involving y; or until x has been checked with all the vertices that were accessed before it, including itself. In the latter case, our new active vertex is taken to be the vertex z which was active at the time that x was accessed, and the next pair involving z is checked, after the pair which accessed x.

Example 3 Consider the following illustrative example (in our example, we do not stop if we show the sink to be accessible but continue to generate other accessible vertices: in fact, we do not even specify a sink). Suppose that our deterministic program scheme is such that the set of rules can be described according to Fig. 1 where the source is u (and where, for clarity, an ϵ denotes that the vertex made accessible by the corresponding pair is one of the vertices of the pair or u).

	u	v	w	x	y
u	w	ε	y	ε	ε
v	ε	ε	y	ε	w
w	y	y	x	ε	v
x	ε	ε	ε	ε	w
y	ε	w	v	w	w

Figure 1. A deterministic path system.

Our algorithm begins with u active and checks the pair $\{u, u\}$; with the result that w is shown to be accessible. Hence, the vertices so far shown to be accessible are, in order, u and w, with w now active.

According to our algorithm, we next check the pair $\{w, u\}$ which shows y to be accessible. Hence, the vertices so far shown to be accessible are, in order, u, w and y, with y now active.

According to our algorithm, we next check the pair $\{y, u\}$, which yields no new accessible vertex, and so we check the pair $\{y, w\}$, which shows v to be accessible. Hence, the vertices so far shown to be accessible are, in order, u, w, y and v, with v now active.

According to our algorithm, we next check the pairs $\{v, u\}$, $\{v, w\}$, $\{v, y\}$ and $\{v, v\}$, yielding no new accessible vertex; thus, we make y active (since it was active when v was shown to be accessible) and resume checking pairs involving y (and vertices accessed before y) starting from the pair $\{y, y\}$. This pair yields no new accessible vertex and so we make w active (since w was active when y was shown to be accessible) and resume checking pairs involving w (and vertices accessed before y) starting from the pair $\{w, w\}$, which shows x to be accessible. Hence, the vertices so far shown to be accessible are, in order, u, w, y, v and x, with x now active.

According to our algorithm, we next check the pairs $\{x, u\}$, $\{x, w\}$, $\{x, y\}$, $\{x, v\}$ and $\{x, x\}$, yielding no new accessible vertex. Thus, w becomes active. But all pairs involving w (and vertices accessed before w) have been checked, so u becomes active. However, all pairs involving u have been checked, so the algorithm halts.

Note that in this case, all accessible vertices are indeed shown to be accessible by our algorithm; and if we repeated the algorithm on our input then the vertices would be shown to be accessible in exactly the same order. $\square$

Our algorithm *DFSearch* can be looked upon as a sort of depth-first search in a deterministic path system; hence its name. However, the analogy is not exact as the 'depth-first search' is not given an *a priori* ordering of the elements upon which the search is performed (as is usually the case in a depth-first search in a graph): it computes the visit-order for itself as it progresses.

A less informal description of the algorithm than that above is given in Fig. 2. Throughout, we use x_0 to denote *source*. Also: we write $(x, y) \mapsto z$ to denote the fact that z is the unique vertex such that there is a rule (x, y, z) or (y, x, z) and z is different from x, y and x_0; and we write $(x, y) \mapsto \epsilon$ to denote the fact that is the unique vertex z such that there is a rule (x, y, z) or (y, x, z) is such that z is identical to one of x, y and x_0. If $(x, y) \mapsto z$ is used to show that z is accessible, given that x and y have already been shown to be accessible, then we say that x and y *access* z and that $(x, y) \mapsto z$ is *applied* to access z: in such a case, the vertex x will always be the active vertex. Also, given x and z, if x and y access z, for some y, then we say that (the active vertex) x *accesses* z.

```
1   suppose that $x_0, x_1, \ldots, x_i$ have been accessed so far and $x_i$ is active;
2   check the ordered pairs $(x_i, x_0), (x_i, x_1), \ldots$ in turn;
3   IF $(x_i, x_j) \mapsto x_{i+1}$ where $x_{i+1}$ is a vertex not yet accessed THEN
4     IF $x_{i+1}$ = $sink$ THEN
5       ACCEPT;
6     ELSE
7       add $x_{i+1}$ to our list of vertices accessed so far;
8       make $x_{i+1}$ the active vertex and repeat from line 2 (with $x_{i+1}$
9       replacing $x_i$) and starting with the pair $(x_{i+1}, x_0)$;
10    FI
11  ELSE
12    it must be the case that each pair $(x_i, x_j)$, for all $j \leq i$, has
13    been checked and nothing new has been shown to be accessible;
14    find the pair $(x_{i_1}, x_{j_1})$ such that $(x_{i_1}, x_{j_1}) \mapsto x_i$ was applied to
15    access $x_i$;
16    make $x_{i_1}$ active;
17    IF $x_{i_1} = x_0$ THEN
18      REJECT;
19    ELSE
20      repeat from line 2 starting from the pair $(x_{i_1}, x_{j_1+1})$;
21      FI
22    FI
```

Figure 2. A less informal description of our algorithm *DFSearch*.

3.2 Proving our algorithm correct

Henceforth, we equate the algorithm *DFSearch* with the description in Fig. 2. The following lemmas are used to prove that *DFSearch* solves the deterministic path system problem. In these lemmas, we write: x_i to denote that it is the ith vertex to be accessed during an execution of *DFSearch*; and *AccessedSet* to denote the set of vertices shown to be accessible by the algorithm *DFSearch* (*AccessedSet* can be regarded as being dynamically constructed, starting off as $\{x_0\}$ and ending up as the set of vertices shown to be accessible by *DFSearch*).

The following lemma proves that if we place the vertices accessed by the algorithm *DFSearch* in a line in the order they are accessed and we draw, above the line, a directed arc from vertex x to vertex y if vertex x accesses vertex y then no two arcs cross.

Lemma 4 *Consider an execution of DFSearch so that the algorithm terminates with AccessedSet* $= \{x_0, x_1, \ldots, x_k\}$. *Suppose that* x_i *accesses* x_{i+r}, *for some* i *such that* $0 \leq i \leq k-2$ *and for some* $r \geq 2$. *Then it is not the case that* x_{i-t} *accesses* x_{i+s}, *for any* s *and* t *such that* $0 < s < r$ *and* $0 < t \leq i$.

Proof Since x_i accesses x_{i+r}, let x_u be the paired vertex such that $(x_i, x_u) \mapsto x_{i+r}$ is applied to access x_{i+r}. Assume that the statement in the lemma is false and let s be the minimal such s for which some x_{i-t} accesses x_{i+s}. Note that when x_i is accessed, it becomes the active vertex and the pairs $(x_i, x_0), (x_i, x_1), \ldots$ are checked in turn until either $(x_i, x_j) \mapsto x_{i+1}$ is applied to access x_{i+1}, for some j, or (x_i, x_i) has been checked and (x_i, x_i) accesses nothing. Since x_i accesses x_{i+r}, we must have that x_i accesses x_{i+1}, and so $s \geq 2$. In fact, by hypothesis, every x_p for which $i < p < i+s$ is accessed by some x_q for which $i \leq q < p$. Putting $s_0 = i + (s-1) > i$, x_{s_0} is accessed by some x_{s_1} such that $i \leq s_1 < s_0$; x_{s_1} is accessed by some x_{s_2} such that $i \leq s_2 < s_1$; and so on until x_{s_v}, for some $v \geq 0$, is accessed by x_i.

When $x_{s_0} (= x_{i+(s-1)})$ is accessed, it becomes active. As x_{i-t} accesses x_{i+s}, x_{s_0} accesses no vertices, and x_{s_1} becomes active. Again, as x_{i-t} accesses x_{i+s}, x_{s_1} accesses no vertices, and x_{s_2} becomes active; and so on until x_i becomes active. Note that the pair (x_i, x_u) has not yet been checked as otherwise the element x_{i+s} would have been accessed. Hence, $x_i \mapsto x_{i+s}$ is applied to access x_{i+s} which yields a contradiction. $\square$

As soon as x_i is accessed, *DFSearch* starts to check the pairs $(x_i, x_0), (x_i, x_1), \ldots,$ (x_i, x_i) in turn. If at some time t during the execution of *DFSearch*, all the pairs have been checked then we say that x_i *is fully checked at time* t. Note that once a vertex becomes fully checked, it stays fully checked.

Lemma 5 *Consider an execution of DFSearch so that the algorithm terminates with AccessedSet* $= \{x_0, x_1, \ldots, x_k\}$. *Suppose that* x_i *accesses* x_{i+r} *where* $r \geq 1$. *Then at the time that* x_{i+r} *is accessed, all the vertices* x_p *with* $i < p < i+r$ *are fully checked.*

Proof We may assume that $r > 1$. We give a proof using induction, where our induction hypothesis IH(j) is as follows: 'At the time that x_i accesses x_{i+r}, all the vertices x_p with $i < j \leq p < i+r$ are fully checked'.

The base case of our induction is when $j = i + r - 1$. Since it is not the case that x_{i+r-1} accesses x_{i+r}, it must be the case that x_{i+r-1} does not access any new vertices and hence becomes fully checked before x_{i+r} is accessed. Thus, the induction hypothesis holds for the base case.

Suppose that IH$(j+1)$ holds, where $j \neq i$. The vertex x_j is accessed before x_{i+r}. Either x_j does not access a new vertex or x_j accesses x_{j+1}. If the former is true then we are done since x_j is fully checked before x_{i+r} is accessed. If the latter is true then x_j accesses at least one new vertex. Let x_s be any vertex such that x_j accesses x_s. By Lemma 4, $s < i + r$ and, by hypothesis, x_s is fully checked before x_{i+r} is accessed. When x_s is fully checked, *DFSearch* resumes checking the vertex that accessed x_s; that is, x_j. Let x_q be the last vertex such that x_j accesses x_q. Since x_j does not access any more new vertices, *DFSearch* continues checking x_j until it is fully checked; and at this point x_{i+r} is still to be accessed. Hence the result follows by induction. □

We can now obtain the following corollary.

Corollary 6 *Consider an execution of DFSearch. Suppose that at time t, AccessedSet $= \{x_0, x_1, \ldots, x_{i+r}\}$ and the vertex x_i is active, where $r \geq 1$. Then at time t, all vertices x_p with $i < p \leq i + r$ are fully checked.*

Proof Suppose that x_j accesses x_{i+r}. When x_{i+r} is accessed, it becomes active; and as x_i is active at time t, x_{i+r} accesses no new vertices before becoming fully checked. At this time (when x_{i+r} is fully checked, which is before time t), by Lemma 5, the vertices of $\{x_p : j < p \leq i + r\}$ are fully checked. If $j \leq i$ then we are done.

Suppose that $i < j$. After x_{i+r} becomes fully checked, x_j becomes active. As x_i is active at (the later) time t, x_j becomes fully checked. Suppose that x_{j_1} accesses x_j. By Lemma 5, when x_j is accessed, which is before time t, the vertices of $\{x_p : j_1 < p < j\}$ are fully checked. If $j_1 \leq i$ then we are done.

Continuing as above, we obtain that there exists some x_{j_k} such that: x_{j_k} accesses $x_{j_{k-1}}$; the vertices of $\{x_p : j_k < p \leq i+r\}$ are fully checked at some time not later than time t; and $j_k \leq i$. Hence, the result follows. □

Now we can prove the correctness of our algorithm.

Proposition 7 *The algorithm DFSearch solves the deterministic path system problem.*

Proof Consider the execution of *DFSearch* with some deterministic path system as input. Initially, *DFSearch* starts with *AccessedSet* consisting only of the source, and if any more vertices are added then they must have been accessed by vertices which have already been placed in *AccessedSet*. Hence, *AccessedSet* only contains vertices which are accessible from the source. Suppose that *DFSearch* accepts its input. Then the sink is accessed from vertices in *AccessedSet*, and so the input is a deterministic path system in which the sink is accessible from the source.

Conversely, suppose that *DFSearch* rejects its input and that *AccessedSet* $= \{x_0, x_1, \ldots, x_k\}$ on termination. For termination to occur, either $k = 0$ or x_0 must have become active again. If $k = 0$ then clearly the input is a deterministic path system in which the sink is not accessible from the source; so assume that x_0 becomes active again. By Corollary 6, at the time that x_0 becomes active again, all the vertices of $\{x_0, x_1, \ldots, x_k\}$ are fully checked. Hence, *AccessedSet* consists of all those vertices which can be shown to be accessible from the source, and the sink is not in *AccessedSet*; that is, the input is a deterministic path system in which the sink is not accessible from the source. □

3.3 Implementing our algorithm

Now that we have developed the algorithm *DFSearch* to solve the deterministic path system problem, let us reconsider the demands on any DPSS(1) program scheme which might implement *DFSearch*. Firstly, it will need to build a set of accessed vertices, *AccessedSet*, and then retrieve vertices from the set in the order in which they were inserted; and it must do this where the only additional storage is the stack. Secondly, it will need to check whether a vertex is already in the set *AccessedSet*. Thirdly, for any accessed vertex it should be able to ascertain the pair from which this vertex was accessed. As we shall see, it is non-trivial to implement *DFSearch* in DPSS(1).

However, we now describe such an implementation of the algorithm *DFSearch*; that is, a program scheme $\rho_0 \in$ DPSS(1) over σ_3' which solves the problem DetPS. The structure of our program scheme ρ_0 is that it consists of the instruction '`PUSH` x_0' followed by one while-loop which loops until the input structure is either accepted or rejected. Changes are made to the stack (starting from an empty stack) during each while-loop iteration such that for any iteration, the changes to the stack are determined by the top (at most) two stack elements, and these changes only involve the top two stack elements with possibly one extra element being pushed onto the stack. Consequently, we describe the program scheme ρ_0 using the table in Fig. 3 (the notation, and underlying encoding, used in Fig. 3 is explained shortly). The 'pre-loop' column shows the top two stack elements, where c is the height of the stack; and the 'post-loop' column shows how the stack changes during one iteration, given the 'pre-loop' conditions. So, our program scheme essentially repeatedly applies the operations specified in each row depending upon the current conditions, as defined in the 'condition' column.

We give each row in the table in Fig. 3 a number. Let i be a row in our table and let β be a stack configuration (that is, the contents of the stack) which satisfies the 'pre-loop' condition of row i. We say that β *satisfies* row i or that row i *holds* for β. In addition, if ρ_0 is such that, prior to an iteration of the while-loop, β satisfies row i then any changes made to β in this iteration are said to be *by* or *via* row i, and we say that row i *is applied*. Note that the rows in the table in 3 are mutually exclusive, i.e., any stack configuration can only satisfy at most one rule; and every possible combination of a pair of stack items is considered in the table.

We now give a definition of the stack items that are introduced in Fig. 3. Note that in the actual program scheme ρ_0 a suitable encoding scheme is used so as to realise the different types of stack item below. Let the input to our program scheme be the σ_3'-structure $\mathcal{P}$. We have stack items of the following types:

(*i*) p, where $p \in |\mathcal{P}|$;

(ii) $\langle p, q, r\rangle$, where $p, q, r \in |\mathcal{P}|$; and

(iii) $[p]$, where $p \in |\mathcal{P}|$.

	Top 2 items on stack pre-loop		Top 3 items on stack post-loop			Condition satisfied by stack pre-loop
Row	$c-1$	c	$c-1$	c	$c+1$	
1						
1.1	$--$	*item*	$--$	*item*	x_0	only one item
2						
2.1	p	q	ACCEPT			$(p, q) \mapsto sink$
2.2	p	q	$\langle p, q, r\rangle$	$--$	$--$	$(p, q) \mapsto r \wedge r \neq sink$
2.3	p	p	$[p]$	$--$	$--$	$(p, p) \mapsto \epsilon \wedge p \neq x_0$
2.4	x_0	x_0	REJECT			$(x_0, x_0) \mapsto \epsilon$
2.5	p	q	p	q	x_0	$(p, q) \mapsto \epsilon \wedge p \neq q$
2.6	p	*item*	p	*item*	x_0	*item* not of type (i)
3						
3.1	$\langle p, q, r\rangle$	$\langle p, q, r\rangle$	r	$--$	$--$	
3.2	$\langle p, q, r\rangle$	$\langle p', q', r\rangle$	p	q	x_0	$(p \neq p' \vee q \neq q') \wedge p \neq q$
3.3	$\langle p, p, r\rangle$	$\langle p', q', r\rangle$	$[p]$	$--$	$--$	$(p \neq p' \vee p \neq q')$
3.4	$\langle p, q, r\rangle$	*item*	$\langle p, q, r\rangle$	*item*	x_0	$item \neq \langle p', q', r\rangle, \forall p', q'$
4						
4.1	$[r]$	$\langle p, q, r\rangle$	p	q	x_0	$p \neq q$
4.2	$[r]$	$\langle p, p, r\rangle$	$[p]$	$--$	$--$	$p \neq x_0$
4.3	$[r]$	$\langle x_0, x_0, r\rangle$	REJECT			
4.4	$[r]$	*item*	$[r]$	*item*	x_0	$item \neq \langle p, q, r\rangle, \forall p, q$

Figure 3. The program scheme ρ_0.

As an example of an encoding scheme alluded to above, we might encode the stack item: $p \in |\mathcal{P}|$ as the 6 stack items u, u, u, p, p, p, for some fixed $u \in |\mathcal{P}|$; $\langle p, q, r\rangle$, where $p, q, r \in |\mathcal{P}|$, as the 6 stack items u, v, v, p, q, r, for some fixed $u, v \in |\mathcal{P}|$ such that $u \neq v$; and $[p]$, where $p \in |\mathcal{P}|$, as the 6 stack items u, v, u, p, p, p, for some fixed $u, v \in |\mathcal{P}|$ such that $u \neq v$. Consequently, popping an 'item' from the stack, for example, really means popping 6 elements, u_1, u_2, u_3, u_4, u_5 and u_6, from the stack and ascertaining, using u_1, u_2 and u_3, the type of the encoded item, with u_4, u_5 and u_6 yielding the parameters of the item.

Having described our program scheme $\rho_0 \in$ DPSS(1) (it is clear that the above description can be implemented in DPSS(1)), let us now set about proving that it is an implementation of the algorithm *DFSearch*.

Some definitions are in order so that we might reason about stacks. Consider the computation of ρ_0 given some σ_3'-structure $\mathcal{P}$ as input. A *stack configuration* simply consists of the contents of the stack at some particular point in the computation. If the stack configuration $\beta = (\beta(1), \beta(2), \ldots, \beta(m))$ (with $\beta(m)$ the top item) then

$ht(\beta) = m$ and the element at *position* i, for $1 \leq i \leq ht(\beta)$, is $\beta(i)$ (the height of the empty stack configuration is 0). The *stack trace* of ρ on input $\mathcal{P}$ is the sequence of stack configurations in the order in which they occur when the flow of control of the execution of ρ_0 on input $\mathcal{P}$ is frozen immediately before executing the while-loop and then immediately after every iteration of the while-loop (and so the first non-empty stack configuration of any stack trace is (x_0)). That is, we do not consider the stack manipulations performed *during* an iteration of the while-loop, but focus on the stack only immediately *after* the iteration. Note that it is conceivable that a stack trace might be infinite; that is, ρ_0 might not halt on input $\mathcal{P}$. In fact, this is never the case but until we have proved this assertion, we must assume that infinite stack traces are possible. The ith stack configuration in the stack trace Σ is Σ_i, and the indices of the stack configurations yield a notion of time; that is, we say that the stack configuration Σ_i is the configuration at time i. If $i < j$ then we say that Σ_i *evolves to* Σ_j. If α and β are stack configurations of heights i and j, respectively, then we denote the stack configuration $(\alpha(1), \alpha(2), \ldots, \alpha(i), \beta(1), \beta(2), \ldots, \beta(j))$ by $\alpha + \beta$; and if x is some stack item then we denote the stack configuration $(x, \alpha(1), \alpha(2), \ldots, \alpha(i))$ by $x + \alpha$.

Before proving that the program scheme ρ_0 simulates our algorithm *DFSearch*, we give an example which illustrates the design of and the philosophy behind the program scheme ρ_0 (in relation to the algorithm *DFSearch*).

Example 8 Consider the deterministic path system $\mathcal{P}$ described in Fig. 4, whose source we take as the vertex u. We shall consider the execution of the program scheme ρ_0 on $\mathcal{P}$. To get the most from our example, we shall not specify a sink in our program scheme but simply let the program scheme run until the input is rejected (if there is no sink then it can never be shown to be accessible).

	u	v	w	y
u	w	ε	y	ε
v	ε	ε	ε	ε
w	y	ε	v	ε
y	ε	ε	ε	ε

Figure 4. A deterministic path system.

We portray the execution of ρ_0 on $\mathcal{P}$ in Fig. 5. In this figure, we depict the stack trace of the execution. Each stack configuration is represented as a column of elements and the row applied in order to alter the stack at any configuration is denoted as a superscript to the symbol $\rightarrow$. Some stack configurations are given a breakpoint number (written below the stack) which we shall use below in our description of the execution.

Initially, u is the only vertex so far shown to be accessible; and this is signalled by the stack configuration initially consisting solely of the item u.

$$
\begin{array}{cccccccccc}
 & & & & & & & & u & \\
 & 1.1 & u & 2.2 & & 1.1 & u & 3.4 & u & 2.2 \\
u & \to & u & \to & \langle u,u,w\rangle & \to & \langle u,u,w\rangle & \to & \langle u,u,w\rangle & \to \\
 & & & & 1 & & & & &
\end{array}
$$

$$
\begin{array}{cccccccccc}
\langle u,u,w\rangle & 3.1 & & 1.1 & u & 2.2 & & 1.1 & u & 3.4 \\
\langle u,u,w\rangle & \to & w & \to & w & \to & \langle w,u,y\rangle & \to & \langle w,u,y\rangle & \to \\
2 & & & & 3 & & & & &
\end{array}
$$

$$
\begin{array}{cccccccccc}
 & & & & & & u & & & \\
u & & & & u & & u & & \langle u,u,w\rangle & \\
u & 2.2 & \langle u,u,w\rangle & 3.4 & \langle u,u,w\rangle & 3.4 & \langle u,u,w\rangle & 2.2 & \langle u,u,w\rangle & 3.1 \\
\langle w,u,y\rangle & \to & \langle w,u,y\rangle & \to & \langle w,u,y\rangle & \to & \langle w,u,y\rangle & \to & \langle w,u,y\rangle & \to
\end{array}
$$

$$
\begin{array}{cccccccccc}
 & & u & & & & & & & \\
w & 3.4 & w & 2.2 & \langle w,u,y\rangle & 3.1 & & 1.1 & u & 2.5 \\
\langle w,u,y\rangle & \to & \langle w,u,y\rangle & \to & \langle w,u,y\rangle & \to & y & \to & y & \to \\
 & & & & 4 & & & & 5 &
\end{array}
$$

$$
\begin{array}{cccccccccc}
 & & & & & & u & & & \\
u & & & & u & & u & & \langle u,u,w\rangle & \\
u & 2.2 & \langle u,u,w\rangle & 2.6 & \langle u,u,w\rangle & 3.4 & \langle u,u,w\rangle & 2.2 & \langle u,u,w\rangle & 3.1 \\
y & \to & y & \to & y & \to & y & \to & y & \to
\end{array}
$$

$$
\begin{array}{cccccccccc}
 & & u & & & & & & & \\
w & 2.5 & w & 2.2 & & 3.1 & y & 2.3 & & 1.1 \\
y & \to & y & \to & \dots & \to & y & \to & [y] & \to \\
6 & & & & & & 7 & & &
\end{array}
$$

$$
\begin{array}{cccccccccc}
 & & u & & & & & & & \\
u & 4.4 & u & 2.2 & \langle u,u,w\rangle & 4.4 & & 2.2 & \langle w,u,y\rangle & 4.1 \\
[y] & \to & [y] & \to & [y] & \to & \dots & \to & [y] & \to \\
 & & & & & & & & 8 &
\end{array}
$$

$$
\begin{array}{cccccccccc}
u & & & & & & & & & \\
u & 2.2 & & 3.1 & w & 2.2 & & 1.1 & u & 3.4 \\
w & \to & \dots & \to & w & \to & \langle w,w,v\rangle & \to & \langle w,w,v\rangle & \to \\
 & & & & 9 & & & & &
\end{array}
$$

$$
\begin{array}{cccccccccc}
u & & & & & & & & & \\
u & 2.2 & & 2.2 & \langle w,w,v\rangle & 3.1 & & 1.1 & u & 2.5 \\
\langle w,w,v\rangle & \to & \dots & \to & \langle w,w,v\rangle & \to & v & \to & v & \to \\
 & & & & 10 & & & & &
\end{array}
$$

$$
\begin{array}{cccccccccc}
u & & & & & & & & & \\
u & 2.2 & & 3.1 & v & 2.3 & & 1.1 & u & 4.4 \\
v & \to & \dots & \to & v & \to & [v] & \to & [v] & \to \\
 & & & & 11 & & & & &
\end{array}
$$

$$
\begin{array}{cccccccccc}
u & & & & & & & & & \\
u & 2.2 & & 2.2 & \langle w,w,v\rangle & 4.2 & & 1.1 & u & 4.4 \\
[v] & \to & \dots & \to & [v] & \to & [w] & \to & [w] & \to \\
 & & & & 12 & & & & &
\end{array}
$$

$$
\begin{array}{ccccc}
u & & & & \\
u & 2.2 & \langle u,u,w\rangle & 4.3 & \\
[w] & \to & [w] & \to & \text{halt} \\
 & & 13 & &
\end{array}
$$

Figure 5. The stack trace of ρ_0.

The computation begins so that the stack configuration evolves until it consists of one item, the item $\langle u, u, w\rangle$ (at breakpoint 1): this comes about because $(u, u) \mapsto w$. Essentially, this configuration is interpreted as 'it may be the case that w is the next vertex to be shown to be accessible (but we must confirm this)'. The stack configuration now evolves so that the whole computation, from the start, is repeated 'above' the item $\langle u, u, w\rangle$ which remains at the bottom of the stack.

This evolution continues until a stack configuration of the form $(\langle u, u, w\rangle, \langle -, -, w\rangle)$ comes about (such a circumstance is at breakpoint 2). Generally: if the two items are different then w must have already been shown to be accessible; otherwise, they are the same and w has not so far been shown to be accessible. At breakpoint 2, the latter case holds and so w is made accessible, an event which is signalled by the stack configuration consisting solely of the item w.

The stack configuration now evolves so that the whole computation is repeated above the item w, until a previously accessed vertex is reached. This happens at breakpoint 3 when the stack configuration is (w, u). As $(w, u) \mapsto y$, the stack evolves so that it consists solely of the item $\langle w, u, y\rangle$ which signals that 'it may be the case that y is the next vertex to be shown to be accessible'. The stack configuration now evolves so that the whole computation is repeated above the item $\langle w, u, y\rangle$ until a stack configuration of the form $(\langle w, u, y\rangle, \langle -, -, y\rangle)$ comes about. This happens at breakpoint 4 when the configuration is $(\langle w, u, y\rangle, \langle w, u, y\rangle)$, which signals that y has not previously been shown to be accessible. The vertex y is now made accessible.

The stack configuration now evolves so that the whole computation is repeated above the item y, until a previously accessed vertex is reached: this happens at breakpoint 5. In this case, $(y, u) \mapsto \epsilon$ and so we continue the repetition, again until a previously accessed vertex is reached (at breakpoint 6). Just as before, $(y, w) \mapsto \epsilon$, and so we yet again continue the repetition. We eventually reach the stack configuration (y, y) (at breakpoint 7). As $(y, y) \mapsto \epsilon$, the stack configuration evolves into $([y])$ which is interpreted as 'all pairs of the form $(y, -)$, where the second component ranges over previously accessed vertices, have been checked and no potentially new accessible vertices obtained'. The computation now evolves so that the whole computation is repeated above the item $[y]$ until a stack configuration of the form $([y], \langle -, -, y\rangle)$ comes about. This happens at breakpoint 8 when the stack configuration is $([y], \langle w, u, y\rangle)$. This signals that the pair of vertices which accessed y was (w, u).

The stack configuration now evolves into (w, u, u), as if it were the case that $(w, u) \mapsto \epsilon$. Of course, in reality $(w, u) \mapsto y$ but, given that ρ_0 is intended to simulate the algorithm *DFSearch*, we wish ρ_0 to search for the vertex accessed *after* the vertex u, and then pair this vertex with the vertex w. This means repeating the computation above w, from the stack configuration (w, u, u), until the next vertex accessed is obtained. The next vertex accessed after vertex u is vertex w and the stack configuration evolves into (w, w) (at breakpoint 9). As $(w, w) \mapsto v$, the stack configuration now evolves into $(\langle w, w, v\rangle)$ (with the interpretation similar to that above). As v has previously not been shown to be accessible, the stack configuration evolves into $(\langle w, w, v\rangle, \langle w, w, v\rangle)$ (at breakpoint 10) and then to (v).

As $(v, u) \mapsto \epsilon$, $(v, w) \mapsto \epsilon$, $(v, y) \mapsto \epsilon$ and $(v, v) \mapsto \epsilon$, the whole computation is repeated above v until the stack configuration evolves to (v, v) (at breakpoint 11); and then to $([v])$. The whole computation is then repeated above $[v]$ in order to ascertain the pair of vertices which accessed v: this comes about at breakpoint 12 when the stack configuration is $([v], \langle w, w, v\rangle)$. The stack configuration now evolves into $([w])$ and the whole computation is then repeated above $[w]$ in order to ascertain the pair of vertices which accessed w: this comes about at breakpoint 13 when the stack configuration is $([w], \langle u, u, w\rangle)$. The execution now halts. Note that this execution is indeed a simulation of the algorithm *DFSearch*. □

The following lemmas will be used to show that *DFSearch* can be implemented as a program scheme of DPSS(1).

Lemma 9 *Let $\mathcal{P}$ be a σ_3'-structure and let Σ be the stack trace of ρ_0 on input $\mathcal{P}$. Suppose that $\Sigma_i = (\iota)$, for some $i > 1$ and for some stack item $\iota \neq x_0$. Then there exists k such that $i < k$ and:*

- $\Sigma_{i+j} = \Sigma_i + \Sigma_j$, *for all* $j \in \{1, 2, \ldots, k-i\}$;
- $ht(\Sigma_k) = 2$; *and*
- Σ_k *evolves to* Σ_{k+1} *by one of the rows* 2.1 – 2.4, 3.1 – 3.3 *and* 4.1 – 4.3.

Proof We have that $\Sigma_1 = (x_0)$ and $\Sigma_2 = (x_0, x_0)$, and, by Fig. 3, $\Sigma_{i+1} = (\iota, x_0)$ and $\Sigma_{i+2} = (\iota, x_0, x_0)$. The application of any row is only dependent upon the top two stack items and only alters the (at most) top two stack items (although a further item might be pushed onto the stack or the height of the stack might be lessened by 1). Let m be the least m such that $m > 1$ and $ht(\Sigma_m) = 1$ (we know that such an m exists as $ht(\Sigma_i) = 1$). Then $\Sigma_{i+1} = \iota + \Sigma_1, \Sigma_{i+2} = \iota + \Sigma_2, \ldots, \Sigma_{i+m} = \iota + \Sigma_m$.

If Σ_{i+m} evolves to Σ_{i+m+1} by one of the rows 2.5, 2.6, 3.4 and 4.4 then, by Fig. 3, $\Sigma_{i+m+1} = \iota + \Sigma_{m+1}$ with $ht(\Sigma_{i+m+1}) = 3$. Thus, we may assume that Σ_i evolves to $\Sigma_k = (\iota, item)$ where $\Sigma_{i+j} = \Sigma_i + \Sigma_j$, for all $j \in \{1, 2, \ldots, k-i\}$, and Σ_k satisfies one of rows 2.1 – 2.4, 3.1 – 3.3 and 4.1 – 4.3. The result follows. □

Lemma 10 *Let $\mathcal{P}$ be a σ_3'-structure and let Σ be the stack trace of ρ_0 on input $\mathcal{P}$. Fix $i \geq 1$ and define:*

$$T(i) = \{t : 1 \leq t \leq i \text{ and } \Sigma_t = (p), \text{ for some } p \in |\mathcal{P}|\},$$

with $T(i)$ ordered as $t_0 < t_1 < \ldots < t_k$, for some $k \geq 0$. Suppose that $\Sigma_{t_j} = (x_j)$, for all $j = 0, 1, \ldots, k$, and that at time i, AccessedSet $= \{x_0, x_1, \ldots, x_k\}$. Suppose further that $\Sigma_i = (x_l, x_m, x_0)$, for some $l, m \in \{0, 1, \ldots, k\}$ where $m < l$. Then Σ_i evolves to Σ_s where:

- $\Sigma_{i+j} = x_l + \Sigma_{(t_m+1)+j}$, *for all* $j = 1, 2, \ldots, s-i$; *and*
- $\Sigma_s = (x_l, x_{m+1})$.

Proof We have that $\Sigma_{t_m} = (x_m)$ and $\Sigma_{t_m+1} = (x_m, x_0)$, and so $\Sigma_i = x_l + \Sigma_{t_m+1}$. By Table 3, the application of any row is only dependent upon the top two stack items and only alters the (at most) top two stack items (although a further item might be pushed onto the stack or the height of the stack might be lessened by 1). Consequently, $\Sigma_{i+1} = x_l + \Sigma_{(t_m+1)+1}, \Sigma_{i+2} = x_l + \Sigma_{(t_m+1)+2}, \ldots, \Sigma_{i+t} = x_l + \Sigma_{(t_m+1)+t}$, for some t such that $ht(\Sigma_{(t_m+1)+t}) = 1$ (we know that such a t exists as $m < l$ and $t_m < t_{m+1} < i$).

If Σ_{i+t} evolves to Σ_{i+t+1} by row 2.6 then $\Sigma_{i+t+1} = x_l + \Sigma_{(t_m+1)+t+1}$ and we can continue as above (as $ht(\Sigma_{i+t+1}) = 3$). Hence, Σ_i evolves to Σ_s where $\Sigma_{i+j} = x_l + \Sigma_{(t_m+1)+j}$, for all $j = 1, 2, \ldots, s-i$, and $\Sigma_s = (x_l, x_{m+1})$. □

Lemma 11 *Let $\mathcal{P}$ be a σ_3'-structure and let Σ be the stack trace of ρ_0 on input $\mathcal{P}$. Let i be such that $\Sigma_i = (\langle p, q, r\rangle)$, for some $p, q, r \in |\mathcal{P}|$. Then there exists k such that $i < k$ and:*

- *$\Sigma_{i+j} = \Sigma_i + \Sigma_j$, for every $j = 1, 2, \ldots, k-i$; and*
- *$\Sigma_k = (\langle p, q, r\rangle, \langle p', q', r\rangle)$, for some $p', q' \in |\mathcal{P}|$.*

Proof By Lemma 9, Σ_i evolves by repeating the computation of ρ_0 on input $\mathcal{P}$ 'above' $\langle p, q, r\rangle$ until the stack height is 2 and one of the rows 2.1 – 2.4, 3.1 – 3.3 and 4.1 – 4.3 is to be applied; that is, in this case, one of the rows 3.1 – 3.3. The result follows. □

Now for our proof that the program scheme ρ_0 implements the algorithm *DFSearch*.

Theorem 12 *For every σ_3'-structure $\mathcal{P}$, the algorithm DFSearch accepts the deterministic path system encoded by $\mathcal{P}$ if, and only if, $\mathcal{P} \models \rho_0$. Hence, the program scheme ρ_0 accepts the problem DetPS.*

Proof Suppose that on input (the deterministic path system encoded by) $\mathcal{P}$, the algorithm *DFSearch* halts with $AccessedSet = \{x_0, x_1, \ldots, x_k\}$, for some $k \geq 0$, and that these vertices have been shown to be accessible in the order given. There are numerous distinguished events in the computation of *DFSearch* on input $\mathcal{P}$, namely: the events when the different vertices are shown to be accessible (line 7 of Fig. 2); the events when pairs of accessible vertices are checked to see whether a new vertex might be accessed (line 2 of Fig. 2); and the events when the search is embarked upon for the pair of vertices that was used to show that a vertex is accessible (line 14 of Fig. 2). These events are all distinct and have associated with them time-stamps denoting when they occur. Let these (finitely-many distinct) time-stamps be ordered as:

$$t_1 < t_2 < t_3 < \ldots$$

(obviously: t_1 is the time-stamp when x_0 is assumed to be accessible; t_2 is the time-stamp associated with the event when the pair (x_0, x_0) is checked; t_3 is the time-stamp associated with the event when x_1 is shown to be accessible, unless $k = 0$; and so on).

In order to prove our theorem we shall proceed by induction. Let Σ be the stack trace of ρ_0 on input $\mathcal{P}$. Our induction hypothesis IH(i) is as follows: 'There exist non-zero natural numbers $s_1 < s_2 < \ldots < s_i$ such that for each $j \in \{1, 2, \ldots, i\}$:

- if t_j is the time-stamp associated with the event when x_l is shown to be accessible then $\Sigma_{s_j} = (x_l)$;
- if t_j is the time-stamp associated with the event when the pair (x_l, x_m) is checked to see whether a new vertex might be accessed then $\Sigma_{s_j} = (x_l, x_m)$;
- if t_j is the time-stamp associated with the event when a search is embarked upon for the pair (x_{l_1}, x_{m_1}) that was used to show that x_l is accessible then $\Sigma_{s_j} = ([x_l])$; and
- if s is such that $1 \leq s \leq s_i$ but $s \notin \{s_1, s_2, \ldots, s_i\}$ then $\Sigma_s \neq (y)$, $\Sigma_s \neq (y, z)$ and $\Sigma_s \neq ([y])$, for any $y, z \in |\mathcal{P}|$.'

The base cases of the induction, when $i = 1$ and $i = 2$, are immediate simply by following the first few steps of the computation of ρ_0 on input $\mathcal{P}$.

Suppose that the induction hypothesis IH(i) holds, for some $i \geq 1$. There are three possibilities:

(1) $\Sigma_{s_i} = (x_l)$, for some l;

(2) $\Sigma_{s_i} = (x_l, x_m)$, for some l and m; and

(3) $\Sigma_{s_i} = ([x_l])$, for some l.

Case (1) $\Sigma_{s_i} = (x_l)$.
The next event in the computation of *DFSearch* on input $\mathcal{P}$ is when the pair (x_l, x_0) is checked to see whether some new vertex might be accessed. As $\Sigma_{s_i+1} = (x_l, x_0)$, IH($i+1$) holds.

Case (2) $\Sigma_{s_i} = (x_l, x_m)$.
There are four possibilities as regards the next event in the computation of *DFSearch* on input $\mathcal{P}$:

(a) $(x_l, x_m) \mapsto sink$, and so *DFSearch* goes on to accept;

(b) $(x_l, x_m) \mapsto y \neq sink$ where y has not yet been accessed, and so the next event is when y is shown to be accessible;

(c) $l > m$ and it is not the case that $(x_l, x_m) \mapsto y$ for some y that has not yet been accessed, and so the next event is when the pair (x_l, x_{m+1}) is checked to see whether a new vertex might be accessed; and

(d) $l = m$ and it is not the case that $(x_l, x_m) \mapsto y$ for some y that has not yet been accessed, and so either $x_l = x_m = 0$ and *DFSearch* goes on to reject or the next event is when a search is embarked upon for the pair (x_{l_1}, x_{m_1}) that accessed x_l.

Case $(2a)$ $(x_l, x_m) \mapsto sink$.
In this case, *DFSearch* accepts $\mathcal{P}$ and ρ_0 accepts $\mathcal{P}$.

Case $(2b)$ $(x_l, x_m) \mapsto y$ where y has not yet been accessed.
We have that $\Sigma_{s_i+1} = (\langle x_l, x_m, y\rangle)$. Suppose that $\Sigma_j = (\langle p, q, y\rangle)$, for some $j < s_i$ and for some $p, q \in |\mathcal{P}|$; and let j be the minimal such j. By Lemma 11 and Fig. 3, Σ_j evolves to $\Sigma_s = (y)$, for some $s < s_i$. This yields a contradiction (as y has not yet been accessed). Hence, by Lemma 11, Σ_{s_i+1} evolves to $(\langle x_l, x_m, y\rangle, \langle x_l, x_m, y\rangle)$ so that all intermediate stack configurations do not correspond to any distinguished events; and then $(\langle x_l, x_m, y\rangle, \langle x_l, x_m, y\rangle)$ evolves to (y) by applying row 3.1. Consequently, IH$(i+1)$ holds.

Case $(2c)$ $l > m$ and it is not the case that $(x_l, x_m) \mapsto y$ for some y that has not yet been accessed.
In this case, t_{i+1} is the time-stamp associated with the event of checking the pair (x_l, x_{m+1}). There are two possibilities: either $(x_l, x_m) \mapsto \epsilon$ or $(x_l, x_m) \mapsto x_r$, for some $x_r \in \mathit{AccessedSet}$ (that is, the current version of *AccessedSet*).

If $(x_l, x_m) \mapsto \epsilon$ then $\Sigma_{s_i+1} = (x_l, x_m, x_0)$, which, by Lemma 10, evolves to the stack configuration (x_l, x_{m+1}) such that no intermediate stack configuration corresponds to a distinguished event; and so IH$(i+1)$ holds.

If $(x_l, x_m) \mapsto x_r$ where $x_r \in \mathit{AccessedSet}$ then $\Sigma_{s_i+1} = (\langle x_l, x_m, x_r\rangle)$. As x_r is in *AccessedSet*, by the induction hypothesis, $\Sigma_{s_j} = (x_r)$, for some $j \leq i$; and consequently (by consulting Fig. 3) $\Sigma_{s_j-1} = (\langle p, q, x_r\rangle, \langle p, q, x_r\rangle)$, for some $p, q \in |\mathcal{P}|$. Let $s < s_j - 1$ be the minimal s such that $\Sigma_s = (\langle p, q, x_r\rangle)$, for some $p, q \in |\mathcal{P}|$ (such an s exists by Fig. 3). By Lemma 11, Σ_{s_i+1} evolves to the stack configuration $(\langle x_l, x_m, x_r\rangle, \langle p, q, x_r\rangle)$ so that no intermediate stack configuration corresponds to a distinguished event. Note that $\Sigma_{s-1} = (p, q)$, and so, by IH(i), we have that $(p, q) \neq (x_l, x_m)$. Hence, the stack configuration $(\langle x_l, x_m, x_r\rangle, \langle p, q, x_r\rangle)$ evolves to (x_l, x_m, x_0) by applying row 3.2, which in turn, by Lemma 10, evolves to the stack configuration (x_l, x_{m+1}) such that no intermediate stack configuration corresponds to a distinguished event. Thus, IH$(i+1)$ holds.

Case $(2d)$ $l = m$ and it is not the case that $(x_l, x_m) \mapsto y$ for some y that has not yet been accessed.
If $l = m = 0$ then *DFSearch* rejects $\mathcal{P}$ and ρ_0 rejects $\mathcal{P}$. Assume that $l = m \neq 0$. The next event in the computation of *DFSearch* on input $\mathcal{P}$ is the event where the search for the pair (x_{l_1}, x_{m_1}) which was used to access x_l is embarked upon.

If $(x_l, x_l) \mapsto \epsilon$ then $\Sigma_{s_i+1} = ([x_l])$ and IH$(i+1)$ holds.

If $(x_l, x_l) \mapsto x_r$ and $x_r \in \mathit{AccessedSet}$ then we proceed as we did in Case $(2c)$ whence Σ_{s_i} evolves to the stack configuration $(\langle x_l, x_l, x_r\rangle, \langle p, q, x_r\rangle)$ (where $(p, q) \neq (x_l, x_l)$) so that no intermediate stack configuration corresponds to a distinguished event. Row 3.3 is now applied so that the stack configuration becomes $([x_l])$. Hence, IH$(i+1)$ holds.

Case (3) $\Sigma_{s_i} = ([x_l])$.
Suppose that the pair (x_{l_1}, x_{m_1}) accessed x_l (note that $l \neq 0$). There are three possibilities for the next event in the computation of *DFSearch* on input $\mathcal{P}$:

(a) if $m_1 < l_1$ then the pair (x_{l_1}, x_{m_1+1}) is checked to see whether a new vertex might be accessed;

(b) if $m_1 = l_1 \neq 0$ then the search for the pair (x_{l_2}, x_{m_2}) which accessed x_{l_1} is embarked upon; and

(c) if $m_1 = l_1 = 0$ then the input is rejected.

By the induction hypothesis, there is a time-stamp t_j, for some $j < i$, when the pair (x_{l_1}, x_{m_1}) was checked and $\Sigma_{s_j} = (x_{l_1}, x_{m_1})$. Consequently, $\Sigma_{s_j+1} = (\langle x_{l_1}, x_{m_1}, x_l\rangle)$. Suppose that $\Sigma_s = (\langle p, q, x_l\rangle)$, for some $s < s_j + 1$ and for some $p, q \in |\mathcal{P}|$ where $(p, q) \neq (x_l, x_m)$. Let s be the minimal such s. By Lemma 11 and Fig. 3, Σ_s evolves to $\Sigma_{s'} = (x_l)$ for some $s' < s_j$. This yields a contradiction as x_l would have already been accessed when the pair (x_{l_1}, x_{m_1}) was later checked (remember, we are assuming that (x_{l_1}, x_{m_1}) accesses x_l). Hence, by Lemma 9 and Fig. 3, Σ_{s_i} evolves to the stack configuration $([x_l], \langle x_{l_1}, x_{m_1}, x_l\rangle)$ so that no intermediate stack configuration corresponds to a distinguished event.

Case ($3a$) $m_1 < l_1$.
The stack configuration $([x_l], \langle x_{l_1}, x_{m_1}, x_l\rangle)$ evolves to (x_{l_1}, x_{m_1}, x_0) which, by Lemma 10, evolves to (x_{l_1}, x_{m_1+1}) so that no intermediate stack configuration corresponds to a distinguished event. Hence, IH$(i + 1)$ holds.

Case ($3b$) $m_1 = l_1 \neq 0$.
The stack configuration $([x_l], \langle x_{l_1}, x_{m_1}, x_l\rangle)$ evolves to $([x_{l_1}])$ by row 4.2, and so IH$(i+1)$ holds.

Case ($3b$) $m_1 = l_1 = 0$.
The computation of ρ_0, from the stack configuration $([x_l], \langle x_{l_1}, x_{m_1}, x_l\rangle)$, leads to a rejection of the input.

Thus, by induction, the program scheme ρ_0 simulates the algorithm *DFSearch*, and the result follows by Proposition 7. □

The reader will have no doubt noted the similarities between the proof of Theorem 12 and the proof of the main theorem in [2]. Cook uses a similar technique to simulate the computation of a polynomial-time deterministic Turing machine as a computation of a log-space deterministic auxiliary pushdown machine. However, note that we provide a much more formal proof of our simulation than Cook does for his.

We can now use Theorem 12 to show that removing non-deterministic guessing from the program schemes of NPSS$_s(1)$ does not diminish the class of problems so captured.

Corollary 13 *Let Ω be a problem over the signature σ. The following are equivalent*:

- $\Omega \in \mathbf{P}$;
- $\Omega \in NPSS_s(1)$;
- $\Omega \in DPSS_s(1)$; *and*

- $\Omega \in (\pm PS)^*[FO_s]$.

Proof Let $\mathcal{P}$ be a σ_3-structure, i.e., a path system, of size n. We shall build a σ_3'-structure $\mathcal{P}'$, i.e., a deterministic path system, such that $\mathcal{P} \in$ PS if, and only if, $\mathcal{P}' \in$ DetPS. In order that we define a σ_3'-structure, our path system $\mathcal{P}'$ will be such that: for every two vertices $x, y \in |\mathcal{P}'|$, there is exactly one $z \in |\mathcal{P}'|$ for which (x, y, z) is a rule; and, furthermore, (x, y, z) is a rule if, and only if, (y, x, z) is a rule.

Our path system $\mathcal{P}'$ has vertex set $|\mathcal{P}|^3$ and we partition this vertex set into the disjoint union:

$$|\mathcal{P}'| = \bigcup_{u \in |\mathcal{P}|} Q_u,$$

where for every $u \in |\mathcal{P}|$, $Q_u = \{(u, v, w) : v, w \in |\mathcal{P}|\}$. We define the set of rules of $\mathcal{P}'$ in three batches. The first two batches describe rules for which the first two components belong to the same Q_u; and the third batch describes rules for which the first two components belong to different sets Q_u and Q_v.

<u>Batch 1</u>

$$\begin{aligned}
&\{((u,0,0),(u,v,w),(u,v,succ(w))),((u,v,w),(u,0,0),(u,v,succ(w))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad : u,v,w \in |\mathcal{P}|, w \neq max\} \\
&\quad \cup\{((u,0,0),(u,v,max),(u,succ(v),0)),((u,0,0),(u,v,max),(u,succ(v),0)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad : u,v \in |\mathcal{P}|, v \neq max\} \\
&\quad \cup\{((u,0,0),(u,max,max),(u,0,0)),((u,max,max),(u,0,0),(u,0,0))\}
\end{aligned}$$

The rules in Batch 1 are essentially such that for any $u \in |\mathcal{P}|$, if $(u, 0, 0)$ is made accessible in $\mathcal{P}'$ then so is every vertex of Q_u.

<u>Batch 2</u>

$$\begin{aligned}
&\{((u,u,w),(u,u,w),(w,0,0)) : u,w \in |\mathcal{P}|, (u,w) \neq (0,0), R(u,u,w) \text{ holds in } \mathcal{P}\} \\
&\quad \cup\{((u,u,w),(u,u,w),(0,0,0)) \\
&\qquad\quad : u,w \in |\mathcal{P}|, (u,w) \neq (0,0), R(u,u,w) \text{ does not hold in } \mathcal{P}\} \\
&\quad \cup\{((u,v,w),(u,v',w'),(0,0,0)),((u,v',w'),(u,v,w),(0,0,0)) \\
&\qquad\quad : u,v,w,v',w' \in |\mathcal{P}|, (v,w) \neq (0,0) \neq (v',w'), \neg(w = w' \text{ and } v = u = v')\}
\end{aligned}$$

The rules in Batch 2 complete the definition for rules whose first 2 components are in the same set Q_u. They are mostly redundant (in that they are there so that $\mathcal{P}'$ has the property described in the first paragraph of this proof) except that if $R(u, u, w)$ holds in $\mathcal{P}$ and $(u, 0, 0)$ is accessible then so is $(w, 0, 0)$ (see the comment subsequent to the definition of the rules in Batch 1).

<u>Batch 3</u>

$$\begin{aligned}
&\{((u,v,w),(v,u,w),(w,0,0)),((v,u,w),(u,w,w),(w,0,0)) \\
&\qquad\qquad\qquad : u,v,w \in |\mathcal{P}|, u \neq v, R(u,v,w) \text{ holds in } \mathcal{P}\} \\
&\quad \cup\{(((u,v,w),(v,u,w),(source,0,0)),((v,u,w),(u,w,w),(source,0,0))
\end{aligned}$$

$$: u, v, w \in |\mathcal{P}|, u \neq v, R(u, v, w) \text{ does not hold in } \mathcal{P}\}$$
$$\cup\{((u, v, w), (u', v', w'), (0, 0, 0))$$
$$: u, v, w, u', v', w' \in |\mathcal{P}|, u \neq u', \neg((u, v) = (u', v') \text{ and } w = w')\}$$

The rules in Batch 3 essentially ensure that if $(u, 0, 0)$ and $(v, 0, 0)$ are accessible in $\mathcal{P}'$, where $u \neq v$, and $R(u, v, w)$ or $R(v, u, w)$ holds in $\mathcal{P}$ then $(w, 0, 0)$ is accessible in $\mathcal{P}'$ (some rules are redundant in terms of making new vertices accessible in $\mathcal{P}'$).

The source of the path system $\mathcal{P}$ is the vertex $(source, 0, 0)$ and the sink is the vertex $(sink, 0, 0)$.

A simple induction, with the vertices of $\{(u, 0, 0) : u \in |\mathcal{P}|\} \subseteq |\mathcal{P}'|$ corresponding to the vertices of $\mathcal{P}$, yields that the sink is accessible in the path system $\mathcal{P}$ if, and only if, the sink is accessible in the path system $\mathcal{P}'$. Moreover, this is true independently of which particular successor function is chosen.

What is more, we can actually describe the deterministic path system $\mathcal{P}'$ in terms of $\mathcal{P}$ using a quantifier-free formula of FO_s. That is, there is a quantifier-free formula $\psi(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in \mathrm{FO}_s$, where $\mathbf{x} = (x_1, x_2, x_3)$, $\mathbf{y} = (y_1, y_2, y_3)$ and $\mathbf{z} = (z_1, z_2, z_3)$, such that for every $\mathbf{u}, \mathbf{v}, \mathbf{w} \in |\mathcal{P}'|$, $F(\mathbf{u}, \mathbf{v}) = \mathbf{w}$ in $\mathcal{P}'$ if, and only if, $\psi(\mathbf{u}, \mathbf{v}, \mathbf{w})$ holds in $\mathcal{P}$. In fact, given variables x_1, x_2, x_3, y_1, y_2 and y_3, we can write a portion of '$\mathrm{DPSS}_s(1)$ code' which gives the variables z_1, z_2 and z_3 the value $F((x_1, x_2, x_3), (y_1, y_2, y_3))$. Consequently, we can clearly amend the program scheme ρ_0 of DPSS(1) so that it becomes a program scheme ρ_1 of $\mathrm{DPSS}_s(1)$ over σ_3 and accepts the problem PS. (In doing so, we essentially replace single variables with 3-tuples of variables and the built-in successor function with the lexicographic successor function on 3-tuples obtained using *succ*. Such constructions are common-place in the literature.)

Let Ω be some problem in $\mathbf{P}$ over the signature σ. By Theorem 1, there exists a quantifier-free formula $\varphi(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in \mathrm{FO}_s$, where $|\mathbf{x}| = |\mathbf{y}| = |\mathbf{z}| = k$, for some $k \geq 1$, such that for every σ-structure $\mathcal{A}$: the path system with vertex set $|\mathcal{A}|^k$, with rules $\{(\mathbf{u}, \mathbf{v}, \mathbf{w}) : \mathbf{u}, \mathbf{v}, \mathbf{w} \in |\mathcal{A}|^k, \varphi(\mathbf{u}, \mathbf{v}, \mathbf{w}) \text{ holds in } \mathcal{A}\}$, with source $(0, 0, 0)$ and with sink (max, max, max) is such that the sink is accessible from the source if, and only if, $\mathcal{A} \in \Omega$. By amending the program scheme ρ_1, in the same way that we amended the program scheme ρ_0 to obtain ρ_1, we can obtain a program scheme $\rho_2 \in \mathrm{DPSS}_s(1)$ which accepts Ω. The result follows by Theorems 1 and 2. $\square$

Notice what Corollary 13 actually says: it says that the deterministic model of computation $\mathrm{DPSS}_s(1)$ captures exactly the complexity class $\mathbf{P}$, and that the non-deterministic extension of this model, $\mathrm{NPSS}_s(1)$, captures $\mathbf{P}$ too. This result can be interpreted as a 'logical reformulation' of Cook's result, mentioned earlier, regarding deterministic and non-deterministic logspace auxiliary pushdown machines.

4 Building an ordering

A different interpretation can be placed on the proof, in the last section, that DetPS can be solved by the program scheme ρ_0 of DPSS(1). By a simple modification of ρ_0 so that it does not accept if the sink is shown to be accessible but simply continues exhibiting new accessible vertices, we can build a canonical ordering of the accessible vertices in any determinstic path system. If we know *a priori* that our deterministic path system is such that every vertex is accessible from the source then we can build a canonical ordering of the vertices whose minimal element is the source.

In more detail, let $\mathcal{P}$ be a σ_3-structure with the property that for every $x, y \in |\mathcal{P}|$ (where possibly $x = y$), there exists exactly one z such that either $R(x, y, z)$ or $R(y, x, z)$ holds. That is, $\mathcal{P}$ encodes a deterministic path system. By the proof of Theorem 12, there is clearly a program scheme $\rho_3 \in \mathrm{NPSS}(1)$ over $\sigma_3 \cup \{C, D\}$, where C and D are two new constant symbols, such that on input $\mathcal{P}$:

- if C and D are accessible and C comes immediately before D in the canonical ordering of accessible vertices of $\mathcal{P}$ then every terminating computation of ρ_3 on input $\mathcal{P}$ signifies this fact and there is at least one terminating computation; and

- if either one of C and D is not accessible or C does not come immediately before D in the canonical ordering of accessible vertices of $\mathcal{P}$ then every terminating computation of ρ_3 on input $\mathcal{P}$ signifies this fact and there is at least one terminating computation.

This observation can be used to show that on certain classes of structures, any problem solvable in polynomial-time can be defined by a sentence of $(\pm\mathrm{PS})^*[\mathrm{FO}]$ (in fact, in a fragment of this logic).

First, we require some definitions. Let Γ be a class of σ-structures that is closed under isomorphism. By a *problem involving structures from* Γ we mean an isomorphism-closed subset of Γ. For any problem Ω involving structures over Γ, we say that a sentence Ψ of some logic *defines* Ω if for every structure $\mathcal{A} \in \Gamma$:

$$\mathcal{A} \in \Omega \text{ if, and only if, } \mathcal{A} \models \Psi.$$

Note that we say nothing about which structures of $\mathrm{STRUCT}(\sigma) \setminus \Gamma$ satisfy Ψ. There is an analogous definition for a program scheme to accept some problem involving structures from Γ; or for a Turing machine to accept some problem involving structures from Γ. Consequently, when we talk of, for example, a logic $\mathcal{L}$ on a class of structures Γ we mean the class of problems involving structures from Γ definable in $\mathcal{L}$.

We begin by examining problems involving strongly-connected locally-ordered digraphs. Let $\mathcal{G}$ be a σ_3-structure with the following property: for every $x \in |\mathcal{G}|$, the set of pairs $N(x) = \{(y, z) : R(x, y, z) \text{ holds in } \mathcal{G}\}$ is of the form $\{(x, x)\}$ or

$$\{(x, y_1), (y_1, y_2), \ldots, (y_k, x) : k \geq 1 \text{ and the } y_i\text{'s are distinct and different from } x\}.$$

The structure $\mathcal{G}$ can be considered to be a digraph with vertex set $|\mathcal{G}|$ where x has no neighbours, if $N(x) = \{(x, x)\}$, and where the neighbours of x are ordered as $y_1, y_2, \ldots, y_k$, otherwise. Such structures are called *locally-ordered digraphs*. A locally-ordered digraph is *strongly-connected* if there is a path from any vertex to any other vertex in the underlying digraph.

Theorem 14 *Any problem involving strongly-connected locally-ordered digraphs that can be solved in polynomial-time can be accepted by a program scheme of* $NPSS(1)$ *and can also be defined by a sentence of* $PS^1[FO]$ *with two built-in constants, of the form:*

$$PS[\lambda \mathbf{x}, \mathbf{y}, \mathbf{z}\psi](\mathbf{0}, \mathbf{max}),$$

where: $|\mathbf{x}| = |\mathbf{y}| = |\mathbf{z}| = k$*, for some* $k \geq 1$*;* $\mathbf{0}$ *(resp.* $\mathbf{max}$*) is the constant symbol* 0 *(resp.* max*) repeated* k *times; and* φ *is a quantifier-free formula of* FO*. Consequently, on the class of strongly-connected locally-ordered digraphs,* $\mathbf{P} = PS^1[FO] = NPSS(1)$*, even when there are no built-in constants in* $PS^1[FO]$*.*

Proof Let $\mathcal{G}$ be a strongly-connected locally-ordered digraph. Define $\psi(\mathbf{x}', \mathbf{y}', \mathbf{z}')$, where $\mathbf{x}' = (x'_1, x'_2)$, $\mathbf{y}' = (y'_1, y'_2)$ and $\mathbf{z}' = (z'_1, z'_2)$, as:

$$(x'_1 = y'_1 \wedge x'_2 = y'_2 \wedge x'_1 = z'_1 \neq z'_2 \wedge R(x'_1, y'_1, z'_2)) \vee (x'_1 = x'_2 = y'_1 \neq y'_2 = z'_1 = z'_2).$$

The vertices of the path system $\mathcal{P}_\mathcal{G}$ obtained by interpreting the formula ψ in $\mathcal{G}$ are $|\mathcal{G}|^2$ and the rules are as follows:

- $((u, v), (u, v), (u, w))$ if $u \neq w$ and $R(u, v, w)$ holds in $\mathcal{G}$; and
- $((u, u), (u, v), (v, v))$ if $u \neq v$.

For every pair of vertices of $\mathcal{P}_\mathcal{G}$, there is at most one rule which can be applied; and also every vertex of the form (u, u) or (u, v), where v is a neighbour of u in $\mathcal{G}$, is accessible no matter which vertex (of the form (u', v'), where $u' = v'$ or v' is a neighbour of u') we choose for the source.

We might be inclined to think that by amending the program scheme ρ_3, defined at the beginning of this section (in a style similar to as in the proof of Corollary 13 so that vertices are replaced by pairs of elements and ψ defines the rules), we can obtain a canonical ordering of the vertices of $\mathcal{P}_\mathcal{G}$ (starting at any vertex we care to choose). However, the program scheme ρ_3 takes as input σ'_3-structures and such structures encode deterministic path systems; that is, path systems where there is *exactly* one rule of the form (u, v, w) or (v, u, w) for every pair of vertices $\{u, v\}$. More to the point, given two vertices u and v, ρ_3 has to ascertain whether there is a rule (u, v, w) with $u \neq w \neq v$. Actually, by considering the proofs of the results in the previous section, ρ_3 need only be able to ascertain whether there is a rule (u, v, w), with $u \neq w \neq v$, for *accessible* vertices u and v. Such a predicate can easily be checked (in NPSS(1)) when the path system is deterministic: we simply guess the unique vertex w, check to see whether there is a rule (u, v, w) or (v, u, w) and whether w is different from both u and v. However, when given $\mathcal{P}_\mathcal{G}$ as input, this can not be done as for some pairs of vertices $(\mathbf{u}, \mathbf{v}) \in |\mathcal{P}_\mathcal{G}|$, there is no vertex $\mathbf{w} \in |\mathcal{P}_\mathcal{G}|$ for which $(\mathbf{u}, \mathbf{v}, \mathbf{w})$ is a rule.

Hence, let us add the following rules to our path system $\mathcal{P}_\mathcal{G}$. Choose some $s \in |\mathcal{P}_\mathcal{G}|$ so that (s, s) is the source of $\mathcal{P}_\mathcal{G}$ and add the rules:

- $((u, v), (u, v'), (s, s))$ if $v' \neq u \neq v$ and $v \neq v'$; and

- $((u, v), (u', v'), (s, s))$ if $u \neq u'$.

These are essentially 'dummy rules' (involving accessible vertices) but their presence allows us to apply the results of the previous section, as these new rules can be defined by conjunctions of atomic and negated atomic formulae. The consequence is that we can obtain a program scheme $\rho_4 \in \text{NPSS}(1)$ which canonically orders the accessible vertices of $\mathcal{P}_\mathcal{G}$, starting from the vertex (s, s) (in ρ_4 we begin by guessing s and leave s fixed throughout). We can now use this ordering of the accessible vertices $\mathcal{P}_\mathcal{G}$ to obtain an ordering of the vertices of $\mathcal{G}$. Our encoding scheme is such that a vertex u of $\mathcal{G}$ is identified with the vertex (u, u) of $\mathcal{P}_\mathcal{G}$. Hence, in a computation of ρ_4 on $\mathcal{G}$, we can always remember the last vertex of $\mathcal{P}_\mathcal{G}$ of the form (u, u) that was shown to be accessible. Thus, to know whether u comes immediately before v in the canonical ordering of $\mathcal{G}$, we simply need to know whether (u, u) comes before (v, v) in the canonical ordering of the accessible vertices of $|\mathcal{P}_\mathcal{G}|$, so that no vertex of the form (w, w) is such that $(u, u) < (w, w) < (v, v)$ in this canonical ordering; and this is what ρ_4 tells us.

Let Ω be any problem involving strongly-connected locally-ordered digraphs that is solvable in polynomial-time. By Theorem 2, Ω can be accepted by a program scheme of $\text{NPSS}_s(1)$. By replacing tests to see whether $succ(x, y)$ or $\neg succ(x, y)$ holds with the code ρ_4, with 0 chosen as the source and max chosen as the last element in our canonical ordering, we obtain a program scheme of NPSS(1) that accepts Ω. By Theorem 2, Ω can be defined by a sentence $\Psi \in \text{PS}^1[\text{FO}]$ as required. Hence, on the class of strongly-connected locally-ordered digraphs, $\mathbf{P} = \text{PS}^1[\text{FO}]$, even in the absence of two built-in constants as we can replace Ψ by:

$$\exists 0 \exists max(\text{`}max \text{ is the last element in the canonical ordering starting at } 0\text{'} \wedge \Psi)$$

(here, we are treating 0 and max as two new variables). □

Theorem 14 should be compared with a result of Etessami and Immerman [5] on strongly-connected locally-ordered digraphs. Their notion of a locally-ordered digraph, which they call a *one-way locally ordered graph*, is the same as ours, i.e., a σ_3-structure with identical restrictions on R, except that in addition they have at their disposal another universe $\{0, 1, \ldots, n-1\}$, in a σ_3-structure of size n, and a built-in total ordering on this universe; that is, their structures are two-sorted. Immerman had previously proven that *transitive closure logic* (see [4] for more details) with a built-in successor relation defines the class of problems solvable in non-deterministic logspace; that is, the complexity class **NL**. The inclusion of this second universe (or 'counting on the side') meant that Immerman and Etessami could prove that on the class of strongly-connected one-way locally ordered graphs, **NL** consists of those problems definable in transitive closure logic (without a built-in successor relation). Looking at transitive closure logic on Etessami and Immerman's one-way locally ordered graphs is a way of removing the built-in successor relation but retaining a weaker notion of ordering. Our result shows that if we dispense with 'counting on the side' in one-way locally ordered graphs, i.e., we consider our locally-ordered digraphs, then whilst we do not show that transitive closure logic captures **NL** on this class of structures, we do show that path system logic captures **P** on the class of such digraphs.

We have another remark concerning Theorem 14. Probably the most commonly occurring locally-ordered digraph is the planar graph when it comes with a plane embedding; that is, for every vertex of the graph, the neighbours are listed in clockwise order. Consequently, Theorem 14 holds for the class of connected planar embeddings. But what if we are just given a connected planar graph *without* an embedding? That is, we are given a structure $\mathcal{G}$ over the signature $\sigma_2 = \langle E \rangle$, where E is a binary relation symbol, and consider $\mathcal{G}$ as an undirected graph with vertex set $|\mathcal{G}|$ and with edges $\{(u, v) : u, v \in |\mathcal{G}|, E(u, v) \text{ or } E(v, u) \text{ holds}\}$. Can we obtain a result similar to Theorem 14 on the class of planar graphs; or at least on a significant sub-class of planar graphs?

A planar graph $\mathcal{G}$ is a *triangulation* if there is a plane embedding of $\mathcal{G}$ such that every face is a cycle of length 3 (in particular, triangulations are connected). A graph is 3-*connected* if no matter which 2 vertices and their incident edges are removed, the graph remains connected. By [10], for example, a triangulation is 3-connected; and by [3], for example, every 3-connected planar graph has a unique plane embedding up to topological isomorphism. Hence, we can talk about the unique set of faces of a triangulation.

Theorem 15 *Any problem involving triangulations that can be solved in polynomial-time can be defined by a sentence of* $(\pm PS)^2[FO]$ *with two built-in constants. Consequently, on the class of triangulations,* $\mathbf{P} = (\pm PS)^2[FO]$ *(even in the absence of two built-in constants).*

Proof Let $\mathcal{G}$ be a σ_2-structure encoding a triangulation. Let $\mathcal{P}_\mathcal{G}$ be a path system with vertex set $|\mathcal{G}|^4 \times \{X, Y, Z\}$. Fix $c_0, c_1, c_2 \in |\mathcal{G}|$ for which (c_0, c_1, c_2) forms a face in $\mathcal{G}$. The path system $\mathcal{P}_\mathcal{G}$ has rules:

(*a*) $(X, (u, v, w, u), (v, w, u, v))$ and $((u, v, w, u), X, (v, w, u, v))$, for all $u, v, w \in |\mathcal{G}|$ for which (u, v, w) forms a face in $\mathcal{G}$;

(*b*) $(Y, (u, v, w, u), (u, w, v, u))$ and $((u, v, w, u), Y, (u, w, v, u))$, for all $u, v, w \in |\mathcal{G}|$ for which (u, v, w) forms a face in $\mathcal{G}$;

(*c*) $(Z, (u, v, w, u), (u, v, w', u))$ and $((u, v, w, u), Z, (u, v, w', u))$, for all $u, v, w, w' \in |\mathcal{G}|$ for which (u, v, w) and (u, v, w') form distinct faces in $\mathcal{G}$;

(*d*) $((u, v, w, u), (u, v, w, u), (u, u, u, u))$, for all $u, v, w \in |\mathcal{G}|$ for which (u, v, w) forms a face in $\mathcal{G}$;

(*e*) $((u, v, w, u), (u', v', w', u'), X)$, for all $u, v, w, u', v', w' \in |\mathcal{G}|$ for which (u, v, w) and (u', v', w') form distinct faces in $\mathcal{G}$ and where $(u, v, w, u) \neq (u', v', w', u')$;

(*f*) $((u, u, u, u), t, X)$ and $(t, (u, u, u, u), X)$, for all $u \in |\mathcal{G}|$ and $t \in |\mathcal{P}_\mathcal{G}|$; and

(*g*) (X, X, Y), (Y, Y, Z), $(Z, Z, (c_0, c_1, c_2, c_0))$ and (s, t, X), for all $s, t \in \{X, Y, Z\}$ for which $s \neq t$.

The source of the path system $\mathcal{P}_{\mathcal{G}}$ is the vertex X. The vertices of $|\mathcal{G}|^4$ of the form (u, v, w, u), where (u, v, w) forms a face in $\mathcal{G}$, can be viewed as rooted partial orientations of the faces of $\mathcal{G}$, via: the vertex (u, v, w, u) is the path $u \to v \to w$ of length 2 partially encompassing the face (u, v, w). The rules are such that they allow us to show that every vertex of the form (u, v, w, u), where (u, v, w) forms a face in $\mathcal{G}$, is accessible: with the rules involving X and Y and Z used to generate all 'rooted 2-paths' around a face; and the rules involving Z used to 'flip' across neighbouring faces. Moreover, all vertices of the form (u, u, u, u), where $u \in |\mathcal{G}|$, are accessible too.

The path system $\mathcal{P}_{\mathcal{G}}$ can easily be defined in terms of $\mathcal{G}$ using a formula of $\mathrm{PS}^1[\mathrm{FO}]$ (to check that (u, v, w) is a face in $\mathcal{G}$, we need to check that for every $u', v' \in |\mathcal{G}| \setminus \{u, v, w\}$, there is a path in $\mathcal{G}$ from u' to v' avoiding u, v and w: this can be verified with a formula of $\mathrm{PS}^1[\mathrm{FO}]$). Additionally, the path system obtained from $\mathcal{P}_{\mathcal{G}}$ by restricting to the vertices of $\{(u, v, w, u) : u, v, w \in |\mathcal{G}|, (u, v, w) \text{ forms a face in } \mathcal{G}\} \cup \{(u, u, u, u) : u \in |\mathcal{G}|\} \cup \{X, Y, Z\}$ is deterministic.

We can amend the program scheme ρ_3 (defined at the beginning of this section), as in the proof of Theorem 14, so that we obtain a program scheme ρ_5 which yields a canonical ordering of the accessible vertices of $\mathcal{P}_{\mathcal{G}}$. However, this program scheme ρ_5 is not in NPSS(1) as the tests in while-loops are allowed to be formulae of $\mathrm{PS}^1[\mathrm{FO}]$. We can now use this ordering to obtain a canonical ordering of the vertices of $\mathcal{G}$. In a computation of ρ_5 on $\mathcal{G}$, we can remember the last vertex of $\mathcal{P}_{\mathcal{G}}$ of the form (u, u, u, u) that was shown to be accessible. This yields a canonical ordering of the vertices of $\mathcal{G}$. Hence, as in the proof of Theorem 14, any polynomial-time solvable problem involving triangulations can be accepted by a program scheme of NPSS(1) with tests from $\mathrm{PS}^1[\mathrm{FO}]$; and so, by Theorem 2, by a sentence of $(\pm\mathrm{PS})^2[\mathrm{FO}]$. The rider in the statement of the result follows as in the proof of Theorem 14. □

Theorem 15 should be compared with a recent result of Grohe [6] who proved that any polynomial-time solvable problem involving 3-connected planar graphs can be defined by a sentence of inflationary fixed-point logic. As was remarked in [1], path system logic is a proper fragment of inflationary fixed-point logic (in fact, there are problems involving trees which are definable in inflationary fixed-point logic but not in path system logic): however, it is not known whether this is the case on the class of 3-connected planar graphs. Theorem 15 shows that on the class of triangulations, a proper sub-class of the class of 3-connected planar graphs, inflationary fixed point logic and (the fragment $(\pm\mathrm{PS})^2[\mathrm{FO}]$ of) path system logic are equally expressive: they express exactly the polynomial-time properties of such graphs.

We end with a remark for those readers acquainted with the hierarchy of program schemes NPSS defined in [1]. An immediate corollary of the proof of Theorem 15 is that on the class of triangulations, this hierarchy collapses to its second level, NPSS(2), and any polynomial-time solvable problem on the class of triangulations can be defined by a program scheme of NPSS(2).

5 Conclusions

In this paper we have essentially developed a new technique for building logically definable successor relations in certain classes of structures. Our technique is established by considering the relationship between certain program schemes with access to a stack and path system logic; and it enables us to (sometimes) build successor relations definable in path system logic as opposed to (the more expressive) inflationary fixed-point logic, as is usually the case in the literature.

Our analysis has resulted in a model of computation which takes arbitrary finite structures as inputs and which captures **P**, but whose non-deterministic version has the same computational power as its deterministic version. It is interesting to note that this equivalence of models comes about essentially because there is a quantifier-free first-order translation (in the parlance of [4]) from the problem PS to the problem detPS. Whilst this translation is not particularly difficult to establish, it is the association of the problem PS and detPS with the classes of program schemes NPSS(1) and DPSS(1) wherein the non-trivial aspects of the equivalence result lie. Another interesting aspect of this equivalence result is that although a program scheme ρ of, for example, $\mathrm{NPSS}_s(1)$ can solve any given problem of **P**, the computation of ρ need not itself be a polynomial-time computation. This point is worthy of further consideration.

There are numerous other obvious directions for further research. For example, it would be interesting to find other (natural) classes of structures over which path system logic captures **P** (such a contender has already been mentioned: the class of 3-connected graphs). A slightly more involved question might be: can we find a class of structures over which path system logic captures **P**, but so that $(\pm\mathrm{PS})^i[\mathrm{FO}]$ captures **P**, for some i, whereas within $(\pm\mathrm{PS})^i[\mathrm{FO}]$ there is a proper hierarchy $(\pm\mathrm{PS})^1[\mathrm{FO}] \subset (\pm\mathrm{PS})^2[\mathrm{FO}] \subset \ldots$? A first step in this direction would be to prove that on the class of triangulations, there are polynomial-time solvable problems that are not definable in $(\pm\mathrm{PS})^1[\mathrm{FO}]$.

References

[1] 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.

[2] S.A. Cook, Characterization of pushdown machines in terms of time-bounded computers, *Journal of the Association for Computing Machinery* **18** (1971) 4–18.

[3] R. Diestel, *Graph Theory*, Springer-Verlag (1997).

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

[5] K. Etessami and N. Immerman, Reachability and the power of local ordering, *Theoretical Computer Science* **148** (1995) 261–279.

[6] M. Grohe, Fixed-point logics on planar graphs, *Proceedings* 13*th Annual IEEE Symposium on Logic in Computer Science* (1998) 6–15.

[7] M. Grohe and J. Mariño, Definability and descriptive complexity on databases of bounded tree-width, *Proceedings of 7th International Conference on Database Theory*, Lecture Notes in Computer Science 1540, Springer-Verlag, Berlin (1999) 70–82.

[8] N. Immerman and E. Lander, Describing graphs: a first-order approach to graph canonization, in: *Complexity theory retrospective* (ed. A. Selman), Springer-Verlag (1990) 59–81.

[9] M. Otto, *Bounded Variable Logics and Counting*, Lecture Notes in Logic Volume 9, Springer-Verlag (1997).

[10] T.L. Saaty and P.C. Kainen, *The Four-Colour Problem*, McGraw-Hill International (1977).

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

[12] I.A. Stewart, Logical description of monotone NP problems, *Journal of Logic and Computation* **4** (1994) 337–357.

[13] I.A. Stewart, Program schemes, queues, the recursive spectrum and zero-one laws, *Proceedings of 7th Annual International Computing and Combinatorics Conference* (*COCOON* 2001) (ed. J. Wang), Lecture Notes in Computer Science 2108, Springer-Verlag, Berlin (2001) 39–48.

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