# Quantitative Aspects in Program Synthesis

Herbert Wiklicky
Department of Computing, Imperial College London, London, United Kingdom
herbert@imperial.ac.uk

## 1 Introduction

The automatic generation or synthesis and optimisation of code is an extremely complex task, nevertheless it constitutes to some extend the holy grail of software engineering. In this paper we consider an approach to this problem via a non-standard semantical model of programs in terms of linear operator or, simply, as matrices. This allows us to employ well-developed techniques of classical mathematical (non-linear) optimisation. More concretely we describe here an experimental implementation of the framework which treats programs more than dynamical systems than as logical entities. In this setting we then aim in generating or transforming programs on the basis of optimising some of their properties. In this way we try to end up with code that exhibits the desired properties (at least as much as possible).

The initial motivation of this approach can be traced back to when we considered Kocher's attack on the RSA algorithm [3, 5]. In very simple terms [7]: the problems is that the execution time of a certain algorithm (e.g. modular exponentiation) is based on some secrete or high information (concretely, the bits in a secrete key) and thus it is possible to reveal or extract the secret by analysing the running time. For example we have repeatedly, for each bit $k[i]$ to execute code which takes very little or a lot of time: `if` $k[i]$ `then` $\langle short \rangle$ `else` $\langle long \rangle$ `fi`.

In, for example, [1] it has been suggested to obfuscate the time signature by using depleted versions $[short]$ and $[long]$ of $\langle short \rangle$ and $\langle long \rangle$, respectively; i.e. code which is executed in the same time as the original version but which does otherwise not change the state in any way. This *padding* means that we are replacing `if` $k[i]$ `then` $\langle short \rangle$ `else` $\langle long \rangle$ `fi` by `if` $k[i]$ `then` $\langle short \rangle$; $[long]$ `else` $[short]$; $\langle long \rangle$ `fi`. The result is then that both branches always take the same maximal time to execute.

As there is a tradeoff between how easy it is to obtain the secrete (key) from the time signature and the increased running time we suggested to introduce the extra time randomly. The result is a whole manifold of programs $P(p)$ in which the padding is performed with a certain probability $p$ or the original code is executed with probability $1-p$. The idea is now to find the $p^*$ for which we have the optimal balance between extra cost and security.

The purpose of this paper is to extend this idea to allow the generation or transformation of programs as an optimisation problem. We will consider a whole family of programs parameterised by a large number of variables $\lambda_i$ and try to identify those which fulfil certain requirements in an optimal way.

## 2 The Language(s)

Program synthesis goes back to the work by Manna and Waldinger in the late 1960s and 70s. It received renewed interest in the last years, in particular in the area of protocol and controller synthesis, see e.g. the recent special issue on Synthesis [2] where various approaches towards program synthesis presented. To some degree our approach is related to Program Sketching [10], we only provide a 'sketch' of a

Submitted to:
QAPL'14

program which leaves certain parts (blocks, statements) open. In order to fulfil a given specification or to meet certain performance objectives one can employ various algorithms in order to determine the appropriate concrete statements, chosen from a set of potential, possible implementations. In this setting one can distinguish between an *implementation* language and a *specification* language which allows the description of certain templates (including valid alternative implementations) and of assertions, i.e. constraints the implementation should ultimately fulfil.

Our approach uses the same language to specify implementations and templates. We use a language which allows for a probabilistic (rather than a non-deterministic) choice. If we utilise this to describe an implementation the idea is that the choice is made at run time according to a given probability (by a 'coin-flipping' device) while as a specification language the probabilities are chosen a priori, at compile-time such as to optimise the behaviour or performance. This could be summarised as: Probabilities are variables in the context of synthesis and constants when executed. The objectives of a synthesis tasks can be expressed by any appropriate function on the space of possible semantics (which in our case has the structure of a vector space or linear algebra), which we see as some kind of semantical abstraction.

We consider a labelled version of the standard (probabilistic) procedural language as one can find it for example in [8]. Further details can be found in the appendix or e.g. in [4, 6].

$$S \quad ::= \quad [\texttt{skip}]^\ell \mid [x := f(x_1,\ldots,x_n)]^\ell \mid [x \,?\!= \rho]^\ell \mid S_1 \,;\, S_2 \mid [\texttt{choose}]^\ell \; p_1 : S_1 \; \texttt{or} \; p_2 : S_2 \; \texttt{ro} \\ \mid \texttt{if} \; [b]^\ell \; \texttt{then} \; S_1 \; \texttt{else} \; S_2 \; \texttt{fi} \mid \texttt{while} \; [b]^\ell \; \texttt{do} \; S \; \texttt{od}$$

The Linear Operator Semantics (LOS) is intended to model probabilistic computations we therefore have to consider probabilistic states. These describe the situation about the computation ate any given moment in time. A classical state $s \in \mathbf{State} = \mathbf{Var} \to \mathbf{Value}$ associates a certain value $s(x)$ with a variable $x$. We assume, in order to keep the mathematical treatment as simple as possible, that the possible values are finite, e.g. $\mathbf{Value} = \{-MININT, \ldots, MAXINT\}$. A probabilistic state $\sigma \in \mathbf{ProbState} = \mathbf{State} \to [0,1]$ can be seen as a probability distribution on classical states or as a (normalised) vector in the free vector space $\mathscr{V}(\mathbf{Value})$ over **Value**.

The set of probabilistic states forms a (sub-set) of a (finite-dimensional) vector (Hilbert) space. The semantics $\mathbf{T}(P) = [\![P]\!]$ of a program $P$ is a linear map or operator on this vector space which encodes the generator of a Discrete Time Markov Chain (DTMC). The construction of the LOS semantics utilises the tensor product construction "$\otimes$". The tensor product – more precisely, the Kronecker product – of an $n \times m$ matrix $\mathbf{A} = (\mathbf{A}_{ij})$ and an $n' \times m'$ matrix $\mathbf{B} = (\mathbf{B}_{kl})$ is constructed as an $nn' \times mm'$ matrix $\mathbf{A} \otimes \mathbf{B} = (\mathbf{A}_{ij}\mathbf{B})$, i.e. each entry $\mathbf{A}_{ij}$ in $\mathbf{A}$ is multiplied with a copy of the matrix or block $\mathbf{B}$. For further details we refer e.g to [9, Chap. 14].

Given a program $P$, our aim is to define compositionally a matrix representing the program behaviour as a DTMC. The domain of the associated linear operator $\mathbf{T}(P)$ is the space of *probabilistic configurations*, that is distributions over classical configurations, defined by $\mathbf{Dist}(\mathbf{Conf}) = \mathbf{Dist}(\mathbb{X}^v \times \mathbf{Lab}) \subseteq \ell_2(\mathbb{X}^v \times \mathbf{Lab})$, where we identify a statement with its label, or more precisely, an SOS configuration $\langle S, s\rangle \in \mathbf{Conf}$ with the pair $\langle s, init(S)\rangle \in \mathbb{X}^v \times \mathbf{Lab}$. The basic building blocks are the *identity matrix* $\mathbf{I}$ and the *matrix units* $\mathbf{E}_{ij}$, a basic *update matrix* $\mathbf{U}(c)$ which assigns to some variable a constant value $c$, and for Boolean expression $b$ a diagonal *projection matrix* $\mathbf{P}(b)$ which filters out those (classical) states which fulfil $b$, cf. the appendix or e.g. [4, 6]. We denote by $e_i$ the unit vector with $(e_i)_i = 1$ and zero otherwise. As we represent distributions by row vectors we use post-multiplication, i.e. $\mathbf{T}(x) = x \cdot \mathbf{T}$.

We first define a multi-variable versions of test matrices $\mathbf{P}$ and update matrices $\mathbf{U}$ via the tensor product (see in the appendix or e.g. [4, 6]). With the help of these auxiliary matrices we can then define for every program $P$ the matrix $\mathbf{T}(P)$ of the DTMC representing the program executions as the sum of the effects of the individual control flow steps. For each individual control flow step it is of the form $[\![[B]^\ell]\!] \otimes \mathbf{E}_{\ell,\ell'}$ or $\underline{[\![[B]^\ell]\!]} \otimes \mathbf{E}_{\ell,\ell'}$, where $(\ell,\ell')$ or $(\ell,\underline{\ell'}) \in \mathscr{F}(P)$ and $[\![[B]^\ell]\!]$ represents the semantics of the

$$\begin{array}{rclcrcl} [\![[x := e]^{\ell}]\!] & = & \mathbf{U}(x \leftarrow e) & \qquad & [\![[v \,?\!= \rho]^{\ell}]\!] & = & \sum_{c \in \mathbb{X}} \rho(c)\mathbf{U}(x \leftarrow c) \\ [\![[b]^{\ell}]\!] & = & \mathbf{P}(b = \mathtt{false}) & & \underline{[\![[b]^{\ell}]\!]} & = & \mathbf{P}(b = \mathtt{true}) \end{array}$$
$$[\![[\mathtt{skip}]^{\ell}]\!] = \underline{[\![[\mathtt{skip}]^{\ell}]\!]} = \underline{[\![[x := e]^{\ell}]\!]} = \underline{[\![[v \,?\!= \rho]^{\ell}]\!]} = \mathbf{I}$$

Table 1: Elements of the LOS

block $B$ labelled by $\ell$. The matrix $\mathbf{E}_{\ell,\ell'}$ represents the control flow from label $\ell$ to $\ell'$; it is a finite $l \times l$ matrix, where $l$ is the number of (unique) distinct labels in $P$. The definitions of $[\![[B]^{\ell}]\!]$ and $\underline{[\![[B]^{\ell}]\!]}$ are given in Table 3. Based on the local semantics of each labelled block, i.e. $[\![[B]^{\ell}]\!]$ and $\underline{[\![[B]^{\ell}]\!]}$, in $P$ we can define the LOS semantics of $P$ as:

$$\mathbf{T}(P) = \sum_{(\ell,\ell') \in \mathscr{F}(P)} [\![[B]^{\ell}]\!] \otimes \mathbf{E}_{\ell,\ell'} + \sum_{(\ell,\underline{\ell'}) \in \mathscr{F}(P)} \underline{[\![[B]^{\ell}]\!]} \otimes \mathbf{E}_{\ell,\ell'}$$

## 3 An Example: Swapping Variables

We consider a simple situation to illustrate how non-linear optimisation can be used to general or transform programs such that certain requirements are fulfilled.

Given a number of basic blocks we aim in constructing a (small) program which exchanges two variables $x$ and $y$. We assume – to keep the setting as simple as possible – that $x$ and $y$ can only take two values 0 and 1. If we consider the state space for these two variables we need to consider the tensor product $\mathscr{V}(\{0,1\} \times \{0,1\}) = \mathscr{V}(\{0,1\}) \otimes \mathscr{V}(\{0,1\}) = \mathbb{R}^2 \otimes \mathbb{R}^2 = \mathbb{R}^4$. In this four dimensional space the first dimension corresponds to the (classical) state $s_1 = [x \mapsto 0, y \mapsto 0]$, the second one to $s_2 = [x \mapsto 0, y \mapsto 0]$, the third to $s_3 = [x \mapsto 1, y \mapsto 0]$, and the forth to $s_4 = [x \mapsto 1, y \mapsto 1]$.

The swapping operation we aim to implement is thus represented by the matrix

$$\mathbf{S} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

If the values of $x$ and $y$ are the same nothing happens when we swap them, thus the entry 1 in the diagonal corresponding to the first and forth classical state. For the second and third coordinate we only have to exchange the probabilities associated to these two classical states.

We consider a few basic building blocks with which we aim to achieve the task of implementing this simple specification. We try to have several options to achieve our aim and thus allow also for a buffer variable $z$ which we might use to swap $x$ and $y$. Another well known way is to use the 'exclusive or' (xor) to swap $x$ and $y$. In our case we implement xor as $x \oplus y = (x+y) \bmod 2$.

What we aim for is a program $P$ of the form (allowing in the obvious way for an n-array choice):

$$\mathtt{choose}\ \lambda_{1,1} : S_1\ \mathtt{or}\ \ldots\ \mathtt{or}\ \lambda_{1,13} : S_{23};\ \mathtt{choose}\ \lambda_{2,1} : S_1\ \mathtt{or}\ \ldots\ \mathtt{or}\ \lambda_{2,13} : S_{13};\ \mathtt{choose}\ \lambda_{3,1} : S_1\ \mathtt{or}\ \ldots\ \mathtt{or}\ \lambda_{3,13} : S_{13}$$

where we have 13 different elementary blocks $S_{i,j}$ which we enumerate as follows:

$$\begin{array}{c} [\mathtt{skip}]^1 \quad [x := y]^2 \quad [x := z]^3 \quad [y := x]^4 \quad [y := z]^5 \quad [z := x]^6 \quad [z := y]^7 \\ [x := (x+y) \bmod 2]^8 \quad [x := (x+z) \bmod 2]^9 \quad [y := (y+x) \bmod 2]^{10} \\ [y := (y+z) \bmod 2]^{11} \quad [z := (z+x) \bmod 2]^{12} \quad [z := (z+y) \bmod 2]^{13} \end{array}$$

such that $\lim_{t\to\infty}\mathbf{T}(P)^t = \mathbf{S}\otimes\mathbf{E}_{i,f}$, i.e. the program does in three steps what we expect from $\mathbf{S}$ (and control transfers from the initial label $i$ to the final $f$). We ignore the control flow, we are just interested in the transfer functions associated to the basic blocks. The LOS program semantics of the program we aim in generating is made up from this 13 transfer functions $\mathbf{F}_1\ldots\mathbf{F}_{13}$ with $\mathbf{F}_j = [\![S_j]\!]$, i.e.

$$\mathbf{T} = \sum_{i=1}^{3}\mathbf{T}_i \quad \text{with} \quad \mathbf{T}_i = \sum_{j=1}^{13}\lambda_{ij}\mathbf{F}_j$$

Each of the $\mathbf{F}_i$ are constructed as $8\times 8$ matrices on the tensor product space $\mathscr{V}(\{0,1\}\times\{0,1\}\times\{0,1\}) = \mathscr{V}(\{0,1\})\otimes\mathscr{V}(\{0,1\})\otimes\mathscr{V}(\{0,1\}) = \mathbb{R}^2\otimes\mathbb{R}^2\otimes\mathbb{R}^2 = \mathbb{R}^8$. As we do not care what value $z$ has in the end we can abstract it away using a technique called Probabilistic Abstract Interpretation (PAI) using the abstraction operator $\mathbf{A} = \mathbf{I}\otimes\mathbf{I}\otimes\mathbf{A}_f$ (with $\mathbf{A}_f$ the forgetful abstraction) and its concretisation $\mathbf{G}$ given by the Moore-Penrose pseudo-inverse (see [4] or appendix).

With this our main objective function, describing the requirement that we want a program $\mathbf{T}(\lambda_{ij})$ which implements the swap of $x$ and $y$, is given by: $[\Phi_{00}(\lambda_{ij}) = \|\mathbf{A}^\dagger\mathbf{T}(\lambda_{ij})\mathbf{A}-\mathbf{S}\|_2$.We also use a general objective function which penalises for reading or writing to the third variable $z$: $\Phi_{\rho\omega}(\lambda_{ij}) = \|\mathbf{A}^\dagger\mathbf{T}(\lambda_{ij})\mathbf{A}-\mathbf{S}\|_2 + \rho R(\lambda_{ij}) + \omega W(\lambda_{ij})$,where the function $R$ and $W$ determine the probability that in each step of our program the variable $z$ is read or written to respectively. Define two projections $\mathbf{P}_r = \mathrm{diag}(0,0,1,0,1,0,0,0,1,0,1,1,1)$ and $\mathbf{P}_w = \mathrm{diag}(0,0,0,0,0,1,1,0,0,0,0,1,1)$ then $R(\lambda_{ij}) = \|\sum_{i=1}^3(\lambda_{1j})\mathbf{P}_r\|_1$ and $W(\lambda_{ij}) = \|\sum_{i=1}^3(\lambda_{1j})\mathbf{P}_w\|_1$.

The optimisation problem we thus have to solve is given by

$$\min\Phi_{\rho\omega}(\lambda_{ij}) \text{ subject to: } \sum_j\lambda_{ij} = 1\ \forall i = 1,2,3 \text{ and } 0\le\lambda_{ij}\le 1\ \forall i=1,2,3, j=1,\ldots,13$$

**Successful Transformation** If we start with a swap which uses $z$, like $[z := x]^6$; $[x := y]^2$; $[y := z]^5$ which corresponds to 39 values for $\lambda_{ij}$ below (each row corresponds to the three computational steps, and each column to the weight of each of the 13 possible blocks).For $\min\Phi_{11}$ we get after 12 iterations of the standard non-linear optimisation algorithm in `octave` a program transformation namely the following set of $\lambda_{ij}$:

$$\begin{pmatrix} 0&0&0&0&0&1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&1&0&0&0&0&0&0&0&0\end{pmatrix} \mapsto \begin{pmatrix} 0&0&0&0&0&0&0&0&0&1&0&0&0\\ 0&0&0&0&0&0&0&1&0&0&0&0&0\\ 0&0&0&0&0&0&0&0&0&1&0&0&0\end{pmatrix}$$

This corresponds to the program: $[y := (y+x) \bmod 2]^{10}$; $[x := (x+y) \bmod 2]^8$; $[y := (y+x) \bmod 2]^{10}$ which indeed also swaps $x$ and $y$ but does not use the variable $z$ in any way.

**Stuck Transformations.** However, sometimes the optimisation does not work, we get stuck in a local minimum. For example, in the case we start with $[y := x]^4$; $[x := y]^2$; $[z := y]^7$ and try to minimise $\Phi_{00}$ we get (in 14 iterations) indeed a program which satisfies the swapping condition:

$$\begin{pmatrix} 0&0&0&1&0&0&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0&0&0&0&0&0\\ 0&0&0&0&0&0&0&1&0&0&0&0&0\end{pmatrix} \mapsto \begin{pmatrix} 0&0&0&0&0&0&1&0&0&0&0&0&0\\ 0&0&0&1&0&0&0&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0&0&0&0&0&0\end{pmatrix}$$

corresponding to $[z := y]^7$; $[y := x]^4$; $[x := z]^3$. But for $\Phi_{11}$ or $\Phi_{100,100}$ we get no improvement, we remain stuck with the initial program.

# References


[1] Johan Agat (2000): *Transforming out timing leaks*. In: *Proceedings of POPL'00*, ACM Press, pp. 40–53.

[2] Rastislav Bodik & Barabara Jobstmann (2013): *Algorithmic Program Synthesis: Introduction*. *Int. J. Softw. Tools Technol. Transfer* 15, pp. 397–411.

[3] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2008): *Quantifying Timing Leaks and Cost Optimisation*. In: *Proceedings of ICICS'08*, *LNCS* 5308, Springer Verlag, pp. 81–96.

[4] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2010): *Probabilistic Semantics and Analysis*. In: *Formal Methods for Quantitative Aspects of Programming Languages*, *LNCS* 6155, Springer Verlag, pp. 1–42.

[5] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2011): *Probabilistic timing covert channels: to close or not to close?* *International Journal of Information Security* 10(2), pp. 83–106.

[6] Alessandra Di Pierro & Herbert Wiklicky (2013): *Semantics of Probabilistic Programs: A Weak Limit Approach*. In Chung chieh Shan, editor: *Proceedings of APLAS13 – 11th Asian Symposium on Programming Languages and Systems*, *Lecture Notes in Computer Science* 8301, Springer Verlag, pp. 241–256.

[7] Paul C. Kocher (1996): *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*. *Lecture Notes in Computer Science* 1109, pp. 104–113.

[8] Dexter Kozen (1981): *Semantics of Probabilistic Programs*. *J. Comput. Syst. Sci.* 22(3), pp. 328–350.

[9] S. Roman (2005): *Advanced Linear Algebra*, 2nd edition. Springer Verlag.

[10] Armando Solar-Lezama (2013): *Program Sketching*. *Int. J. Softw. Tools Technol. Transfer* 15, pp. 475–495.

# A The Language

## A.1 Syntax

We consider a labelled version of the standard (probabilistic) procedural language as one can find it for example in [8]. Further details can be found, e.g., [4]

$$
\begin{array}{lcl}
S & ::= & [\texttt{skip}]^{\ell} \\
 & | & [x := f(x_1,\ldots,x_n)]^{\ell} \\
 & | & [x \texttt{ ?= } \rho]^{\ell} \\
 & | & S_1 \texttt{ ; } S_2 \\
 & | & [\texttt{choose}]^{\ell}\ p_1 : S_1 \texttt{ or } p_2 : S_2 \texttt{ ro} \\
 & | & \texttt{if } [b]^{\ell} \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi} \\
 & | & \texttt{while } [b]^{\ell} \texttt{ do } S \texttt{ od}
\end{array}
$$

Table 2: The Labelled Syntax

The statement `skip` does not have any operational effect but can be used, for example, as a placeholder in conditional statements. We have the usual (deterministic) assignment $x := e$, sometimes also in the form $x := f(x_1,\ldots,x_n)$.

In the random assignment $x \texttt{ ?= } \rho$, the value of a variable $x$ is set to a value according to some random distribution $\rho$. In [8] it is left open how to define or specify distributions $\rho$ in detail. We will use occasionally an ad-hoc notation as sets of tuples $\{(v_i, p_i)\}$ expressing the fact that value $v_i$ will be selected with probability $p_i$; or just as a set $\{v_i\}$ assuming a uniform distribution on the values $v_i$. It might be useful to assume that the random number generator or scheduler which implements this construct can only implement choices over finite ranges, but in principle we can also use distributions with infinite support. For the rest we have the usual sequential composition, conditional statement and loop. We leave the detailed syntax of functions $f$ or expressions $e$ open as well as for boolean expressions or test $b$ in conditionals and loop statements. For each (labelled) statement in this language we identify the initial and final label

## A.2 Semantics

The Linear Operator Semantics (LOS) is intended to model probabilistic computations we therefore have to consider probabilistic states. These describe the situation about the computation at any given moment in time. Our model is based on a discrete time model. The information will specify the probability that the computational system in question is in a particular classical state.

A classical state $s \in \mathbf{State} = \mathbf{Var} \to \mathbf{Value}$ associates a certain value $s(x)$ with a variable $x$. We assume, in order to keep the mathematical treatment as simple as possible, that the possible values are finite, e.g. $\mathbf{Value} = \{-MININT, \ldots, MAXINT\}$. A probabilistic state $\sigma \in \mathbf{ProbState} = \mathbf{State} \to [0,1]$ can be seen as a probability distribution on classical states or as a (normalised) vector in the free vector space $\mathscr{V}(\mathbf{Value})$ over **Value**.

The set of probabilistic states forms a (sub-set) of a (finite-dimensional) vector (Hilbert) space. The semantics $\mathbf{T}(P) = [\![P]\!]$ of a program $P$ is a linear map or operator on this vector space which encodes the generator of a Discrete Time Markov Chain (DTMC). DTMC are non-terminating processes: it is assumed that there is always a next state and the process goes on forever. In order to reflect this property in our semantics, we introduce a terminal statement `stop` which indicates successful termination. Then

the termination with a state $s$ in the classical setting is represented here by reaching the final configuration $\langle \mathtt{stop}, s\rangle$ which then 'loops' forever after. This means that we implicitly extend a statement $S$ to construct full programs of the form $P \equiv S;\ [\mathtt{stop}]^{\ell^*}$.

The *tensor product* is an essential element of the description of probabilistic states and the semantical operator $\mathbf{T}(P)$. The tensor product – more precisely, the Kronecker product, i.e. the coordinate based version of the abstract concept of a tensor product – of two vectors $(x_1,\ldots,x_n)$ and $(y_1,\ldots,y_m)$ is given by $(x_1y_1,\ldots,x_1y_m,\ldots,x_ny_1,\ldots,x_ny_m)$ an $nm$ dimensional vector. For an $n\times m$ matrix $\mathbf{A} = (\mathbf{A}_{ij})$ and an $n'\times m'$ matrix $\mathbf{B} = (\mathbf{B}_{kl})$ we construct similarly an $nn'\times mm'$ matrix $\mathbf{A}\otimes\mathbf{B} = (\mathbf{A}_{ij}\mathbf{B})$, i.e. each entry $\mathbf{A}_{ij}$ in $\mathbf{A}$ is multiplied with a copy of the matrix or block $\mathbf{B}$. For further details we refer e.g to [9, Chap. 14].

Given a program $P$, our aim is to define compositionally an infinite matrix representing the program behaviour as a DTMC. The domain of the associated linear operator $\mathbf{T}(P)$ is the space of *probabilistic configurations*, that is distributions over classical configurations, defined by $\mathbf{Dist}(\mathbf{Conf}) = \mathbf{Dist}(\mathbb{X}^v\times\mathbf{Lab}) \subseteq \ell_2(\mathbb{X}^v\times\mathbf{Lab})$, where we identify a statement with its label, or more precisely, an SOS configuration $\langle S, s\rangle \in \mathbf{Conf}$ with the pair $\langle s, \mathit{init}(S)\rangle \in \mathbb{X}^v\times\mathbf{Lab}$.

Among the building blocks of the construction of $T(P)$ are the *identity matrix* $\mathbf{I}$ and the *matrix units* $\mathbf{E}_{ij}$ containing only a single non zero entry $(\mathbf{E}_{ij})_{ij} = 1$ and zero otherwise. We denote by $e_i$ the unit vector with $(e_i)_i = 1$ and zero otherwise. As we represent distributions by row vectors we use post-multiplication, i.e. $\mathbf{T}(x) = x\cdot\mathbf{T}$.

A basic operator is the *update matrix* $\mathbf{U}(c)$ which implements state changes. The intention is that from an initial probabilistic state $\sigma$, e.g. a distribution over classical states, we get a new probabilistic state $\sigma'$ by the product $\sigma' = \sigma\cdot\mathbf{U}$. The matrix $\mathbf{U}(c)$ implements the deterministic update of a variable to a constant $c$ via $(\mathbf{U}(c))_{ij} = 1$ if $\xi(c) = j$ and 0 otherwise, with $\xi : \mathbb{X}\to\mathbb{N}$ the underlying enumeration of values in $\mathbb{X}$. In other words, this is a matrix which has only one column (corresponding to $c$) containing 1s while all other entries are 0. Whatever the value of a variable is, after applying $\mathbf{U}(c)$ to the state vector describing the current situation we get a *point* distribution expressing the fact that the value of our variable is now $c$.

We also define for any Boolean expression $b$ on $\mathbb{X}$ a diagonal *projection matrix* $\mathbf{P}$ with $(\mathbf{P}(b))_{ii} = 1$ if $b(c)$ holds and $\xi(c) = i$ and 0 otherwise. The purpose of this diagonal matrix is to "filter out" only those states which fulfil the condition $b$. If we want to apply an operator with matrix representation $\mathbf{T}$ only if a certain condition $b$ is fulfilled then pre-multiplying this $\mathbf{P}(b)\cdot\mathbf{T}$ achieves this effect.

In Table 3 we first define a multi-variable versions of the test matrices and the update matrices via the tensor product '$\otimes$'.

With the help of the auxiliary matrices we can now define for every program $P$ the matrix $\mathbf{T}(P)$ of the DTMC representing the program executions as the sum of the effects of the individual control flow steps. For each individual control flow step it is of the form $[\![[B]^\ell]\!]\otimes\mathbf{E}_{\ell,\ell'}$ or $\underline{[\![[B]^\ell]\!]}\otimes\mathbf{E}_{\ell,\ell'}$, where $(\ell,\ell')$ or $(\ell,\underline{\ell'}) \in \mathscr{F}(P)$ and $[\![[B]^\ell]\!]$ represents the semantics of the block $B$ labelled by $\ell$. The matrix $\mathbf{E}_{\ell,\ell'}$ represents the control flow from label $\ell$ to $\ell'$; it is a finite $l\times l$ matrix, where $l$ is the number of (unique) distinct labels in $P$.

The definitions of $[\![[B]^\ell]\!]$ and $\underline{[\![[B]^\ell]\!]}$ are given in Table 3. The semantics of an assignment block is obviously given by $\mathbf{U}(\mathtt{x}\leftarrow e)$. For the random assignment we simply take the linear combination of assignments to all possible values, weighted by the corresponding probability given by the distribution $\rho$. The semantics of a test block $[b]^\ell$ is given by its positive and its negative part, both are test operators $\mathbf{P}(b = \mathtt{true})$ and $\mathbf{P}(b = \mathtt{false})$ as described before. The meaning of $\underline{[\![[B]^\ell]\!]}$ is non-trivial only for tests $b$ while it is the identity for all the other blocks. The positive and negative semantics of all blocks is independent of the context and can be studied and analysed in isolation from the rest of the program $P$.

$$\begin{array}{rclcrcl}
\mathbf{P}(s) & = & \bigotimes^{v} \mathbf{P}(s(\mathtt{x}_i)) & & \mathbf{U}(\mathtt{x}_k \leftarrow c) & = & \bigotimes^{k-1} \mathbf{I} \otimes \mathbf{U}(c) \otimes \bigotimes^{v}_{i=k+1} \mathbf{I} \\
\mathbf{P}(e=c) & = & \sum_{\mathscr{E}(e)s=c}^{i=1} \mathbf{P}(s) & & \mathbf{U}(\mathtt{x}_k \leftarrow e) & = & \sum_{c}^{i=1} \mathbf{P}(e=c)\mathbf{U}(\mathtt{x}_k \leftarrow c)
\end{array}$$

$$\begin{array}{rclcrcl}
[\![[x := e]^{\ell}]\!] & = & \mathbf{U}(x \leftarrow e) & & [\![[v \mathtt{ ?= } \rho]^{\ell}]\!] & = & \sum_{c \in \mathbb{X}} \rho(c) \mathbf{U}(x \leftarrow c) \\
[\![[b]^{\ell}]\!] & = & \mathbf{P}(b = \mathtt{false}) & & \underline{[\![[b]^{\ell}]\!]} & = & \mathbf{P}(b = \mathtt{true})
\end{array}$$

$$[\![[\mathtt{skip}]^{\ell}]\!] = \underline{[\![[\mathtt{skip}]^{\ell}]\!]} = \underline{[\![[x := e]^{\ell}]\!]} = \underline{[\![[v \mathtt{ ?= } \rho]^{\ell}]\!]} = \mathbf{I}$$

Table 3: Elements of the LOS

Based on the local (forward) semantics of each labelled block, i.e. $[\![[B]^{\ell}]\!]$ and $\underline{[\![[B]^{\ell}]\!]}$, in $P$ we can define the LOS semantics of $P$ as:

$$\mathbf{T}(P) = \sum_{(\ell,\ell') \in \mathscr{F}(P)} [\![[B]^{\ell}]\!] \otimes \mathbf{E}_{\ell,\ell'} + \sum_{(\ell,\underline{\ell'}) \in \mathscr{F}(P)} \underline{[\![[B]^{\ell}]\!]} \otimes \mathbf{E}_{\ell,\ell'}$$

A minor adjustment is required to make our semantics conform to the DTMC model. As paths in a DTMC are *maximal* (i.e. infinite) in the underlying directed graph, we will add a single final loop via a virtual label $\ell^*$. This corresponds to adding to $\mathbf{T}(P)$ the factor $\mathbf{I} \otimes \mathbf{E}_{\ell^*,\ell^*}$.

# B The XOR Example

## B.1 Further Details

The eight dimensional state space has a base which corresponds to the (classical) states:

$$\begin{array}{lll}
s_1 & \ldots & [x \mapsto 0, y \mapsto 0, z \mapsto 0] \\
s_2 & \ldots & [x \mapsto 0, y \mapsto 0, z \mapsto 1] \\
s_3 & \ldots & [x \mapsto 0, y \mapsto 1, z \mapsto 0] \\
s_4 & \ldots & [x \mapsto 0, y \mapsto 1, z \mapsto 1] \\
s_5 & \ldots & [x \mapsto 1, y \mapsto 0, z \mapsto 0] \\
s_6 & \ldots & [x \mapsto 1, y \mapsto 0, z \mapsto 1] \\
s_7 & \ldots & [x \mapsto 1, y \mapsto 1, z \mapsto 0] \\
s_8 & \ldots & [x \mapsto 1, y \mapsto 1, z \mapsto 1]
\end{array}$$

The abstraction $\mathbf{A}$ and its concretisation $\mathbf{A}^{\dagger}$ are concretely given by:

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

and its concretisation function given by the Moore-Penrose pseudo-inverse:

$$\mathbf{G} = \mathbf{A}^{\dagger} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} \frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

## B.2 Concrete Implementation

The program we implement concretely with and which contains all our basic blocks is:

```
var
x : {0,1};
y : {0,1};
z : {0,1};

begin
skip;
#
x := y;
x := z;
y := x;
y := z;
z := x;
z := y;
#
x := (x+y)%2;
x := (x+z)%2;
y := (y+x)%2;
y := (y+z)%2;
z := (z+x)%2;
z := (z+y)%2;
end
```

Executing a OCAML tool `pwc` on this program results in the output

```
pwc 0.95 - pWhile compiler - (c) 2008-13 H.Wiklicky

1: skip
2: x := y
3: x := z
4: y := x
5: y := z
6: z := x
7: z := y
8: x := ((x + y) % 2)
9: x := ((x + z) % 2)
10: y := ((y + x) % 2)
```

```
11: y := ((y + z) % 2)
12: z := ((z + x) % 2)
13: z := ((z + y) % 2)

(1, 1., 2)
(2, 1., 3)
(3, 1., 4)
(4, 1., 5)
(5, 1., 6)
(6, 1., 7)
(7, 1., 8)
(8, 1., 9)
(9, 1., 10)
(10, 1., 11)
(11, 1., 12)
(12, 1., 13)

Identifiers:
x in {0, 1}
y in {0, 1}
z in {0, 1}

x starts at 1
y starts at 2
z starts at 3

Done with ‘XOREx.pw’.
```