Probabilistic data flow analysis: a linear equational approach

Alessandra Di Pierro
University of Verona
Verona, Italy
alessandra.dipierro@univr.it

Herbert Wiklicky
Imperial College London
London, UK
herbert@doc.ic.ac.uk

Speculative optimisation relies on the estimation of the probabilities that certain properties of the control flow are fulfilled. Concrete or estimated branch probabilities can be used for searching and constructing advantageous speculative and bookkeeping transformations. We present a probabilistic extension of the classical equational approach to data-flow analysis that can be used to this purpose. More precisely, we show how the probabilistic information introduced in a control flow graph by branch prediction can be used to extract a system of linear equations from a program and present a method for calculating correct (numerical) solutions.

1 Introduction

In the last two decades probabilistic aspects of software have become a particularly popular subject of research. The reason for this is arguably in *economical* and *resource conscious* questions involving modern computer systems. While program verification and analysis originally focused on qualitative issues, e.g. whether code is correct or if compiler optimisations are valid, the focus is now more often also on the costs of operations.

Speculative optimisation is part of this trend; it plays an important role in the design of modern compiler and run time architectures. A speculative approach has been adopted in various models where cost optimisation claims for a more optimistic interpretation of the results of a program analysis. It is in fact often the case that possible optimisations are discarded because the analysis cannot guarantee their correctness. The alternative to this sometimes overly pessimistic analysis is to speculatively assume in those cases that optimisations are correct and then eventually backtrack and redo the computation if at a later check the assumption turns out to be incorrect.

Speculative optimisation relies on the optimal estimation of the probabilities that certain properties of the control flow are fulfilled. This is different from the classical (pessimistic) thinking where one aims in providing bounds for what can happen during execution [8].

A number of frameworks and tools to analyse systems's probabilistic aspects have been developed, which can be seen as probabilistic versions of classical techniques such as model checking and abstract interpretation. To provide a basis for such analysis various semantical model involving discrete and continuous time and also non-deterministic aspects have been developed (e.g. DTMCs, CTMCs, MDPs, process algebraic approaches etc.). There also exist some powerful tools which implement these methods, e.g. PRISM [14], just to name one.

Our own contribution in this area has been a probabilistic version of the abstract interpretation framework [6], called Probabilistic Abstract Interpretation (PAI) [12, 9]. This analysis framework, in its basic form, is concerned with purely probabilistic, discrete time models. Its purpose is to give optimal estimates of the probability that a certain property holds rather than providing probabilities bounds. As such, we think it is well suited as a base for speculative optimisation.

To appear in EPTCS.

S	::=	`skip`	S	::=	$[\texttt{skip}]^\ell$
	|	x `:=` $e(x_1,\ldots,x_n)$		|	$[x$ `:=` $e(x_1,\ldots,x_n)]^\ell$
	|	x `?=` ρ		|	$[x$ `?=` $\rho]^\ell$
	|	S_1 `;` S_2		|	S_1 `;` S_2
	|	`if` b `then` S_1 `else` S_2 `fi`		|	`if` $[b]^\ell$ `then` S_1 `else` S_2 `fi`
	|	`while` b `do` S `od`		|	`while` $[b]^\ell$ `do` S `od`

Table 1: The syntax

The aim of this paper is to provide a framework for a probabilistic analysis of programs in the style of a classical data flow approach [18, 1]. In particular, we are interested in a formal basis for (non-static) branch prediction. The analysis technique we present consists of three phases: (i) abstract branch prediction, (ii) specification of the actual data-flow equations based on the estimates of the branch probabilities, and (iii) finding solutions. We will use vector space structures to specify the properties and analysis of a program. This allows for the construction of solutions via numerical (linear algebraic) methods as opposed to the lattice-theoretic fixed-point construction of the classical analysis.

2 A Probabilistic Language

2.1 Syntax and Operational Semantics

We use as a reference language a simple imperative language whose syntax is given in Table 1. Following the approach in [18] we extend this syntax with unique program labels $\ell \in \mathbf{Lab}$ in order to be able to refer to certain program points during the analysis.

The dummy statement `skip` has no computational effect. For the arithmetic *expressions* $e(x_1,\ldots,x_n)$ on the right hand side (RHS) of the assignment as well as for the tests $b = b(x_1,\ldots,x_n)$ in `if` and `while` statements, we leave the details of the syntax open as they are irrelevant for our treatment. The RHS of a random assignment x `?=` ρ is a distribution ρ over some set of values with the meaning that x is assigned one of the possible constant values c with probability $\rho(c)$.

An operational semantics in the SOS style is given in Table 2.2. This defines a probabilistic transition relation on configurations in $\mathbf{Conf} = \mathbf{Stmt} \times \mathbf{State}$ with **Stmt** the set of all statements in our language together with `stop` which indicates termination and $\mathbf{State} = \mathbf{Var} \to \mathbf{Value}$. The details of the semantics of arithmetic and boolean expressions $[\![a]\!] = \mathscr{E}(a)$ and $[\![b]\!] = \mathscr{E}(b)$ respectively are again left open in our treatment here and can be found in [10].

2.2 Computational States

In any concrete computation or execution – even when it is involving probabilistic elements – the computational situation is uniquely defined by a mapping $s : \mathbf{Var} \to \mathbf{Value}$ to which we refer to as a *classical state*. Every variable in **Var** has a unique value in **Value** possibly including $\perp \in \mathbf{Value}$ to indicate undefinedness. We denote by **State** the set of all classical states.

In order to keep the mathematical treatment simple we will assume here that every variable can take values in a finite set **Value**. These sets can be nevertheless quite large and cover, for example, all finitely representable integers on a given machine.

R0	$\langle \texttt{stop}, s\rangle \Rightarrow_1 \langle \texttt{stop}, s\rangle$	**R4**$_1$	$\dfrac{\langle S_1, s\rangle \Rightarrow_p \langle S_1', s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow_p \langle S_1'; S_2, s'\rangle}$
R1	$\langle \texttt{skip}, s\rangle \Rightarrow_1 \langle \texttt{stop}, s\rangle$		
R2	$\langle v \texttt{ := } e, s\rangle \Rightarrow_1 \langle \texttt{stop}, s[v \mapsto \mathscr{E}(e)s]\rangle$	**R4**$_2$	$\dfrac{\langle S_1, s\rangle \Rightarrow_p \langle \texttt{stop}, s'\rangle}{\langle S_1; S_2, s\rangle \Rightarrow_p \langle S_2, s'\rangle}$
R3	$\langle v \texttt{ ?= } \rho, s\rangle \Rightarrow_{\rho(r)} \langle \texttt{stop}, s[v \mapsto r]\rangle$		
R5$_1$	$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}, s\rangle \Rightarrow_1 \langle S_1, s\rangle$		if $\mathscr{E}(b)s = \textbf{true}$
R5$_2$	$\langle \texttt{if } b \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}, s\rangle \Rightarrow_1 \langle S_2, s\rangle$		if $\mathscr{E}(b)s = \textbf{false}$
R6$_1$	$\langle \texttt{while } b \texttt{ do } S \texttt{ od}, s\rangle \Rightarrow_1 \langle S; \texttt{ while } b \texttt{ do } S \texttt{ od}, s\rangle$		if $\mathscr{E}(b)s = \textbf{true}$
R6$_2$	$\langle \texttt{while } b \texttt{ do } S \texttt{ od}, s\rangle \Rightarrow_1 \langle \texttt{stop}, s\rangle$		if $\mathscr{E}(b)s = \textbf{false}$

Table 2: The rules of the SOS semantics

For a finite set X we denote by $\mathscr{P}(X)$ the power-set of X and by $\mathscr{V}(X)$ the free vector space over X, i.e. the set of formal linear combinations of elements in X. We represent vectors via their coordinates $(x_1,\ldots,x_n)$ as rows, i.e. elements in $\mathbb{R}^{|X|}$ with $|X|$ denoting the cardinality of X and use post-multiplication with matrices representing linear maps, i.e. $\mathbf{A}(x) = x \cdot \mathbf{A}$. The set $\mathbf{Dist}(X)$ of distributions on X – i.e. $\rho : X \to [0,1]$ and $\sum_i \rho(x_i) = 1$ – clearly correspond to a sub-set of $\mathscr{V}(X)$. We will also use a tuple notation for distributions: $\rho = \{\langle a, \frac{1}{2}\rangle, \langle b, \frac{1}{4}\rangle, \langle c, \frac{1}{4}\rangle\}$ will denote a distribution where a has probability $\rho(a) = \frac{1}{2}$ and b and c both have probability $\frac{1}{4}$. For uniform distributions we will simply specify the underlying set, e.g. $\{a,b,c\}$ instead of $\langle a, \frac{1}{3}\rangle, \langle b, \frac{1}{3}\rangle, \langle c, \frac{1}{3}\rangle\}$.

The tensor product is an essential element of the description of probabilistic states. The tensor product[1] 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. Similarily for matrices. The tensor product of two vector spaces $\mathscr{V} \otimes \mathscr{W}$ 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 $\mathscr{V}$ and $\mathscr{W}$, respectively. For further details we refer e.g to [19, Chap. 14].

Importantly, the isomorphism $\mathscr{V}(X \times Y) = \mathscr{V}(X) \otimes \mathscr{V}(Y)$ allows us to identify set of all distributions on the cartesian product of two sets with the tensor product of the spaces of distributions on X and Y.

We define a *probabilistic state* σ as any probability distribution over classical states, i.e. $\sigma \in \mathbf{Dist}(\mathbf{State})$. This can also be seen as $\sigma \in \mathscr{V}(\mathbf{State}) = \mathscr{V}(\mathbf{Var} \to \mathbf{Value}) = \mathscr{V}(\mathbf{Value}^{|\mathbf{Var}|}) = \mathscr{V}(\mathbf{Value})^{\otimes v}$ the v-vold tensor product of $\mathscr{V}(\mathbf{Value})$ with $v = |\mathbf{Var}|$.

In our setting we represent (semantical) functions and predicates or tests as linear operators on the probabilistic state space, i.e. as matrices. For any function $f : X \mapsto Y$ we define a linear representation $|X| \times |Y|$ matrix by:

$$(\mathbf{F}_f)_{ij} = (\mathbf{F}(f))_{ij} = (\mathbf{F})_{ij} = \begin{cases} 1 & \text{if } f(x_i) = y_j \\ 0 & \text{otherwise.} \end{cases}$$

where we assume some fixed enumeration on both X and Y. For an equivalence relation on X we can also represent the function which maps every element in X to its equivalence class $c : x \mapsto [x]$ in this way. Such a *classification matrix* contains in every row exactly one non-zero entry 1. Classification matrices (modulo reordering of indices) are in a one-to-one correspondence with the equivalence relations on a set X and we will use them to define probabilistic abstractions for our analysis (cf. Section 4.2). A predicate $p : X \to \{\textbf{true}, \textbf{false}\}$ is represented by a diagonal $|X| \times |X|$ matrix:

$$(\mathbf{P}_p)_{ij} = (\mathbf{P}(p))_{ij} = (\mathbf{P})_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } p(x_i) = \textbf{true} \\ 0 & \text{otherwise.} \end{cases}$$

[1] More precisely, the Kronecker product – the coordinate based version of the abstract concept of a tensor product.

2.3 Probabilistic Abstraction

The analysis technique we present in this paper will make use of a particular notion of abstraction of the state space (given as a vector space) which is formalised in terms of Moore-Penrose pseudo-inverse [19].

Definition 1 *Let $\mathscr{C}$ and $\mathscr{D}$ be two finite dimensional vector spaces, and let $\mathbf{A} : \mathscr{C} \to \mathscr{D}$ be a linear map between them. The linear map $\mathbf{A}^\dagger = \mathbf{G} : \mathscr{D} \to \mathscr{C}$ is the* Moore-Penrose pseudo-inverse *of $\mathbf{A}$ iff*

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

where $\mathbf{P}_A$ and $\mathbf{P}_G$ denote orthogonal projections onto the ranges of $\mathbf{A}$ and $\mathbf{G}$.

An operator or matrix is an *orthogonal projection* if $\mathbf{P}^* = \mathbf{P}^2 = \mathbf{P}$ where $.^*$ denotes the *adjoint* which for real matrices correspond simply to the transpose matrix $\mathbf{P}^* = \mathbf{P}^t$ [19, Ch 10].

For invertible matrices the Moore-Penrose pseudo-inverse is the same as the inverse. A special example is the *forgetful abstraction* $\mathbf{A}_f$ which corresponds to a map $f : X \to \{*\}$ which maps all elements of X onto a single abstract one. It is represented by a $|X| \times 1$ matrix containing only 1, and its Moore-Penrose pseudo-inverse is given by $1 \times |X|$ matrix with all entries $\frac{1}{|X|}$.

The Moore-Penrose pseudo-inverse allows us to construct the closest, in a least square sense (see for example [5, 3]), approximation $\mathbf{F}^\# : \mathscr{D} \to \mathscr{D}$ of a concrete linear operator $\mathbf{F} : \mathscr{C} \to \mathscr{C}$ for a given abstraction $\mathbf{A} : \mathscr{C} \to \mathscr{D}$ as

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

This notion of probabilistic abstraction is central in the Probabilistic Abstract Interpretation (PAI) framework. For further details we refer to e.g. [10]. As we will use this notion later for abstracting branching probabilities, it is important here to point out the guarantees that such abstractions are able to provide. In fact, these are not related to any correctness notion in the classical sense. The theory of the least-square approximation [7, 3] tells us that if $\mathscr{C}$ and $\mathscr{D}$ be two finite dimensional vector spaces, $\mathbf{A} : \mathscr{C} \mapsto \mathscr{D}$ a linear map between them, and $\mathbf{A}^\dagger = \mathbf{G} : \mathscr{D} \mapsto \mathscr{C}$ its Moore-Penrose pseudo-inverse, then the vector $x_0 = y \cdot \mathbf{G}$ is the one minimising the distance between $x \cdot \mathbf{A}$, for any vector x in $\mathscr{C}$, and y, i.e.

$$\inf_{x \in \mathscr{C}} \|x \cdot \mathbf{A} - y\| = \|x_0 \cdot \mathbf{A} - y\|.$$

This guarantees that our probabilistic abstractions correspond to the **closest** approximations in a metric sense of the concrete situations, as they are constructed using the Moore-Penrose pseudo-inverse.

3 Data-Flow Analysis

Data-flow analysis is based on a statically determined flow relation. This is defined in terms of two auxiliary operations, namely $\mathit{init} : \mathbf{Stmt} \to \mathbf{Lab}$ and $\mathit{final} : \mathbf{Stmt} \to \mathscr{P}(\mathbf{Lab})$, defined as follows:

$\mathit{init}([\texttt{skip}]^\ell) = \ell$	$\mathit{final}([\texttt{skip}]^\ell) = \{\ell\}$
$\mathit{init}([v \texttt{ := } e]^\ell) = \ell$	$\mathit{final}([v \texttt{ := } e]^\ell) = \{\ell\}$
$\mathit{init}([v \texttt{ ?= } e]^\ell) = \ell$	$\mathit{final}([v \texttt{ ?= } e]^\ell) = \{\ell\}$
$\mathit{init}(S_1; S_2) = \mathit{init}(S_1)$	$\mathit{final}(S_1; S_2) = \mathit{final}(S_2)$
$\mathit{init}(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) = \ell$	$\mathit{final}(\texttt{if } [b]^\ell \texttt{ then } S_1 \texttt{ else } S_2 \texttt{ fi}) = \mathit{final}(S_1) \cup \mathit{final}(S_2)$
$\mathit{init}(\texttt{while } [b]^\ell \texttt{ do } S \texttt{ od}) = \ell$	$\mathit{final}(\texttt{while } [b]^\ell \texttt{ do } S \texttt{ od}) = \{\ell\}.$

The control flow $\mathscr{F}(S)$ in $S \in \mathbf{Stmt}$ is defined via the function $flow : \mathbf{Stmt} \to \mathscr{P}(\mathbf{Lab} \times \mathbf{Lab})$:

$$flow([\texttt{skip}]^\ell) = flow([v\ \texttt{:=}\ e]^\ell) = flow([v\ \texttt{?=}\ e]^\ell) = \emptyset$$
$$flow(S_1;S_2) = flow(S_1) \cup flow(S_2) \cup \{(\ell, init(S_2)) \mid \ell \in final(S_1)\}$$
$$flow(\texttt{if}\ [b]^\ell\ \texttt{then}\ S_1\ \texttt{else}\ S_2\ \texttt{fi}) = flow(S_1) \cup flow(S_2) \cup \{(\ell, \underline{init(S_1)}), (\ell, init(S_2))\}$$
$$flow(\texttt{while}\ [b]^\ell\ \texttt{do}\ S\ \texttt{od}) = flow(S) \cup \{(\ell, \underline{init(S)})\} \cup \{(\ell', \ell) \mid \ell' \in final(S)\}$$

The definition of flow only records that a certain control flow step is possible. For tests b in conditionals and loops we indicate the branch corresponding to the case when the test is successful by underlining it. We identify a statement S with the block $[S]^\ell$ that contains it and with the (unique) label ℓ associated to the block. We will denote by $\mathbf{Block} = \mathbf{Block}(P)$ the set of all the blocks occurring in P, and use indistinctly **Block** and **Lab** to refer to blocks.

3.1 Monotone Framework

The classical data-flow analysis is made up of two components: a "local" part which describes how the information representing the analysis changes when execution passes through a given block/label, and a "global" collection part which describes how information is accumulated when a number of different control flow paths (executions) come together.

This is formalised in a general scheme, called Monotone Framework in [18, Section 2.3], where a data-flow analysis is defined via a number of equations over the lattice L modelling the property to be analysed. For every program label ℓ we have two equations: one describing the generalised 'entry' in terms of the generalised 'exit' of the block in question, and the other describing 'exit' in terms of 'entry' – for forward analysis we have $\circ$=entry and $\bullet$=exit, for a backward analysis the situation is reversed.

$$\begin{aligned} Analysis_\bullet(\ell) &= f_\ell(Analysis_\circ(\ell)) \\ Analysis_\circ(\ell) &= \begin{cases} \iota, \text{if } \ell \in E \\ \bigsqcup\{Analysis_\bullet(\ell') \mid (\ell', \ell) \in F\}, \text{otherwise} \end{cases} \end{aligned}$$

For the typical classical analyses, such as Live Variable *LV* and Reaching Definition *RD*, the property lattice L is often the power-set of some underlying set (like **Var** as in the case of the LV analysis). For a may-analysis the collecting operation $\sqcup$ of L is represented by set union $\cup$ and for must-analysis it is the intersection operation $\cap$. The flow relation F can be the forward or backward flow. ι specifies the initial or final analysis information on "extreme" labels in E, where E is $\{init(S_\star)\}$ or $final(S_\star)$, and f_ℓ is the transfer function associated with $B^\ell \in \mathbf{Block}(S)$ [18, Section 2.3].

3.2 Live Variable Analysis

We will illustrate the basic principles of the equational approach to data flow analysis by considering Live Variable analysis (*LV*) following the presentation in [18, Section 2.1]. The problem is to identify at any program point those variables which are *live*, i.e. which may later be used in an assignment or test.

There are two phases of classical *LV* analysis: (i) formulation of data-flow equations as set equations (or more generally over a property lattice L), (ii) finding or constructing solutions to these equations, for example, via a fixed-point construction. In the classical analysis we associate to every program point or label ℓ – to be precise the entry and the exit of each label – the information which describes (a super-set of) those variables which are alive at this program point.

Based on the auxiliary functions $gen_{\mathsf{LV}} : \mathbf{Block} \to \mathscr{P}(\mathbf{Var})$ and $kill_{\mathsf{LV}} : \mathbf{Block} \to \mathscr{P}(\mathbf{Var})$ which only depend on the syntax of the local block $[B]^\ell$ and are defined as

$$\begin{array}{rclcrcl} kill_{\mathsf{LV}}([x := a]^\ell) & = & \{x\} & & gen_{\mathsf{LV}}([x := a]^\ell) & = & FV(a) \\ kill_{\mathsf{LV}}([x ?= \rho]^\ell) & = & \{x\} & & gen_{\mathsf{LV}}([x ?= \rho]^\ell) & = & \emptyset \\ kill_{\mathsf{LV}}([\mathtt{skip}]^\ell) & = & \emptyset & & gen_{\mathsf{LV}}([\mathtt{skip}]^\ell) & = & \emptyset \\ kill_{\mathsf{LV}}([b]^\ell) & = & \emptyset & & gen_{\mathsf{LV}}([b]^\ell) & = & FV(b) \end{array}$$

we can define the transfer functions for the *LV* analysis $f_\ell^{LV} : \mathscr{P}(\mathbf{Var}_\star) \to \mathscr{P}(\mathbf{Var}_\star)$ by

$$f_\ell^{LV}(X) = X \setminus kill_{\mathsf{LV}}([B]^\ell) \cup gen_{\mathsf{LV}}([B]^\ell)$$

This allows us to define equations over the property space $L = \mathscr{P}(\mathbf{Var})$, i.e. set equations, which associate to every label entry and exit the analysis information $\mathsf{LV}_{entry} : \mathbf{Lab} \to \mathscr{P}(\mathbf{Var})$ and $\mathsf{LV}_{exit} : \mathbf{Lab} \to \mathscr{P}(\mathbf{Var})$. These set equations are of the general form for a backward may analysis:

$$\begin{array}{rcl} \mathsf{LV}_{entry}(\ell) & = & f_\ell^{LV}(\mathsf{LV}_{exit}(\ell)) \\ \mathsf{LV}_{exit}(\ell) & = & \bigcup_{(\ell,\ell') \in flow} \mathsf{LV}_{entry}(\ell') \end{array}$$

At the beginning of the analysis (i.e. for final labels, as this is a backward analysis) we set $\mathsf{LV}_{exit}(\ell) = \emptyset$.

Example 1 *Consider the following program:*

$$[x ?= \{0,1\}]^1;\ [y ?= \{0,1,2,3\}]^2;\ [x := x+y \bmod 4]^3;$$
$$\mathtt{if}\ [x > 2]^4\ \mathtt{then}\ [z := x]^5\ \mathtt{else}\ [z := y]^6\ \mathtt{fi}$$

Although the program is probabilistic we still can perform a classical analysis by considering non-zero probabilities simply as possibilities. The flow is given by $\{(1,2),(2,3),(3,4),(4,\underline{5}),(4,6)\}$.

With the auxiliary functions $kill_{\mathsf{LV}}$ *and* gen_{LV} *we can now specify the data-flow equations:*

	$gen_{\mathsf{LV}}(\ell)$	$kill_{\mathsf{LV}}(\ell)$
1	$\emptyset$	$\{x\}$
2	$\emptyset$	$\{y\}$
3	$\{x,y\}$	$\{x\}$
4	$\{x\}$	$\emptyset$
5	$\{x\}$	$\{z\}$
6	$\{y\}$	$\{z\}$

$$\begin{array}{rclcrcl} \mathsf{LV}_{entry}(1) & = & \mathsf{LV}_{exit}(1) \setminus \{x\} & & \mathsf{LV}_{exit}(1) & = & \mathsf{LV}_{entry}(2) \\ \mathsf{LV}_{entry}(2) & = & \mathsf{LV}_{exit}(2) \setminus \{y\} & & \mathsf{LV}_{exit}(2) & = & \mathsf{LV}_{entry}(3) \\ \mathsf{LV}_{entry}(3) & = & \mathsf{LV}_{exit}(3) \setminus \{x\} \cup \{x,y\} & & \mathsf{LV}_{exit}(3) & = & \mathsf{LV}_{entry}(4) \\ \mathsf{LV}_{entry}(4) & = & \mathsf{LV}_{exit}(4) \cup \{x\} & & \mathsf{LV}_{exit}(4) & = & \mathsf{LV}_{entry}(5) \cup \mathsf{LV}_{entry}(6) \\ \mathsf{LV}_{entry}(5) & = & \mathsf{LV}_{exit}(5) \setminus \{z\} \cup \{x\} & & \mathsf{LV}_{exit}(5) & = & \emptyset \\ \mathsf{LV}_{entry}(6) & = & \mathsf{LV}_{exit}(6) \setminus \{z\} \cup \{y\} & & \mathsf{LV}_{exit}(6) & = & \emptyset \end{array}$$

Then the classical LV analysis of our program gives the solutions:

$$\begin{array}{rclcrcl} \mathsf{LV}_{entry}(1) & = & \emptyset & & \mathsf{LV}_{exit}(1) & = & \{x\} \\ \mathsf{LV}_{entry}(2) & = & \{x\} & & \mathsf{LV}_{exit}(2) & = & \{x,y\} \\ \mathsf{LV}_{entry}(3) & = & \{x,y\} & & \mathsf{LV}_{exit}(3) & = & \{x,y\} \\ \mathsf{LV}_{entry}(4) & = & \{x,y\} & & \mathsf{LV}_{exit}(4) & = & \{x,y\} \\ \mathsf{LV}_{entry}(5) & = & \{x\} & & \mathsf{LV}_{exit}(5) & = & \emptyset \\ \mathsf{LV}_{entry}(6) & = & \{y\} & & \mathsf{LV}_{exit}(6) & = & \emptyset. \end{array}$$

4 The Probabilistic Setting

In order to specify a probabilistic data flow analysis using the analogue of the classical equational approach (as presented in the previous sections), we have to define the main ingredients of the analysis in a probabilistic setting namely a vector space as property space (replacing the property lattice L), a linear operator representing the transfer functions f_ℓ, and a method for the information collection (in place of the $\bigsqcup$ operation of the classical monotone framework). Moreover, as we will work with probabilistic states, the second point implies that the control-flow graph will be labelled by some probability information.

As a **property space** we consider distributions $\mathbf{Dist}(L) \subseteq \mathscr{V}(L)$ over a set L, e.g. the corresponding classical property space. For a relational analysis, where the classical property lattice corresponds to $L = L_1 \times L_2$ (cf [11]), the probabilistic property space will be the tensor product $\mathscr{V}(L_1) \otimes \mathscr{V}(L_2)$; this allows us to represent properties via joint probabilities which are able to express the dependency or correlation between states.

We can define probabilistic **transfer functions** by using the linear representation of the classical f_ℓ, i.e. a matrix $\mathbf{F}_\ell = \mathbf{F}_{f_\ell}$ as introduced above in Section 2.2. In general, we will define a probabilistic transfer function by means of an appropriate abstraction of the concrete semantics $[\![[B]^\ell]\!]$ of a given block $[B]^\ell$ according to PAI, i.e. $\mathbf{F}_\ell = \mathbf{A}^\dagger [\![[B]^\ell]\!] \mathbf{A}$ for the relevant abstraction matrix $\mathbf{A}$.

In the classical analysis we treat tests b non-deterministically, to avoid problems with the potential undecidability of predicates. Moreover, we take everything which is possible i.e. the collection of what can happen along the different execution paths, e.g. the two branches of an `if` statement. In the probabilistic setting we **collect information** by means of weighted sums, where the 'weights' are the probabilities associated to each branch. These probabilities come from an estimation of the (concrete or abstract) branch probabilities and are propagated along the control flow graph representing the **flow relation**.

4.1 Control Flow Probabilities

If we execute a program in classical states s which have been chosen randomly according to some probability distribution ρ then this also induces a probability distribution on the possible control flow steps.

Definition 2 *Given a program S_ℓ with init$(S_\ell) = \ell$ and a probability distribution ρ on* **State***, the probability $p_{\ell,\ell'}(\rho)$ that the control is flowing from ℓ to ℓ' is defined as:*

$$p_{\ell,\ell'}(\rho) = \sum_s \left\{ p \cdot \rho(s) \mid \exists s' \text{ s.t. } \langle S_\ell, s\rangle \Rightarrow_p \langle S_{\ell'}, s'\rangle \right\}.$$

In other words, if we provide with a certain probability $\rho(s)$ a concrete execution environment or classical state s for a program S_ℓ, then the control flow probability $p_{\ell,\ell'}(\rho)$ is the probability that we end up with a configuration $\langle S_{\ell'}, \ldots\rangle$ for whatever state in the successor configuration.

Example 2 *Consider the program:* $[x \texttt{ ?= } \{0,1\}]^1$; `if` $[x > 0]^2$ `then` $[\texttt{skip}]^3$ `else` $[x \texttt{ := } 0]^4$ `fi`. *We can have two possible states at label 2, namely $s_0 = [x \mapsto 0]$ and $s_1 = [x \mapsto 1]$. After the first statement has been executed in one of two possible ways (with any intial state s):*

$$\begin{aligned} & \left\langle [x \texttt{ ?= } \{0,1\}]^1; \texttt{ if } [x>0]^2 \texttt{ then } [\texttt{skip}]^3 \texttt{ else } [x \texttt{ := } 0]^4 \texttt{ fi}, s\right\rangle \Rightarrow_{\frac{1}{2}} \\ & \quad \Rightarrow_{\frac{1}{2}} \left\langle \texttt{if } [x>0]^2 \texttt{ then } [\texttt{skip}]^3 \texttt{ else } [x \texttt{ := } 0]^4 \texttt{ fi}, s_0\right\rangle \\ \textit{or} \quad & \left\langle [x \texttt{ ?= } \{0,1\}]^1; \texttt{ if } [x>0]^2 \texttt{ then } [\texttt{skip}]^3 \texttt{ else } [x \texttt{ := } 0]^4 \texttt{ fi}, s\right\rangle \Rightarrow_{\frac{1}{2}} \\ & \quad \Rightarrow_{\frac{1}{2}} \left\langle \texttt{if } [x>0]^2 \texttt{ then } [\texttt{skip}]^3 \texttt{ else } [x \texttt{ := } 0]^4 \texttt{ fi}, s_1\right\rangle \end{aligned}$$

the distribution over states is obviously $\rho = \{\langle s_0, \frac{1}{2}\rangle \langle s_1, \frac{1}{2}\rangle\}$. *However, in each execution path we have at any moment a definite value for x (the distribution* ρ *describes a property of the set of all executions, not of one execution alone).*

The branch probability in this case (independently of the state s and of any distribution ρ*) is simply* $p_{1,2}(\rho) = 1$ *because, although there are two possible execution steps, the successor configurations are 'coincidently' equipped with the same program* $\texttt{if}\ [x > 0]^2\ \texttt{then}\ [\texttt{skip}]^3\ \texttt{else}\ [x := 0]^4\ \texttt{fi}$.

The successive control steps from label 2 *to* 3 *and* 4*, respectively, both occur with probability* 1 *as in each state* s_0 *and* s_1 *the value of x is a definite one.*

$$\begin{array}{rll} & \langle \texttt{if}\ [x > 0]^2\ \texttt{then}\ [\texttt{skip}]^3\ \texttt{else}\ [x := 0]^4\ \texttt{fi}, s_0\rangle & \Rightarrow_1 \quad \langle [x := 0]^4, s_0\rangle \\ \textit{and} & \langle \texttt{if}\ [x > 0]^2\ \texttt{then}\ [\texttt{skip}]^3\ \texttt{else}\ [x := 0]^4\ \texttt{fi}, s_1\rangle & \Rightarrow_1 \quad \langle [\texttt{skip}]^3, s_1\rangle \end{array}$$

Thus the branch probabilities with $\rho = \{\langle s_0, \frac{1}{2}\rangle, \langle s_1, \frac{1}{2}\rangle\}$ *are* $p_{2,3}(\rho) = \frac{1}{2}$ *and* $p_{2,4}(\rho) = \frac{1}{2}$. *In general for any* $\rho = \{\langle s_0, p_0\rangle, \langle s_1, p_1\rangle\}$ *we have* $p_{2,3}(\rho) = p_1$ *and* $p_{2,4}(\rho) = p_0$ *despite the fact that the transitions are deterministic. It is the randomness in the probabilistic state that determines in this case the branch probabilities.*

For all blocks in a control flow graph – except for the tests b – there is always only one next statement $S_{\ell'}$ so that the branch probability $p_{\ell,\ell'}(\rho)$ is always 1 for all ρ. For tests b in `if` and `while` statements we have only two different successor statements, one corresponding to the case where $[b]^\ell$ evaluates to **true** and one for **false**. As the corresponding probabilities must sum up to 1 we only need to specify the first case which we denote by $p_\ell(\rho)$.

The probability distributions over states at every execution point are thus critical for the analysis as they determine the branch probabilities for tests, and we need to provide them. The problem is, of course that analysing these probabilities is nearly as expensive as analysing the concrete computation or program executions. It is therefore reasonable to investigate abstract branch probabilities, based on classes of states, or abstract states. It is always possible to lift concrete distributions to ones over (equivalence) classes.

Definition 3 *Given a probability distribution* ρ *on* **State** *and an equivalence relation* $\sim$ *on states then we denote by* $\rho^\# = \rho^\#_\sim$ *the probability distribution on the set of equivalence classes* $\mathbf{State}^\# = \mathbf{State}/\sim$ *defined by*

$$\rho^\#([s]_\sim) = \sum_{s' \in [s]_\sim} \rho(s')$$

where $[s]_\sim$ *denotes the equivalence classes of s wrt* $\sim$.

4.2 Estimating Abstract Branch Probabilities

In order to determine concrete or abstract branch probabilities we need to investigate – as we have seen in Example 2 – the interplay between distribution over states and the test $[b]^\ell$ we are interested in. We need for this the linear representation $\mathbf{P}_b$ of the test predicate b as defined in Section 2.2, which for a given distribution over states determines a sub-distribution of those states that lead into one of the two branches by filtering out those states where this happens.

Example 3 *Consider the simple program* $\texttt{if}\ [x >= 1]^1\ \texttt{then}\ [x := x - 1]^2\ \texttt{else}\ [\texttt{skip}]^3\ \texttt{fi}$ *and assume that x has values in* $\{0,1,2\}$ *(enumerated in the obvious way). Then the test* $b = (x >= 1)$ *is represented*

by the projection matrix:

$$\mathbf{P}(x >= 1) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \text{ and } \mathbf{P}(x >= 1)^{\perp} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \mathbf{P}(x = 0)$$

For any given concrete probability distribution over states $\rho = \{\langle 0, p_0\rangle, \langle 1, p_1\rangle, \langle 2, p_2\rangle\} = (p_0, p_1, p_2)$ *we can easily compute the probabilities to go from label* 1 *to label* 2 *as* $\rho\mathbf{P}(x >= 1) = (0, p_1, p_2)$ *and thus*

$$p_{1,2}(\rho) = \|\rho \cdot \mathbf{P}(x >= 1)\|_1 = p_1 + p_2,$$

where $\|.\|_1$ *is the 1-norm of vectors, i.e.* $\|(x_i)_i\| = \sum_i |x_i|$*, which we use here to aggregate the total probabilities. Similarly, for the else branch, with* $\mathbf{P}^{\perp} = \mathbf{I} - \mathbf{P}$*:*

$$p_{1,3}(\rho) = \|\rho \cdot \mathbf{P}^{\perp}(x >= 1)\|_1 = p_0.$$

In general, the branching behaviour at a test b is described by the projection operator $\mathbf{P}(b)$ and its complement $\mathbf{P}^{\perp}(b) = \mathbf{P}(\neg b)$. For a branching point $[b]^{\ell}$ with $(\ell, \underline{\ell'}), (\ell, \ell'') \in \mathit{flow}$, we denote $\mathbf{P}(b)$ by $\mathbf{P}(\ell, \ell')$ and $\mathbf{P}(\neg b) = \mathbf{P}(b)^{\perp}$ by $\mathbf{P}(\ell, \ell'')$. Each branch probability can be computed for any given input distribution as $p_{\ell,\ell'}(\rho) = \|\rho\mathbf{P}(\ell, \ell')\|_1$ and $p_{\ell,\ell''}(\rho) = \|\rho\mathbf{P}(\ell, \ell'')\|_1$, respectively.

Sometimes it could be useful or practically more appropriate to consider abstract branch probabilities. These can be obtained by means of abstractions on the state space corresponding to classifications c : **State** $\to$ **State**$^{\#}$ that, as explained in Section 2.2, can be lifted to *classification matrices*. Given an equivalence relation $\sim$ on the states and its matrix representation $\mathbf{A}_{\sim}$, we can compute the individual chance of abstract states (i.e. equivalence classes of states) to take the **true** or **false** branch of a test by multiplying the abstract distribution $\rho^{\#}$ by an abstract version $\mathbf{P}(b)^{\#}$ of $\mathbf{P}(b)$ that we can use to select those classes of states satisfying b. In doing so we must guarantee that:

$$\begin{aligned} \rho\mathbf{P}(b)\mathbf{A} &= \rho^{\#}\mathbf{P}^{\#}(b) \\ \rho\mathbf{P}(b)\mathbf{A} &= \rho\mathbf{A}\mathbf{P}^{\#}(b) \\ \mathbf{P}(b)\mathbf{A} &= \mathbf{A}\mathbf{P}^{\#}(b) \end{aligned}$$

In order to give an explicit description of $\mathbf{P}^{\#}$ we only would need to multiply the last equation from the left with $\mathbf{A}^{-1}$. However, $\mathbf{A}$ is in general not a square matrix and thus not invertible. So we use instead the Moore-Penrose pseudo-inverse to have the closest, least-square approximation possible.

$$\begin{aligned} \mathbf{A}^{\dagger}\mathbf{P}(b)\mathbf{A} &= \mathbf{A}^{\dagger}\mathbf{A}\mathbf{P}^{\#}(b) \\ \mathbf{A}^{\dagger}\mathbf{P}(b)\mathbf{A} &= \mathbf{P}^{\#}(b) \end{aligned}$$

The abstract test matrix $\mathbf{P}^{\#}(b)$ contains all the information we need in order to estimate the abstract branch probabilities. Again, we denote by $\mathbf{P}(\ell, \ell')^{\#} = \mathbf{P}^{\#}(b)$ and $\mathbf{P}(\ell, \ell'')^{\#} = \mathbf{P}^{\#}(\neg b) = \mathbf{P}^{\#}(b)^{\perp}$ for a branching point $[b]^{\ell}$ with $(\ell, \underline{\ell'}), (\ell, \ell'') \in \mathit{flow}$.

Branch prediction/predictors in hardware design has long history [16, 20]. It is used at test points $[b]^{\ell}$ to allow pre-fetching of instructions of the expected branch before the test is actually evaluated. If the prediction is wrong the prefetched instructions need to be discarded and the correct ones to be fetched. Ultimately, wrong predictions "just" lead to longer running times, the correctness of the program is not concerned. It can be seen as a form of speculative optimisation. Typical applications or cases where branch prediction is relevant is for nested tests (loops or ifs). Here we get exactly the interplay between different tests and/or abstractions. We illustrate this in the following example.

Example 4 *Consider the following program that counts the prime numbers.*

$$[i := 2]^1;\ \texttt{while}\ [i < 100]^2\ \texttt{do if}\ [prime(i)]^3\ \texttt{then}\ [p := p+1]^4\ \texttt{else}\ [\texttt{skip}]^5\ \texttt{fi};\ [i := i+1]^6\ \texttt{od}$$

Within our framework we can simulate to a certain degree a history dependent branch prediction. If the variable p has been updated in the previous iteration it is highly unlikely it will so again in the next – in fact that only happens in the first two iterations. One can also interpret this as follows: For i even the branch probability $p_{3,4}(\rho_e)$ at label 3 is practically zero for any reasonable distribution, e.g. a uniform distribution ρ_e, on evens. To see this, we need to investigate only the form of

$$\mathbf{P}(prime(i))^{\#} = \mathbf{A}_e^{\dagger}\mathbf{P}(prime(i))\mathbf{A}_e,$$

where A_e is the abstraction corresponding to the classification in even and odd.

In order to understand how an abstract property interacts with the branching in the program, as in the previous example we look at $\mathbf{A}^{\dagger}\mathbf{P}(b)\mathbf{A}$ in order to evaluate how good a branch prediction is for a certain predicate/test b if it is based on a certain abstraction/property $\mathbf{A}$. This is explained in the following example where we consider two properties/abstractions and corresponding tests.

Example 5 *Let us consider two tests for numbers in the range $i = 0,1,2,3,\ldots,n$):*

$$\mathbf{P}_e = (\mathbf{P}(even(n)))_{ii} = \begin{cases} 1 & \text{if } i = 2k \\ 0 & \text{otherwise} \end{cases} \qquad \mathbf{P}_p = (\mathbf{P}(prime(n)))_{ii} = \begin{cases} 1 & \text{if } prime(i) \\ 0 & \text{otherwise} \end{cases}$$

Likewise we can consider two corresponding abstractions ($j \in \{1 = \mathbf{true}, 2 = \mathbf{false}\}$):

$$(\mathbf{A}_e)_{ij} = \begin{cases} 1 & \text{if } i = 2k+1 \ \wedge\ j = 2 \\ 1 & \text{if } i = 2k \ \wedge\ j = 1 \\ 0 & \text{otherwise} \end{cases} \qquad (\mathbf{A}_p)_{ij} = \begin{cases} 1 & \text{if } prime(i) \ \wedge\ j = 2 \\ 1 & \text{if } \neg prime(i) \ \wedge\ j = 1 \\ 0 & \text{otherwise} \end{cases}$$

Then we can use $\mathbf{P}^{\#}$ and its orthogonal complement, $(\mathbf{P}^{\#})^{\perp} = \mathbf{I} - \mathbf{P}^{\#}$ to determine information about the quality of a certain property or its corresponding abstraction via the number of false positives. In fact, this will tell us how precise the abstraction is with respect to tests (such as those controlling a loop or conditional). With rounding the values to 2 significant digits we get, for example the following results for different concrete ranges of the concrete values $0,\ldots,n$.

	$\mathbf{A}_e^{\dagger}\mathbf{P}_p\mathbf{A}_e$	$\mathbf{A}_e^{\dagger}\mathbf{P}_p^{\perp}\mathbf{A}_e$	$\mathbf{A}_p^{\dagger}\mathbf{P}_e\mathbf{A}_p$	$\mathbf{A}_p^{\dagger}\mathbf{P}_e^{\perp}\mathbf{A}_p$
$n = 10$	$\begin{pmatrix} 0.20 & 0.00 \\ 0.00 & 0.60 \end{pmatrix}$	$\begin{pmatrix} 0.80 & 0.00 \\ 0.00 & 0.40 \end{pmatrix}$	$\begin{pmatrix} 0.25 & 0.00 \\ 0.00 & 0.67 \end{pmatrix}$	$\begin{pmatrix} 0.75 & 0.00 \\ 0.00 & 0.33 \end{pmatrix}$
$n = 100$	$\begin{pmatrix} 0.02 & 0.00 \\ 0.00 & 0.48 \end{pmatrix}$	$\begin{pmatrix} 0.98 & 0.00 \\ 0.00 & 0.52 \end{pmatrix}$	$\begin{pmatrix} 0.04 & 0.00 \\ 0.00 & 0.65 \end{pmatrix}$	$\begin{pmatrix} 0.96 & 0.00 \\ 0.00 & 0.35 \end{pmatrix}$
$n = 1000$	$\begin{pmatrix} 0.00 & 0.00 \\ 0.00 & 0.33 \end{pmatrix}$	$\begin{pmatrix} 1.00 & 0.00 \\ 0.00 & 0.67 \end{pmatrix}$	$\begin{pmatrix} 0.01 & 0.00 \\ 0.00 & 0.60 \end{pmatrix}$	$\begin{pmatrix} 0.99 & 0.00 \\ 0.00 & 0.40 \end{pmatrix}$
$n = 10000$	$\begin{pmatrix} 0.00 & 0.00 \\ 0.00 & 0.25 \end{pmatrix}$	$\begin{pmatrix} 1.00 & 0.00 \\ 0.00 & 0.75 \end{pmatrix}$	$\begin{pmatrix} 0.00 & 0.00 \\ 0.00 & 0.57 \end{pmatrix}$	$\begin{pmatrix} 1.00 & 0.00 \\ 0.00 & 0.43 \end{pmatrix}$

Note that the positive and negative versions of these matrices always add up to the identity matrix **I**. *Also, the entries in the upper left corner of $\mathbf{A}_e^{\dagger}\mathbf{P}_p\mathbf{A}_e$ give us information about the chances that an even*

number is also a prime number: For small n the percentage is a fifth (indeed 2 *is a prime and it is one out of* 5 *even numbers under* 10*); the larger n gets the less relevant is this single even prime. With* $\mathbf{A}_p^\dagger \mathbf{P}_e \mathbf{A}_p$ *we get the opposite information: Among the prime numbers* $\{2,3,5,7\}$ *smaller than* 10 *there is one which is even, i.e. 25%; again this effect diminishes for larger n. Finally, the lower right entry in these matrices gives us the percentage that a non-prime number is odd and/or that an odd number is not prime, respectively.*

4.3 Linear Equations Framework

A general framework for our probabilistic data-flow analysis can be defined in analogy with the classical monotone framework by defining the following linear equations:

$$\begin{array}{rcl} \mathit{Analysis}_\bullet(\ell) & = & \mathit{Analysis}_\circ(\ell) \cdot \mathbf{F}_\ell \\ \mathit{Analysis}_\circ(\ell) & = & \begin{cases} \iota, \text{if } \ell \in E \\ \sum\{\mathit{Analysis}_\bullet(\ell') \cdot \mathbf{P}(\ell',\ell)^\# \mid (\ell',\ell) \in F\}, \text{otherwise} \end{cases} \end{array}$$

The first equation is a straight forward generalisation of the classical case, while the second one is defined by means of the linear sums over vectors. A simpler version is obtained by considering static branch prediction:

$$\mathit{Analysis}_\circ(\ell) = \sum\{p_{\ell',\ell} \cdot \mathit{Analysis}_\bullet(\ell') \mid (\ell',\ell) \in F\}$$

with $p_{\ell',\ell}$ is a numerical value representing a *static* branch probability.

We have as many variables in this systems of equations as there are individual equations. As a result we get unique solutions rather than least fix-points as in the classical setting.

This general scheme must be extended to include a preliminary phase of probability estimation if one wants to improve the quality of the branch prediction. In this case, the abstract state should carry two kinds of information: One, Prob, to provide estimates for probabilities, the other, Analysis, to analyse the actual property in question. The same abstract branch probabilities $\mathbf{P}(\ell',\ell)^\#$ – which we obtain via Prob – can then be used in both cases, but we have different information or properties and different transfer functions for Prob and Analysis.

4.4 Probabilistic Live Variable Analysis

We can use the previously defined probabilistic setting for a data flow analysis, to define a probabilistic version of the Live Variable analysis extending the one in [18] in order to also cover for random assignments and to provide estimates for 'live' probabilities.

The transfer functions, which describe how the program analysis information changes when we pass through a block $[B]^\ell$, is for the classical analysis given via the two auxiliary functions gen_{LV} and $kill_{\mathsf{LV}}$ (cf. Example 1). Probabilistic versions of these operations can be defined as follows. Consider two properties d for 'dead', and l for 'live' and the space $\mathscr{V}(\{0,1\}) = \mathscr{V}(\{d,l\}) = \mathbb{R}^2$ as the property space corresponding to a single variable. On this space define the operators:

$$\mathbf{L} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \quad \text{and} \quad \mathbf{K} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

The matrix $\mathbf{L}$ changes the "liveliness" of a variable from whatever it is (dead or alive) into alive, while $\mathbf{K}$ does the opposite. The local transfer operators

$$\mathbf{F}_\ell = \mathbf{F}_\ell^{LV} : \mathscr{V}(\{0,1\})^{\otimes|\mathbf{Var}|} \to \mathscr{V}(\{0,1\})^{\otimes|\mathbf{Var}|}$$

for the block $[x := a]^\ell$ can thus be defined as (with $\mathbf{I}$ the identity matrix)

$$\mathbf{F}_\ell = \bigotimes_{x_i \in \mathbf{Var}} \mathbf{X}_i \text{ with } \mathbf{X}_i = \begin{cases} \mathbf{L} & \text{if } x_i \in FV(a) \\ \mathbf{K} & \text{if } x_i = x \wedge x_i \notin FV(a) \\ \mathbf{I} & \text{otherwise.} \end{cases}$$

and similarly for tests $[b]^\ell$

$$\mathbf{F}_\ell = \bigotimes_{x_i \in \mathbf{Var}} \mathbf{X}_i \text{ with } \mathbf{X}_i = \begin{cases} \mathbf{L} & \text{if } x_i \in FV(b) \\ \mathbf{I} & \text{otherwise.} \end{cases}$$

For $[\mathtt{skip}]^\ell$ and random assignments $[x \mathtt{?=} \rho]^\ell$ we simply have $\mathbf{F}_\ell = \bigotimes_{x_i \in \mathbf{Var}} \mathbf{I}$.

In the following example we demonstrate the use of our general framework for probabilistic data-flow analysis by defining a probabilistic *LV* analysis for the program in Example 1.

Example 6 *For the program in Example 1 we present a LV analysis based on concrete branch probabilities. That means that in the first phase of the analysis (which determines the branch probabilities) we will not abstract the values of x and y (and ignore z all together). If the concrete state of each variable is a value in* $\{0,1,2,3\}$*, then the probabilistic state is an element in* $\mathscr{V}(\{0,1,2,3\})^{\otimes 3} = \mathbb{R}^{4^3} = \mathbb{R}^{64}$*. The abstraction we use when we compute the concrete branch probabilities is* $\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{A}_f$*, i.e. z is ignored. This allows us to reduce the dimensions of the probabilistic state space from* 64 *down to just* 16*. The abstract transfer functions for the first* 3 *statements are given in the Appendix.*

We can now compute the probability distribution at label 4 *for any given input distribution. The abstract transfer functions* $\mathbf{F}_5^\#$ *and* $\mathbf{F}_6^\#$ *are the identity as we have restricted ourselves only to the variables x and y.*

We can now set the linear equations for the joint distributions over x and y at the entry and exit to each of the labels:

$$\begin{array}{lcllcl}
\mathsf{Prob}_{entry}(1) & = & \rho & \mathsf{Prob}_{exit}(1) & = & \mathsf{Prob}_{entry}(1) \cdot \mathbf{F}_1^\# \\
\mathsf{Prob}_{entry}(2) & = & \mathsf{Prob}_{exit}(1) & \mathsf{Prob}_{exit}(2) & = & \mathsf{Prob}_{entry}(1) \cdot \mathbf{F}_2^\# \\
\mathsf{Prob}_{entry}(3) & = & \mathsf{Prob}_{exit}(2) & \mathsf{Prob}_{exit}(3) & = & \mathsf{Prob}_{entry}(1) \cdot \mathbf{F}_3^\# \\
\mathsf{Prob}_{entry}(4) & = & \mathsf{Prob}_{exit}(3) & \mathsf{Prob}_{exit}(4) & = & \mathsf{Prob}_{entry}(4) \\
\mathsf{Prob}_{entry}(5) & = & \mathsf{Prob}_{exit}(4) \cdot \mathbf{P}_4^\# & \mathsf{Prob}_{exit}(5) & = & \mathsf{Prob}_{entry}(5) \\
\mathsf{Prob}_{entry}(6) & = & \mathsf{Prob}_{exit}(4) \cdot (\mathbf{I} - \mathbf{P}_4^\#) & \mathsf{Prob}_{exit}(6) & = & \mathsf{Prob}_{entry}(6)
\end{array}$$

These equations are easy to solve. In particular we can explicitly determine

$$\begin{array}{lcl}
\mathsf{Prob}_{entry}(5) & = & \rho \cdot \mathbf{F}_1^\# \cdot \mathbf{F}_2^\# \cdot \mathbf{F}_3^\# \cdot \mathbf{P}_4^\# \\
\mathsf{Prob}_{entry}(6) & = & \rho \cdot \mathbf{F}_1^\# \cdot \mathbf{F}_2^\# \cdot \mathbf{F}_3^\# \cdot \mathbf{P}_4^\#,
\end{array}$$

that give us the static branch probabilities $p_{4,5}(\rho) = \|\mathsf{Prob}_{entry}(5)\|_1 = \frac{1}{4}$ *and* $p_{4,6}(\rho) = \|\mathsf{Prob}_{entry}(6)\|_1 = \frac{3}{4}$*. These distributions can explicitly be computed and do not depend on the initial distribution* ρ*.*

We then perform a probabilistic LV analysis using these probabilities as required. Using the abstract property space and the auxiliary operators we get:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{LV}_{entry}(1) &=& \mathsf{LV}_{exit}(1)\cdot(\mathbf{K}\otimes\mathbf{I}\otimes\mathbf{I}) & \mathsf{LV}_{exit}(1) &=& \mathsf{LV}_{entry}(2)\\
\mathsf{LV}_{entry}(2) &=& \mathsf{LV}_{exit}(2)\cdot(\mathbf{I}\otimes\mathbf{K}\otimes\mathbf{I}) & \mathsf{LV}_{exit}(2) &=& \mathsf{LV}_{entry}(3)\\
\mathsf{LV}_{entry}(3) &=& \mathsf{LV}_{exit}(3)\cdot(\mathbf{L}\otimes\mathbf{L}\otimes\mathbf{I}) & \mathsf{LV}_{exit}(3) &=& \mathsf{LV}_{entry}(4)\\
\mathsf{LV}_{entry}(4) &=& \mathsf{LV}_{exit}(4)\cdot(\mathbf{L}\otimes\mathbf{I}\otimes\mathbf{I}) & \mathsf{LV}_{exit}(4) &=& p_{4,5}\mathsf{LV}_{entry}(5)+p_{4,6}\mathsf{LV}_{entry}(6)\\
\mathsf{LV}_{entry}(5) &=& \mathsf{LV}_{exit}(5)\cdot(\mathbf{L}\otimes\mathbf{I}\otimes\mathbf{K}) & \mathsf{LV}_{exit}(5) &=& (1,0)\otimes(1,0)\otimes(1,0)\\
\mathsf{LV}_{entry}(6) &=& \mathsf{LV}_{exit}(6)\cdot(\mathbf{I}\otimes\mathbf{L}\otimes\mathbf{K}) & \mathsf{LV}_{exit}(6) &=& (1,0)\otimes(1,0)\otimes(1,0)
\end{array}
$$

And thus the solutions for the probabilistic LV analysis are given by:

$$
\begin{array}{rcl@{\qquad}rcl}
\mathsf{LV}_{entry}(1) &=& (1,0)\otimes(1,0)\otimes(1,0) & \mathsf{LV}_{exit}(1) &=& (0,1)\otimes(1,0)\otimes(1,0)\\
\mathsf{LV}_{entry}(2) &=& (0,1)\otimes(1,0)\otimes(1,0) & \mathsf{LV}_{exit}(2) &=& (0,1)\otimes(0,1)\otimes(1,0)\\
\mathsf{LV}_{entry}(3) &=& 0.25\cdot(0,1)\otimes(0,1)\otimes(1,0)+ & \mathsf{LV}_{exit}(3) &=& 0.25\cdot(0,1)\otimes(1,0)\otimes(1,0)+\\
&+& 0.75\cdot(0,1)\otimes(0,1)\otimes(1,0) &&+& 0.75\cdot(0,1)\otimes(0,1)\otimes(1,0)\\
&=& (0,1)\otimes(0,1)\otimes(1,0) & \mathsf{LV}_{exit}(4) &=& 0.25\cdot(0,1)\otimes(1,0)\otimes(1,0)+\\
\mathsf{LV}_{entry}(4) &=& 0.25\cdot(0,1)\otimes(1,0)\otimes(1,0)+ &&+& 0.75\cdot(1,0)\otimes(0,1)\otimes(1,0)\\
&+& 0.75\cdot(0,1)\otimes(0,1)\otimes(1,0) & \mathsf{LV}_{exit}(5) &=& (1,0)\otimes(1,0)\otimes(1,0)\\
\mathsf{LV}_{entry}(5) &=& (0,1)\otimes(1,0)\otimes(1,0) & \mathsf{LV}_{exit}(6) &=& (1,0)\otimes(1,0)\otimes(1,0)\\
\mathsf{LV}_{entry}(6) &=& (1,0)\otimes(0,1)\otimes(1,0) &&&
\end{array}
$$

This means that, for example, at the beginning label 4*, i.e. the test* $x > 2$ *there are two situations: It can be with probability* $\frac{1}{4}$ *that only the variable x is alive, or with probability* $\frac{3}{4}$ *both variables x and y are alive. One could say that x for sure is alive and y only with a 75% chance. At the exit of label* 4 *the probabilistic LV analysis tells us that with 25% chance* only *x is alive and with 75% that y is the* only *live variable. To say that x is alive with probability* 0.25 *and y with* 0.75 *probability would be wrong: It is either x or y which is alive and this is reflected in the joint distributions represented as tensors, which we obtain as solution. This illustrates that the probabilistic property space cannot be just* $\mathscr{V}(\{x,y,z\})$ *but that we need indeed* $\mathscr{V}(\{d,l\})^{\otimes 3}$*.*

5 Conclusions and Related Work

This paper highlights two important aspects of probabilistic program analysis in a data-flow style: (i) the use of tensor products in order to represent the correlation between a number of variables, and (ii) the use of Probabilistic Abstract Interpretation to estimate branch probabilities and to construct probabilistic transfer functions. In particular, we argue that static program analysis does not mean necessarily considering *static branch prediction*. Instead – by extending single numbers $p_{\ell,\ell'}$ as branch probabilities to matrices as abstract branch probabilities $\mathbf{P}(\ell,\ell)^{\#}$ – the PAI framework allows us to express dynamic or conditional aspects.

The framework presented here aims in providing a formal basis for speculative optimisation. Speculative optimisation [15, 2] has been an element of hardware design for some time, in particular to branch prediction [16] or for cache optimisation [17]. More recently, related ideas have also been discussed in the context of speculative multi-threading [4] or probabilistic pointer analysis [9, 13].

The work we have presented in this paper concentrates on the conceptual aspects of probabilistic analysis and not on optimal realisation of, for example, concrete branch predictors. Further work should however include practical implementations of the presented framework in order to compare its performance with the large number of predictors in existence. Another research direction concerns the automatic construction of abstractions so that the induced $\mathbf{P}(\ell, \ell)^{\#}$ are optimal and maximally predictive.

References

[1] A.V. Aho, M.S. Lam, R. Sethi & J.D. Ullman (2007): *Compilers: Principles, Techniques, and Tools*, second edition. Pearson Education.

[2] A. A. Belevantsev, S. S. Gaisaryan & V. P. Ivannikov (2008): *Construction of Speculative Optimization Algorithms*. Programming and Computer Software 34(3), pp. 138–153, doi:10.1134/S036176880803002X.

[3] A. Ben-Israel & T.N.E. Greville (2003): *Generalised Inverses*, 2nd edition. Springer Verlag.

[4] A. Bhowmik & M. Franklin (2004): *A General Compiler Framework for Speculative Multithreaded Processors*. IEEE Transactions on Parallel and Distributed Syststems 15(8), pp. 713–724, doi:10.1109/TPDS.2004.26.

[5] S.L. Campbell & D. Meyer (1979): *Generalized Inverse of Linear Transformations*. Constable, London.

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

[7] F. Deutsch (2001): *Bet Approximation in Inner Product Spaces*. CMS Books in Mathematics 7, Springer Verlag, New York — Berlin.

[8] A. Di Pierro, C. Hankin & H. Wiklicky (2007): *Abstract Interpretation for Worst and Average Case Analysis*. In: Program Analysis and Compilation, Theory and Practice, *LNCS* 4444, Springer Verlag, pp. 160–174, doi:10.1007/978-3-540-71322-7_8.

[9] A. Di Pierro, C. Hankin & H. Wiklicky (2007): *A Systematic Approach to Probabilistic Pointer Analysis*. In Z. Shao, editor: Proceedings of *APLAS'07*, *LNCS* 4807, Springer Verlag, pp. 335–350, doi:10.1007/978-3-540-76637-7_23.

[10] A. Di Pierro, C. Hankin & H. Wiklicky (2010): *Probabilistic Semantics and Analysis*. In: Formal Methods for Quantitative Aspects of Programming Languages, *LNCS* 6155, Springer Verlag, pp. 1–42, doi:10.1007/978-3-642-13678-8_1.

[11] A. Di Pierro, P. Sotin & H. Wiklicky (2008): *Relational Analysis and Precision via Probabilistic Abstract Interpretation*. In C. Baier & A. Aldini, editors: Proceedings of *QAPL'08*, Electronic Notes in Theoretical Computer Science, Elsevier, pp. 23–42, doi:10.1016/j.entcs.2008.11.017.

[12] A. Di Pierro & H. Wiklicky (2000): *Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation*. In: *PPDP'00*, pp. 127–138, doi:10.1145/351268.351284.

[13] M.-Y. Hung, P.-S. Chen, Y-S. Hwang, R. D.-C. Ju & J. K. Lee (2012): *Support of Probabilistic Pointer Analysis in the SSA Form*. IEEE Transactions on Parallel Distributed Syststems 23(12), pp. 2366–2379, doi:10.1109/TPDS.2012.73.

[14] M.Z. Kwiatkowska, G. Norman & D. Parker (2004): *PRISM 2.0: A Tool for Probabilistic Model Checking*. In: International Conference on Quantitative Evaluation of Systems (QEST 2004), IEEE Computer Society, pp. 322–323, doi:10.1109/QEST.2004.10016.

[15] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai & S. Chan (2003): *A compiler framework for speculative analysis and optimizations*. In: Proceedings Conference on Programming Language Design and Implementation (PLDI), pp. 289–299, doi:10.1145/781131.781164.

[16] S. McFarling (1993): *Combining Branch Predictors*. Technical Report WLR TN-36, Digital.

[17] D. Nicolaescu, B. Salamat & A.V. Veidenbaum (2006): *Fast Speculative Address Generation and Way Caching for Reducing L1 Data Cache Energy*. In: *Proceedings of the 24th International Conference on Computer Design (ICCD 2006)*, IEEE, pp. 101–107, doi:10.1109/ICCD.2006.4380801.

[18] F. Nielson, H. Riis Nielson & C. Hankin (1999): *Principles of Program Analysis*. Springer Verlag, Berlin – Heidelberg.

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

[20] H. Styles & W. Luk (2004): *Exploiting Program Branch Probabilities in Hardware Compilation*. *IEEE Transaction on Computers* 53(11), pp. 1408–1419, doi:10.1109/TC.2004.96.

Appendix

For completeness, we present here the abstract transfer functions in the probabilistic analysis of Example 6.

$$\mathbf{F}_1^{\#} = \begin{pmatrix}
\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

$$\mathbf{F}_2^{\#} = \begin{pmatrix}
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & \frac{1}{4}
\end{pmatrix}$$

$$\mathbf{F}_3^{\#} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{pmatrix}$$

$$\mathbf{P}_4^{\#} = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$