Probabilistic Semantics and Program Analysis

Alessandra Di Pierro[1], Chris Hankin[2], and Herbert Wiklicky[2]

[1] University of Verona, Ca' Vignal 2 - Strada le Grazie 15, 37134 Verona, Italy
dipierro@sci.univr.it
[2] Imperial College London, 180 Queen's Gate, London SW7 2AZ, United Kingdom
{clh,herbert}@doc.ic.ac.uk

Abstract. The aims of these lecture notes are two-fold: (i) we investigate the relation between the operational semantics of probabilistic programming languages and Discrete Time Markov Chains (DTMCs), and (ii) we present a framework for probabilistic program analysis which is inspired by the classical Abstract Interpretation framework by Cousot & Cousot and which we introduced as Probabilistic Abstract Interpretation (PAI) in [1]. The link between programming languages and DTMCs is the construction of a so-called Linear Operator semantics (LOS) in a syntax-directed or compositional way. The main element in this construction is the use of tensor product to combine information about different aspects of a program. Although this inevitably results in a combinatorial explosion of the size of the semantics of program, the PAI approach allows us to keep some control and to obtain reasonably sized abstract models.

1 Introduction

These lecture notes aim in establishing a formal link between the semantics of deterministic and probabilistic programming languages and Markov Chains. We will consider only discrete time models but, as we have shown in [2], it is possible to use similar constructions also to model continuous time systems. Our motivation is based on concrete systems rather than specifications of systems as we find it for example in the area of process algebras; we therefore eliminate any non-probabilistic or pure non-determinism. To a certain degree non-deterministic models can be simulated by using "unknown" probability variables rather than constants to express choice probabilities. However, this leads to slightly different outcomes as even "unknown" probabilities, for example, are able to express correlations between different choices.

A further (didactic) restriction we will use throughout these notes is the finiteness of our state and configuration spaces. Although it is possible to develop a similar framework also for infinite spaces, this requires certain mathematical tools from Functional Analysis and Operator Theory (e.g. C*algebras, Hilbert and Banach spaces) which are beyond what a short introduction can provide. We will therefore consider only a finite-dimensional algebraic theory for which a basic knowledge of linear algebra is sufficient.

In the following we will use a simple but intriguing example to illustrate our approach:

Example 1 (Monty Hall). The origins of this example are legendary. Allegedly, it goes back to some TV show in which the contestant was given the chance to win a car or other prizes by picking the right door behind which the desired prize could be found.

The game proceeds as follows: First the contestant is invited to pick one of three doors (behind one is the prize) but the door is not yet opened. Instead, the host – legendary Monty Hall – opens one of the other doors which is empty. After that the contestant is given a last chance to stick with his/her door or to switch to the other closed one. Note that the host (knowing where the prize is) has always at least one door he can open.

The problem is whether it is better to stay stubborn or to switch the chosen door. Assuming that there is an equal chance for all doors to hide the prize it is a favourite exercise in basic probability theory to demonstrate that it is better to switch to a new door.

We will analyse this example using probabilistic techniques in program analysis - rather than more or less informal mathematical arguments. An extensive discussion of the problem can be found in [3] where it is also observed that a bias in hiding the car (e.g. because the architecture of the TV studio does not allow for enough room behind a door to put the prize there) changes the analysis dramatically.

Note that it is pointless to investigate a non-deterministic version of the Monty Hall problem: If we are only interested in a possibilistic analysis then both strategies have exactly the same possible outcomes: The contestant might win or lose – everything is possible. As in many walks of life it is not what is possible that determines success, but the chances of achieving one's aim.

2 Mathematical Preliminaries

We assume that the reader of these lecture notes is well acquainted with basic ideas from linear algebra and probability theory. We will consider here only finite dimensional spaces and thus avoid a detailed consideration of finite dimensional spaces, as in functional analysis, and general measure theoretic concepts. However, it is often possible to generalise the concepts to such an infinite dimensional setting and we may occasionally mention this or give hints in this direction.

We need to introduce a few basic mathematical concepts – the acquainted readers may skip immediately to Section 3. The aim of this section is to sketch the basic constructions and to provide some motivation and intuition of the mathematical framework we use. A more detailed discussion of the notions and concepts we need can be found in the appropriate textbooks on probability and linear algebra.

2.1 Vector Spaces

In all generality, the real *vector space* $\mathcal{V}(S, \mathbb{R}) = \mathcal{V}(S)$ over a set S is defined as the formal[3] linear combinations of elements in S which we can also see as tuples of real numbers x_s indexed by elements in S

$$\mathcal{V}(S) = \{\langle x_s, s\rangle_{s\in S} \mid x_s \in \mathbb{R}\} = \left\{\sum_{s\in S} x_s \boldsymbol{s}\right\} = \{(x_s)_{s\in S}\}\,,$$

with the usual point-wise algebraic operations, i.e. scalar multiplication for $\lambda \in \mathbb{R}$:

$$\lambda \cdot (x_s)_s = (\lambda \cdot x_s)_s$$

and vector addition

$$(x_s)_s + (y_s)_s = (x_s + y_s)_s.$$

We denote tuples like $(x_s)_s$ or $(y_s)_s$ as vectors $\boldsymbol{x}$ and $\boldsymbol{y}$.

We consider in the following only finite dimensional vector spaces, i.e. $\mathcal{V}(S)$ over finite sets S, as they possess a unique topological structure, see e.g. [4, 1.22]. By imposing additional constraints one could equip $\mathcal{V}(S)$ with an appropriate topological structure even for infinite sets S, e.g. by considering Banach or Hilbert spaces like $\ell^1(S)$, $\ell^2(S)$, etc. (see for example [5]).

The importance of vector spaces in the context of these notes comes from the fact that we can use them to represent *probability distributions* ρ, i.e. normalised functions which associate to elements in S some probability in the interval $[0, 1]$

$$\rho : S \to [0, 1] \quad \text{s.t.} \quad \sum_{s\in S} \rho(s) = 1.$$

The set of all distributions $\mathbf{Dist}(S)$ on S is isomorphic to a sub-set (however, *not* a sub-space) of $\mathcal{V}(S)$. This helps to transfer the algebraic structures of $\mathcal{V}$ like, for example, the tensor product (see below) immediately into the context of distributions.

The important class of structure preserving maps between vector spaces $\mathcal{V}$ and $\mathcal{W}$ are *linear* maps $\mathbf{T} : \mathcal{V} \to \mathcal{W}$ which fulfil:

$$\mathbf{T}(\boldsymbol{v}) = \lambda \cdot \mathbf{T}(\boldsymbol{v}) \quad \text{and} \quad \mathbf{T}(\boldsymbol{v}_1 + \boldsymbol{v}_2) = \mathbf{T}(\boldsymbol{v}_1) + \mathbf{T}(\boldsymbol{v}_2).$$

For linear maps $\mathbf{T} : \mathcal{V} \to \mathcal{V}$ we usually use the term *operator*.

Vectors in any vector space can be represented – as in the above definition of $\mathcal{V}(S)$ – as a linear combination of elements a certain basis, or even simpler as a tuple, i.e. a row, of coordinates. Usually, we will use here the defining basis $\{\boldsymbol{s} \mid s \in S\}$ so that we do not need to consider the problem of base changes.

As with vectors we can also represent linear maps in a standardised way as *matrices*. We will treat here the terms linear map and operator as synonymous of

[3] We allow for any – also infinite – linear combinations. For the related notion of a *free* vector space one allows only finite linear combinations.

matrix. The standard representation of a linear map $\mathbf{T} : \mathcal{V} \to \mathcal{W}$ simply records the image of all basis vectors of the basis in $\mathcal{V}$ and collects them as row vectors of a matrix. It is sufficient to just specify what happens to the (finitely many) basis vectors to completely determine $\mathbf{T}$ as by linearity this can be extended to all (uncountably infinitely many) vectors in $\mathcal{V}$. Given a (row) vector $\boldsymbol{x} = (x_s)_s$ and the matrix $(\mathbf{T}_{st})_{st}$, with the first index indicating the row and the second the column of the matrix entry, representing a linear map $\mathbf{T}$ we can implement the application of $\mathbf{T}$ to $\boldsymbol{x}$ as a matrix multiplication:

$$\mathbf{T}(\boldsymbol{x}) = \boldsymbol{x} \cdot \mathbf{T} = (x_s)_s \cdot (\mathbf{T}_{st})_{st} = (\sum_s x_s \mathbf{T}_{st})_t.$$

2.2 Discrete Time Markov Chains

The standard and most popular model for probabilistic processes are Markov Chains. We assume a basic knowledge as presented for example in [6–10], to mention just a few of the many monographs on this topic.

Markov chains have the important property that they are *memory-less* in the sense that the "next state" does not depend on anything else but the current state. Markov Chains come in two important versions as Discrete Time Markov Chains (DTMC) and Continuous Time Markov Chains (CTMC). We will deal here only with DTMCs, i.e. probabilistic descriptions of a system only at discrete time steps. This allows us to talk about the next state in the obvious way (for CTMC this concept is a bit more complicated).

The DTMCs we will use to model the semantics of a programming language will be based on finitely many states S.[4]. For such a system a description at a given point in time is represented by a distribution over the finite state space S, we will refer to the elements in s also as *classical states* and to the elements in $\mathbf{Dist}(S)$ as *probabilistic states*. In general, we would need measures or vectors in Banach or Hilbert spaces to describe probabilistic states.

Once we have an enumeration of states in S we can represent probabilistic states, i.e. distributions on S, as normalised tuples or simply as vectors in $\mathcal{V}(S)$. The fact that DTMCs are *memory-less* means that we only need to specify how the description of a system changes into the one at the next step, i.e. how to transform one probabilistic state $\boldsymbol{d}_t$ into the next one $\boldsymbol{d}_{t+1}$. Intuitively, we need to describe how much of the probability of an $s_i \in S$ is "distributed" to the other s_j in the next moment. Again, we can use matrices to do this. More precisely, we need to consider *stochastic* matrices $\mathbf{M}$, where all rows must sum up to 1, i.e.

$$\sum_t \mathbf{M}_{st} = 1 \quad \text{for all } s,$$

so that for a distribution represented by $\boldsymbol{d}$ the image $\boldsymbol{x} \cdot \mathbf{M}$ is again a (normalised) distribution. Note that we follow in these notes the convention of postmultiplying $\mathbf{M}$ and that vectors are implemented as row vectors.

[4] Unfortunately, the term "state" is used differently in probability theory and semantics: The (probabilistic) state space for the semantics we represent is made up of so-called configurations which are pairs of (semantical) states and statements.

We will consider here only homogenous DTMCs where the way the system changes does not change itself over time, i.e. $\boldsymbol{d}_0$ is transformed into $\boldsymbol{d}_1$ in the same way as $\boldsymbol{d}_t$ becomes $\boldsymbol{d}_{t+1}$ at any time t. The change to matrix $\mathbf{M}$, thus, does not depend on t. In fact, we can define a DTMC as we use it here just by specifying its state space S and its *generator* matrix $\mathbf{M}$, which has to be stochastic.

2.3 Kronecker and Tensor Product

For the definition of our semantics we will use the *tensor product* construction. The tensor product $\mathcal{U} \otimes \mathcal{V}$ of two vector spaces $\mathcal{U}$ and $\mathcal{V}$ can be defined in a purely abstract way via the following *universal* property: For each *bi-linear* function $f : \mathcal{U} \times \mathcal{V} \to \mathcal{W}$ there exists a unique *linear* function $f_\otimes : \mathcal{U} \otimes \mathcal{V} \to \mathcal{W}$ such that $f(u, v) = f_\otimes(u \otimes v)$, see e.g. [11, Ch 14].

In the case of infinite dimensional topological vector spaces one usually imposes additional requirements on the tensor product ensuring, for example, that the tensor product of two Hilbert spaces is again a Hilbert space, see e.g. [12, 2.6]. Product measures on the Cartesian product of measure spaces as characterised by Fubini's Theorem, see e.g. [13, 4.5], can also be seen as tensor products.

For finite dimensional vector spaces we can realise $\mathcal{U} \otimes \mathcal{V}$ as the space of the tensor product of vectors in $\mathcal{V}$ and $\mathcal{U}$. More concretely, we can construct the tensor product of two finite dimensional matrices or vectors – seen as $1 \times n$ or $n \times 1$ matrices – via the so-called *Kronecker product*: Given an $n \times m$ matrix $\mathbf{A}$ and a $k \times l$ matrix $\mathbf{B}$ then $\mathbf{A} \otimes \mathbf{B}$ is the $nk \times ml$ matrix

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{1,1} \ldots a_{1,m} \\ \vdots \quad \ddots \quad \vdots \\ a_{n,1} \ldots a_{m,n} \end{pmatrix} \otimes \begin{pmatrix} b_{1,1} \ldots b_{1,l} \\ \vdots \quad \ddots \quad \vdots \\ b_{k,1} \ldots b_{k,l} \end{pmatrix} = \begin{pmatrix} a_{1,1}\mathbf{B} \ldots a_{1,m}\mathbf{B} \\ \vdots \quad \ddots \quad \vdots \\ a_{n,1}\mathbf{B} \ldots a_{n,m}\mathbf{B} \end{pmatrix}$$

For a d_1 dimensional vector $\boldsymbol{u}$ and a d_2 dimensional vector $\boldsymbol{v}$ we get a $d_1 \cdot d_2$ dimensional vector $\boldsymbol{u} \otimes \boldsymbol{v}$. The ith entry in $\boldsymbol{u} \otimes \boldsymbol{v}$ is the product of the i_1th coordinate of $\boldsymbol{u}$ with the i_2th coordinate of $\boldsymbol{v}$. The relation between index i and the indices i_1 and i_2 is as follows:

$$\begin{aligned} i &= (i_1 - 1) \cdot d_2 + (i_2 - 1) + 1 \\ i_1 &= (i - 1) \text{ div } d_2 + 1 \\ i_2 &= (i - 1) \bmod d_2 + 1 \end{aligned}$$

Note that the concrete realisation of the tensor product via the Kronecker product is *not* base independent, i.e. if we use a different basis to represent $\mathbf{A}$ and $\mathbf{B}$ then it is non-trivial to see how the coordinates of $\mathbf{A} \otimes \mathbf{B}$ change. Thus many texts prefer the abstract definition of tensor products. However, our discussions will not involve base changes and we thus can work with Kronecker and tensor products as synonyms.

The binary tensor/Kronecker product can easily be generalised to an n-ary version which is associative but not commutative. Among the important algebraic properties of the tensor/Kronecker product (of matrices and vectors with matching dimensions) we have for example, see e.g. [11, 12]:

$$(\lambda\mathbf{A}) \otimes \mathbf{B} = \lambda(\mathbf{A} \otimes \mathbf{B}) = \mathbf{A} \otimes (\lambda\mathbf{B})$$
$$(\mathbf{A}_1 + \mathbf{A}_2) \otimes \mathbf{B} = (\mathbf{A}_1 \otimes \mathbf{B}) + (\mathbf{A}_1 \otimes \mathbf{B})$$
$$\mathbf{A} \otimes (\mathbf{B}_1 + \mathbf{B}_2) = (\mathbf{A} \otimes \mathbf{B}_1) + (\mathbf{A} \otimes \mathbf{B}_2)$$
$$(\mathbf{A}_1 \otimes \mathbf{B}_1)(\mathbf{A}_2 \otimes \mathbf{B}_2) = (\mathbf{A}_1\mathbf{A}_2) \otimes (\mathbf{B}_1\mathbf{B}_2)$$

If we consider the tensor product of vector spaces $\mathcal{V}(X)$ and $\mathcal{V}(Y)$ over some (finite) sets X and Y then we get the following important isomorphism which relates the Cartesian product and the tensor product:

$$\mathcal{V}(X \times Y) = \mathcal{V}(X) \otimes \mathcal{V}(Y)$$

This follows directly from the universal properties of the tensor product. In terms of distribution this provides a way to construct and understand the space of distributions over product spaces.

3 Probabilistic While

We now introduce a simple *imperative* language, **pWhile**, with constructs for probabilistic choice and random assignment, which is based on the well known **While** language one can find for example in [14, 15]. We will use this language to investigate static program analysis techniques based on its semantics. We first present the syntax and operational semantics (in an SOS style) of **pWhile**; then we develop a syntax-directed semantics which will immediately give the generator of the corresponding DTMC.

3.1 Syntax

The overall structure of a **pWhile** program is made up from a possibly empty declaration part D of variables and a single statement S which represents the actual program:

$$\begin{array}{rcl} P & ::= & \texttt{begin}\ S\ \texttt{end} \\ & \mid & \texttt{var}\ D\ \texttt{begin}\ S\ \texttt{end} \end{array}$$

The declarations D of variables v associate to them a certain basic type e.g. **int**, **bool**, or a simple value range r, which determine the possible values of the variable v. Each variable can have only one type, i.e. possible values are in the disjoint union of $\mathbb{Z}$ representing integers, $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$ for booleans.

$$\begin{array}{rcl} r & ::= & \texttt{bool} \\ & \mid & \texttt{int} \\ & \mid & \texttt{\{}\ c_1, \ldots c_n\ \texttt{\}} \\ & \mid & \texttt{\{}\ c_1\ \texttt{..}\ c_n\ \texttt{\}} \\ D & ::= & v\ \texttt{:}\ r \\ & \mid & v\ \texttt{:}\ r\ \texttt{;}\ D \end{array}$$

The syntax of statements S is as follows:

$$
\begin{array}{rl}
S ::= & \texttt{stop} \\
\mid & \texttt{skip} \\
\mid & v \texttt{ := } a \\
\mid & v \texttt{ ?= } r \\
\mid & S_1 \texttt{ ; } S_2 \\
\mid & \texttt{choose } p_1 : S_1 \texttt{ or } p_2 : S_2 \texttt{ ro} \\
\mid & \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi} \\
\mid & \texttt{while } b \texttt{ do } S \texttt{ od}
\end{array}
$$

We have in **pWhile** two types of "empty" statements, namely `stop` and the usual `skip` statement. We can use both as final statements in a program but while `skip` represents actual termination the meaning of `stop` is an infinite loop which replicates the terminal configuration forever – this is a behaviour we need in order to avoid "probability leaks" and to obtain proper DTMCs. The meaning of the assignment "`:=`", sequential composition "`;`", "`if`" and "`while`" are as usual – we only change the syntax slightly to allow for an easier implementation of a **pWhile** parser in `ocaml`. We have two additional probabilistic statements: a random assignment "`?=`" which assigns a random value to a variable using a uniform distribution over the possible values in the range r; and a probabilistic choice "`choose`", which executes either S_1 or S_2 with probabilities p_1 and p_2, respectively. Here p_1 and p_2 are constants and we assume without loss of generality that they are normalised, i.e. that $p_1 + p_2 = 1$; if this is not the case, we can also require that at compile time these values are normalised to obtain $\tilde{p}_i = \frac{p_i}{p_1+p_2}$. It is obvious how to generalise the "`choose`" construct from a binary to an n-ary version. We will also use brackets, indentation and comment lines "`#`" to improve the readability of programs.

Expressions e in **pWhile** are either boolean expressions b or arithmetic expressions a. Arithmetic expressions are of the form

$$
\begin{array}{rl}
a ::= & n \\
\mid & a_1 \odot a_2
\end{array}
$$

with $n \in \mathbb{Z}$ a constant and '$\odot$' representing one of the usual arithmetic operations like '$+$', '$-$', '$\times$', '$/$' or '$\%$' (representing the remainder of an integer division).

The syntax of boolean expressions b is defined by

$$
\begin{array}{rl}
b ::= & \texttt{true} \\
\mid & \texttt{false} \\
\mid & \texttt{not } b \\
\mid & b_1 \texttt{ \&\& } b_2 \\
\mid & b_1 \texttt{ || } b_2 \\
\mid & a_1 \lessgtr a_2
\end{array}
$$

The symbol '$\lessgtr$' denotes one of the standard comparison operators for arithmetic expressions, i.e. $<, \leq, =, \neq, \geq, >$.

3.2 Operational Semantics

The semantics of **pWhile** follows essentially the standard one for **While** as presented, e.g., in [15]. The only two differences concern (i) the probabilistic choice and (ii) random assignments. The structured operational semantics (SOS) is given as usual via a transition system on configurations $\langle S, \sigma \rangle$, i.e. pairs of statements and (classical) states. To allow for probabilistic choices we label these transitions with probabilities; except for the **choose** construct and the random assignment these probabilities will always be 1 as all other statements in **pWhile** are deterministic.

A state $\sigma \in \mathbf{State}$ describes how variables in **Var** are associated to values in $\mathbf{Value} = \mathbb{Z} + \mathbb{B}$ (with '+' denoting the disjoint union). The value of a variable can be either an integer or a boolean constant, i.e.

$$\mathbf{State} = \mathbf{Var} \to \mathbb{Z} + \mathbb{B}$$

The expressions a and b evaluate to values of type $\mathbb{Z}$ and $\mathbb{B}$ in the usual way. The value represented by an arithmetic expression can be computed by:

$$\begin{aligned} \mathcal{E}(n)\sigma &= n \\ \mathcal{E}(v)\sigma &= \sigma(\llbracket v \rrbracket \sigma) \\ \mathcal{E}(a_1 \odot a_2)\sigma &= \mathcal{E}(a_1)\sigma \odot \mathcal{E}(a_2)\sigma \end{aligned}$$

The result is always an integer (i.e. $\mathcal{E}(.)_a \in \mathbb{Z}$). Boolean expressions are also handled in a similar way; their semantics is given by an element in $\mathbb{B} = \{\texttt{true}, \texttt{false}\}$:

$$\begin{aligned} \mathcal{E}(\texttt{true})\sigma &= \texttt{true} \\ \mathcal{E}(\texttt{false})\sigma &= \texttt{false} \\ \mathcal{E}(\texttt{not } b)\sigma &= \neg \mathcal{E}(b)\sigma \\ \mathcal{E}(b_1 \texttt{ || } b_2)\sigma &= \mathcal{E}(b_1)\sigma \vee \mathcal{E}(b_2)\sigma \\ \mathcal{E}(b_1 \texttt{ \&\& } b_2)\sigma &= \mathcal{E}(b_1)\sigma \wedge \mathcal{E}(b_2)\sigma \\ \mathcal{E}(a_1 \texttt{ <= } a_2)\sigma &= \mathcal{E}(a_1)\sigma \leqslant \mathcal{E}(a_2)\sigma \end{aligned}$$

If we denote by **Expr** the set of all expressions e then the evaluation function $\mathcal{E}(.)$. is a function from $\mathbf{Expr} \times \mathbf{State}$ into $\mathbb{Z} + \mathbb{B}$.

Based on the functions $\llbracket . \rrbracket$. and $\mathcal{E}(.)$. the semantics of an assignment is given, for example, by:

$$\langle v \texttt{ := } e, \sigma \rangle \longrightarrow_1 \langle \texttt{stop}, \sigma[v \mapsto \mathcal{E}(e)\sigma] \rangle.$$

The state σ stays unchanged except for the variable v. The value of this variable is changed so that it now contains the value represented by the expression e.

The formal definition of the transition rules defining the operational semantics of **pWhile** in the SOS style is given in Table 3.2.

3.3 Examples

To illustrate the the use of **pWhile** to formulate probabilistic programs we present two small examples which we will use throughout these lecture notes.

R0 $\langle \texttt{skip}, \sigma \rangle \longrightarrow_1 \langle \texttt{stop}, \sigma \rangle$

R1 $\langle \texttt{stop}, \sigma \rangle \longrightarrow_1 \langle \texttt{stop}, \sigma \rangle$

R2 $\langle v \texttt{ := } e, \sigma \rangle \longrightarrow_1 \langle \texttt{stop}, \sigma[v \mapsto \mathcal{E}(e)\sigma] \rangle$

R3 $\langle v \texttt{ ?= } r, \sigma \rangle \longrightarrow_{\frac{1}{|r|}} \langle \texttt{stop}, \sigma[v \mapsto r_i \in r] \rangle$

R4$_1$ $\dfrac{\langle S_1, \sigma \rangle \longrightarrow_p \langle S_1', \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \longrightarrow_p \langle S_1'; S_2, \sigma' \rangle}$

R4$_2$ $\dfrac{\langle S_1, \sigma \rangle \longrightarrow_p \langle \texttt{stop}, \sigma' \rangle}{\langle S_1; S_2, \sigma \rangle \longrightarrow_p \langle S_2, \sigma' \rangle}$

R5$_1$ $\langle \texttt{choose } p_1 : S_1 \texttt{ or } p_2 : S_2 \texttt{ ro}, \sigma \rangle \longrightarrow_{p_1} \langle S_1, \sigma \rangle$

R5$_2$ $\langle \texttt{choose } p_1 : S_1 \texttt{ or } p_2 : S_2 \texttt{ ro}, \sigma \rangle \longrightarrow_{p_2} \langle S_2, \sigma \rangle$

R6$_1$ $\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}, \sigma \rangle \longrightarrow_1 \langle S_1, \sigma \rangle$ if $\mathcal{E}(b)\sigma = \texttt{true}$

R6$_2$ $\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}, \sigma \rangle \longrightarrow_1 \langle S_2, \sigma \rangle$ if $\mathcal{E}(b)\sigma = \texttt{false}$

R7$_1$ $\langle \texttt{while } b \texttt{ do } S \texttt{ od}, \sigma \rangle \longrightarrow_1 \langle S; \texttt{ while } b \texttt{ do } S \texttt{ od}, \sigma \rangle$ if $\mathcal{E}(b)\sigma = \texttt{true}$

R7$_2$ $\langle \texttt{while } b \texttt{ do } S \texttt{ od}, \sigma \rangle \longrightarrow_1 \langle \texttt{stop}, \sigma \rangle$ if $\mathcal{E}(b)\sigma = \texttt{false}$

Table 1. The rules of the SOS semantics of **pWhile**

Example 2 (Factorial). This example concerns the Factorial of a natural number, i.e. $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ (with $0! = 1$). The two programs below compute the usual factorial $n!$ and the "double factorial $2 \cdot n!$.

```
var
  m : {0..2};
  n : {0..2};

begin
m := 1;
while (n>1) do
  m := m*n;
  n := n-1;
od;
stop; # looping
end
```

```
var
  m : {0..2};
  n : {0..2};

begin
m := 2;
while (n>1) do
  m := m*n;
  n := n-1;
od;
stop; # looping
end
```

Though these two programs are deterministic, we will still analyse them using probabilistic techniques.

Example 3 (Monty Hall). Let us consider again Example 1 in Section 1. We can implement the two possible strategies of the contestant: Either to *stick* to his/her initial choice no matter what the show host is doing, or to *switch* doors once one of the empty doors has been opened.

```
var
  d :{0,1,2};
  g :{0,1,2};
  o :{0,1,2};

begin
# Pick winning door
d ?= {0,1,2};
# Pick guessed door
g ?= {0,1,2};
# Open empty door
o ?= {0,1,2};
while ((o == g) || (o == d)) do
  o := (o+1)%3;
od;
# Stick with guess
stop; # looping
end
```

```
var
  d :{0,1,2};
  g :{0,1,2};
  o :{0,1,2};

begin
# Pick winning door
d ?= {0,1,2};
# Pick guessed door
g ?= {0,1,2};
# Open empty door
o ?= {0,1,2};
while ((o == g) || (o == d)) do
  o := (o+1)%3;
od;
# Switch guess
g := (g+1)%3;
while (g == o) do
  g := (g+1)%3;
od;
stop; # looping
end
```

3.4 Linear Operator Semantics

In order to study the semantic properties of a **pWhile** program we will investigate the stochastic process which corresponds to the program executions. More precisely, we will construct the generator of a Discrete Time Markov Chain (DTMC) which represents the operational semantics of the program in question.

The generator matrix of the DTMC which we will construct for any given **pWhile** program defines a linear operator – thus we refer to it as a *Linear Operator Semantics* (LOS) – on a vector space based on the labelled blocks and classical states of the program in question.

The SOS transition relation – and in particular its restriction to the reachable configurations of a given program – can be directly encoded in a linear operator (cf. [16]), i.e. a matrix $\mathbf{T}$ defined for all configurations c_i, c_j by

$$(\mathbf{T})_{c_i,c_j} = \begin{cases} p \text{ if } \langle S_i, \sigma_i \rangle \longrightarrow_p \langle S_j, \sigma_j \rangle \\ 0 \text{ otherwise,} \end{cases}$$

However, this approach is in fact only a matrix representation of the SOS semantics and requires the construction of all possible execution trees. This is

in itself not compositional, i.e. if we know already the DTMC of a part of the program (e.g. a while loop) it is impossible or at least extremely difficult to describe the operational semantics of a program which contains this part.

Instead we present here a different construction which has the advantage of being compositional and therefore provides a more suitable basis for the compositional analysis in Section 4.2.

In order to be able to refer to particular program points in an unambiguous way we introduce a standard labelling (cf. [15])

$$
\begin{array}{rl}
S ::= & [\texttt{stop}]^\ell \\
\mid & [\texttt{skip}]^\ell \\
\mid & [v \texttt{ := } a]^\ell \\
\mid & [v \texttt{ ?= } r]^\ell \\
\mid & [S_1 \texttt{ ; } S_2 \\
\mid & [\texttt{choose}]^\ell\ p_1 : S_1 \texttt{ or } p_2 : S_2 \texttt{ ro} \\
\mid & \texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi} \\
\mid & \texttt{while } [b]^\ell \texttt{ do } S \texttt{ od}
\end{array}
$$

where ℓ is a label in **Lab** – typically just a unique number.

Classical and Probabilistic States. The probabilistic state of the computation is described via a probability measure over the space of (classical) states $\mathbf{State} = (\mathbf{Var} \to \mathbb{Z} + \mathbb{B})$.

In order to keep the mathematical treatment as simple as possible we will exploit the fact that **Var** is finite for any given program. We furthermore restrict the actual range of integer variables to a finite sub-set $\underline{\mathbb{Z}}$ of $\mathbb{Z}$. Although such a finite restriction is somewhat unsatisfactory from a purely theoretical point of view, it appears to be justified in the context of static program analysis (one could argue that any "real world" program has to be executed on a computer with certain memory limitations). As a result we can restrict our construction to probability *distributions* on **State**, i.e. $\mathbf{Dist}(\mathbf{State}) \subseteq \mathcal{V}(\mathbf{State})$ rather than referring to the more general notion of probability *measures* on states.

While in discrete, i.e. finite, probability spaces every measure can be defined via a distribution, the same does not hold any more for infinite state spaces, even for countable ones: it is, for example, impossible to define on the set of rationals in the interval $[0, 1]$ a kind of "uniform distribution" which would correspond to the Lebesgue measure.

As we consider only finitely many variables, $v = |\mathbf{Var}|$, we can represent the space of all possible states $\mathbf{Var} \to \underline{\mathbb{Z}} + \mathbb{B}$ as the Cartesian product $(\underline{\mathbb{Z}} + \mathbb{B})^v$, i.e. for every variable $\mathtt{v}_i \in \mathbf{Var}$ we specify its associated value in (a separate copy of) $\underline{\mathbb{Z}} + \mathbb{B}$. As the declarations of variables fix their types – in effect their possible range – we can exploit this information by presenting the state in a slightly more effective way:

$$\mathbf{State} = \mathbf{Value}_1 \times \mathbf{Value}_2 \ldots \times \mathbf{Value}_v$$

with $\mathbf{Value}_i = \underline{\mathbb{Z}}$ or $\mathbb{B}$. We will use the convention that, given v variables, we enumerate them according to the sequence in which they are declared in D.

Probabilistic Control Flow. We base the compositional construction of our LOS semantics on a probabilistic version of the *control flow* [15] or *abstract syntax* [17] of **pWhile** programs.

The flow $\mathcal{F} = \mathit{flow}$ is a set of triples $\langle \ell_i, p_{ij}, \ell_j \rangle$ which record the fact that control passes with probability p_{ij} from block B_i to block B_j, where a block is of the form $B_i = [\ldots]^{\ell_i}$. We assume label consistency, i.e. the labels on blocks are unique. We denote by $\mathbf{Block}(P)$ the set of all blocks and by $\mathbf{Lab}(P)$ the set of all labels in a program P. Except for the `choose` statement and the random assignment the probability p_{ij} is always equal to 1. For the `if` statement we indicate the control step into the `then` branch by underlining the target label; the same is the case for `while` statements.

The formal definition of the control flow of a program following the presentation in [15] is based on two auxiliary operations *init* and *final*

$$\begin{aligned} \mathit{init} &: \mathbf{Stmt} \to \mathbf{Lab} \\ \mathit{final} &: \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab}) \end{aligned}$$

which return the initial label and the final labels of a statement (whereas a sequence of statements has a single entry, it may have multiple exits, as for example in the conditional).

$$\begin{aligned} \mathit{init}([\texttt{skip}]^{\ell}) &= \ell \\ \mathit{init}([\texttt{stop}]^{\ell}) &= \ell \\ \mathit{init}([v \texttt{ := } e]^{\ell}) &= \ell \\ \mathit{init}([v \texttt{ ?= } e]^{\ell}) &= \ell \\ \mathit{init}(S_1; S_2) &= \mathit{init}(S_1) \\ \mathit{init}([\texttt{choose}]^{\ell}\ p_1 : S_1 \texttt{ or } p_2 : S_2 \texttt{ ro}) &= \ell \\ \mathit{init}(\texttt{if } [b]^{\ell} \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) &= \ell \\ \mathit{init}(\texttt{while } [b]^{\ell} \texttt{ do } S \texttt{ od}) &= \ell \end{aligned}$$

and

$$\begin{aligned} \mathit{final}([\texttt{skip}]^{\ell}) &= \{\ell\} \\ \mathit{final}([\texttt{stop}]^{\ell}) &= \{\ell\} \\ \mathit{final}([v \texttt{ := } e]^{\ell}) &= \{\ell\} \\ \mathit{init}([v \texttt{ ?= } e]^{\ell}) &= \{\ell\} \\ \mathit{final}(S_1; S_2) &= \mathit{final}(S_2) \\ \mathit{final}([\texttt{choose}]^{\ell}\ p_1 : S_1 \texttt{ or } p_2 : S_2 \texttt{ ro}) &= \mathit{final}(S_1) \cup \mathit{final}(S_2) \\ \mathit{final}(\texttt{if } [b]^{\ell} \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) &= \mathit{final}(S_1) \cup \mathit{final}(S_2) \\ \mathit{final}(\texttt{while } [b]^{\ell} \texttt{ do } S \texttt{ od}) &= \{\ell\} \end{aligned}$$

The probabilistic control flow $\mathcal{F}(S) = \mathit{flow}(S)$ is then defined via the a function *flow*

$$\mathit{flow} : \mathbf{Stmt} \to \mathcal{P}(\mathbf{Lab} \times [0,1] \times \mathbf{Lab})$$

which maps statements to sets of triples which represent the probabilistic control flow graph:

$$\begin{aligned}
\mathit{flow}([\texttt{skip}]^\ell) &= \emptyset \\
\mathit{flow}([\texttt{stop}]^\ell) &= \{\langle \ell, 1, \ell\rangle\} \\
\mathit{flow}([v\ \texttt{:=}\ e]^\ell) &= \emptyset \\
\mathit{flow}([v\ \texttt{?=}\ e]^\ell) &= \emptyset \\
\mathit{flow}(S_1; S_2) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \\
&\quad \cup \{(\ell, 1, \mathit{init}(S_2)) \mid \ell \in \mathit{final}(S_1)\} \\
\mathit{flow}([\texttt{choose}]^\ell\ p_1 : S_1\ \texttt{or}\ p_2 : S_2\ \texttt{ro}) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \\
&\quad \cup \{(\ell, p_1, \mathit{init}(S_1)), (\ell, p_2, \mathit{init}(S_2))\} \\
\mathit{flow}(\texttt{if}\ [b]^\ell\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}) &= \mathit{flow}(S_1) \cup \mathit{flow}(S_2) \cup \\
&\quad \cup \{(\ell, 1, \underline{\mathit{init}(S_1)}), (\ell, 1, \mathit{init}(S_2))\} \\
\mathit{flow}(\texttt{while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}) &= \mathit{flow}(S) \cup \\
&\quad \cup \{(\ell, 1, \underline{\mathit{init}(S)})\} \cup \{(\ell', 1, \ell) \mid \ell' \in \mathit{final}(S)\}
\end{aligned}$$

Example 4. Consider the following labelled program P:

```
var
  z : {0..200};
begin
while [z<100]^1 do
  [choose]^2 1/3:[x:=3]^3 or 2/3:[x:=1]^4 ro;
od;
[stop]^5;
end
```

The flow of this program is given by:

$$\mathit{flow}(P) = \{\langle 1, 1, \underline{2}\rangle, \langle 1, 1, 5\rangle, \langle 2, \frac{1}{3}, 3\rangle, \langle 2, \frac{2}{3}, 4\rangle, \langle 3, 1, 1\rangle, \langle 4, 1, 1\rangle\}.$$

Probabilistic Configurations. The construction of the DTMC representing the probabilistic semantics of a **pWhile** program will – as in the SOS case, represent a transition relation on configurations in **Conf**. In the classical case configurations are pairs formed by the (remaining) program S which is to be executed and the computational state σ representing the current values of all the variables. For **pWhile** it is in fact enough to just record the initial $\mathit{init}(S)$ statement of the program S. In other words, a classical configuration is an element in $\mathbf{Stmt} \times \mathbf{State}$ or just in $\mathbf{Block} \times \mathbf{State}$.

For probabilistic programs there is in general no unique configuration which describes the current situation when we execute a program. Instead we need to use distributions (in general measures) over classical configurations, i.e. configurations are elements in

$$\mathbf{Dist}(\mathbf{State} \times \mathbf{Block}) \subseteq \mathcal{V}(\mathbf{State} \times \mathbf{Block}).$$

In order to distinguish between classical and probabilistic states we reverse the order between the value part and the syntactic part of configurations.

Exploiting the fact that states can also be described as tuples in the Cartesian products of values of each variable and that blocks are uniquely labelled we identify as the space of configurations describing the computational situation in the probabilistic case as

$$\mathbf{Dist}(\mathbf{Value}_1 \times \ldots \times \mathbf{Value}_v \times \mathbf{Lab}) \subseteq \mathcal{V}(\mathbf{Value}_1 \times \ldots \times \mathbf{Value}_v \times \mathbf{Lab}).$$

Finally we observe the important isomorphism between the vector space over Cartesian products and the tensor product of vector spaces. This allows us to construct probabilistic configurations as elements in

$$\mathcal{V}(\mathbf{Value}_1 \times \ldots \times \mathbf{Value}_v \times \mathbf{Lab}) = \mathcal{V}(\mathbf{Value}_1) \otimes \ldots \otimes \mathcal{V}(\mathbf{Value}_v) \otimes \mathcal{V}(\mathbf{Lab}).$$

This decomposition of the space of configurations is the basis for a compositional description of the DTMC generator which defines the semantics of a program. In particular, by the finiteness condition for $\mathbf{Value}_i$ and the fact that **Lab** is always finite we know immediately the finite set of (potentially) reachable states and thus the state of probabilistic configurations. Furthermore, we will exploit the tensor product to describe the DTMC generator as a linear combination of local updates. These factors themselves are given as tensor products of operators which describe the computational dynamics of individual statements (blocks).

Basic Operators. In order to construct the concrete semantics we need to identify those states which satisfy certain conditions, e.g. all those states where a variable has a value larger than 5. This is achieved by "filtering" states which fulfil some conditions via *projection operators*, which are concretely represented by diagonal matrices.

Consider a variable $\mathtt{x}$ together with the set of its possible values $\mathbf{Value} = \{v_1, v_2, \ldots\}$, and the vector space $\mathcal{V}(\mathbf{Value})$. The probabilistic state of the variable v can be described by a distribution over its possible values, i.e. a vector in $\mathcal{V}(\mathbf{Value})$. For example, if we know that $\mathtt{x}$ holds the value v_1 or v_3 with probabilities $\frac{1}{3}$ and $\frac{2}{3}$ respectively (and no other values) then this situation is represented by the vector $(\frac{1}{3}, 0, \frac{2}{3}, 0, \ldots)$.

As we represent distributions by row vectors $\boldsymbol{x}$ the application of a linear map corresponds to a post-multiplication by the corresponding matrix $\mathbf{T}$, i.e. $\mathbf{T}(\boldsymbol{x}) = \boldsymbol{x} \cdot \mathbf{T}$.

We might need to apply a transformation $\mathbf{T}$ to the probabilistic state of the variable $\mathtt{x}_i$ only when a certain condition is fulfilled. We can express such a condition by a predicate q on $\mathbf{Value}_i$. Defining a diagonal matrix $\mathbf{P}$ with otherwise,

$$(\mathbf{P})_{ii} = \begin{cases} 1 \text{ if } p(v_i) \text{ holds} \\ 0 \text{ otherwise.} \end{cases}$$

allows us to "filter out" only those states which fulfil the condition q, i.e. $\mathbf{P} \cdot \mathbf{T}$ applies $\mathbf{T}$ only to those states.

$$(\mathbf{E}(m,n))_{ij} = \begin{cases} 1 \text{ if } m = i \wedge n = j \\ 0 \text{ otherwise.} \end{cases}$$

$$(\mathbf{I})_{ij} = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ otherwise.} \end{cases}$$

Table 2. Basic Operators for **pWhile**

$$(\mathbf{P}(c))_{ij} = \begin{cases} 1 \text{ if } i = c = j \\ 0 \text{ otherwise.} \end{cases}$$

$$\mathbf{P}(\sigma) = \bigotimes_{i=1}^{v} \mathbf{P}(\sigma(\mathtt{x}_i))$$

$$\mathbf{P}(e = c) = \sum_{\mathcal{E}(e)\sigma = c} \mathbf{P}(\sigma)$$

Table 3. Test or Filter Operators **pWhile**

The Linear Operator Semantics (LOS) of **pWhile** is built using a number of basic operators which can be represented by the (sparse) square matrices specified in Table 2. The matrix units $\mathbf{E}(m,n)$ contains only one non-zero entry, and $\mathbf{I}$ is the identity operator.

Using these basic building blocks we can define a number of "filters" $\mathbf{P}$ as depicted in Table 3. The operator $\mathbf{P}(c)$ has only one non-zero entry: the diagonal element $\mathbf{P}_{cc} = 1$, i.e. $\mathbf{P}(c) = \mathbf{E}(c,c)$. This operator extracts the probability corresponding to the c-th coordinate of a vector, i.e. for $\boldsymbol{x} = (x_i)_i$ the multiplication with $\mathbf{P}(c)$ results in a vector $\boldsymbol{x'} = \boldsymbol{x} \cdot \mathbf{P}(c)$ with only one non-zero coordinate, namely $x'_c = x_c$.

The operator $\mathbf{P}(\sigma)$ performs a similar test for a vector representing the probabilistic state of the computation. It filters the probability that the computation is in a classical state σ. This is achieved by checking whether each variable $\mathtt{x}_i$ has the value specified by σ namely $\sigma(\mathtt{x}_i)$. Finally, the operator $\mathbf{P}(e = c)$ filters those states where the values of the variables $\mathtt{x}_i$ are such that the evaluation of the expression e results in c. The number of (diagonal) non-zero entries of this operator is exactly the number of states σ for which $\mathcal{E}(e)\sigma = c$.

$$(\mathbf{U}(c))_{ij} = \begin{cases} 1 \text{ if } j = c \\ 0 \text{ otherwise.} \end{cases}$$

$$\mathbf{U}(\mathtt{x}_k \leftarrow c) = \bigotimes_{i=1}^{k-1} \mathbf{I} \otimes \mathbf{U}(c) \otimes \bigotimes_{i=k+1}^{v} \mathbf{I}$$

$$\mathbf{U}(\mathtt{x}_k \leftarrow e) = \sum_{c} \mathbf{P}(e = c)\mathbf{U}(\mathtt{x}_k \leftarrow c)$$

Table 4. Update Operators for **pWhile**

LOS Semantics. The update operators (see Table 4) implement state changes. From an initial probabilistic state σ, i.e. a distribution over classical states, we get a new probabilistic state σ' via $\sigma' = \sigma \cdot \mathbf{U}$

The simple operator $\mathbf{U}(c)$ implements the deterministic update of a variable $\mathtt{x}_i$: Whatever the value(s) of $\mathtt{x}_i$ are, after applying $\mathbf{U}(c)$ to the state vector describing $\mathtt{x}_i$ we get a point distribution expressing the fact that the value of $\mathtt{x}_i$ is now certainly c. The operator $\mathbf{U}(\mathtt{x}_k \leftarrow c)$ puts $\mathbf{U}(c)$ into the context of other variables: Most factors in the tensor product are identities, i.e. most variables keep their previous values, only $\mathtt{x}_k$ is deterministically updated to its new value c using the previously defined $\mathbf{U}(c)$ operator. The operator $\mathbf{U}(\mathtt{x}_k \leftarrow e)$ updates a variable not to a constant but to the value of an expression e. This update is realised using the filter operator $\mathbf{P}(e = c)$: For all possible values c of e we select those states where e evaluates to c and then update $\mathtt{x}_k$ to this c.

The full LOS semantics of a **pWhile** program P is defined as the operator $\mathbf{T} = \mathbf{T}(P)$ on $\mathcal{V}(\mathbf{State} \times \mathcal{B}(P))$. This concrete semantics of a program P is given by:

$$\mathbf{T}(P) = \sum_{\langle i, p_{ij}, j \rangle \in \mathcal{F}(P)} p_{ij} \cdot \mathbf{T}(\ell_i, \ell_j).$$

The meaning of $\mathbf{T}(P)$ is to collect for every triple in the probabilistic flow $\mathcal{F}(P)$ of P its effects, weighted according the probability associated to this triple. The operators $\mathbf{T}(\ell_i, \ell_j)$ which implement the local state updates and control transfers from ℓ_i to ℓ_j are presented in Table 5.

Each local operator $\mathbf{T}(\ell_i, \ell_j)$ is of the form $\mathbf{N} \otimes \mathbf{E}(\ell_i, \ell_j)$ where the first factor $\mathbf{N}$ represents a state update or, in the case of tests, a filter operator while the second factor realises the transfer of control from label ℓ_i to label ℓ_j. For the `skip` and `stop` no changes to the state happen, we only transfer control (deterministically) to the next statement or loop on the current (terminal) statement using matrix units $\mathbf{E}$. Also in the case of a **choose** there is no change to the state but only a transfer of control, however the probabilities p_{ij} will in general

$\mathbf{T}(\ell_1, \ell_2) = \mathbf{I} \otimes \mathbf{E}(\ell_1, \ell_2)$	for $[\texttt{skip}]^{\ell_1}$
$\mathbf{T}(\ell, \ell) = \mathbf{I} \otimes \mathbf{E}(\ell, \ell)$	for $[\texttt{stop}]^{\ell}$
$\mathbf{T}(\ell_1, \ell_2) = \mathbf{U}(v \leftarrow e) \otimes \mathbf{E}(\ell_1, \ell_2)$	for $[v \texttt{ := } e]^{\ell_1}$
$\mathbf{T}(\ell_1, \ell_2) = \left(\frac{1}{\lvert r \rvert} \sum_{c \in r} \mathbf{U}(v \leftarrow c)\right) \otimes \mathbf{E}(\ell_1, \ell_2)$	for $[v \texttt{ ?= } r]^{\ell_1}$
$\mathbf{T}(\ell, \ell_k) = \mathbf{I} \otimes \mathbf{E}(\ell, \ell_k)$	for $[\texttt{choose}]^{\ell}$
$\mathbf{T}(\ell, \underline{\ell_t}) = \mathbf{P}(b = \texttt{true}) \otimes \mathbf{E}(\ell, \ell_t)$	for $[b]^{\ell}$
$\mathbf{T}(\ell, \ell_f) = \mathbf{P}(b = \texttt{false}) \otimes \mathbf{E}(\ell, \ell_f)$	for $[b]^{\ell}$

Table 5. Linear Operator Semantics for **pWhile**

be different from 1, unlike `skip`. With assignments we have both a state update, implemented using $\mathbf{U}(p \leftarrow e)$ as well as a control flow step. For tests b we use a filter operator $\mathbf{P}(b = \texttt{true})$ to select those states which pass the test or $\mathbf{P}(b = \texttt{false})$ fail it to determine to which label control will pass.

Proposition 1. *The operator $\mathbf{T}(P)$ is a stochastic matrix for any* **pWhile** *program P, i.e. the sum of all elements in each row add up to one.*

Thus, $\mathbf{T}$ is indeed the generator of a DTMC. Furthermore, by the construction of $\mathbf{T}$ it also follows immediately that the SOS and LOS semantics are equivalent in the following sense.

Proposition 2. *For any* **pWhile** *program P and any classical state $\sigma \in$* **State***, we have: $\langle S, \sigma\rangle \longrightarrow_p \langle S', \sigma'\rangle$ iff $(\mathbf{T}(P))_{\langle \sigma, \ell\rangle, \langle \sigma', \ell'\rangle} = p$,where ℓ and ℓ' label the first block in the statement S and S', respectively.*

It is an easy exercise to introduce additional languages features, e.g. pointers (see [18]) or (probabilistic) jumps, i.e. `goto`s, or sub-routines.

3.5 Example

Example 5 (Monty Hall). Consider again our running Example 1. The labelled version of the program H_w (with s**w**itching doors) is:

```
var
  d :{0,1,2};
  g :{0,1,2};
  o :{0,1,2};
begin
[d ?= {0,1,2}]^1;
[g ?= {0,1,2}]^2;
[o ?= {0,1,2}]^3;
while [((o == g)||(o == d))]^4 do
  [o := (o+1)%3]^5;
od;
[g := (g+1)%3]^6;
while [(g == o)]^7 do
  [g := (g+1)%3]^8;
od;
[stop]^9;
end
```

The blocks for this program H_w are thus

$$\begin{aligned}\mathbf{Block}(H_w) = \{&\texttt{[d ?= \{ 0, 1, 2 \}]}^1, \texttt{[g ?= \{ 0, 1, 2 \}]}^2,\\ &\texttt{[o ?= \{ 0, 1, 2 \}]}^3, \texttt{[((o == g)||(o == d))]}^4,\\ &\texttt{[o := ((o + 1) \% 3)]}^5, \texttt{[g := ((g + 1) \% 3)]}^6,\\ &\texttt{[(g == o)]}^7, \texttt{[g := ((g + 1) \% 3)]}^8, \texttt{[stop]}^9\}\end{aligned}$$

and the flow

$$\begin{aligned}\mathbf{Flow}(H_w) = \{&(1,1,2),(2,1,3),(3,1,4),(4,1,\underline{5}),(5,1,4),(4,1,6),\\ &(6,1,7),(7,1,\underline{8}),(8,1,7),(7,1,9),(9,1,9)\}\end{aligned}$$

The elements describing the version H_t where we s**t**ick to the original door is identical to this one, except that it just involves labels 1 to 5 and then a final $\texttt{[stop]}^6$ (instead of $\texttt{[stop]}^9$). Note that this program does not use probabilistic choices and so the second element of all entries in flow are 1 (though we use random assignments here).

Following the definition of the LOS semantics we can construct the transition operators, i.e. the generators of the corresponding DTMCs, in a straight forward way.

$$\begin{aligned}\mathbf{T}(H_t) = &\frac{1}{3}\left(\mathbf{U}(\texttt{d} \leftarrow 0) + \mathbf{U}(\texttt{d} \leftarrow 1) + \mathbf{U}(\texttt{d} \leftarrow 2)\right) \otimes \mathbf{E}(1,2) +\\ &\frac{1}{3}\left(\mathbf{U}(\texttt{g} \leftarrow 0) + \mathbf{U}(\texttt{g} \leftarrow 1) + \mathbf{U}(\texttt{g} \leftarrow 2)\right) \otimes \mathbf{E}(2,3) +\\ &\frac{1}{3}\left(\mathbf{U}(\texttt{o} \leftarrow 0) + \mathbf{U}(\texttt{o} \leftarrow 1) + \mathbf{U}(\texttt{o} \leftarrow 2)\right) \otimes \mathbf{E}(3,4) +\\ &\mathbf{P}((\texttt{o} == \texttt{g})||(\texttt{o} == \texttt{d}) = \texttt{true}) \otimes \mathbf{E}(4,5) +\\ &\mathbf{P}((\texttt{o} == \texttt{g})||(\texttt{o} == \texttt{d}) = \texttt{false}) \otimes \mathbf{E}(4,6) +\\ &\mathbf{I} \otimes \mathbf{E}(6,6)\end{aligned}$$

and

$$\begin{aligned}\mathbf{T}(H_w) = & \frac{1}{3}\left(\mathbf{U}(\mathtt{d} \leftarrow 0) + \mathbf{U}(\mathtt{d} \leftarrow 1) + \mathbf{U}(\mathtt{d} \leftarrow 2)\right) \otimes \mathbf{E}(1,2) + \\ & \frac{1}{3}\left(\mathbf{U}(\mathtt{g} \leftarrow 0) + \mathbf{U}(\mathtt{g} \leftarrow 1) + \mathbf{U}(\mathtt{g} \leftarrow 2)\right) \otimes \mathbf{E}(2,3) + \\ & \frac{1}{3}\left(\mathbf{U}(\mathtt{o} \leftarrow 0) + \mathbf{U}(\mathtt{o} \leftarrow 1) + \mathbf{U}(\mathtt{o} \leftarrow 2)\right) \otimes \mathbf{E}(3,4) + \\ & \mathbf{P}((\mathtt{o} == \mathtt{g}) || (\mathtt{o} == \mathtt{d}) = \mathtt{true}) \otimes \mathbf{E}(4,5) + \\ & \mathbf{P}((\mathtt{o} == \mathtt{g}) || (\mathtt{o} == \mathtt{d}) = \mathtt{false}) \otimes \mathbf{E}(4,6) + \\ & \mathbf{U}(\mathtt{g} \leftarrow (g+1)\%3) \otimes \mathbf{E}(6,7) + \\ & \mathbf{P}((\mathtt{g} == \mathtt{o}) = \mathtt{true}) \otimes \mathbf{E}(7,8) + \\ & \mathbf{P}((\mathtt{g} == \mathtt{o}) = \mathtt{false}) \otimes \mathbf{E}(7,9) + \\ & \mathbf{U}(\mathtt{g} \leftarrow (g+1)\%3) \otimes \mathbf{E}(6,7) + \\ & \mathbf{I} \otimes \mathbf{E}(9,9)\end{aligned}$$

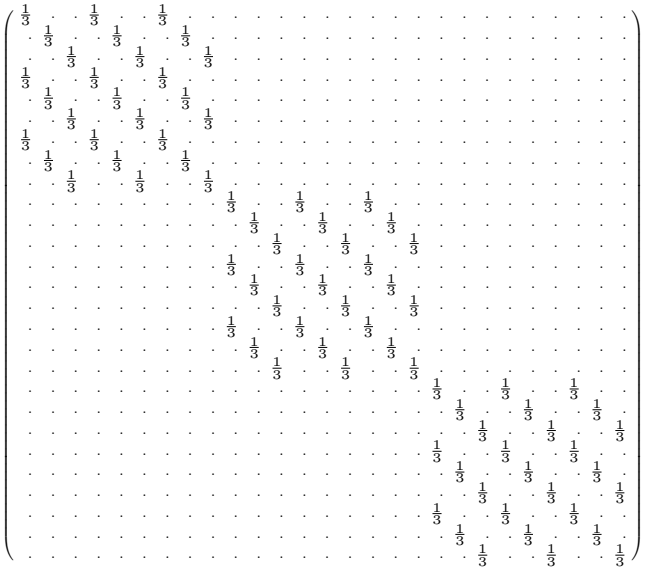
The individual update operators $\mathbf{T}(i,j)$ are then given by the following matrices where the enumeration of elements in **State** is as follows

$1 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 0)$	$10 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 0)$	$19 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 0)$
$2 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 1)$	$11 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 1)$	$20 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 1)$
$3 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 2)$	$12 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 2)$	$21 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 0, \mathtt{o} \mapsto 2)$
$4 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 0)$	$13 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 0)$	$22 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 0)$
$5 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 1)$	$14 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 1)$	$23 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 1)$
$6 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 2)$	$15 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 2)$	$24 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 1, \mathtt{o} \mapsto 2)$
$7 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 0)$	$16 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 0)$	$25 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 0)$
$8 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 1)$	$17 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 1)$	$26 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 1)$
$9 \ldots (\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 2)$	$18 \ldots (\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 2)$	$27 \ldots (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 2, \mathtt{o} \mapsto 2)$

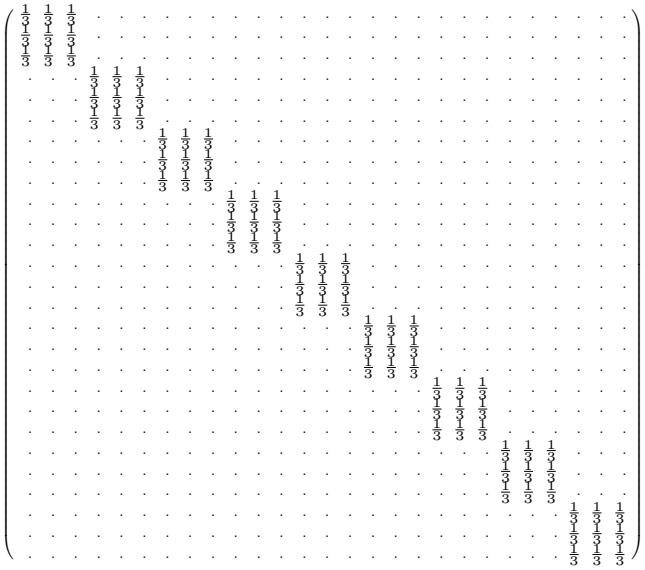
For the first three random assignments we have

$$\mathbf{T}(1,2) = \begin{pmatrix}
\frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . \\
. & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . \\
. & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . \\
. & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . \\
. & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . \\
. & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . \\
. & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . \\
. & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . \\
. & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} \\
\frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . \\
. & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . \\
. & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . \\
. & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . \\
. & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . \\
. & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . \\
. & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . \\
. & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . \\
. & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} \\
\frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . \\
. & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . \\
. & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . \\
. & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . \\
. & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . \\
. & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . \\
. & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . \\
. & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . \\
. & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3} & . & . & . & . & . & . & . & . & \frac{1}{3}
\end{pmatrix} \otimes \mathbf{E}(1,2)$$

$$\mathbf{T}(2,3) = \left(\begin{array}{c} \\ \end{array} \right) \otimes \mathbf{E}(2,3)$$



$$\mathbf{T}(3,4) = \left(\begin{array}{c} \\ \end{array} \right) \otimes \mathbf{E}(3,4)$$



In each of these cases we have an update of the values of one of the three variables `d`, `g` and `o` together with a (deterministic) transfer control from one label to the next implemented via a matrix unit $\mathbf{E}(1,2)$, $\mathbf{E}(2,3)$ and $\mathbf{E}(3,4)$.

After that we have to consider the projections which implement the guard of the first while loop. If the current state of the variables `d`, `g` and `o` fulfils the condition that `(o == g) || (o ==d)` then we transfer control to label 5, otherwise to label 6. The filter or projections which determine which control

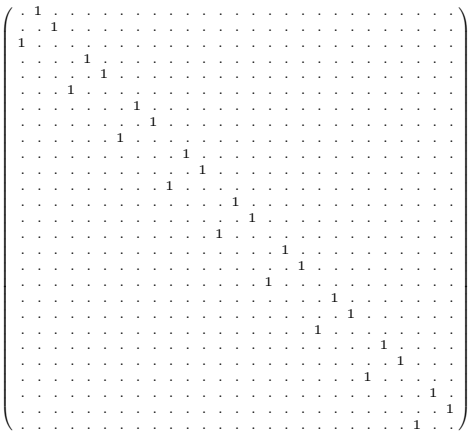
transfer will be executed are diagonal matrices such that the transfer operators are given by:

$$\mathbf{T}(4,\underline{5}) = \text{diag}\left(1\,0\,0\,1\,1\,0\,1\,0\,1\,1\,1\,0\,0\,1\,0\,0\,1\,1\,1\,0\,1\,0\,1\,1\,0\,0\,1\right) \otimes \mathbf{E}(4,5)$$

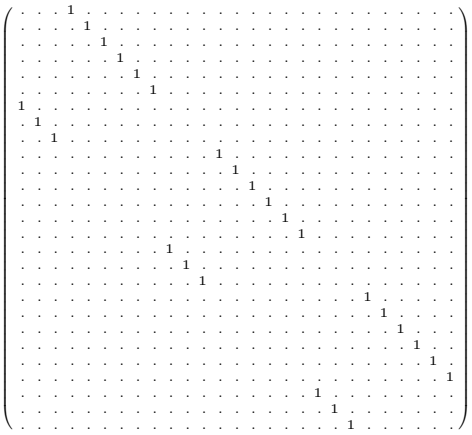
and

$$\mathbf{T}(4,6) = \text{diag}\left(0\,1\,1\,0\,0\,1\,0\,1\,0\,0\,0\,1\,1\,0\,1\,1\,0\,0\,0\,1\,0\,1\,0\,0\,1\,1\,0\right) \otimes \mathbf{E}(4,6).$$

The single statement labelled with 5 is a deterministic update and is combined with a return to the guard of the loop labelled with 4, i.e.

$$\mathbf{T}(5,4) = \left(\begin{array}{c}\text{  }\end{array}\right) \otimes \mathbf{E}(5,4)$$

If we are not switching doors then we can construct essentially with these matrices the full LOS semantics of H_t. For the switching strategy we also need to execute the switch. This starts with the statement at label 6 which its associated control step from label 6 to 7.

$$\mathbf{T}(6,7) = \left(\begin{array}{c}\text{  }\end{array}\right) \otimes \mathbf{E}(6,7)$$

After this we have another loop which is guarded by the test `g == o` labelled with 7. To implement this we need again two diagonal matrices which allow us to construct

$$\mathbf{T}(7,\underline{8}) = \left(1\,0\,0\,0\,1\,0\,0\,0\,1\,1\,0\,0\,0\,1\,0\,0\,0\,1\,1\,0\,0\,0\,1\,0\,0\,0\,1\right) \otimes \mathbf{E}(7,8)$$

and

$$\mathbf{T}(7,9) = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \otimes \mathbf{E}(7,9)$$

Finally, we only need the LOS semantics of the single statement which makes up the loop we have to avoid changing our guess to the already open door. The matrix for this is given by

$$\mathbf{T}(8,7) = \begin{pmatrix}
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{pmatrix} \otimes \mathbf{E}(8,7)$$

Note that the update part of $\mathbf{T}(8,7)$ is exactly the same as for $\mathbf{T}(6,7)$. This reflects the fact that we have the same statement at label 6 and 8: `g := (g+1)%3`. The LOS semantics is compositional and syntax directed, it treats syntactically equivalent parts in exactly the same way. The only difference between $\mathbf{T}(8,7)$ and $\mathbf{T}(6,7)$ is the different "continuation" after the update.

The final $[\texttt{stop}]^\ell$ statement will just introduce an infinite loop on the terminal state we have reached (in order to "preserve" the final result). It is implemented both for the program H_t as well as for the program H_w as the matrix $\mathbf{I} \otimes \mathbf{E}(\ell, \ell)$.

Putting everything together we get the LOS semantics for both programs as;

$$\begin{aligned}\mathbf{T}(H_s) = \mathbf{T}(1,2) + \mathbf{T}(2,3) + \mathbf{T}(3,4) + \mathbf{T}(4,\underline{5}) + \mathbf{T}(5,4) + \mathbf{T}(4,6) + \\ + \mathbf{I} \otimes \mathbb{E}(6,6)\end{aligned}$$

and

$$\begin{aligned}\mathbf{T}(H_w) = \mathbf{T}(1,2) + \mathbf{T}(2,3) + \mathbf{T}(3,4) + \mathbf{T}(4,\underline{5}) + \mathbf{T}(5,4) + \mathbf{T}(4,6) + \\ + \mathbf{T}(6,7) + \mathbf{T}(7,\underline{8}) + \mathbf{T}(8,7) + \mathbf{T}(7,9) + \mathbf{I} \otimes \mathbb{E}(9,9)\end{aligned}$$

This is for H_t a $(27 \cdot 5) \times (27 \cdot 5) = 162 \times 162$ matrix and for H_t a $(27 \cdot 9) \times (27 \cdot 9) = 243 \times 243$ matrix which we omit to present in full. These matrices encode all possible transitions between all (potentially) reachable configurations. For any probabilistic state, i.e. distributions over **State**, we can construct the successor configuration using just this matrices.

4 Probabilistic Static Analysis

Program analysis is a collection of techniques to predict in advance what will happen when a program is executed. Classically, such information could be used to optimise the code produced by a compiler; more recently this has formed the basis for the automatic debugging, verification and certification of code. Since well known un-decidability results tell us that it is impossible to know everything about the behaviour of every program, we can only aim for partial answers to some of the questions. Program analysis techniques allows us to obtain such partial answers via the use of approximation and abstraction aiming at reducing/simplifying the problem under consideration. It is now widely recognised that considering probabilistic information allows one to obtain more precise and practically useful results from a static analysis. Statistical or other types of information can be encoded in the form of probabilities in the program semantics and used to weight the execution paths according to their likelihood to actually be executed. We will show here how one of the basic techniques of static analysis, namely Abstract Interpretation, can be extended so as to include quantitative information in the form of both probabilities associated to the analysed programs, and estimates of the precision of the resulting analyses.

4.1 Classical Abstract Interpretation

The basic idea behind abstract interpretation is to analyse a program not in terms of its standard or *concrete semantics*, but rather in terms of an appropriately simplified approximated or *abstract semantics*, which only registers aspects of the program which are of relevance with respect to a specific analysis (cf. [19, 20, 15]). Typically these aspects are encoded in the definition of an abstract domain, which is usually structured, like the concrete domain, as a complete partially ordered set.

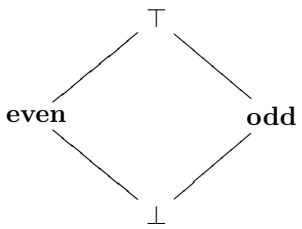
The idea is to execute the program using *abstract* instead of *concrete* values to describe the current state or configuration. For an informal introduction let us consider the factorial programs.

Example 6 (Factorial). Considering the two programs in Example 2 we get the following labelling:

```
var                          var
  m : {0..2};                  m : {0..2};
  n : {0..2};                  n : {0..2};
begin                        begin
[m := 1]^1;                  [m := 2]^1;
while [(n>1)]^2 do           while [(n>1)]^2 do
  [m := m*n]^3;;               [m := m*n]^3;
  [n := n-1]^4;;               [n := n-1]^4;
od;                          od;
[stop]^5;                    [stop]^5;
end                          end
```

The idea is now to analyse the properties of the states during the execution of the program rather than the actual or concrete values of the variables. To demonstrate this idea let us look at the *parity* of the variables, i.e. whether they are even or odd.

The abstract property we are interested in is the description of the possible parities of the variables `m` and `n`: If we can guarantee that a variable is *always* 'even' when we reach a certain program point then we associate to it the abstract value or property **even**; if on the other hand we are certain that a variable is *always* 'odd', then we use **odd** as its abstract value. However, we also have to take care of the case when we are not sure about the parity of a variable: it could be sometimes even or sometimes odd. We use the value $\top$ to indicate this **ambiguous** situation. We can distinguish this situation from another kind of **unknown value** $\bot$ we use to handle non-initialised variables which are neither even nor odd. This situation can be formalised using the notion of a *lattice* $\mathcal{L}$, cf. [21]:



which expresses the relation between abstract values as an *order* relation, e.g. $\top$ is more general than **even** and **odd**, i.e. if we know that a variable could be even and odd then the general statement which describes its (abstract) value is to say that its parity is **ambiguous** or $\top$. We can interprete this property lattice also as the power-set of $\{\mathbf{even}, \mathbf{odd}\}$, i.e. $\mathcal{L} = \mathcal{P}(\{\mathbf{even}, \mathbf{odd}\})$, identifying $\top = \{\mathbf{even}, \mathbf{odd}\}$ and $\bot = \emptyset$ and ordered by inclusion "$\subseteq$".

We now consider the abstract execution of the "double factorial" program (on the left-hand side above). Two cases are possible: One where the guard in label 2 fails, and one where we enter the loop. The abstract values we can associate in these two cases (assuming that we start with unknown rather than non-initialised values) are:

$1 : \mathtt{m} \mapsto \top, \quad \mathtt{n} \mapsto \top$	$1 : \mathtt{m} \mapsto \top, \quad \mathtt{n} \mapsto \top$
$2 : \mathtt{m} \mapsto \mathbf{even}, \mathtt{n} \mapsto \top$	$2 : \mathtt{m} \mapsto \mathbf{even}, \mathtt{n} \mapsto \top$
$3 :$	$3 : \mathtt{m} \mapsto \mathbf{even}, \mathtt{n} \mapsto \top$
$4 :$	$4 : \mathtt{m} \mapsto \mathbf{even}, \mathtt{n} \mapsto \top$
$5 : \mathtt{m} \mapsto \mathbf{even}, \mathtt{n} \mapsto \top$	$5 : \mathtt{m} \mapsto \mathbf{even}, \mathtt{n} \mapsto \top$

We observe that the parity of `n` remains ambiguous throughout the execution of the program. However, whether or not the loop is executed, the parity of `m` will always be **even** when we reach the final label 5: If we omit the loop then the even value 2 we assigned to `m` is directly used; if we execute the loop, then `m` enters the loop at the first iteration with an **even** value and remains **even** despite the fact that in label 3 it is multiplied with an unknown `n` because we know that the

product of an even number with any number results again in an even number. In any subsequent iteration the same argument holds. Thus, whenever the loop terminates, we will always be certain that `m` is **even** when we reach label 5. The "double factorial" always produces an even result.

If we consider the program on the right-hand side, which implements the simple "factorial" then our arguments break down. The abstract executions in this case give us:

$1 : \mathtt{m} \mapsto \top, \quad \mathtt{n} \mapsto \top$	$1 : \mathtt{m} \mapsto \top, \quad \mathtt{n} \mapsto \top$
$2 : \mathtt{m} \mapsto \mathbf{odd}, \mathtt{n} \mapsto \top$	$2 : \mathtt{m} \mapsto \mathbf{odd}, \mathtt{n} \mapsto \top$
$3 :$	$3 : \mathtt{m} \mapsto \top, \quad \mathtt{n} \mapsto \top$
$4 :$	$4 : \mathtt{m} \mapsto \top, \quad \mathtt{n} \mapsto \top$
$5 : \mathtt{m} \mapsto \mathbf{odd}, \mathtt{n} \mapsto \top$	$5 : \mathtt{m} \mapsto \top, \quad \mathtt{n} \mapsto \top$

If the loop is not executed we can guarantee that `m` is **odd**; but if we execute the loop then we have to multiply (in the first iteration) an odd `m` with an unknown `n` and we cannot guarantee any particular parity for `m` from then on. As a result the analysis will return $\top$ for the parity of `m` at label 5.

The factorial indeed may give an odd value (for 0 and 1) but it is obvious that for "most" values of `n` it will be an even number. The classical analysis is conservative and unable to extract this information. The remainder of these notes aims in developing a framework which allows for a formal analysis which captures such a "probabilistic" intuition.

A detailed formal discussion of the relation between the concrete values of `m` and `n` as sub-sets of $\mathbb{Z}$, i.e. as elements in the power-set $\mathcal{P}(\mathbb{Z})$ (which also forms a lattice in a canonical way via the sub-set relation) and their abstract values in $\mathcal{L}$ is beyond the the scope of these notes. For our purposes, it is sufficient to say that there exists a *abstraction* function α between the concrete and abstract values of `m` and `n` and a formal way to define an abstract semantics describing our factorial programs in terms of these abstract values by constructing the "right" *concretisation* function γ.

In the standard theory of *abstract interpretation*, which was introduced by Cousot & Cousot 30 years ago [22, 23], the correctness of an abstract semantics is guaranteed by ensuring that we have a pair of functions α and γ which form a *Galois connection* between two lattices $\mathcal{C}$ and $\mathcal{D}$ representing concrete and abstract properties.

Definition 1. *Let $\mathcal{C} = (\mathcal{C}, \leq_{\mathcal{C}})$ and $\mathcal{D} = (\mathcal{D}, \leq_{\mathcal{D}})$ be two partially ordered set (e.g. lattices). If there are two functions $\alpha : \mathcal{C} \mapsto \mathcal{D}$ and $\gamma : \mathcal{D} \mapsto \mathcal{C}$ such that for all $c \in \mathcal{C}$ and all $d \in \mathcal{D}$:*

$$c \leq_{\mathcal{C}} \gamma(d) \text{ iff } \alpha(c) \leq_{\mathcal{D}} d,$$

then $(\mathcal{C}, \alpha, \gamma, \mathcal{D})$ forms a Galois connection.

The intended meaning is that an abstract element d approximates a concrete one c if $c \leq_{\mathcal{C}} \gamma(d)$ or equivalently (by adjunction) if $\alpha(c) \leq_{\mathcal{D}} d$. Therefore,

the concrete value corresponding to an abstract denotation d is $\gamma(d)$, while the adjunction guarantees that $\alpha(c)$ is the best possible approximation of c in $\mathcal{D}$ (because whenever d is a correct approximation of c, then $\alpha(c) \leq_{\mathcal{D}} d$).

An abstract function $f^{\#} : \mathcal{D} \mapsto \mathcal{D}$ is a *correct* approximation of a concrete function $f : \mathcal{C} \mapsto \mathcal{C}$ if

$$\alpha \circ f \leq_{\mathcal{A}} f^{\#} \circ \alpha$$

If α and γ form a Galois connection then correctness is automatically guaranteed. The important case is when f describes the (concrete) semantics of a program. An easy way to define a correct abstract function (e.g. a semantics) $f^{\#}$ is to induce it simply via $f^{\#} = \alpha \circ f \circ \gamma$.

An alternative characterisation of a Galois connection is as follows:

Theorem 1. *Let $\mathcal{C} = (\mathcal{C}, \leq_{\mathcal{C}})$ and $\mathcal{D} = (\mathcal{D}, \leq_{\mathcal{D}})$ be two partially ordered set together with two functions $\alpha : \mathcal{C} \mapsto \mathcal{D}$ and $\gamma : \mathcal{D} \mapsto \mathcal{C}$. Then $(\mathcal{C}, \alpha, \gamma, \mathcal{D})$ form a Galois connection iff*

1. *α and γ are order-preserving,*
2. *$\alpha \circ \gamma$ is reductive (i.e. for any $d \in \mathcal{D}$, $\alpha \circ \gamma(d) \leq_{\mathcal{D}} d$),*
3. *$\gamma \circ \alpha$ is extensive (i.e. for any $c \in \mathcal{C}$, $c \leq_{\mathcal{C}} \gamma \circ \alpha(c)$).*

A further important property of Galois connections guarantees that the approximation of a concrete semantics by means of two functions α and γ related by a Galois connection is not only safe but also *conservative* in as far as repeating the abstraction or the concretisation gives the same results as by a single application of these functions. Formally, this property is expressed by the following proposition: Let $(\mathcal{C}, \alpha, \gamma, \mathcal{D})$ be a Galois connection, then α and γ are *quasi-inverse*, i.e. $\alpha \circ \gamma \circ \alpha = \alpha$, and $\gamma \circ \alpha \circ \gamma = \gamma$.

4.2 Probabilistic Abstract Interpretation

The general approach for constructing simplified versions of a concrete (collecting) semantics via *abstract interpretation* is based on order-theoretic and not on linear structures. One can define a number of orderings (lexicographic, etc.) as an additional structure on a given vector space, and then use this order to compute over- or under-approximations using classical Abstract Interpretation.

Though such approximations will always be safe, they might also be quite unrealistic, addressing a *worst case* scenario rather than the *average case* [24]. Furthermore, there is no *canonical* order on a vector space (e.g. the lexicographic order depends on the base). In order to provide probabilistic estimates we have previously introduced, cf. [1, 25], a quantitative version of the Cousot & Cousot framework, which we have called *Probabilistic Abstract Interpretation* (PAI).

The PAI approach is based, as in the classical case, on a concrete and abstract domain $\mathcal{C}$ and $\mathcal{D}$ – except that $\mathcal{C}$ and $\mathcal{D}$ are now vector spaces (or in general, Hilbert spaces) instead of lattices. We assume that the pair of abstraction and concretisation function $\alpha : \mathcal{C} \to \mathcal{D}$ and $\gamma : \mathcal{D} \to \mathcal{C}$ are again structure preserving, i.e. in our setting they are (bounded) linear maps represented by matrices $\mathbf{A}$ and $\mathbf{G}$. Finally, we replace the notion of a Galois connection by the notion of a Moore-Penrose pseudo-inverse.

Definition 2. *Let $\mathcal{C}$ and $\mathcal{D}$ be two finite dimensional vector spaces, and let $\mathbf{A} : \mathcal{C} \to \mathcal{D}$ be a linear map between them. The linear map $\mathbf{A}^\dagger = \mathbf{G} : \mathcal{D} \to \mathcal{C}$ is the* Moore-Penrose pseudo-inverse *of $\mathbf{A}$ iff*

$$\mathbf{A} \circ \mathbf{G} = \mathbf{P}_A \quad \textit{and} \quad \mathbf{G} \circ \mathbf{A} = \mathbf{P}_G$$

where $\mathbf{P}_A$ and $\mathbf{P}_G$ denote orthogonal projections (i.e. $\mathbf{P}_A^* = \mathbf{P}_A = \mathbf{P}_A^2$ and $\mathbf{P}_G^* = \mathbf{P}_G = \mathbf{P}_G^2$ where $.^*$ denotes the adjoint *[11, Ch 10]) onto the ranges of $\mathbf{A}$ and $\mathbf{G}$.*

Alternatively, if $\mathbf{A}$ is Moore-Penrose invertible, its Moore-Penrose pseudo-inverse, $\mathbf{A}^\dagger$ satisfies the following:

$$\begin{array}{l}\textbf{(i)}\ \ \mathbf{A}\mathbf{A}^\dagger\mathbf{A} = \mathbf{A},\\ \textbf{(ii)}\ \ \mathbf{A}^\dagger\mathbf{A}\mathbf{A}^\dagger = \mathbf{A}^\dagger,\\ \textbf{(iii)}\ \ (\mathbf{A}\mathbf{A}^\dagger)^* = \mathbf{A}\mathbf{A}^\dagger,\\ \textbf{(iv)}\ \ (\mathbf{A}^\dagger\mathbf{A})^* = \mathbf{A}^\dagger\mathbf{A}.\end{array}$$

It is instructive to compare these equations with the classical setting. For example, if (α, γ) is a Galois connection we similarly have $\alpha \circ \gamma \circ \alpha = \alpha$ and $\gamma \circ \alpha \circ \gamma = \gamma$.

This allows us to construct the closest (i.e. least square, see for example [26, 27]) approximation $\mathbf{T}^\# : \mathcal{D} \to \mathcal{D}$ of the concrete semantics $\mathbf{T} : \mathcal{C} \to \mathcal{C}$ as:

$$\mathbf{T}^\# = \mathbf{G} \cdot \mathbf{T} \cdot \mathbf{A} = \mathbf{A}^\dagger \cdot \mathbf{T} \cdot \mathbf{A} = \mathbf{A} \circ \mathbf{T} \circ \mathbf{G}.$$

As our concrete semantics is constructed using tensor products it is important that the Moore-Penrose pseudo-inverse of a tensor product can easily be computed as follows [27, 2.1,Ex 3]:

$$(\mathbf{A}_1 \otimes \mathbf{A}_2 \otimes \ldots \otimes \mathbf{A}_n)^\dagger = \mathbf{A}_1^\dagger \otimes \mathbf{A}_2^\dagger \otimes \ldots \otimes \mathbf{A}_n^\dagger.$$

Example 7 (Parity). Let us consider as abstract and concrete domains $\mathcal{C} = \mathcal{V}(\{-n, \ldots, n\})$ and $\mathcal{D} = \mathcal{V}(\{\text{even}, \text{odd}\})$. The abstraction operator $\mathbf{A}_p$ and its concretisation operator $\mathbf{G}_p = \mathbf{A}_p^\dagger$ corresponding to a parity analysis are represented by the following $n \times 2$ and $2 \times n$ matrices (assuming w.l.o.g. that n is even) with $.^T$ denoting the matrix transpose, $(\mathbf{A}^T)_{ij} = (\mathbf{A})_{ji}$:

$$\mathbf{A}_p = \begin{pmatrix} 1\ 0 \\ 0\ 1 \\ 1\ 0 \\ 0\ 1 \\ \vdots\ \vdots \\ 1\ 0 \end{pmatrix} \qquad \mathbf{A}_p^\dagger = \begin{pmatrix} \frac{1}{n+1} & 0 & \frac{1}{n+1} & 0 & \ldots & \frac{1}{n+1} \\ 0 & \frac{1}{n} & 0 & \frac{1}{n} & \ldots & 0 \end{pmatrix}$$

The concretisation operator $\mathbf{A}_p^\dagger$ represents uniform distributions over the $n+1$ even numbers in the range $-n, \ldots, n$ (as the first row) and the n odd numbers in the same range (in the second row).

Example 8 (Sign). With $\mathcal{C} = \mathcal{V}(\{-n, \ldots, 0, \ldots, n\})$ and $\mathcal{D} = \mathcal{V}(\{-, 0, +\})$ we can represent the usual sign abstraction by the following matrices:

$$\mathbf{A}_s = \begin{pmatrix} 1\,0\,0 \\ \vdots\,\vdots\,\vdots \\ 1\,0\,0 \\ 0\,1\,0 \\ 0\,0\,1 \\ \vdots\,\vdots\,\vdots \\ 0\,0\,1 \end{pmatrix} \qquad \mathbf{A}_s^\dagger = \begin{pmatrix} \frac{1}{n} \ldots \frac{1}{n}\ 0\ 0 \ldots 0 \\ 0 \ldots 0\ 1\ 0 \ldots 0 \\ 0 \ldots 0\ 0\ \frac{1}{n} \ldots \frac{1}{n} \end{pmatrix}$$

Example 9 (Forget). We can also abstract all details of the concrete semantics. Although this is in general a rather unusual abstraction it is quite useful in the context of a tensor product state and/or abstraction. Let the concrete domain be the vector space over any range, i.e. $\mathcal{C} = \mathcal{V}(\{n, \ldots, 0, \ldots, m\})$, and the abstract domain a one dimensional space $\mathcal{D} = \mathcal{V}(\{\star\})$. Then the forgetful abstraction and concretisation can be defined by:

$$\mathbf{A}_f^T = \begin{pmatrix} 1\ 1\ 1 \ldots 1 \end{pmatrix} \qquad \mathbf{A}_f^\dagger = \begin{pmatrix} \frac{1}{m-n+1} & \frac{1}{m-n+1} & \frac{1}{m-n+1} & \cdots & \frac{1}{m-n+1} \end{pmatrix}$$

For any matrix $\mathbf{M}$ operating on $\mathcal{C} = \mathcal{V}(\{n, \ldots, 0, \ldots, m\})$ the abstraction $\mathbf{A}_f^\dagger \cdot \mathbf{M} \cdot \mathbf{A}_f$ gives a one dimensional matrix, i.e. a single scalar μ. For stochastic matrices, such as our $\mathbf{T}$ generating the DTMC representing the concrete semantics we have: $\mu = 1$. If we consider a tensor product of two matrices $\mathbf{M} \otimes \mathbf{N}$, then the abstraction $\mathbf{A}_f \otimes \mathbf{I}$ extracts (essentially) $\mathbf{N}$,

$$\begin{aligned} &(\mathbf{A}_f \otimes \mathbf{I})^\dagger \cdot (\mathbf{M} \otimes \mathbf{N}) \cdot (\mathbf{A}_f \otimes \mathbf{I}) = \\ &= (\mathbf{A}_f^\dagger \otimes \mathbf{I}^\dagger) \cdot (\mathbf{M} \otimes \mathbf{N}) \cdot (\mathbf{A}_f \otimes \mathbf{I}) = \\ &= (\mathbf{A}_f^\dagger \cdot \mathbf{M} \cdot \mathbf{A}_f) \otimes (\mathbf{I} \cdot \mathbf{N} \cdot \mathbf{I}) = \\ &= \mu \otimes \mathbf{N} = \mu\mathbf{N}. \end{aligned}$$

4.3 Abstract LOS Semantics

The abstract semantics $\mathbf{T}^\#(P)$ of a program P is constructed exactly like the concrete one, except that we will use abstract tests and update operators. This is possible as abstractions and concretisations distribute over sums and tensor products. More precisely, we can construct $\mathbf{T}^\#$ for a program P as:

$$\mathbf{T}^\#(P) = \sum_{\langle i, p_{ij}, j \rangle \in \mathcal{F}(P)} p_{ij} \cdot \mathbf{T}^\#(\ell_i, \ell_j)$$

where the transfer operator along a computational step from label ℓ_i to ℓ_j can be abstracted "locally": Abstracting each variable separately and using the concrete control flow we get the operator

$$\mathbf{A} = (\bigotimes_{i=1}^{v} \mathbf{A}_i) \otimes \mathbf{I} = \mathbf{A}_1 \otimes \mathbf{A}_2 \otimes \ldots \otimes \mathbf{A}_v \otimes \mathbf{I}.$$

Then the abstract transfer operator $\mathbf{T}^{\#}(\ell_i, \ell_j)$ can be defined as:

$$\mathbf{T}^{\#}(\ell_i, \ell_j) = (\mathbf{A}_1^{\dagger}\mathbf{N}_{i1}\mathbf{A}_1) \otimes (\mathbf{A}_2^{\dagger}\mathbf{N}_{i2}\mathbf{A}_2) \otimes \ldots \otimes (\mathbf{A}_v^{\dagger}\mathbf{N}_{iv}\mathbf{A}_v) \otimes \mathbf{E}(\ell_i, \ell_j).$$

This operator implements the (abstract) effect to each of the variables in the individual statement at ℓ_i and combines it with the concrete control flow. This follows directly from a short calculation:

$$\begin{aligned}
\mathbf{T}^{\#} &= \mathbf{A}^{\dagger}\mathbf{T}\mathbf{A} = \\
&= \mathbf{A}^{\dagger}(\sum_{i,j} p_{ij} \cdot \mathbf{T}(\ell_i, \ell_j))\mathbf{A} = \\
&= \sum_{i,j} p_{ij} \cdot (\mathbf{A}^{\dagger}\mathbf{T}(\ell_i, \ell_j)\mathbf{A}) = \\
&= \sum_{i,j} p_{ij} \cdot (\bigotimes_k \mathbf{A}_k \otimes \mathbf{I})^{\dagger}\mathbf{T}(\ell_i, \ell_j)(\bigotimes_k \mathbf{A}_k \otimes \mathbf{I}) = \\
&= \sum_{i,j} p_{ij} \cdot (\bigotimes_k \mathbf{A}_k^{\dagger} \otimes \mathbf{I}^{\dagger})(\bigotimes_k \mathbf{N}_{ik} \otimes \mathbf{E}(\ell_i, \ell_j))(\bigotimes_k \mathbf{A}_k \otimes \mathbf{I}) = \\
&= \sum_{i,j} p_{ij} \cdot \bigotimes_k (\mathbf{A}_k^{\dagger}\mathbf{N}_{ik}\mathbf{A}_k) \otimes \mathbf{E}(\ell_i, \ell_j).
\end{aligned}$$

It is of course also possible to abstract the control flow, or to use abstractions which abstract several variables at the same time, e.g. by specifying the abstract state via the difference of two variables. The dramatic reduction in size, i.e. dimensions, achieved via PAI illustrated also by the examples in these notes lets us hope that our approach could ultimately lead to scalable analyses, despite the fact that the concrete semantics is non-feasibly large. As many people have observed – the use of tensor products or similar constructions in probabilistic models leads to a combinatorial explosion of the size of the formal model. However, the PAI approach allows us to keep some control and to obtain reasonably sized abstract models. Further work in the form of practical implementations and experiments is needed to decide whether this is indeed the case.

The LOS represents the SOS via the generator of a DTMC. It describes the stepwise evolution of the state of a computation and does not provide a fixed-point semantics. Therefore, neither in the concrete nor in the abstract case can we guarantee that $\lim_{n\to\infty}(\mathbf{T}(P))^n$ or $\lim_{n\to\infty}(\mathbf{T}(P)^{\#})^n$ always exist. The analysis of a program P based on the abstract operator $\mathbf{T}(P)^{\#}$ is considerably *simpler* than by considering the concrete one but still not entirely trivial. Various properties of $\mathbf{T}(P)^{\#}$ can be extracted by iterative methods (e.g. computing $\lim_{n\to\infty}(\mathbf{T}(P)^{\#})^n$ or some averages). As often in numerical computation, these methods will converge only for $n \to \infty$ and any result obtained after only a finite number of steps will only be an approximation. However, one can study *stopping criteria* which guarantee a certain quality of this approximation. The development or adaptation of iterative methods and formulation of appropriate stopping criteria might be seen as the numerical analog to *widening* and *narrowing* techniques within the classical setting.

4.4 Classical vs Probabilistic Abstract Interpretation

Classical abstract interpretation and probabilistic abstract interpretation provide "approximations" for different mathematical structures, namely partial orders vs vector spaces. In order to illustrate and compare their features we therefore need a setting where the domain in question in some way naturally provides both structures. One such situation is in the context of classical function interpolation or approximation.

The set of real-valued functions on a real interval $[a, b]$ obviously comes with a canonical partial order, namely the point-wise ordering, and at the same time is equipped with a vector space structure, where again addition and scalar multiplication are defined point-wise. Some care has to be taken in order to define an inner product – which we need to obtain a Hilbert space structure, e.g. one could consider only the square integrable functions $L^2([a, b])$. In order to avoid mathematical (e.g. measure-theoretic) details we simplify the situation by just considering the step functions on the interval $[a, b]$.

For a (closed) real interval $[a, b] \subseteq \mathbb{R}$ we call the set of subintervals $[a_i, b_i]$ with $i = 1, \ldots, n$ the *n-subdivision* of $[a, b]$ if $\bigcup_{i=1}^{n} [a_i, b_i] = [a, b]$ and $b_i - a_i = \frac{b-a}{n}$ for all $i = 1, \ldots, n$. We assume that the sub-intervals are enumerated in the obvious way, i.e. $a_i < b_i = a_{i+1} < b_{i+1}$ for all i and in particular that $a = a_1$ and $b_n = b$.

Definition 3. *The set of* n-step functions $\mathcal{T}_n([a, b])$ *on* $[a, b]$ *is the set of real-valued functions* $f : [a, b] \to \mathbb{R}$ *such that* f *is constant on each subinterval* (a_i, b_i) *in the n-subdivision of* $[a, b]$.

We define a partial order on $\mathcal{T}_n([a, b])$ in the obvious way for $f, g \in \mathcal{T}_n([a, b])$:

$$f \sqsubseteq g \text{ iff } f(\frac{b_i - a_i}{2}) \leq g(\frac{b_i - a_i}{2}), \text{ for all } 1 \leq i \leq n$$

i.e. iff the value of f (which we obtain by evaluating it on the mid-point in (a_i, b_i)) on all subintervals (a_i, b_i) is less or equal to the value of g.

It is also obvious to see that $\mathcal{T}_n([a, b])$ has a vector space structure isomorphic to $\mathbb{R}^n$ and thus is also provided with an inner product. More concretely we define the vector space operations $. \cdot . : \mathbb{R} \times \mathcal{T}_n([a, b]) \to \mathcal{T}_n([a, b])$ and $. + . : \mathcal{T}_n([a, b]) \times \mathcal{T}_n([a, b]) \to \mathcal{T}_n([a, b])$ pointwise as follows:

$$(\alpha \cdot f)(x) = \alpha f(x)$$

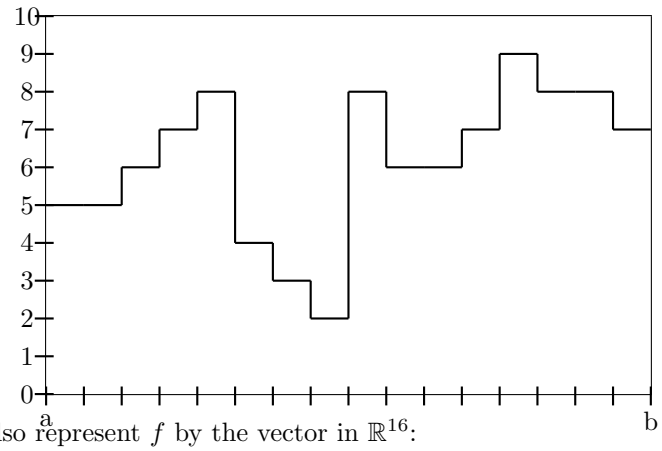
$$(f + g)(x) = f(x) + g(x)$$

for all $\alpha \in \mathbb{R}$, $f, g \in \mathcal{T}_n([a, b])$ and $x \in [a, b]$. The inner product is given by:

$$\langle f, g \rangle = \sum_{i=1}^{n} f(\frac{b_i - a_i}{2}) g(\frac{b_i - a_i}{2}).$$

In this setting we now can apply and compare both the classical and the quantitative version of abstract interpretation as in the following example.

Example 10. Let us consider a step function f in $\mathcal{T}_{16}$ (the concrete values of a and b don't really play a role in our setting) which can be depicted as:



We can also represent f by the vector in $\mathbb{R}^{16}$:

$$\begin{pmatrix} 5 & 5 & 6 & 7 & 8 & 4 & 3 & 2 & 8 & 6 & 6 & 7 & 9 & 8 & 8 & 7 \end{pmatrix}$$

We then construct a series of abstractions which correspond to coarser and coarser sub-divisions of the interval $[a, b]$, e.g. considering 8, 4 etc. subintervals instead of the original 16. These abstractions are from $\mathcal{T}_{16}([a, b])$ to $\mathcal{T}_8([a, b])$, $\mathcal{T}_4([a, b])$ etc. and can be represented by 16×8, 16×4, etc. matrices. For example, the abstraction which joins two sub-intervals and which corresponds to the abstraction $\alpha_8 : \mathcal{T}_{16}([a, b]) \to \mathcal{T}_8([a, b])$ together with its Moore-Penrose pseudo-inverse is represented by:

$$\mathbf{A}_8 = \begin{pmatrix} 1&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&0&1 \end{pmatrix} \quad \mathbf{G}_8 = \begin{pmatrix} \frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2}&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0&\frac{1}{2}&\frac{1}{2} \end{pmatrix}$$

With the help of $\mathbf{A}_j$, $j \in \{1, 2, 4, 8\}$, we can easily compute the abstraction of f as $f\mathbf{A}_j$, which in order to compare it with the original f we can then again

concretise using $\mathbf{G}$, i.e. computing $f\mathbf{AG}$. In a similar way we can also compute the over- and under-approximation of f in $\mathcal{T}_i$ based on the above pointwise ordering and its reverse ordering. The result of these abstractions is depicted geometrically in Figure 1.

With the help of $\mathbf{A}_j$, $j \in \{1, 2, 4, 8\}$, we can easily compute the abstraction of f as $f\mathbf{A}_j$, which in order to compare it with the original f we can then again concretise using $\mathbf{G}$, i.e. computing $f\mathbf{AG}$. In a similar way we can also compute the over- and under-approximation of f in $\mathcal{T}_i$ based on the above pointwise ordering and its reverse ordering. The result of these abstractions is depicted geometrically in Figure 1.

The individual diagrams in this figure depict the original, i.e. concrete step function $f \in \mathcal{T}_{16}$ together with its approximations in $\mathcal{T}_8$, $\mathcal{T}_4$, etc. On the left hand side the PAI abstractions show how coarser and coarser interval subdivisions result in a series of approximations which try to interpolate the given function as closely as possible, sometimes below, sometimes above the concrete values. The diagrams on the right hand side depict the classical over- and under-approximations: In each case the function f is entirely below or above these approximations, i.e. we have safe but not necessarily close approximations. Additionally, one can also see from these figures not only that the PAI interpolation is in general closer to the original function than the classical abstractions (in fact it is the closest possible) but also that the PAI interpolation is always between the classical over- and under-approximations.

The vector space framework also allows us to judge the quality of an abstraction or approximation via the Euclidian distance between the concrete and abstract version of a function. We can compute the *least square error* as

$$\|f - f\mathbf{AG}\|.$$

In our case we get for example:

$$\begin{aligned}\|f - f\mathbf{A}_8\mathbf{G}_8\| &= 3.5355\\ \|f - f\mathbf{A}_4\mathbf{G}_4\| &= 5.3151\\ \|f - f\mathbf{A}_2\mathbf{G}_2\| &= 5.9896\\ \|f - f\mathbf{A}_1\mathbf{G}_1\| &= 7.6444\end{aligned}$$

which illustrates, as expected, that the coarser our abstraction is the larger is also the mistake or error.

4.5 Examples

We conclude by discussing in detail how probabilistic abstraction allows us to analyse the properties of programs. In the first example we are going to present, the aim is to reduce the size (dimension) of the concrete semantics so as to allow for an immediate understanding of the results of a computation. The second example will look more closely at the efficiency of an analysis, i.e. how PAI

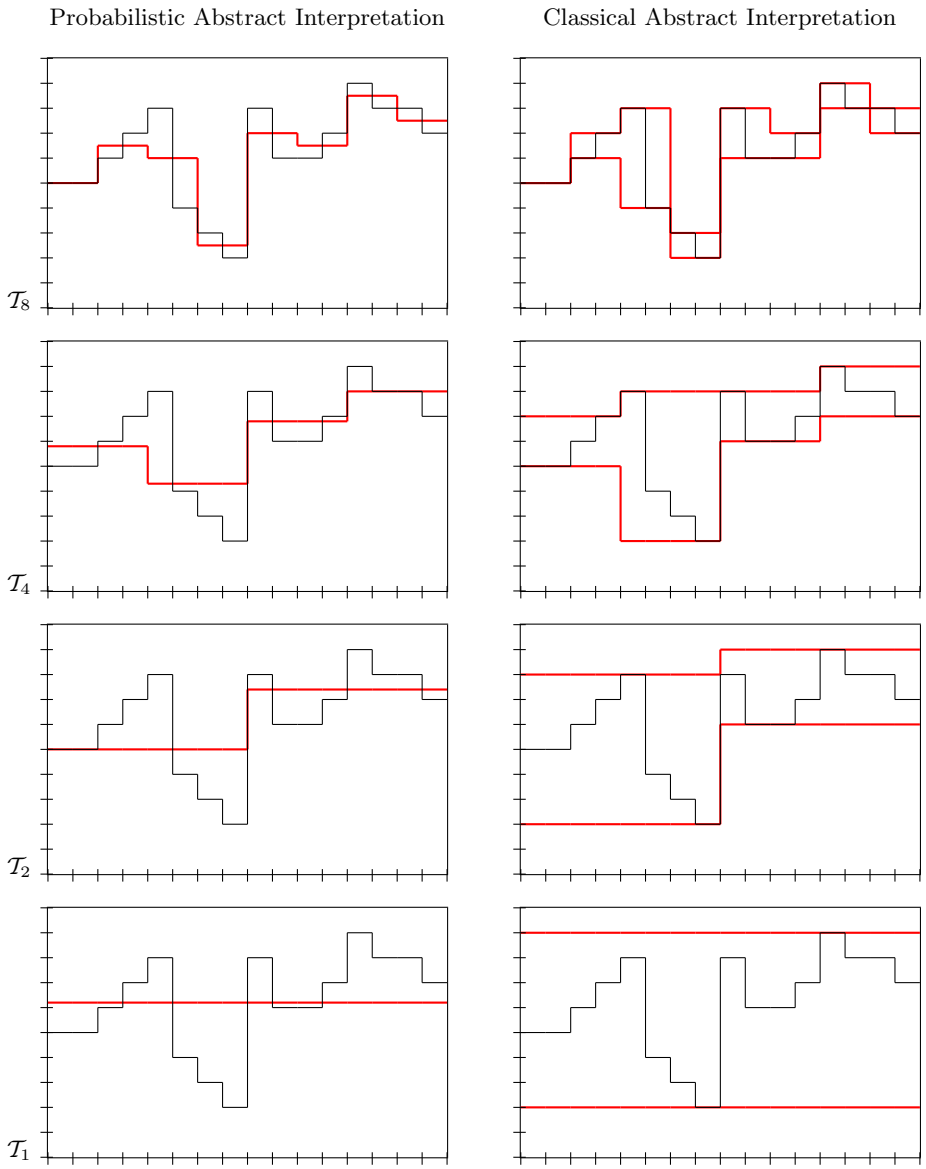


Fig. 1. Average, Over- and Under-Approximation

can be deployed in order to beat the combinatorial explosion or the curse of dimensionality.

Example 11 (Monty Hall). We have already investigated the LOS semantics of the Monty Hall program in Example 5. We still have to analyse whether it is H_t or H_w that implements the better strategy. In principle, we can do this using the concrete semantics we constructed above. However, it is rather cumbersome to work with "relatively large" 162×162 or 243×243 matrices, even when they are sparse, i.e. contain almost only zeros (in fact only about 1.2% of entries in H_t and 0.7% of entries in H_w are non-zero).

If we want to analyse the final states, i.e. which of the two programs has a better chance of getting the right door, we need to start with an initial configuration and then iterate $\mathbf{T}(H)$ until we reach a/the final configuration. For our programs it is sufficient to indicate that we start in label 1, while the state is irrelevant as we initialise all three variables at the beginning of the program; we could take – for example – a state with $d = o = g = 0$. The vector or distribution which describes this initial configuration is a 162 or 243 dimensional vector. We can describe it in a rather compact form as:

$$\boldsymbol{x}_0 = \big(1\,0\,0\big) \otimes \big(1\,0\,0\big) \otimes \big(1\,0\,0\big) \otimes \big(1\,0\,0\,\ldots\,0\big)\,,$$

where the last factor is 6 or 9 dimensional, depending on whether we deal with H_t or H_w. This represents a point distribution on 162 or 243 relevant distributions.

Assuming that our program terminates for all initial states, as it is the case here, then there exists a certain number of iterations t such that $\boldsymbol{x}_0\mathbf{T}(H)^t = \boldsymbol{x}_0\mathbf{T}(H)^{t+1}$, i.e. we will eventually reach a fix-point which gives us a distribution over configurations. In general, as in our case here, this will not be just a point distribution. Again we get vectors of dimension 162 or 243, respectively. For H_t and H_w there are 12 configurations which have non-zero probability.

$$\text{for } H_t \begin{cases} x_{12} = 0.074074 \\ x_{18} = 0.037037 \\ x_{36} = 0.11111 \\ x_{48} = 0.11111 \\ x_{72} = 0.11111 \\ x_{78} = 0.037037 \\ x_{90} = 0.074074 \\ x_{96} = 0.11111 \\ x_{120} = 0.11111 \\ x_{132} = 0.11111 \\ x_{150} = 0.074074 \\ x_{156} = 0.037037 \end{cases} \qquad \text{for } H_w \begin{cases} x_{18} = 0.11111 \\ x_{27} = 0.11111 \\ x_{54} = 0.037037 \\ x_{72} = 0.074074 \\ x_{108} = 0.074074 \\ x_{117} = 0.11111 \\ x_{135} = 0.11111 \\ x_{144} = 0.037037 \\ x_{180} = 0.037037 \\ x_{198} = 0.074074 \\ x_{225} = 0.11111 \\ x_{234} = 0.11111 \end{cases}$$

It is anything but easy to determine from this information which of the two strategies is more successful. In order to achieve this we will abstract away all unnecessary information. First, we ignore the syntactic information: If we are in the terminal state, then we have reached the final `stop` state, but even if this

would not be the case we only need to know whether in the final state we have guessed the right door, i.e. whether `d==g` or not. We thus also don't need to know the value of `o` as it ultimately is of no interest to us which door had been opened during the game. Therefore, we can use the forgetful abstraction $\mathbf{A}_f$ to simplify the information contained in the terminal state. Regarding `d` and `g` we want to know everything, and thus use the trivial abstraction $\mathbf{A} = \mathbf{I}$, i.e. the identity. The result for H_t is for $\boldsymbol{x}$ the terminal configuration distribution:

$$\boldsymbol{x} \cdot (\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{A}_f \otimes \mathbf{A}_f) = \begin{pmatrix} 0.11 & 0.11 & 0.11 & 0.11 & 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \end{pmatrix}$$

and for H_w we get:

$$\boldsymbol{x} \cdot (\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{A}_f \otimes \mathbf{A}_f) = \begin{pmatrix} 0.22 & 0.04 & 0.07 & 0.07 & 0.22 & 0.04 & 0.04 & 0.07 & 0.22 \end{pmatrix}$$

The nine coordinates of these vectors correspond to $(\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 0)$, $(\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 1)$, $(\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 2)$, $(\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 0)$, $\ldots$, $(\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 2)$. This is in principle enough to conclude that H_w is the better strategy.

However, we can go a step further and abstract not the values of `d` and `g` but their relation, i.e. whether they are equal or different. For this we need the abstraction:

$$\mathbf{A}_w = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

where the first column corresponds to a winning situation (i.e. `d` and `g` are equal), and the second to unequal `d` and `g`. With this we get for H_t:

$$\boldsymbol{x} \cdot (\mathbf{A}_w \otimes \mathbf{A}_f \otimes \mathbf{A}_f) = \begin{pmatrix} 0.33333 & 0.66667 \end{pmatrix}$$

and for H_w

$$\boldsymbol{x} \cdot (\mathbf{A}_w \otimes \mathbf{A}_f \otimes \mathbf{A}_f) = \begin{pmatrix} 0.66667 & 0.33333 \end{pmatrix}$$

It is now obvious that H_t has just a $\frac{1}{3}$ chance of winning, while H_w has a $\frac{2}{3}$ probability of picking the winning door.

This example illustrates how abstraction can be used in order to obtain useful information from a large collection of data – so to say, how to use abstractions to do *statistics*. We did not utilise PAI here to simplify the semantics itself but only the final results. We will now consider this issue in our second running example.

Example 12 (Factorial). Classical abstraction allows us to determine the parity properties of the "double factorial" in Example 2. However, we cannot use it to justify our intuition that even the plain factorial itself almost always produces a even result. In order to do this, let us first consider the concrete semantics of our program using the following labelling:

```
var
  m : {0..2};
  n : {0..2};
begin
[m := 1]^1;
while [(n>1)]^2 do
  [m := m*n]^3;
  [n := n-1]^4;
od;
[stop]^5;
end
```

The flow of this program F is given as follows:

$$\mathbf{Flow}(F) = \{(1,1,2),(2,1,\underline{3}),(3,1,4),(4,1,2),(2,1,5),(5,1,5)\}$$

The operator $\mathbf{T}(F)$ is then constructed as

$$\begin{aligned}\mathbf{T}(F) = {} & \mathbf{U}(\mathtt{m} \leftarrow \mathtt{1}) \otimes \mathbf{E}(1,2) + \\ & \mathbf{P}(\mathtt{(n > 1)}) \otimes \mathbf{E}(2,3) + \\ & \mathbf{U}(\mathtt{m} \leftarrow \mathtt{(m * n)}) \otimes \mathbf{E}(3,4) + \\ & \mathbf{U}(\mathtt{n} \leftarrow \mathtt{(n - 1)}) \otimes \mathbf{E}(4,2) + \\ & \mathbf{P}(\mathtt{(n <= 1)}) \otimes \mathbf{E}(2,5) + \\ & \mathbf{I} \otimes \mathbf{E}(5,5)\end{aligned}$$

using the matrices $\mathbf{T}(\ell,\ell') = \mathbf{S}(\ell) \otimes \mathbf{E}(\ell,\ell')$ (where we indicate the `then` and `else` branches again by underlining in the obvious way):

$$\mathbf{S}(1) = \begin{pmatrix} 0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \end{pmatrix} \qquad \mathbf{S}(\underline{2}) = \begin{pmatrix} 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1 \end{pmatrix}$$

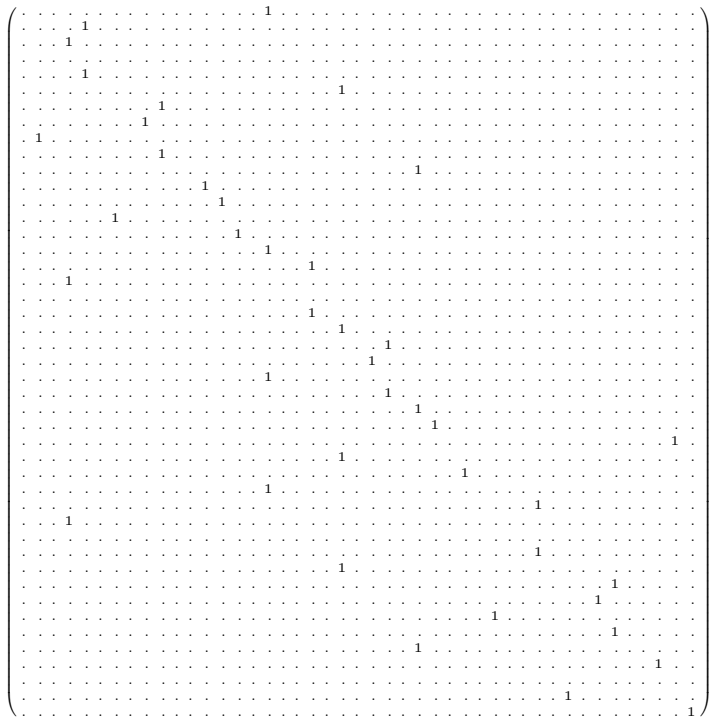
$$\mathbf{S}(3) = \begin{pmatrix} 1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&0&0 \end{pmatrix} \qquad \mathbf{S}(4) = \begin{pmatrix} 0&0&0&0&0&0&0&0&0 \\ 1&0&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&1&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&1&0 \end{pmatrix}$$

$$\mathbf{S}(2) = \begin{pmatrix} 1&0&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&0&0 \end{pmatrix} \qquad \mathbf{S}(5) = \begin{pmatrix} 1&0&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0&0\\ 0&0&0&0&0&1&0&0&0&0\\ 0&0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&0&0&1 \end{pmatrix}$$

Note that for the updates in label 3 and 4 we have "empty rows", i.e. rows where we have no non-zero entries. These correspond to over- and under-flows as we are dealing only with finite values in $\underline{\mathbb{Z}}$. We could clarify the situation in various ways, e.g. by introducing an additional value $\perp$ for undefined (concrete) values of variables, or by introducing an `error` configuration, etc. In the analysis we present in this example these over- and under-flows do not play any relevant role and we therefore leave things as they are – one should however keep in mind that this violates the observation that $\mathbf{T}(F)$ is a stochastic matrix.

The full operator representing the LOS semantics of this (small) factorial program is given by a $(3 \cdot 3 \cdot 5) \times (3 \cdot 3 \cdot 5) = 45 \times 45$ matrix:

$$\mathbf{T}(F) =$$



We can now construct an abstract version $\mathbf{T}^{\#}(F)$ of $\mathbf{T}(F)$ by recording only the parity of m as even and odd. We will not abstract n nor the labels defining

the current configuration during the execution. We thus get

$$\mathbf{T}^{\#}(F) = (\mathbf{A}_p \otimes \mathbf{I} \otimes \mathbf{I})^{\dagger}\mathbf{T}(F)(\mathbf{A}_p \otimes \mathbf{I} \otimes \mathbf{I})$$

a $(2 \cdot 3 \cdot 5) \times (2 \cdot 3 \cdot 5) = 30 \times 30$ matrix.

Though this abstract semantics does have some interesting properties, it appears to be only a minor improvement with regard to the concrete semantics: We managed to reduce the dimension only from 45 to 30. However, the simplification becomes substantially more dramatic once we increase the possible values of `m` and `n`, and combinatorial explosion really takes a hold. If we allow `n` to take values between 0 and n then we must allow for `m` values between 0 and $n!$. Concrete values of the dimensions of $\mathbf{T}(F)$ and $\mathbf{T}^{\#}(F)$ are given in the following table:

n	$\dim(\mathbf{T}(F))$	$\dim(\mathbf{T}^{\#}(F))$
2	45	30
3	140	40
4	625	50
5	3630	60
6	25235	70
7	201640	80
8	1814445	90
9	18144050	100

The problem is that the size of $\mathbf{T}(F)$ explodes so quickly that it is impossible to simulate it for values of n much larger than 5 on a normal PC. If we want to analyse the abstract semantics, things remain much smaller. Importantly, we can construct the abstract semantics in the same way as the concrete one, just using "smaller" matrices:

$$\begin{aligned}\mathbf{T}^{\#}(F) = {} & \mathbf{U}^{\#}(\texttt{m} \leftarrow \texttt{1}) \otimes \mathbf{E}(1,2) + \\ & \mathbf{P}^{\#}(\texttt{(n > 1)}) \otimes \mathbf{E}(2,3) + \\ & \mathbf{U}^{\#}(\texttt{m} \leftarrow \texttt{(m * n)}) \otimes \mathbf{E}(3,4) + \\ & \mathbf{U}^{\#}(\texttt{n} \leftarrow \texttt{(n - 1)}) \otimes \mathbf{E}(4,2) + \\ & \mathbf{P}^{\#}(\texttt{(n <= 1)}) \otimes \mathbf{E}(2,5) + \\ & \mathbf{I}^{\#} \otimes \mathbf{E}(5,5)\end{aligned}$$

Fortunately, most of the operators $\mathbf{T}^{\#}(\ell, \ell')$ are very easy to construct. These matrices are $2 \cdot (n+1) \cdot 5 \times 2 \cdot (n+1) \cdot 5 = 10(n+1) \times 10(n+1)$ matrices if we consider the control transfer and only $2(n+1) \times 2n+1)$ matrices if we deal only with the update of the current state. In principle we could abstract the $\mathbf{T}^{\#}(\ell, \ell')$ from their concrete versions $\mathbf{T}(\ell, \ell')$ using our abstraction and its Moore-Penrose pseudo-inverse as concretisation.

However, by considering the matrices $\mathbf{T}^{\#}(\ell, \ell')$ in detail it is possible to come up with an even more direct construction. Except for label 3 only either `m` or `n` (but not both) are involved in each statement: We thus can express the

$\mathbf{T}^{\#}(\ell, \ell')$'s as tensor products of a 2×2 and a $(n+1) \times (n+1)$ matrix.

$$\mathbf{U}^{\#}(\texttt{m} \leftarrow \texttt{1}) = \begin{pmatrix} 0\,1 \\ 0\,1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0\,0 \ldots 0 \\ 0 & 1 & 0\,0 \ldots 0 \\ 0 & 0 & 1\,0 \ldots 0 \\ 0 & 0 & 0\,1 \ldots 0 \\ \vdots & \vdots & \vdots\,\vdots\,\ddots\,\vdots \\ 0 & 0\,0\,0 & \ldots 1 \end{pmatrix}$$

$$\mathbf{U}^{\#}(\texttt{n} \leftarrow \texttt{(n-1)}) = \begin{pmatrix} 1\,0 \\ 0\,1 \end{pmatrix} \otimes \begin{pmatrix} 0\,0\,0\,0 \ldots 0 \\ 1\,0\,0\,0 \ldots 0 \\ 0\,1\,0\,0 \ldots 0 \\ 0\,0\,1\,0 \ldots 0 \\ \vdots\,\vdots\,\vdots\,\vdots\,\ddots\,\vdots \\ 0\,0\,0\,0 \ldots 0 \end{pmatrix}$$

$$\mathbf{P}^{\#}(\texttt{(n > 1)}) = \begin{pmatrix} 1\,0 \\ 0\,1 \end{pmatrix} \otimes \begin{pmatrix} 0\,0\,0\,0 \ldots 0 \\ 0\,0\,0\,0 \ldots 0 \\ 0\,0\,1\,0 \ldots 0 \\ 0\,0\,0\,1 \ldots 0 \\ \vdots\,\vdots\,\vdots\,\vdots\,\ddots\,\vdots \\ 0\,0\,0\,0 \ldots 1 \end{pmatrix}$$

$$\mathbf{P}^{\#}(\texttt{(n <= 1)}) = \begin{pmatrix} 1\,0 \\ 0\,1 \end{pmatrix} \otimes \begin{pmatrix} 1\,0\,0\,0 \ldots 0 \\ 0\,1\,0\,0 \ldots 0 \\ 0\,0\,0\,0 \ldots 0 \\ 0\,0\,0\,0 \ldots 0 \\ \vdots\,\vdots\,\vdots\,\vdots\,\ddots\,\vdots \\ 0\,0\,0\,0 \ldots 0 \end{pmatrix}$$

Finally, we need just to construct the update for label 3. It is easy to see that for even `m` the result is again even and for odd `m` the parity of `n` determines the parity of the resulting `m`. We can thus write this update as:

$$\mathbf{U}^{\#}(\texttt{m} \leftarrow \texttt{(m * n)}) = \begin{pmatrix} 1\,0 \\ 0\,0 \end{pmatrix} \otimes \begin{pmatrix} 1\,0\,0\,0 \ldots 0 \\ 0\,1\,0\,0 \ldots 0 \\ 0\,0\,1\,0 \ldots 0 \\ 0\,0\,0\,1 \ldots 0 \\ \vdots\,\vdots\,\vdots\,\vdots\,\ddots\,\vdots \\ 0\,0\,0\,0 \ldots 1 \end{pmatrix} +$$

$$+ \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \ddots \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & \ddots \end{pmatrix}$$

With this we can now approximate the probabilistic properties of the factorial function. In particular, if we look at the terminal configurations with the initial abstract configuration:

$$\boldsymbol{x}_0 = \left(\tfrac{1}{2} \ \tfrac{1}{2} \right) \otimes \left(\tfrac{1}{n+1} \ \cdots \ \tfrac{1}{n+1} \right) \otimes \left(1 \, 0 \, 0 \, 0 \, 0 \right)$$

which corresponds to a uniform distribution over all possible abstract values for our variables `m` and `n` (in fact, the part describing `m` could be any other distribution) then we get as final probabilistic configuration:

$$\boldsymbol{x} = \left(\tfrac{n-1}{n+1} \ \tfrac{2}{n+1} \right) \otimes \left(\tfrac{1}{n+1} \ \tfrac{n}{n+1} \ 0 \dots 0 \right) \otimes \left(0 \, 0 \, 0 \, 0 \, 1 \right)$$

This expresses the fact that indeed in most cases (with probability $\frac{n-1}{n+1}$) we get an even factorial – only in two cases out of $n+1$ (for 0 and 1) we get an odd result (namely 1). The final value of `n` is nearly always 1 except when we start with 0 and we always reach the final statement with label 5.

If we start with the abstract initial state $\boldsymbol{x}_0$ above and execute $\mathbf{T}^{\#}(F)$ until we get a fix-pint $\boldsymbol{x}$ we can use (as before in the Monty Hall example) abstractions not to simplify the semantics but instead in order to extract the relevant information. Concretely we can use:

$$\mathbf{A} = \mathbf{I} \otimes \mathbf{A}_f \otimes \mathbf{A}_f$$

i.e. once we reached the terminal configuration (of the abstract execution) we ignore the value of n and the final label ℓ and only concentrate on the abstract, i.e. parity, values of m. Concretely we have to compute:

$$(\lim_{i \to \infty} \boldsymbol{x}_0 \cdot (\mathbf{T}^{\#}(F))^i) \cdot \mathbf{A}$$

Note that we always reach the fix-point after a finite number of iterations (namely at most n) so this can be computed in finite time. The concrete probabilities we get are for various n are:

n	**even**	**odd**
10	0.81818	0.18182
100	0.98019	0.019802
1000	0.99800	0.0019980
10000	0.99980	0.00019998

We see that we can easily compute the final distribution on $\{\mathbf{even}, \mathbf{odd}\}$ for quite large n despite the fact that, as said, it is virtually impossible to compute the explicit representation of the concrete semantics $\mathbf{T}(F)$ already for $n = 6$.

Acknowledgements. These lecture notes are partly based on previously published material in [18] and [24]. The example (matrices) in these notes were generated using a parser written in `ocaml` and using `octave` [28].

References

1. Di Pierro, A., Wiklicky, H.: Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation. In: PPDP'00. (2000) 127–138
2. Di Pierro, A., Hankin, C., Wiklicky, H.: Continuous-time probabilistic KLAIM. In Focardi, R., Zavattaro, G., eds.: SecCo'04 — CONCUR Workshop on Security Issues in Coordination Models, Languages, and Systems. Volume 128:5 of Electronic Notes in Theoretical Computer Science., Elsevier (2005)
3. Stirzaker, D.: Probability and Random Variables – A Beginners Guide. Cambridge University Press (1999)
4. Greub, W.: Linear Algebra. third edn. Volume 97 of Grundlehren der mathematischen Wissenschaften. Springer Verlag, New York (1967)
5. Conway, J.: A Course in Functional Analysis. second edn. Volume 96 of Garduate Texts in Mathematics. Springer Verlag, New York (1990)
6. Seneta, E.: Non-negative Matrices and Markov Chains. second edn. Springer Verlag, New York – Heidelberg – Berlin (1981)
7. Grimmett, G., Stirzaker, D.: Probability and Random Processes. second edn. Clarendon Press, Oxford. (1992)
8. Tijms, H.: Stochastic Models – An Algorithmic Approach. John Wiley & Sons, Chichester (1994)
9. Grinstead, C., Snell, J.: Introduction to Probability. second revised edn. American Mathematical Society, Providence, Rhode Island (1997)
10. Norris, J.: Markov Chains. Cambidge Series in Statistical and Probabilistic Mathematics. Cambridge University Press, Cambridge (1997)
11. Roman, S.: Advanced Linear Algebra. 2nd edn. Springer Verlag (2005)
12. Kadison, R., Ringrose, J.: Fundamentals of the Theory of Operator Algebras: Volume I – Elementary Theory. Volume 15 of Graduate Studies in Mathematics. American Mathematical Society, Providence, Rhode Island (1997) reprint from Academic Press edition 1983.
13. Filter, W., Weber, K.: Integration Theory. Chapman & Hall (1997)
14. Nielson, F., Nielson, H.R.: Flow logics and operational semantics. Electronic Notes of Theoretical Computer Science **10** (1998)
15. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag, Berlin – Heidelberg (1999)
16. Di Pierro, A., Hankin, C., Wiklicky, H.: Measuring the confinement of probabilistic systems. Theoretical Computer Science **340**(1) (June 2005) 3–56
17. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: POPL'02. (2002) 178–190
18. Di Pierro, A., Hankin, C., Wiklicky, H.: A systematic approach to probabilistic pointer analysis. In Shao, Z., ed.: Proceedings of APLAS'07 – 5th Asian Symposium on Programming Languages and Systems. Volume 4807 of Lecture Notes in Computer Science., Springer Verlag (2007) 335–350
19. Cousot, P., Cousot, R.: Abstract Interpretation and Applications to Logic Programs. Journal of Logic Programming **13**(2-3) (July 1992) 103–180

20. Abramsky, S., Hankin, C., eds.: Abstract Interpretation of Declarative Languages. Ellis-Horwood, Chichester, England (1987)
21. Davey, B., Priestley, H.: Introduction to Lattices and Order. Cambridge University Press, Cambridge (1990)
22. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL'77. (1977) 238–252
23. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of POPL'79, San Antonio, Texas (1979) 269–282
24. Di Pierro, A., Hankin, C., Wiklicky, H.: Abstract interpretation for worst and average case analysis. In: Program Analysis and Compilation, Theory and Practice. Volume 4444 of LNCS. Springer Verlag (2007) 160–174
25. Di Pierro, A., Wiklicky, H.: Measuring the precision of abstract interpretations. In: LOPSTR'00. Volume 2042 of LNCS., Springer Verlag (2001) 147–164
26. Campbell, S., Meyer, D.: Generalized Inverse of Linear Transformations. Constable and Company, London (1979)
27. Ben-Israel, A., Greville, T.: Generalised Inverses. 2nd edn. Springer Verlag (2003)
28. Eaton, J.: Gnu Octave Manual. www.octave.org (2002)