Program Synthesis and Linear Operator Semantics

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

For deterministic and probabilistic programs we investigate the problem of program synthesis and program optimisation (with respect to non-functional properties) in the general setting of global optimisation. This approach is based on the representation of the semantics of programs and program fragments in terms of linear operators, i.e. as matrices. We exploit in particular the fact that we can automatically generate the representation of the semantics of elementary blocks. These can then can be used in order to compositionally assemble the semantics of a whole program, i.e. the generator of the corresponding Discrete Time Markov Chain (DTMC). We also utilise a generalised version of Abstract Interpretation suitable for this linear algebraic or functional analytical framework in order to formulate semantical constraints (invariants) and optimisation objectives (for example performance requirements).

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 as 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 [10, 12]. In very simple terms [19]: 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 λ_i and try to identify those which fulfil certain requirements in an optimal way.

2 The General Approach

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” [3] where various approaches towards program synthesis presented. To some degree our approach is related to “Program Sketching” [24], we only provide a ‘sketch’ of a 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.

The central idea of our approach is to interpret the probabilistic choice in a program not as a choice made at *run-time* but as a parameter which describes a whole manifold of possible programs or perhaps better their semantics. The aim is the identification of the “right” parameters at *compile-time* and thus to generate or synthesise the desired program behaviour.

2.1 A Manifold of Sketches

Our approach can be also seen as a form of *continuous* sketching in the sense of Solar-Lezama [24]. Instead of allowing for a sketch like (cf [23, p26])

```
int W = 32;
void main(bit[W] x, bit[W] y){
  bit[W] xold = x;
  bit[W] yold = y;
  if(??) { x = x ^ y;} else { y = x ^ y}; }
  if(??) { x = x ^ y;} else { y = x ^ y}; }
  if(??) { x = x ^ y;} else { y = x ^ y}; }
  assert y == xold && x == yold;
}
```

our approach we would consider something like

```
int W = 32;
void main(bit[W] x, bit[W] y){
  bit[W] xold = x;
  bit[W] yold = y;
  choose p:{ x = x ^ y;} or 1-p:{ y = x ^ y}; } ro
  choose q:{ x = x ^ y;} or 1-q:{ y = x ^ y}; } ro
  choose r:{ x = x ^ y;} or 1-r:{ y = x ^ y}; } ro
  assert y == xold && x == yold;
}
```

where the `choose p:S1 or 1-p:S2 ro` statements implements random choices: With probability p we execute `S1` and with probability $1-p$ we execute `S2`. The aim is to identify in the design space the correct or optimal parameters $p,q,r,\ldots \in [0,1]$ such that certain conditions or assertions are fulfilled.

We will specify the semantics of programs as Linear Operators on an appropriate vector space (containing all probabilistic states) or simply as matrices, i.e. $[\![P]\!] \in \mathscr{L}(\mathscr{V})$, which describe the generator of a Discrete Time Markov Chain (DTMC). The desired properties – i.e. the objectives of a certain synthesis problem – will be recast as properties of this linear operator. In this way, the assertions to be fulfilled are translated into a (non-linear) optimisation problem with the appropriate objective function $\Phi : \mathscr{L}(\mathscr{V}) \to \mathbb{R}$ and constraints (e.g. guaranteeing the normalisation of probabilities). The synthesis problem becomes in this way a global (in general non-linear) optimisation (minimisation or maximisation) problem.

2.2 Probabilistic Abstract Interpretation

In order to relate usual notions of program properties and the objective function Φ we will utilise our theory of Probabilistic Abstract Interpretation (PAI) which generalises Cousot's Abstract Interpretation framework (though it is different from the approach in, for example, [21, 7]).

Classically the correctness of a program analysis is asserted with respect to the semantics in terms of a correctness relation. The theory of Abstract Interpretation allows for constructing analyses that are automatically correct without having to prove it a posteriori [5, 6]. The main applications of this theory are for the analysis of safety-critical systems as it guarantees correct answers at the cost of precision.

For probabilistic systems or the probabilistic analysis of (non-)deterministic ones, the theory of Probabilistic Abstract Interpretation (PAI) allows for the construction of analyses that are possibly unsafe but maximally precise [14, 15]. Its main applications are therefore in fields like speculative optimisation and the analysis of trade-offs. PAI has been used for the definition of various analyses based on the LOS (see e.g. [13, 9, 8]. In the following we will, for the sake of a simpler mathematical treatment, only consider finite dimensional versions of PAI (although it is also possible to extend the framework to infinite dimensional spaces, e.g. [16, 17]).

PAI relies on the notion of generalised (or pseudo-)inverse in place of the notion of a Galois connection as in classical Abstract Interpretation. This notion is well-established in mathematics where it is used for finding approximate, so-called least-square solutions (cf. e.g. [2]).

Definition 1 *Let $\mathscr{H}_1$ and $\mathscr{H}_2$ be two Hilbert spaces and $\mathbf{A} : \mathscr{H}_1 \mapsto \mathscr{H}_2$ a linear map between them. A linear map $\mathbf{A}^\dagger = \mathbf{G} : \mathscr{H}_2 \mapsto \mathscr{H}_1$ is the* Moore-Penrose pseudo-inverse *of $\mathbf{A}$ iff $\mathbf{A} \circ \mathbf{G} = \mathbf{P_A}$ and $\mathbf{G} \circ \mathbf{A} = \mathbf{P_G}$, where $\mathbf{P_A}$ and $\mathbf{P_G}$ denote orthogonal projections onto the ranges of $\mathbf{A}$ and $\mathbf{G}$.*

An linear operator $\mathbf{P} : \mathscr{H} \to \mathscr{H}$ is an *orthogonal projection* if $\mathbf{P}^* = \mathbf{P}^2 = \mathbf{P}$, where $.^*$ denotes the *adjoint*. The adjoint is defined implicitly via the condition: $\langle x \cdot \mathbf{P}, y \rangle = \langle x, y \cdot \mathbf{P}^* \rangle$ for all $x, y \in \mathscr{H}$, where $\langle .,. \rangle$ denotes the inner product on $\mathscr{H}$. For real matrices the adjoint correspond simply to the transpose matrix $\mathbf{P}^* = \mathbf{P}^t$ [22, Ch 10].

If $\mathscr{C}$ an $\mathscr{D}$ are two Hilbert spaces, and $\mathbf{A} : \mathscr{C} \to \mathscr{D}$ and $\mathbf{G} : \mathscr{D} \to \mathscr{C}$ are linear operators between the concrete domain $\mathscr{C}$ and the abstract domain $\mathscr{D}$, such that $\mathbf{G}$ is the Moore-Penrose pseudo-inverse of $\mathbf{A}$, then we say that $(\mathscr{C}, \mathbf{A}, \mathscr{D}, \mathbf{G})$ forms a *probabilistic abstract interpretation*.

A very simple example of such a abstraction is the *forgetful abstraction* $\mathbf{A}_f : \mathbb{R}^n \to \mathbb{R}$ which is represented by an $n \times 1$ dimensional (column) matrix with all entries equal to 1. Its Moore-Penrose pseudo-inverse $\mathbf{A}_f^\dagger$ is simply a $1 \times n$ (row) matrix with all entries $\frac{1}{n}$. This abstraction "forgets" about all details and just records the existence of a system (part).

3 The Language

Our approach uses the same language to specify implementations and templates or sketches. 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 a kind of semantical abstraction.

3.1 Syntax

We consider a labelled version of the standard (probabilistic) procedural language as one can find it for example in [20]. Further details can be found, e.g., [11]. All statements are labelled in order to allow a convenient construction of its semantics (indicating relevant program points). These labels can always be reconstructed if a unlabelled version of a program is considered.

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

Table 1: 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 ρ. In [20] it is left open how to define or specify distributions ρ 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. The statement `choose` $p_1 : S_1$ `or` $p_2 : S_2$ `ro` executes randomly either S_1 or S_2, assuming an implicit normalisation of probabilities, i.e. $p_1 + p_2 = 1$. 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

3.2 Linear Operator 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 $\mathcal{V}(\mathbf{Value})$ over $\mathbf{Value}$.

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 $\langle \texttt{stop}, s\rangle$ which then 'loops' forever after. This means that we implicitly extend a statement S to construct full programs of the form $P \equiv S;\ [\texttt{stop}]^{\ell^*}$.

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 $\mathbf{B}$. That is: 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

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

The tensor product of two vector spaces $\mathcal{V} \otimes \mathcal{W}$ can be defined as the formal linear combinations of the tensor products $v_i \otimes w_j$ with v_i and w_j base vectors in $\mathcal{V}$ and $\mathcal{W}$, respectively. For further details we refer e.g to [22, 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 $\langle s, init(S)\rangle \in \mathbb{X}^v \times \mathbf{Lab}$.

Among the building blocks of the construction of $\mathbf{T}(P)$ 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 σ, e.g. a distribution over classical states, we get a new probabilistic state σ' 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.

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

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

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

Table 2: Elements of the LOS

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 2 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 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 (ℓ,ℓ') or $(\ell,\underline{\ell'}) \in \mathscr{F}(P)$ and $[\![[B]^{\ell}]\!]$ represents the semantics of the block B labelled by ℓ. The matrix $\mathbf{E}_{\ell,\ell'}$ represents the control flow from label ℓ to ℓ'; 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 2. 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 ρ. 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.

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 ℓ^*. This corresponds to adding to $\mathbf{T}(P)$ the factor $\mathbf{I} \otimes \mathbf{E}_{\ell^*,\ell^*}$.

We first define a multi-variable versions of test matrices $\mathbf{P}$ and update matrices $\mathbf{U}$ via the tensor product (see e.g. [11, 16]). 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 (ℓ,ℓ') or $(\ell,\underline{\ell'}) \in \mathscr{F}(P)$ and $[\![[B]^{\ell}]\!]$ represents the semantics of the block B labelled by ℓ. The matrix $\mathbf{E}_{\ell,\ell'}$ represents the control flow from label ℓ to ℓ'; 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 2. 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:

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

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 ℓ^* as discussed in Section 3. This corresponds to adding to $\mathbf{T}(P)$ the factor $\mathbf{I}\otimes\mathbf{E}_{\ell^*,\ell^*}$.

3.3 LOS and PAI

Important for the applicability of PAI in the context of program analysis is the fact that it possesses nice compositionality properties. These allows us to construct the abstract semantics by abstracting the single blocks of the concrete semantics $\mathbf{T}(S)$:

$$\mathbf{T}(S)^\# = \mathbf{A}^\dagger\mathbf{T}(S)\mathbf{A} = \mathbf{A}^\dagger\left(\sum [\![[B]^\ell]\!]\right)\mathbf{A} = \sum\left(\mathbf{A}^\dagger [\![[B]^\ell]\!] \mathbf{A}\right) = \sum [\![[B]^\ell]\!]^\#$$

(where, for simplicity, we do not distinguish between the positive and negative semantics of blocks). The fact that we can work with the abstract semantics of individual blocks instead of the full operator obviously reduces the complexity of the analysis substantially. Another important fact is that the Moore-Penrose pseudo-inverse of a tensor product can be computed as [2, 2.1,Ex 3]:

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

We can therefore abstract properties of individual variables and then combine them in the global abstraction. This is also made possible by the definition of the concrete LOS semantics which is heavily based on the use of tensor product. Typically we have $[\![[B]^\ell]\!] = (\bigotimes_{i=1}^v \mathbf{T}_{i\ell})\otimes\mathbf{E}_{\ell\ell'}$ or a sum of a few of such terms. The $\mathbf{T}_{i\ell}$ represents the effect of $\mathbf{T}(S)$, and in particular of $[\![[B]^\ell]\!]$, on variable i at label ℓ (both labels and variables only form a finite set). For example, we can define an abstraction $\mathbf{A}$ for one variable and apply it individually to all variables (e.g. extracting their even/odd property), or use different abstractions for different variables (maybe even forgetting about some of them by using $\mathbf{A}_f = (1,1,\ldots)^t$) and define $\mathbf{A} = \bigotimes_{i=1}^v \mathbf{A}_i$ such that $\mathbf{A}^\dagger = \bigotimes_{i=1}^v \mathbf{A}_i^\dagger$ in order to get an analysis on the full state space.

3.4 Relation to Other Probabilistic Semantics

We can use the LOS to reconstruct the semantics of Kozen [20] by simply taking the limit of $\mathbf{T}(S)^n(s_0)$ for $n\to\infty$ for all initial states s_0. The limit state $\mathbf{T}(S)^n(s_0)$ still contains too much information in relation to Kozen's semantics; in fact we only need the probability distributions on the possible values of the variables at the final label.

In order to extract information about the probability that variables have certain values at a certain label, i.e. program point ℓ, we can use the operator $\mathbf{I}\otimes\ldots\otimes\mathbf{I}\otimes\mathbf{E}_{\ell,\ell}$. In particular, for extracting the information about a probabilistic state we will use $\mathbf{S}_\ell = \mathbf{I}\otimes\ldots\otimes\mathbf{I}\otimes e_\ell$ which forgets about all distributions at other labels than ℓ. In particular we use $\mathbf{S}_f$ for the final looping `stop` statement and $\mathbf{S}_i$ for the initial label $init(P)$ of the program. We denote by e_0 the base vector in $\mathbb{R}^l$ which expresses the fact that we are in the initial label, i.e. $e_0 = e_{init(P)}$.

Proposition 1 ([16]) *Given a program P and an initial state s_0 as a distribution over the program variables, then $(s_0 \otimes e_0)\mathbf{T}(P)^n\mathbf{S}_f$ corresponds to the distribution over all states on which P terminates in n or less computational steps.*

We can now show that the effect of the LOS operator we obtain as solution to Kozen's fixed-point equations agrees with the "output" $\lim_{n\rightarrow\infty}(s_0 \otimes e_0)\mathbf{T}^n\mathbf{S}_f$ we get via the LOS. Essentially, both semantics define the same I/O operator, provided we supply them with the appropriate input. However, the LOS also provides information about internal labels and reflects the relation between different variables via the tensor product representation.

Proposition 2 ([16]) *Given a program P and an initial probabilistic state s_0 as a distribution over the program variables, let $[\![P]\!]$ be Kozen's semantics of P and $\mathbf{T}(P)$ the LOS. Then $(s_0 \otimes e_0)(\lim_{n\rightarrow\infty}\mathbf{T}^n)\mathbf{S}_f = s_0[\![P]\!]$.*

4 An Example: 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 [25] 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.

We first consider two programs H_t and H_w in which the contestant is either sticking to his/her initial choice or where there is a switch to (the other closed) door:

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

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

We introduce a short hand notation (macros) with ℓ any program label:

$$\begin{array}{rcl}
\mathbf{T}_{\text{pick}} & = & \frac{1}{3}\left(\mathbf{U}(\mathtt{d}\leftarrow 0)+\mathbf{U}(\mathtt{d}\leftarrow 1)+\mathbf{U}(\mathtt{d}\leftarrow 2)\right)\otimes\mathbf{E}(1,2)+ \\
& & \frac{1}{3}\left(\mathbf{U}(\mathtt{g}\leftarrow 0)+\mathbf{U}(\mathtt{g}\leftarrow 1)+\mathbf{U}(\mathtt{g}\leftarrow 2)\right)\otimes\mathbf{E}(2,3)+ \\
& & \frac{1}{3}\left(\mathbf{U}(\mathtt{o}\leftarrow 0)+\mathbf{U}(\mathtt{o}\leftarrow 1)+\mathbf{U}(\mathtt{o}\leftarrow 2)\right)\otimes\mathbf{E}(3,4)+ \\
& & \mathbf{P}((\mathtt{o}==\mathtt{g})||(\mathtt{o}==\mathtt{d})=\mathtt{true})\otimes\mathbf{E}(4,5)+ \\
& & \mathbf{P}((\mathtt{o}==\mathtt{g})||(\mathtt{o}==\mathtt{d})=\mathtt{false})\otimes\mathbf{E}(4,6)+ \\
& & \mathbf{U}(\mathtt{o}\leftarrow(o+1)\%3)\otimes\mathbf{E}(5,4)
\end{array}$$

and (parametric in the initial label of the final part of H_w):

$$\begin{array}{rcl}
\mathbf{T}_{\text{flip}}(\ell) & = & \mathbf{U}(\mathtt{g}\leftarrow(g+1)\%3)\otimes\mathbf{E}(\ell,\ell+1)+ \\
& & \mathbf{P}((\mathtt{g}==\mathtt{o})=\mathtt{true})\otimes\mathbf{E}(\ell+1,\ell+2)+ \\
& & \mathbf{P}((\mathtt{g}==\mathtt{o})=\mathtt{false})\otimes\mathbf{E}(\ell+1,\ell+3)+ \\
& & \mathbf{U}(\mathtt{g}\leftarrow(g+1)\%3)\otimes\mathbf{E}(\ell,\ell+1)
\end{array}$$

The LOS semantics of the two programs can then be easily specified:

$$\begin{array}{rcl}
\mathbf{T}(H_t) & = & \mathbf{T}_{\text{pick}}+\mathbf{I}\otimes\mathbf{E}(6,6) \\
\mathbf{T}(H_w) & = & \mathbf{T}_{\text{pick}}+\mathbf{T}_{\text{flip}}(6)+\mathbf{I}\otimes\mathbf{E}(9,9)
\end{array}$$

The matrix representations of the individual transfer operators **P**, **U** etc. and the complete LOS semantics of both programs (as 162×162 or 243×243 matrices) are given in detail in [11]. We can compute these matrices via an experimental tool "`pwc`" which has been written in `OCaml` and which generates the matrices to be used with the numerical tool `octave`. These matrices are based on the enumeration of elements in **State** as follows:

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

We can use the LOS semantics 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:

$$\vec{x}_0 = \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \end{pmatrix},$$

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 $\vec{x}_0\mathbf{T}(H)^t = \vec{x}_0\mathbf{T}(H)^{t+1}$, 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.

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

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 $\vec{x}_t$ the terminal configuration distribution as well as for H_w with terminal distribution $\vec{x}_w$

$$\begin{array}{rcl} \vec{x}_t \cdot (\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{A}_f \otimes \mathbf{A}_f) &=& \begin{pmatrix} 0.11 & 0.11 & 0.11 & 0.11 & 0.11 & 0.11 & 0.11 & 0.11 & 0.11 \end{pmatrix} \\ \vec{x}_w \cdot (\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{A}_f \otimes \mathbf{A}_f) &=& \begin{pmatrix} 0.22 & 0.04 & 0.07 & 0.07 & 0.22 & 0.04 & 0.04 & 0.07 & 0.22 \end{pmatrix} \end{array}$$

The nine coordinates of these vectors correspond to $(\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 0)$, $(\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 1)$, $(\mathtt{d} \mapsto 0, \mathtt{g} \mapsto 2)$, $(\mathtt{d} \mapsto 1, \mathtt{g} \mapsto 0), \dots, (\mathtt{d} \mapsto 2, \mathtt{g} \mapsto 2)$. 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 and H_w respectively:

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

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.

We can also consider a more general strategy which is a combination of *switching* and *sticking*: With a certain probability p the contestant switches or sticks with his/her first choice. This corresponds to the following program(s) $H(p)$ parametrised using the probability $p \in [0,1]$:

```
# Pick winning door
d ?= {0,1,2};
# Pick guess
g ?= {0,1,2};
# Open empty door
o ?= {0,1,2};
while ((o == g) || (o == d)) do
  o := (o+1)%3;
od;
# Switch or stick with probability p
choose p: g := (g+1)%3;
          while (g == o) do
          g := (g+1)%3; od
or (1-p): skip
ro;
```

One can now analyse the winning chance of these programs $H(p)$ depending on the parameter p and see what chance of winning we have. The LOS is given by

$$\mathbf{T}(H(p)) \quad = \quad \mathbf{T}_{\text{pick}} + p \cdot \mathbf{I} \otimes \mathbf{E}(6,7) + (1-p) \cdot \mathbf{I} \otimes \mathbf{E}(6,10) + \mathbf{T}_{\text{flip}}(7) + \mathbf{I} \otimes \mathbf{E}(10,11) + \mathbf{I} \otimes \mathbf{E}(11,11)$$

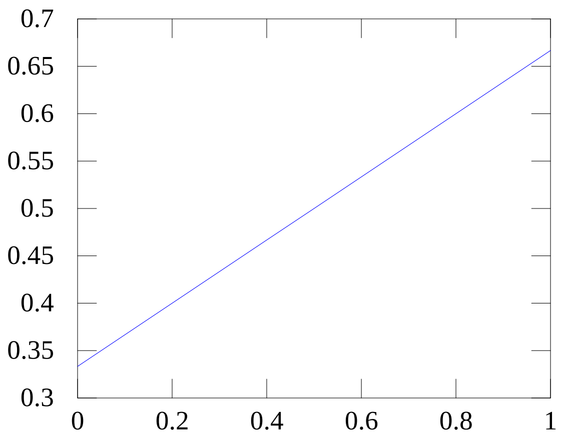
One way to interpret this is to see this as the problem of determining the optimal strategy by identifying the optimal value for p for which the winning chance is the largest, i.e. as an optimisation problem with objective function:

$$\Phi(p) = s_0 \cdot (\lim_{n\to\infty} \mathbf{T}(H(p))^n) \cdot (\mathbf{A}_w \otimes \mathbf{A}_f \otimes \mathbf{A}_f) \cdot \mathbf{P}_1$$

where $\mathbf{T}(H(p))$ is the LOS semantics of $H(p)$ for a certain value of p and $\mathbf{P}_1$ is the projection of the first coordinate (in $\mathbb{R}^2$). In effect we can avoid the limit as our program always terminates after a finite number of steps; it is also independent of the initial state s_0. This way we have the optimisation problem:

$$\max \Phi(p) \ \text{ subject to } \ 0 \le p \le 1$$

The relationship between the probability or parameter p and the chances of winning (obtained using `octave` for various values of p) can be seen in the following diagram.



As one would expect the worst chance of winning is for $p = 0$, i.e. for the *sticking* strategy, and the best (of $\frac{2}{3}$) for $p = 1$, i.e. *switching*. This is a very simple case of a synthesis problem, but it illustrated the basic idea: We optimise a continuous parameter p in order to get a program with optimal performance (winning chance).

5 An Example: Swapping Variables

We consider another 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 $\mathcal{V}(\{0,1\} \times \{0,1\}) = \mathcal{V}(\{0,1\}) \otimes \mathcal{V}(\{0,1\}) = \mathbb{R}^2 \otimes \mathbb{R}^2 = \mathbb{R}^4$. 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 $x \oplus y = (x+y)$ mod 2.

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

$$\begin{array}{l}
\texttt{choose}\ \lambda_{1,1} : S_1\ \texttt{or}\ \lambda_{1,2} : S_2\ \texttt{or}\ \ldots\ \texttt{or}\ \lambda_{1,13} : S_{23}\ \texttt{ro;} \\
\texttt{choose}\ \lambda_{2,1} : S_1\ \texttt{or}\ \lambda_{2,2} : S_2\ \texttt{or}\ \ldots\ \texttt{or}\ \lambda_{2,13} : S_{13}\ \texttt{ro;} \\
\texttt{choose}\ \lambda_{3,1} : S_1\ \texttt{or}\ \lambda_{3,2} : S_2\ \texttt{or}\ \ldots\ \texttt{or}\ \lambda_{3,13} : S_{13}\ \texttt{ro;}
\end{array}$$

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

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

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 8×8 matrices on the tensor product space $\mathcal{V}(\{0,1\} \times \{0,1\} \times \{0,1\}) = \mathcal{V}(\{0,1\}) \otimes \mathcal{V}(\{0,1\}) \otimes \mathcal{V}(\{0,1\}) = \mathbb{R}^2 \otimes \mathbb{R}^2 \otimes \mathbb{R}^2 = \mathbb{R}^8$. In this eight dimensional space the dimensions corresponds to the (classical) states:

s_1	$\ldots$	$[x \mapsto 0, y \mapsto 0, z \mapsto 0]$	s_5	$\ldots$	$[x \mapsto 1, y \mapsto 0, z \mapsto 0]$
s_2	$\ldots$	$[x \mapsto 0, y \mapsto 0, z \mapsto 1]$	s_6	$\ldots$	$[x \mapsto 1, y \mapsto 0, z \mapsto 1]$
s_3	$\ldots$	$[x \mapsto 0, y \mapsto 1, z \mapsto 0]$	s_7	$\ldots$	$[x \mapsto 1, y \mapsto 1, z \mapsto 0]$
s_4	$\ldots$	$[x \mapsto 0, y \mapsto 1, z \mapsto 1]$	s_8	$\ldots$	$[x \mapsto 1, y \mapsto 1, z \mapsto 1]$

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.

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

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

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

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

$$\mathbf{P}_r = \mathrm{diag}(0,0,1,0,1,0,0,0,1,0,1,1,1) \quad \text{and} \quad \mathbf{P}_w = \mathrm{diag}(0,0,0,0,0,1,1,0,0,0,0,1,1)$$

then

$$R(\lambda_{ij}) = \|\sum_{i=1}^{3}(\lambda_{ij})_j \mathbf{P}_r\|_1 \;\text{ and }\; W(\lambda_{ij}) = \|\sum_{i=1}^{3}(\lambda_{ij})_j \mathbf{P}_w\|_1$$

The optimisation problem we thus have to solve is given by

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

Using the builtin non-linear optimisation in `octave` we get for certain initial λ_{ij}'s the some interesting results when we minimise the objective function $\Phi_{\rho\omega}$.

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

For randomly chosen initial values (which do not even fulfill the normalisation conditions for λ_{ij}) we get with the objective function Φ_{11} from

$$\begin{pmatrix} .70&.30&.72&.84&.51&.70&.76&.47&.63&.63&.93&.55&.68 \\ .74&.22&.37&.70&.67&.13&.93&.69&.30&.88&.03&.52&.80 \\ .59&.49&.01&.69&.22&.23&.10&.01&.10&.22&.03&.55&.11 \end{pmatrix}$$

after 9 iterations with `octave`:

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

This corresponds to the program:

$$[y := (y+x) \text{ mod } 2]^{10};\; [x := (x+y) \text{ mod } 2]^{8};\; [y := (y+x) \text{ mod } 2]^{10}$$

For Φ_{00} we may also get $[z := x]^6$; $[x := y]^2$; $[y := z]^5$. Sometimes however the optimisation does not work, we get stuck in a local minimum. This can usually be overcome by using stronger punishing terms for read and writes, e.g. $\Phi_{100,100}$. It might be interesting to observe that we obtain the desired program without reference to any *algebraic* properties of the xor operation.

6 Conclusions and Further Work

The idea of our approach is to consider a continuous manifold of program sketches which are parametrised using some (real-valued, probability) parameters. The objectives of the synthesis are encoded as a global optimisation problem in terms of the LOS semantics of these programs. The PAI framework can often be used in order to extract the relevant properties or information.

This setting allows us to address quantitative and qualitative problems within the same setting. Initial numerical experiments also indicate that although we allow probabilistic choices in the intermediate programs the ultimate solution are often purely deterministic programs.

The approach is in many ways orthogonal to the one in a recent paper by Chaudhuri et.al. [4]. We also consider a probabilistic semantics but while [4] is in essence based on Kozen's denotational semantics our approach utilises LOS which is more in the spirit of a compositional small-step collecting semantics. In the presentation here we restrict ourselves to a finite dimensional version, though it is possible, see [16], to drop this restriction: however the result is a Hilbert rather than a Banach space semantics (based on measure theoretic structures as in the case of Kozen's semantics). Our semantics is compositional because it exploits tensor products in order to describe individual effects (similar to SAN models). We also make use of Probabilistic Abstract Interpretation as introduced in [14] rather than Abstract Interpretation of a probabilistic semantics (as in the case of Monniaux's approach [21], and more recently in [7]). We also use PAI in a different way: in order to define or extract quantitative properties of programs rather than to construct (smooth) semantical approximations as in [4]. Our setting provides for continuous optimisation problems because of the parametrisation of the program semantics not its abstraction.

For the time being it is unclear if and in which way the presented approach would scale. Problems in this respect could be overcome by the compositional nature of our semantics, but additional techniques will be needed to accelerate the optimisation process which could include for example: guided search using ideas from differential (information) geometry or exploiting the structure, symmetry etc. of a particular problems via PAI in maybe a similar way as classical AI has been used in the synthesis of synchronisation in [26]; it could also be useful to combine this with a staged optimisation, addressing more detailed constraints after more global ones have been resolved, e.g. incorporating elements of [4]. It would also be desirable to combine our framework with formal specifications (logics) of program properties and synthesis objectives. Finally, the experimental tools used up to now should be developed into more efficient and user-friendly versions.

References

[1] Johan Agat (2000): *Transforming out timing leaks*. In: *Proceedings of POPL'00*, ACM, pp. 40–53, doi:10.1145/325694.325702.

[2] Adi Ben-Israel & Thomas Nall Eden Greville (2003): *Gereralized Inverses – Theory and Applications*, second edition. CMS Books in Mathematics, Springer Verlag, New York.

[3] Rastislav Bodik & Barabara Jobstmann (2013): *Algorithmic Program Synthesis: Introduction*. *Int. J. Softw. Tools Technol. Transfer* 15, pp. 397–411, doi:10.1007/s10009-013-0287-9.

[4] Swarat Chaudhuri, Martin Clochard & Armando Solar-Lezama (2014): *Bridging Boolean and Quantitative Synthesis Using Smoothed Proof Search*. In: *Proceedings of POPL '14*, ACM, pp. 207–220, doi:10.1145/2535838.2535859.

[5] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In: *Proceedings of POPL'77*, pp. 238–252, doi:10.1145/512950.512973.

[6] Patrick Cousot & Radhia Cousot (1979): *Systematic Design of Program Analysis Frameworks*. In: *Proceedings of POPL'79*, pp. 269–282, doi:10.1145/567752.567778.

[7] Patrick Cousot & Michaël Monerau (2012): *Probabilistic Abstract Interpretation*. In H. Seidel, editor: *Proceedings of ESOP12*, *Lecture Notes in Computer Science* 7211, Springer Verlag, pp. 166–190, doi:10.1007/978-3-642-28869-2_9.

[8] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2005): *Measuring the Confinement of Probabilistic Systems*. *Theoretical Computer Science* 340(1), pp. 3–56, doi:10.1016/j.tcs.2005.03.002.

[9] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2007): *A Systematic Approach to Probabilistic Pointer Analysis*. In: *Proceedings of APLAS'07*, *Lecture Notes in Computer Science* 4807, Springer Verlag, pp. 335–350, doi:10.1007/978-3-540-76637-7_23.

[10] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2008): *Quantifying Timing Leaks and Cost Optimisation*. In: *Proceedings of ICICS'08*, *Lecture Notes in Computer Science* 5308, Springer Verlag, pp. 81–96, doi:10.1007/978-3-540-88625-9_6.

[11] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2010): *Probabilistic Semantics and Analysis*. In: *Formal Methods for Quantitative Aspects of Programming Languages*, *Lecture Notes in Computer Science* 6155, Springer Verlag, pp. 1–42, doi:10.1007/978-3-642-13678-8_1.

[12] 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, doi:10.1007/s10207-010-0107-0.

[13] Alessandra Di Pierro, Pascal Sotin & Herbert Wiklicky (2008): *Relational Analysis and Precision via Probabilistic Abstract Interpretation*. In: *Proceedings of QAPL'08*, *ENTCS* 220(3), Elsevier, pp. 23–42, doi:10.1016/j.entcs.2008.11.017.

[14] Alessandra Di Pierro & Herbert Wiklicky (2000): *Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation*. In: *Proceedings of PPDP'00*, ACM, pp. 127–138, doi:10.1145/351268.351284.

[15] Alessandra Di Pierro & Herbert Wiklicky (2001): *Measuring the Precision of Abstract Interpretations*. In: *Proceedings of LOPSTR'00*, *Lecture Notes in Computer Science* 2042, Springer Verlag, Berlin – New York, pp. 147–164, doi:10.1007/3-540-44651-6.

[16] 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, doi:10.1007/978-3-319-03542-0_18.

[17] Alessandra Di Pierro & Herbert Wiklicky (2014): *Probabilistic Analysis of Programs: A Weak Limit Approach*. In: *FOPARA'13*, Lecture Notes in Computer Science, Springer Verlag, p. (to appear).

[18] John W. Eaton, David Bateman & Soren Hauberg (2007): *GNU Octave – A high-level interactive language for numerical computations*, 3rd edition.

[19] 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, doi:10.1007/3-540-68697-5_9.

[20] Dexter Kozen (1981): *Semantics of Probabilistic Programs*. *J. Comput. Syst. Sci.* 22(3), pp. 328–350, doi:10.1016/0022-0000(81)90036-2.

[21] David Monniaux (2000): *Abstract Interpretation of Probabilistic Semantics*. In: *SAS'00*, *Lecture Notes in Computer Science* 1824, Springer Verlag, pp. 322–339, doi:10.1007/978-3-540-45099-3_17.

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

[23] Armando Solar Lezama (2008): *Program Synthesis By Sketching*. Ph.D. thesis, University of California, Berkeley.

[24] Armando Solar-Lezama (2013): *Program Sketching*. *Int. J. Softw. Tools Technol. Transfer* 15, pp. 475–495, doi:10.1007/s10009-012-0249-7.

[25] David Stirzaker (1999): *Probability and Random Variables – A Beginners Guide*. Cambridge University Press, doi:10.1017/CBO9780511813627.

[26] Martin Vechev, Eran Yahav & Greta Yorsh (2010): *Abstraction-guided Synthesis of Synchronization*. In: *Proceedings of POPL '10*, ACM, pp. 327–338, doi:10.1145/1706299.1706338.