Program schemes, arrays, Lindström quantifiers and zero-one laws*

Iain A. Stewart†
Department of Mathematics and Computer Science,
University of Leicester, Leicester LE1 7RH, U.K.

June 21, 2001

Abstract

We characterize the class of problems accepted by a class of program schemes with arrays, NPSA, as the class of problems defined by the sentences of a logic formed by extending first-order logic with a particular uniform (or vectorized) sequence of Lindström quantifiers. A simple extension of a known result thus enables us to prove that our logic, and consequently our class of program schemes, has a zero-one law. However, we use another existing result to show that there are problems definable in a basic fragment of our logic, and so also accepted by basic program schemes, which are not definable in bounded-variable infinitary logic. As a consequence, the class of problems NPSA is not contained in the class of problems defined by the sentences of partial fixed-point logic even though in the presence of a built-in successor relation, both NPSA and partial fixed-point logic capture the complexity class **PSPACE**.

1 Introduction

This paper is a continuation of the study of the classes of problems captured by different classes of program schemes (in this study, the particular emphasis is on a comparison with the classes of problems defined by the sentences of well-known logics from finite model theory). *Program schemes* form 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. Program schemes were extensively studied in the seventies (for example, see [3, 7, 16, 35]), without much regard being paid to an analysis of resources, before a closer complexity analysis was undertaken in, mainly, the eighties (for example, see [24, 26, 44]). There are connections between program schemes and logics of programs, especially dynamic logic [9, 30]. One might also view many query languages from database theory as classes of program schemes, although query

*An extended abstract of this paper appeared in *Proc. Computer Science Logic*, Lecture Notes in Computer Science Vol. 1683, Springer-Verlag (1999) 374–388.
†Supported by EPSRC Grants GR/K 96564 and GR/M 12933.

languages tend to operate on relations as opposed to individual elements (for example, see the *while* language from [1, 4, 5] and the language BQL from [4, 33]).

One of the most basic classes of program schemes is that obtained by allowing assignments, while instructions with quantifier-free tests and non-determinism. In [6], the relative expressibilities of this class of program schemes, NPS(1), in the presence of a built-in successor, a built-in linear-order, a built-in multiplication, a built-in addition and combinations of such were completely classified. It was shown in [2] that NPS, an extension of NPS(1) obtained by allowing universally quantified program schemes to appear as tests in while instructions, is none other than transitive closure logic and that a computational analysis (as opposed to a model-theoretic, and in particular a game-theoretic, analysis) of such program schemes yields proper infinite hierarchies within NPS (and so within transitive closure logic). Also in [2], the class of program schemes obtained from NPS by allowing additional access to a stack, NPSS, was shown to be none other than path system logic (which had previously been shown to be none other than stratified fixed point logic and stratified Datalog [28, 22]), and again a computational analysis of the program schemes of NPSS was shown to yield proper infinite hierarchies within NPSS (and so within path system logic). Subsequently, in [43], a detailed analysis of certain program schemes of NPSS yielded that any polynomial-time problem involving strongly-connected locally-ordered digraphs, connected planar embeddings or triangulations (that is, planar graphs embeddable in the plane so that every face is a cycle of length 3) can be defined in (a proper fragment of) path system logic (without any built-in relations).

The results mentioned above show that the study of program schemes is intimately related with more mainstream logics from finite model theory. In [38], program schemes allowing assignments, while instructions with quantifier-free tests, non-determinism and access to arrays were studied but only in the presence of a built-in successor relation (the class of problems accepted by such program schemes was shown to be **PSPACE**). It is with these program schemes and their extensions, obtained by allowing universally quantified program schemes to appear as tests in while instructions, that we are concerned in this paper but in the absence of any built-in relations; that is, the class of program schemes NPSA. Our class of program schemes NPSA is quite natural. It consists of the union of an infinite hierarchy of classes of program schemes

$$\mathrm{NPSA}(1) \subseteq \mathrm{NPSA}(2) \subseteq \mathrm{NPSA}(3) \subseteq \ldots$$

The program schemes of NPSA(1) are built by allowing assignments, while instructions with quantifier-free tests, non-determinism and access to arrays (full details follow later). The program schemes of NPSA(2) are built from program schemes of NPSA(1) by universally quantifying free variables. The program schemes of NPSA(3) are built as are the program schemes of NPSA(1) except that tests in while instructions can be program schemes of NPSA(2). The program schemes of NPSA(4) are built from program schemes of NPSA(3) by universally quantifying free variables; and so on.

What is crucial is our definition of the semantics. Consider, for example, a while instruction in a program scheme ρ of NPSA(3) where the test is a program scheme ρ' of NPSA(2). In order to evaluate whether the test is true or not, the arrays from ρ are not 'passed over' to the program scheme ρ': the evaluation of ρ' has no access to the arrays of ρ. After evaluation of ρ' has been completed, the computation of

the program scheme ρ resumes accordingly with its arrays having exactly the same values as they had immediately prior to the evaluation of ρ'. It is essentially our semantic definition that enables us to characterize the class of problems accepted by the program schemes of NPSA as the class of problems defined by the sentences of a logic $(\pm\Omega)^*[\mathrm{FO}]$ formed by extending first-order logic with a particular uniform (or vectorized) sequence of Lindström quantifiers (where this uniform sequence of Lindström quantifiers corresponds to a **PSPACE**-complete problem Ω). Moreover, we show that the logic $(\pm\Omega)^*[\mathrm{FO}]$ has a zero-one law; but not because it is a fragment of bounded-variable infinitary logic, as is so often the case in finite model theory, for we show that there are problems definable in NPSA (in NPSA(1) even) which are not definable in bounded-variable infinitary logic. Consequently, whilst both NPSA and partial fixed-point logic capture the complexity class **PSPACE** in the presence of a built-in successor relation, there are problems in NPSA which are not definable in partial-fixed point logic. If our semantics were such as to allow for universal quantification over arrays then we could simply guess a successor relation and hold our guesses in an array, use universal quantification to verify that the guessed relation was indeed a successor relation and subsequently use this guessed relation as our successor relation throughout. Consequently, we would have captured **PSPACE** and not the interesting logics (with zero-one laws but which are not fragments of bounded-variable infinitary logic) encountered in this paper.

Section 2 includes our preliminary definitions (with [13] serving as our basic reference text for finite model theory). In Section 3, we establish complete problems for NPSA(1) via quantifier-free first-order translations with 2 constants, and in Section 4, we use these completeness results to obtain our logical characterizations of NPSA. We then use a result due to Stewart to show that NPSA(1) (and so NPSA) is not contained in bounded-variable infinitary logic. In Section 5, we extend a result due to Dawar and Grädel and hence show that our logics from the previous section, and consequently NPSA, have a zero-one law. Finally, we present our conclusions and directions for further research.

2 Preliminaries

2.1 Logic

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 consider signatures in which there are no constant symbols; that is, *relational signatures*. *First-order logic over the signature* σ, $\mathrm{FO}(\sigma)$, consists of those formulae built from atomic formulae over σ using $\wedge$, $\vee$, $\neg$, $\forall$ and $\exists$; and $\mathrm{FO} = \cup\{\mathrm{FO}(\sigma) : \sigma \text{ is some signature}\}$.

A *finite structure* $\mathcal{A}$ *over the signature* σ, or σ*-structure*, consists of a finite *universe* or *domain* $|\mathcal{A}|$ together with a relation R_i of arity a_i, for every relation symbol R_i of σ, 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). 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 of size at least 2 over the signature σ is denoted $\mathrm{STRUCT}(\sigma)$. A *problem* over some

signature σ consists of a subset of STRUCT(σ) 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.

2.2 Lindström quantifiers

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 the complexity class **L** (logspace) which can not be defined in first-order logic (one such being the problem consisting of all those structures, over any 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 [13] 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 the complexity class **P** (polynomial-time) via logspace reductions [8].

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

$$\{\mathcal{P} \in \text{STRUCT}(\sigma_{3++}) \quad : \quad \text{the vertex } \textit{sink} \text{ is accessible from the vertex } \textit{source} \text{ in the path system } \mathcal{P}\}.$$

Let us return to increasing the expressibility of FO. Corresponding to the problem PS is an operator of the same name. The logic $(\pm\text{PS})^*[\text{FO}]$, 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 $\varphi(\mathbf{x}, \mathbf{y}, \mathbf{z}) \in (\pm\text{PS})^*[\text{FO}]$ over some 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 $\text{PS}[\lambda\mathbf{x}, \mathbf{y}, \mathbf{z}\varphi](\mathbf{u}, \mathbf{v})$, where $\mathbf{u}$ and $\mathbf{v}$ are k-tuples of (not necessarily distinct) constant symbols and variables, is also a formula of $(\pm\text{PS})^*[\text{FO}]$. The

free variables of Φ are 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, [13] for a more detailed semantic definition of operators such as PS). Note 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 [13] 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 that 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 $(\pm\mathrm{PS})^*[\mathrm{FO}_s]$ (and adopt a similar notation for other logics). 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 [13] and [34] for a detailed discussion of this and related points. However, it turns out that a problem is in the complexity class $\mathbf{P}$ if, and only if, it can be defined by a sentence of $(\pm\mathrm{PS})^*[\mathrm{FO}_s]$ [40].

Our notation for $(\pm\mathrm{PS})^*[\mathrm{FO}]$ 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 $(\pm\mathrm{PS})^k[\mathrm{FO}]$, for some $k \geq 1$, is obtained by allowing at most k nestings of applications of PS, and the fragment $\mathrm{PS}^k[\mathrm{FO}]$ is obtained by further disallowing any application of PS to appear within the scope of a negation sign.

In [40], it was shown that there is a very restricted normal form for sentences of $\mathrm{PS}^1[\mathrm{FO}_s]$. This normal form is such that any problem in $\mathbf{P} = (\pm\mathrm{PS})^*[\mathrm{FO}_s]$ can be defined by a sentence of $\mathrm{PS}^1[\mathrm{FO}_s]$ of the form

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

where: $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are k-tuples of distinct variables, for some $k \geq 1$; φ is a quantifier-

free formula of FO_s; and **0** and **max** are k-tuples consisting of the constant symbols 0 and *max* repeated k times, respectively. (Note that in the absence of built-in relations, the hierarchy

$$\mathrm{PS}^1[\mathrm{FO}] \subset \mathrm{PS}^2[\mathrm{FO}] \subset \ldots$$

is proper [20].)

Saying that any problem in **P** can be described by a sentence of the above normal form is equivalent to saying that there is a *quantifier-free first-order translation with successor* from any problem in **P** to the problem PS; that is, PS is complete for **P** via quantifier-free first-order translations with successor. The reader is referred to [13] for more on logical translations where they go under the name of logical interpretations. However, in [13] logical translations involving only relational signatures are considered. If the target problem of a logical translation, PS above, is over a signature containing constant symbols then we assume that such constant symbols are specified by an appropriate tuple of other constant symbols (as is the case in the normal form result above where the constant symbols *source* and *sink* of σ_{3++} are specified by the k-tuples **0** and **max**, respectively). Naturally, we have notions such as a *quantifier-free first-order translation* (where *succ* and 0 and *max* are not involved) and a *quantifier-free first-order translation with* 2 *constants* (where there are two built-in constants, which are always interpreted differently, but no built-in successor relation), and we can have logical translations involving formulae of other logics, not just quantifier-free first-order formulae.

A number of extensions of FO using operators corresponding to some problem Ω have been studied, as indeed has the whole notion of using such operators to extend FO. For example: numerous complexity classes have been 'captured' by (fragments of) logics of the form $(\pm\Omega)^*[\mathrm{FO}]$ (sometimes in which there are built-in relations) and a variety of problems have been shown to be complete for different complexity classes via different logical translations (see, for example, the papers [19, 25, 36, 37] and the references therein); proper infinite hierarchies have been established in logics of the form $(\pm\Omega)^*[\mathrm{FO}]$ (see, for example, [2, 20, 21]); logics of the form $(\pm\Omega)^*[\mathrm{FO}]$ have been shown to have zero-one laws [12] (we shall talk about zero-one laws for such logics in more detail later); and it has been shown that if there is a logic for **P** (where 'logic' is as in [13] and [34]) then there is a logic for **P** of the form $(\pm\Omega)^*[\mathrm{FO}]$ [11].

2.3 Program schemes

An alternative and more computational means for defining classes of problems is to use program schemes. A *program scheme* $\rho \in \mathrm{NPSA}(1)$ 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 '$x_i := A[y_1, y_2, \ldots, y_d]$' or '$A[y_1, y_2, \ldots, y_d] := y_0$', for some $i \in \{1, 2, \ldots, k\}$, where 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 which do not appear in any signature, and where A is an array symbol of dimension d;

- a *guess instruction* of the form 'GUESS x_i', where $i \in \{1, 2, \ldots, k\}$; or
- a *while instruction* of the form 'WHILE φ DO $\alpha_1; \alpha_2; \ldots; \alpha_q$ OD', where φ is a quantifier-free formula of FO($\sigma \cup \{0, max\}$), 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 one of the forms given here (note that there may be nested while instructions).

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 nor in a guess instruction. Essentially, free variables appear in ρ as if they were constant symbols.

A program scheme $\rho \in$ NPSA(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:

- execution of the instruction 'GUESS x_i' non-deterministically assigns an element of $|\mathcal{A}|$ to the variable x_i;
- the constants 0 and max are interpreted as two arbitrary but distinct elements of $|\mathcal{A}|$; and
- initially, every input-output variable and every array element 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, [38]. 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 NPSA(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 or 2 constant symbols into a logic. Indeed, we can build a successor relation into our program schemes of NPSA(1) so as to obtain the class of program schemes NPSA$_s$(1). As with our logics, we write NPSA(1) and NPSA$_s$(1) to also denote the class of problems accepted by the program schemes of NPSA(1) and NPSA$_s$(1), respectively. It was proven in [38]

that a problem is in the complexity class **PSPACE** (polynomial-space) if, and only if, it is in $\mathrm{NPSA}_s(1)$.

Henceforth, we think of our program schemes as being written in the style of a computer program. That is, each instruction is written on one line and while instructions (and, similarly, if and if-then-else instructions) are split so that 'WHILE φ DO' appears on one line, 'α_1' appears on the next, 'α_2' on the next, and so on (of course, if any α_i is a while, if or if-then-else instruction then it is split over a number of lines in the same way). The instructions are labelled 1, 2, and so on, according to the line they appear on. In particular, every instruction is considered to be an assignment, a guess or a test. An *instantaneous description* (*ID*) of a program scheme on some input consists of a value for each variable, the number of the instruction about to be executed and values for all array elements. A *partial ID* consists of just a value for each variable and the number of the instruction about to be executed. One *step* in a program scheme computation is the execution of one instruction, which takes one ID to another, and we say that a program scheme can *move* from one ID to another if there exists a sequence of steps taking the former ID to the latter.

3 Complete problems

We begin by examining the class of problems NPSA(1) and we show that this class has a complete problem via quantifier-free first-order translations with 2 constants. This problem, which to our knowledge has not been studied before, is also shown to be complete for **PSPACE** via quantifier-free first-order translations with successor.

Definition 1 Let the signature $\sigma_{TR} = \langle E, P, T, C, D \rangle$, where E is a binary relation symbol, P and T are unary relation symbols and C and D are constant symbols. We can envisage any σ_{TR}-structure $\mathcal{A}$ as a digraph (possibly with self-loops) whose edge relation is E and with distinguished vertices C, the *source*, and D, the *sink*. The relation P can be seen as providing a partition of the vertices and the relation T a subset of the vertices upon which *tokens* are initially placed. All tokens are indistinguishable and any vertex has upon it at most one token. Let us call a σ_{TR}-structure $\mathcal{A}$ a *token digraph*.

Just as one can traverse a path in a digraph by moving along edges, so one can traverse a path in a token digraph $\mathcal{A}$. However, as to how edges can be traversed is different from the usual notion. Consider an edge $(u, v) \in E$ for which both u and v are in P and such that a *traveller* is at vertex u (the traveller traverses a path of edges in the digraph). The edge (u, v) can only be traversed by the traveller moving as follows.

- The traveller moves from u via the edge (u, u') to a vertex u' not in P upon which exactly one token resides;
- then from u' via the edge (u', v') to a vertex v' not in P upon which no token resides, if $v' \neq u'$, and at the same time taking the token previously at u' to v', or by moving from u' via the edge (u', u') (if it exists) to u' (so that the token remains at u'); and finally
- by moving from the vertex v' or u', whichever is the case, via the edge (v', v) or (u', v) to v.