Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 190 (2007) 59–77

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

On Probabilistic Techniques for Data Flow Analysis

Alessandra Di Pierro[1] Chris Hankin[2] Herbert Wiklicky[3]

Department of Computing
Imperial College London
180 Queen's Gate
London SW7 2AZ
United Kingdom

Abstract

We present a semantics-based technique for analysing probabilistic properties of imperative programs. This consists in a probabilistic version of classical data flow analysis. We apply this technique to **pWhile** programs, i.e programs written in a probabilistic version of a simple While language. As a first step we introduce a syntax based definition of a linear operator semantics (LOS) which is equivalent to the standard structural operational semantics of While. The LOS of a **pWhile** program can be seen as the generator of a Discrete Time Markov Chain and plays a similar role as a collecting or trace semantics for classical While. Probabilistic Abstract Interpretation techniques are then employed in order to define data flow analyses for properties like Parity and Live Variables.

Keywords: Probabilistic Programs, Linear Operators Semantics, Probabilistic Abstract Interpretation, Data Flow Analysis

1 Introduction

Typically a classical Data Flow Analysis (DFA) considers all the paths in a control-flow graph and computes safe solutions according to a 'meet-over-all-paths' approach, i.e. all paths contribute equally to the solution independently on how feasible or likely to be executed they are. However, in practice even moderately large programs execute only a small part of the billions of paths that are possible for the program to execute. For a given analysis, the specific nature of the problem under consideration may help to reduce the number of paths. Moreover, it is now widely recognised that considering probabilistic information allows one to obtain more precise and practically useful results from a static analysis. Statistical or other types

[1] Email: adip@doc.ic.ac.uk
[2] Email: clh@doc.ic.ac.uk
[3] Email: herbert@doc.ic.ac.uk

doi:10.1016/j.entcs.2007.07.005

of information can be encoded in the form of probabilities in the program semantics and used to weight the execution paths according to their likelihood to actually be executed. Following this approach, in this paper we define probabilistic DFA whose solutions are not safe in the classical sense but 'close' enough to the concrete behaviour of the program. This is similar to the approach in [7,8], where probabilistic techniques are used for the construction of static analyses of approximate properties. Considering approximate rather than absolute properties corresponds to asserting the property up to a given quantifiable error margin rather than as a Boolean relation. The significance of this approach is that it allows in general for more realistic analyses. As an example, a security property such as confinement is never satisfied by real systems in its absolute formulation asserting whether the system is confined or not.

We show how a classical DFA can be turned into a probabilistic one via the use of an appropriate collecting semantics. This is defined as a linear operator on a linear space representing the operational configurations (current state and instruction to be executed) of a given probabilistic program. We will show that the use of tensor product is essential in the construction of the semantics in order to obtain a correct result of the analysis. This is demonstrated via the application of our probabilistic technique to two classical DFA examples, namely Parity Analysis and Live Variables Analysis. These examples also show how probabilistic analysis can result in more practically useful information compared to the standard classical techniques.

2 Probabilistic While

The language we consider is a simple classical **While** language extended with a probabilistic choice statement. A different but equivalent approach to defining probabilistic languages is the use of random assignment as in e.g. [14]. We choose the first approach because it is more suited as a base for the treatment we present in this paper. The same approach is also adopted in [6] where a Hoare-style proof system is introduced for reasoning about probabilistic sequential programs.

2.1 Syntax

The syntax of the language, which we call **pWhile**, is given by:

$$\begin{array}{c} S \;::=\; \textbf{skip} \;\Big|\; \textbf{stop} \;\Big|\; \texttt{x} \leftarrow a \;\Big|\; S_1; S_2 \;\Big|\; \textbf{choose}\ p_1 : S_1\ \textbf{or}\ p_2 : S_2 \;\Big| \\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end if} \;\Big|\; \textbf{while}\ b\ \textbf{do}\ S\ \textbf{end while} \end{array}$$

In the probabilistic choice construct we assume that p_i are some constants representing the probability that statement S_i is executed; they define a probability distribution, i.e. $p_1 + p_2 = 1$. Arithmetic expressions a follow the usual syntax:

$$a \;::=\; n \;\Big|\; \texttt{x} \;\Big|\; a_1 \odot a_2$$

where n represents a constant value, `x` a program variable (for which we use typeface characters) and '$\odot$' represents one of the usual arithmetic operations, like '$+$', '$-$', or '$\times$'. We also need Boolean expressions which have the following form:

R0	$\langle \textbf{skip}, \sigma\rangle \longrightarrow_1 \langle \textbf{stop}, \sigma\rangle$	
R1	$\langle \textbf{stop}, \sigma\rangle \longrightarrow_1 \langle \textbf{stop}, \sigma\rangle$	
R2	$\langle \mathrm{x} \leftarrow a, \sigma\rangle \longrightarrow_1 \langle \textbf{stop}, \sigma[x \mapsto [\![a]\!]\sigma]\rangle$	
R3$_1$	$\dfrac{\langle S_1, \sigma\rangle \longrightarrow_p \langle S_1', \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \longrightarrow_p \langle S_1'; S_2, \sigma'\rangle}$	
R3$_2$	$\dfrac{\langle S_1, \sigma\rangle \longrightarrow_p \langle \textbf{stop}, \sigma'\rangle}{\langle S_1; S_2, \sigma\rangle \longrightarrow_p \langle S_2, \sigma'\rangle}$	
R4$_1$	$\langle \textbf{choose}\ p_1 : S_1\ \textbf{or}\ p_2 : S_2, \sigma\rangle \longrightarrow_{p_1} \langle S_1, \sigma\rangle$	
R4$_2$	$\langle \textbf{choose}\ p_1 : S_1\ \textbf{or}\ p_2 : S_2, \sigma\rangle \longrightarrow_{p_2} \langle S_2, \sigma\rangle$	
R5$_1$	$\langle \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end if}, \sigma\rangle \longrightarrow_1 \langle S_1, \sigma\rangle$	if $[\![b]\!]\sigma = \text{TRUE}$
R5$_2$	$\langle \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{end if}, \sigma\rangle \longrightarrow_1 \langle S_2, \sigma\rangle$	if $[\![b]\!]\sigma = \text{FALSE}$
R6$_1$	$\langle \textbf{while}\ b\ \textbf{do}\ S\ \textbf{end while}, \sigma\rangle \longrightarrow_1 \langle S, \sigma\rangle$	if $[\![b]\!]\sigma = \text{TRUE}$
R6$_2$	$\langle \textbf{while}\ b\ \textbf{do}\ S\ \textbf{end while}, \sigma\rangle \longrightarrow_1 \langle \textbf{stop}, \sigma\rangle$	if $[\![b]\!]\sigma = \text{FALSE}$

Table 1
A Probabilistic Transition System for **pWhile**

$$b \ ::= \ \text{TRUE} \ \Big| \ \text{FALSE} \ \Big| \ \neg b \ \Big| \ b_1 \vee b_2 \ \Big| \ b_1 \wedge b_2 \ \Big| \ a_1 \lessgtr a_2$$

where '$\lessgtr$' denotes one of the standard comparison operators for arithmetic expressions like: $<, \leq, =, \geq, >$.

2.2 Operational Semantics

In this section we define a small step operational semantics for **pWhile** in the usual SOS style [17]. This corresponds to a probabilistic transition system [13] where states encode information on both the memory state (values of the program's variables) and the instruction to be executed. The transition relation is probabilistic in the sense that the configuration reached in one step from another configuration is associated with a number expressing the probability of actually performing that transition; moreover, all probabilities associated to the outgoing transitions of a configuration form a probability distribution (i.e. they sum up to one). In other words, we define the operational semantics of our language according to the *generative* model of probability [20] and the resulting execution process is a Discrete Time Markov Chain, [19,1].

2.2.1 Configurations and Transitions

We define a *state* as a mapping from variables to values:

$$\sigma \in \textbf{State} = \textbf{Var} \rightarrow \textbf{Value},$$

$$[\![a]\!]_a\sigma = n \qquad [\![\mathrm{x}]\!]_a\sigma = \sigma(\mathrm{x}) \qquad [\![a_1 \odot a_2]\!]_a\sigma = [\![a_1]\!]_a\sigma \odot [\![a_2]\!]_a\sigma$$

$$\begin{array}{lllll}
[\![\mathrm{TRUE}]\!]_b\sigma & = \mathrm{TRUE} & \quad [\![b_1 \vee b_2]\!]_b\sigma & = [\![b_1]\!]_b\sigma \vee [\![b_2]\!]_b\sigma \\
[\![\mathrm{FALSE}]\!]_b\sigma & = \mathrm{FALSE} & \quad [\![b_1 \wedge b_2]\!]_b\sigma & = [\![b_1]\!]_b\sigma \wedge [\![b_2]\!]_b\sigma \\
[\![\neg b]\!]_b\sigma & = \neg[\![b]\!]_b\sigma & \quad [\![a_1 \lessgtr a_2]\!]_b\sigma & = [\![a_1]\!]_b\sigma \lessgtr [\![a_2]\!]_b\sigma
\end{array}$$

Table 2
Semantics of Expressions

The value of arithmetic and Boolean expressions is given by the functions $[\![.]\!]_a$ and $[\![.]\!]_b$ defined in Table 2. We will omit the index from the function $[\![.]\!]$ when it is clear from the context.

A *configuration* is a pair $\langle S, \sigma\rangle$ consisting of a statement S and a state $\sigma \in$ **State**. The probabilistic transition relation $\longrightarrow_p$ is defined by the rules in Table 1. These reflect the typical semantics of a classical while language but for the presence of a probability labelling each transition. Final states are represented via a looping on configurations $\langle \mathbf{stop}, \sigma\rangle$. This allows us to model a computation as a Markov chain as it guarantees that the matrix representing the execution of a **pWhile** program is a stochastic matrix (cf. Section 3.1).

2.2.2 Abstract Syntax

Following the approach in [16], in order to determine the control flow (graph) of a program, regardless of the concrete information on the actual states in which each statement is executed, we associate a label from a set **Lab** to each assignment statement, to the **skip** statement and to each test that appears in conditional and loop statements. We call the resulting program a *labelled program* and define its syntax, which we call *abstract syntax*, as follows:

$$S ::= [\mathbf{skip}]^\ell \;\Big|\; [\mathbf{stop}]^\ell \;\Big|\; [\mathrm{x} \leftarrow a]^\ell \;\Big|\; S_1; S_2 \;\Big|\; [\mathbf{choose}]^\ell\, p_1 : S_1 \;\mathbf{or}\; p_2 : S_2$$
$$\Big|\; \mathbf{if}\; [b]^\ell \;\mathbf{then}\; S_1 \;\mathbf{else}\; S_2 \;\mathbf{end\ if} \;\Big|\; \mathbf{while}\; [b]^\ell \;\mathbf{do}\; S \;\mathbf{end\ while}$$

We will refer to the elementary labelled instructions as *commands* and denote by **Cmd** the set of all commands with typical elements $c_1, c_2, \ldots$, corresponding to statements of the form $[\mathbf{skip}]^\ell$ or $[b]^\ell$, etc. We will assume a unique labelling of commands, i.e. we will use ℓ interchangeably with c, where c is the elementary instruction with label ℓ; this is sometimes called the label consistency property [16]. The control flow $\mathcal{F}_S$ of a program S is then defined as a set of pairs (ℓ, ℓ') or $\langle c_1, c_2\rangle$ which record the fact that it is possible that after executing c_1 the next command executed could be c_2 (abstracting from the concrete outcomes of tests).

In principle, it is possible to reconstruct the syntax of the original program S from the pairs $\langle \ell, \ell'\rangle$ in $\mathcal{F}_S$: we only have to indicate which branch is taken when a test succeeds or fails; and for probabilistic choices we need to record the transition probabilities. If we do this the control flow $\mathcal{F}_S$ of S, which is sometimes also referred to as its abstract syntax [5], can be taken as an alternative description of the syntax of the program. In this paper, we will use the terms control flow and abstract syntax interchangeably to refer to a labelled program.

3 Probabilistic Data Flow Analysis

Classically, Data Flow Analysis is based on a graph representation of a program usually defined via a collecting semantics. For our probabilistic language we will define such a collecting semantics as a linear operator (more precisely a Markov chain); this represents the states and the program statements in a way that the probabilistic information can be appropriately derived in the final result of the analysis. We will consider here the more common equational approach to DFA and introduce a probabilistic technique based on this approach and the linear operator semantics in Section 3.1.

The probabilistic DFA we present is intra-procedural (we do not deal with functions or procedures). We will demonstrate our technique by presenting two classical examples of a forward and a backward analysis, the former being specifically a Parity Analysis and the latter Live Variables Analysis.

3.1 Collecting Semantics

A probabilistic transition system has a straightforward representation as a matrix, i.e. a linear operator on the vector space over configurations. In general, for any set X we can define the *vector space* $\mathcal{V}(X)$ over X as the space of formal linear combinations of elements in X with coefficients in a field $\mathbb{W}$. We can represent the elements in $\mathcal{V}(X)$ as vectors with coefficients in $\mathbb{W}$ indexed by X:

$$\mathcal{V}(X) = \{\ (v_x)_{x \in X} \mid v_x \in \mathbb{W}\}.$$

The field $\mathbb{W}$ is typically taken to be the set of complex numbers $\mathbb{C}$. In our case it simply corresponds to $[0,1] \subseteq \mathbb{R} \subseteq \mathbb{C}$, as we are interested in representing probability distributions. The support set X corresponds to the set of configurations **Conf**.

A convenient way to represent the vector space on configurations is via the *tensor product* operation. The tensor product is a useful tool when it comes to describing composite systems, such as in the theory of stochastic systems, performance analysis, quantum physics, etc. Its properties allow us to elegantly represent the vector space over a Cartesian product $\mathcal{V}(X \times Y)$ as $\mathcal{V}(X) \otimes \mathcal{V}(Y)$. These properties and an abstract definition can be found e.g. in [18].

Concretely, we can define the tensor or Kronecker product of two finite dimensional matrices as follows. Given an $n \times m$ matrix $\mathbf{A}$ and and a $k \times l$ matrix $\mathbf{B}$ then $\mathbf{A} \otimes \mathbf{B}$ is the $nk \times ml$ matrix:

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & \ldots & a_{1n}\mathbf{B} \\ \vdots & \ddots & \vdots \\ a_{1m}\mathbf{B} & \ldots & a_{nm}\mathbf{B} \end{pmatrix}$$

Based on the abstract syntax or control flow of a program S we define the collecting semantics of S as a linear operator on $\mathcal{V}(\mathbf{State} \times \mathbf{Cmd}) = \mathcal{V}(\mathbf{Value}) \otimes$

$$\begin{aligned}
\mathbf{T}(\langle[\mathbf{skip}]^{\ell_1}, c_2\rangle) &= \mathbf{I} \otimes \mathbf{E}_{\ell_1,c_2} \\
\mathbf{T}(\langle[\mathrm{x}_i \leftarrow a]^{\ell_1}, c_2\rangle) &= \mathbf{A}(a) \otimes \mathbf{E}_{\ell_1,c_2} \\
\mathbf{T}(\langle[b]^{\ell}, c_t\rangle) &= \mathbf{B}(b) \otimes \mathbf{E}_{\ell,c_t} \\
\mathbf{T}(\langle[b]^{\ell}, c_f\rangle) &= (\mathbf{I} - \mathbf{B}(b)) \otimes \mathbf{E}_{\ell,c_f} \\
\mathbf{T}(\langle[\mathbf{choose}]^{\ell}_{p_k}, c_k\rangle) &= p_k \cdot \mathbf{I} \otimes \mathbf{E}_{\ell,c_k} \\
\mathbf{T}(\langle[\mathbf{stop}]^{\ell}, [\mathbf{stop}]^{\ell}\rangle) &= \mathbf{I} \otimes \mathbf{E}_{\ell,\ell}
\end{aligned}$$

Table 3
Collecting Semantics

$\mathcal{V}(\mathbf{Cmd}) = \mathcal{V}(\mathbf{Value}_1) \otimes \ldots \otimes \mathcal{V}(\mathbf{Value}_n) \otimes \mathcal{V}(\mathbf{Cmd})$. Here we assume an enumeration of the program variables $x_1, \ldots, x_n$ and denote by $\mathbf{Value}_i$ the set of values for x_i. Thus, we have that $\mathcal{V}(\mathbf{Value}) = \mathcal{V}(\mathbf{Value}_1 \times \ldots \times \mathbf{Value}_n) = \mathcal{V}(\mathbf{Value}_1) \otimes \ldots \otimes \mathcal{V}(\mathbf{Value}_n)$.

For all possible control flow steps in $\mathcal{F}_S$, i.e. pairs of commands $\langle c_1, c_2\rangle$ in a program S we will describe the possible changes to the variables and then sum up these effects together with the actual transfer of control from one statement to another. In other words, we define the collecting semantics $[\![S]\!]$ of a program S in terms of the linear operators $\mathbf{T}(\langle c_1, c_2\rangle)$, which for every pair $\langle c_1, c_2\rangle \in \mathcal{F}_S$ transform a vector configuration $\vec{x} \in \mathcal{V}(\mathbf{State} \times \mathbf{Cmd})$ into the vector configuration describing the effect of the control flow step from c_1 to c_2. Formally:

$$[\![S]\!] = \sum_{\langle c_1, c_2\rangle \in \mathcal{F}_S} \mathbf{T}(\langle c_1, c_2\rangle)$$

where $\mathbf{T}$ can be expressed via the tensor product:

$$\mathbf{T}(\langle c_1, c_2\rangle) = \mathbf{M}(\langle c_1, c_2\rangle) \otimes \mathbf{N}(\langle c_1, c_2\rangle)$$

with $\mathbf{M}(\langle c_1, c_2\rangle)$ an operator on $\mathcal{V}(\mathbf{State})$ describing the possible state changes, and $\mathbf{N}(\langle c_1, c_2\rangle)$ an operator on $\mathcal{V}(\mathbf{Cmd})$ describing the control flow. For all possible commands $c_1, c_2 \in \mathbf{Cmd}$ we define $\mathbf{T}(\langle c_1, c_2\rangle)$ in Table 3, where we use the control and state operators explained below.

Note that the last rule in Table 3 which expresses the looping on the final configuration guarantees that $[\![S]\!]$ is a stochastic matrix, and that the collecting semantics of a program is indeed a Markov chain.

Control Operators

In the deterministic case, i.e. the case where for a command c_1 there is only one command c_2 with $\langle c_1, c_2\rangle \in \mathcal{F}_S$, the control flow part $\mathbf{N}$ is very simple: it is given by a so called *matrix unit* $\mathbf{E}_{c_1,c_2}$ containing a single non-zero entry in row c_i and column c_j, namely

$$\mathbf{N}(\langle c_1, c_2\rangle) = (\mathbf{E}_{c_1,c_2})_{i,j} = \begin{cases} 1 \text{ for } i = c_1 \ \wedge \ j = c_2 \\ 0 \text{ otherwise} \end{cases}$$

This is the case for $[\mathbf{skip}]^\ell$, $[\mathbf{stop}]^\ell$ and $[\mathrm{x} \leftarrow a]^\ell$. This kind of control transition applies in particular to the case of the control flow between the statements in a sequential composition.

In the case of the random choice we have to construct the control transfer operator as a linear combination between two deterministic transitions, from the choice point to the initial label of each of the branches. This is effectively achieved by defining

$$\mathbf{N}(\langle c, c_k\rangle)_{i,j} = p_k \cdot \mathbf{E}_{c,c_k}$$

where c is the choice command and c_k the first command in the kth branch.

For tests $c = [b]^\ell$ in **while**'s and **if**'s we also have to construct a linear combination, however in this case we have to filter in each case those transitions which actually happen when the test succeeds from those when the test does not succeed. Thus, we define the control flow at a test point by

$$\begin{aligned}\mathbf{T}(\langle c, c_t\rangle) &= \mathbf{B}(b) \cdot (\mathbf{I} \otimes \mathbf{E}_{c,c_t})\\ \mathbf{T}(\langle c, c_f\rangle) &= (\mathbf{I} - \mathbf{B}(b)) \cdot (\mathbf{I} \otimes \mathbf{E}_{c,c_f})\end{aligned}$$

where c is the test command $[b]^\ell$ and c_t and c_f the successors in case the test succeeds or fails, respectively. In this definition the operator $\mathbf{I}$ is the identity while the test operator $\mathbf{B}$ is defined by:

$$\mathbf{B}(b) = \sum_{b(v_1,\ldots,v_n)=\mathrm{TRUE}} \left(\bigotimes_{k=1}^{n} \mathbf{D}(e_{v_k}) \right)$$

where $\mathbf{D}(x)$ denotes the diagonal matrix with diagonal vector x and e_i the unit-vectors:

$$(e_i) = \begin{cases} 1 \text{ if } i = j \\ 0 \text{ otherwise.}\end{cases}$$

Thus, $\mathbf{D}(e_i) = \mathbf{E}_{i,i}$, and $\mathbf{B}$ contains a '1' in the diagonal if for some values of the variables the test b succeeds.

Example 3.1 Suppose that we have two variables x_1 and x_2 which can both have values between 0 and 4. Then

$$[\mathrm{x}_1 < 2]^\ell$$

is represented by a 25×25 matrix given as the tensor product:

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

which expresses the fact that the value of $\mathtt{x}_2$ does not contribute to the result of the test.

State Operators

For all commands c except for the assignment $[\mathtt{x} \leftarrow a]^\ell$ the variables will not change, i.e. we will have $\mathbf{M} = \mathbf{I}$, i.e. the identity on the vector space representing the value of the program variables. In the case of the assignment we have for constants

$$\mathbf{M}([\mathtt{x}_i \leftarrow n]^{\ell_1}, c_2) = \left(\bigotimes_{k=1}^{i-1} \mathbf{I}\right) \otimes \left(\sum_j \mathbf{E}_{jn}\right) \otimes \left(\bigotimes_{k=i+1}^{n} \mathbf{I}\right),$$

and for general expressions a involving (other) variables

$$\mathbf{M}([\mathtt{x}_i \leftarrow a]^{\ell_1}, c_2) = \sum_{a(v_1,\dots,v_n)=v} \mathbf{A}_{v_1,\dots,v_i \mapsto v,\dots,v_n}$$

where $\mathbf{A}_{v_1,\dots,v_i \mapsto v,\dots,v_n}$ is a matrix with a single non-zero entry defined as:

$$\left(\bigotimes_{k=1}^{i-1} \mathbf{D}(e_{v_k})\right) \otimes \left(\sum_j \mathbf{E}_{v_i,v}\right) \otimes \left(\bigotimes_{k=i+1}^{n} \mathbf{D}(e_{v_k})\right).$$

In other words, if a certain combination of values v_i for the variables in a produces a certain result v, then the corresponding matrix $\mathbf{A}_{v_1,\dots,v_i \mapsto v,\dots,v_n}$ indicating this single update contributes to the sum.

Example 3.2 Consider two variables x_1 and x_2 which both can take values in $\{0, 1, 2\}$. Then the update

$$[\mathtt{x}_1 \leftarrow \mathtt{x}_1 + 1]^\ell$$

is represented by a 9×9 matrix, which is the tensor product of two 3×3 matrices, namely:

$$\mathbf{M}([\mathtt{x}_1 \leftarrow \mathtt{x}_1 + 1]^\ell, c_2) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

while

$$[\mathtt{x}_2 \leftarrow 1]^\ell$$

is represented by

$$\mathbf{M}([\mathtt{x}_2 \leftarrow 1]^\ell, c_2) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

In the following example we show how to construct the full collecting semantics for a simple **pWhile** program S.

Example 3.3 Consider the program S defined by:

```
if [x == 0]^a then
    [x ← 0]^b;
else
    [x ← 1]^c;
end if;
[stop]^d
```

If we enumerate the four relevant program points by a, b, c and d and assume that x can have only two values, namely 0 and 1 then the whole program is represented by (where $\mathbf{I}_d$ denotes the $d \times d$ identity matrix):

$$
\begin{aligned}
[\![S]\!] = & \left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \mathbf{I}_4 \right) \cdot (\mathbf{I}_2 \otimes \mathbf{E}_{ab}) + \\
& + \left(\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \mathbf{I}_4 \right) \cdot (\mathbf{I}_2 \otimes \mathbf{E}_{ac}) + \\
& + \left(\begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix} \otimes \mathbf{E}_{bd} \right) + \\
& + \left(\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \otimes \mathbf{E}_{cd} \right) + \\
& + \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \mathbf{E}_{dd} \right) = \\
& = \left(\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right) \cdot \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \right) + \\
& + \left(\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \right) \cdot \left(\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \right) +
\end{aligned}
$$

$$+\left(\begin{pmatrix}1&0\\1&0\end{pmatrix}\otimes\begin{pmatrix}0&0&0&0\\0&0&0&1\\0&0&0&0\\0&0&0&0\end{pmatrix}\right)+\left(\begin{pmatrix}0&1\\0&1\end{pmatrix}\otimes\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&1\\0&0&0&0\end{pmatrix}\right)+$$

$$+\left(\begin{pmatrix}1&0\\0&1\end{pmatrix}\otimes\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{pmatrix}\right)=$$

$$=\left(\begin{pmatrix}1&0\\0&0\end{pmatrix}\otimes\begin{pmatrix}0&1&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\right)+\left(\begin{pmatrix}0&0\\0&1\end{pmatrix}\otimes\begin{pmatrix}0&0&1&0\\0&0&0&0\\0&0&0&0\\0&0&0&0\end{pmatrix}\right)+$$

$$+\left(\begin{pmatrix}1&0\\1&0\end{pmatrix}\otimes\begin{pmatrix}0&0&0&0\\0&0&0&1\\0&0&0&0\\0&0&0&0\end{pmatrix}\right)+\left(\begin{pmatrix}0&1\\0&1\end{pmatrix}\otimes\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&1\\0&0&0&0\end{pmatrix}\right)+$$

$$+\left(\begin{pmatrix}1&0\\0&1\end{pmatrix}\otimes\begin{pmatrix}0&0&0&0\\0&0&0&0\\0&0&0&0\\0&0&0&1\end{pmatrix}\right)$$

The collecting semantics results therefore in the following stochastic matrix:

$$[\![S]\!]=\begin{array}{cl}\begin{pmatrix}0&1&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&1&0&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&0&1\end{pmatrix} & \begin{array}{l}\ldots\ \langle x=0,[\mathtt{x}==0]^a\rangle\\ \ldots\ \langle x=0,[\mathtt{x}\leftarrow 0]^b\rangle\\ \ldots\ \langle x=0,[\mathtt{x}\leftarrow 1]^c\rangle\\ \ldots\ \langle x=0,[\mathbf{stop}]^d\rangle\\ \ldots\ \langle x=1,[\mathtt{x}==0]^a\rangle\\ \ldots\ \langle x=1,[\mathtt{x}\leftarrow 0]^b\rangle\\ \ldots\ \langle x=1,[\mathtt{x}\leftarrow 1]^c\rangle\\ \ldots\ \langle x=1,[\mathbf{stop}]^d\rangle\end{array}\end{array}$$

3.2 Probabilistic Abstract Interpretation

In order to construct abstract DFA equations for a **pWhile** program out of the collecting semantics we need a way to abstract the semantics. A quantitative approach for obtaining DFA equations is provided by our framework of Probabilistic Abstract Interpretation (PAI) [10,11]. This is a general framework for the probabilistic analysis of (probabilistic) programs which recasts classical Abstract Interpretation [3,4] in a vector space setting. The basic idea is to define abstractions as some kinds of linear operators on some appropriate vector spaces. More precisely, domains (both abstract and concrete) are Hilbert spaces that is vector spaces with inner product. The crucial notion of concretisation is expressed in this framework via a notion of generalised inverse of a linear operator, and in particular the notion of *Moore-Penrose pseudo-inverse* (MP). The properties of this kind of linear inverse makes the pair of a linear operator and its MP into an adjunction similar to the classical notion of *Galois connection.*

Let $\mathcal{C}$ and $\mathcal{D}$ be two Hilbert spaces and $\mathbf{A} : \mathcal{C} \mapsto \mathcal{D}$ a bounded linear map between them. A bounded linear map $\mathbf{A}^\dagger = \mathbf{G} : \mathcal{D} \mapsto \mathcal{C}$ is the Moore-Penrose pseudo-inverse of $\mathbf{A}$ iff

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

where $\mathbf{P}_A$ and $\mathbf{P}_G$ denote orthogonal projections (i.e. $\mathbf{P}_A^* = \mathbf{P}_A = \mathbf{P}_A^2$ and $\mathbf{P}_G^* = \mathbf{P}_G = \mathbf{P}_G^2$ where $.^*$ denotes the *adjoint* [18, Ch 10]) onto the ranges of $\mathbf{A}$ and $\mathbf{G}$.

A *probabilistic abstract interpretation* is defined by a pair of linear maps, $\mathbf{A} : \mathcal{C} \mapsto \mathcal{D}$ and $\mathbf{G} : \mathcal{D} \mapsto \mathcal{C}$, between the concrete domain $\mathcal{C}$ and the abstract domain $\mathcal{D}$, such that $\mathbf{G}$ is the Moore-Penrose pseudo-inverse of $\mathbf{A}$, and vice versa.

As shown in [10] any classical abstract interpretation can be seen as a probabilistic abstract interpretation by lifting the classical domains into probabilistic domains (Hilbert spaces). Another approach to the analysis of probabilistic programs applies classical Abstract Interpretation techniques to probabilistic semantics [15]. This approach corresponds to a special case of classical abstract interpretation and it therefore results in safe, i.e. worst case analyses. As shown in [9], the more general PAI approach allows us to construct additionally statistical information which is more in the spirit of an average case analysis.

Let $\mathbf{T}$ be a linear operator on some Hilbert space $\mathcal{V}$ representing the probabilistic semantics of the concrete program, and let $\mathbf{A} : \mathcal{V} \mapsto \mathcal{W}$ be a linear abstraction function with $\mathbf{G} = \mathbf{A}^\dagger$. An abstract semantics can then be defined on $\mathcal{W}$ as:

$$\mathbf{T}^\# = \mathbf{GTA}.$$

This so-called *induced semantics* is guaranteed to be the best correct approximation of the concrete semantics for classical abstractions based on Galois connections. In the linear space based setting of PAI where the order of the classical domains is replaced by some notion of metric distance, the induced abstract semantics is the *closest* one to the concrete semantics [10].

4 Parity Analysis

The first example of an analysis based on the LOS of **pWhile** concerns the parity of variables. Although, it is arguable whether this analysis by itself is of any practical use it is certainly a quite useful example in order to illustrate the basic methodology.

The purpose of the **Parity** Analysis is to determine at every program point whether a variable is *even* or *odd*. It is a simple forward analysis where the concrete value of variables is abstracted to just their parity information.

We will consider two simple programs which compute the factorial and twice the factorial of n which is left in m when the programs terminate.

```
[$m \leftarrow 1$]$^1$;                    [$m \leftarrow 2$]$^1$;
while [$n > 1$]$^2$ do                 while [$n > 1$]$^2$ do
    [$m \leftarrow m \times n$]$^3$;                [$m \leftarrow m \times n$]$^3$;
    [$n \leftarrow n - 1$]$^4$                  [$n \leftarrow n - 1$]$^4$
end while                          end while
[stop]$^5$                           [stop]$^5$
```

Without going into too many details, it should be obvious that a classical **Parity** Analysis should detect that the parity of $m = 2 \times n!$ at the end of the second program is always *even* even when the original value (and parity) of n is "unknown". A safe classical analysis of the first program will however fail in the sense that it will return "unknown" or $\top$ for the parity of m at the end of the program. This is in some sense rather unsatisfactory as it is obvious that $m = n!$ is "nearly always" *even.* Only in the "rare" case that the initial n is less than or equal to 1 will m be odd at the end of the program. The purpose of a probabilistic program analysis is a formal derivation of this intuition about the parity of m when the program terminates.

The idea is to (re)construct the abstract LOS of a program in the same way as the concrete LOS with the only difference that in the abstract case (some of the) operators which implement tests and assignments are replaced by their abstract versions. The general scheme for this program is given by:

$$\begin{aligned}\mathbf{T} = &\ \mathbf{M}_i \otimes \mathbf{I} \otimes \mathbf{E}_{1,2}\\ &+ \mathbf{T}_\top(\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{E}_{2,3})\\ &+ \mathbf{T}_\perp(\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{E}_{2,5})\\ &+ \mathbf{A} \otimes \mathbf{E}_{3,4}\\ &+ \mathbf{I} \otimes \mathbf{N} \otimes \mathbf{E}_{4,2}\\ &+ \mathbf{I} \otimes \mathbf{I} \otimes \mathbf{E}_{5,5}\end{aligned}$$

where the first factor, i.e. first variable, represents m and the second one n. By a slight abuse of notation we use the same symbol $\mathbf{I}$ for identities of different dimensions. The last component in each contribution $\mathbf{E}_{ij}$ is a 5×5 matrix which represents the control-flow steps.

The first term specifies the dynamics at program point 1: m is initialised using an operator $\mathbf{M}_i$ with $i = 1, 2$ – depending which of the two programs we consider –

of the form:

$$\mathbf{M}_1 = \begin{pmatrix} 0 & 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad \mathbf{M}_2 = \begin{pmatrix} 0 & 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

which implement a change of m's value to 1 and 2, respectively. The variable n stays unchanged, represented by the identity $\mathbf{I}$ and control is transfered from statement 1 to statement 2.

The next two terms deal with the test $[n > 1]^2$: The actual effect is just a control transfer with no changes to the two variables. However, depending on the test's result we get different control-flow transfers. The test operators are realised as:

$$\mathbf{T}_\top = \mathbf{I} \otimes \begin{pmatrix} 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 1 & 0 & \dots \\ 0 & 0 & 0 & 1 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \otimes \mathbf{I} \qquad \mathbf{T}_\perp = \mathbf{I} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 & \dots \\ 0 & 1 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ 0 & 0 & 0 & 0 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \otimes \mathbf{I}$$

$\mathbf{T}_\top$ "filters" all those configurations where n is $2, 3, \dots$ while $\mathbf{T}_\perp$ "blocks" all configurations except where n is equal to 0 or 1. In both cases the value of m and the current position in the program are irrelevant; this is indicated by the identity operators used to define the test operators.

The fourth term represents the most complicated arithmetic operation namely the statement $[m \leftarrow m \times n]^3$. The changes to m and n are rather complicated so that $\mathbf{A}$, the operator describing the effects on m and n, is given as an "entangled" operator of the form:

$$\mathbf{A} = \sum_{i,j} ((\mathbf{I} \otimes \mathbf{E}_{j,i \cdot j} \otimes \mathbf{I})(\mathbf{T}_i \otimes \mathbf{T}_j \otimes \mathbf{I}))$$

with $\mathbf{T}_k = \mathbf{E}_{k,k}$ for all k. This can be interpreted as follows: For all pairs (m, n) we first test whether m and n have certain values i and j, respectively (ignoring the current statement label), then we perform the update by leaving n (and the current statement label) unchanged but changing the value of m from j to its new value $i \cdot j$.

The penultimate term in the definition of $\mathbf{T}$ results in the decrement of n via the (shift) operator:

$$\mathbf{N} = \begin{pmatrix} 0\ 0\ 0\ 0 \ldots \\ 1\ 0\ 0\ 0 \ldots \\ 0\ 1\ 0\ 0 \ldots \\ 0\ 0\ 1\ 0 \ldots \\ \vdots\ \vdots\ \vdots\ \vdots\ \ddots \end{pmatrix}$$

and the last term just specifies the terminal configuration $[\mathbf{stop}]^5$ as an infinite loop which leave m and n unchanged.

Concrete Semantics

Both programs have five program points, which means that the control transition matrices are 5×5 matrices. The state of the two variables are in principle represented by infinite dimensional vectors. However, if we fix a maximal input size $n \leq N$ then we can also restrict the maximum size of m. For this finite approximation we need a vector in $\mathcal{V}(\{0, 1, \ldots, N!\}) \otimes \mathcal{V}(\{0, 1, \ldots, N\}) \otimes \mathcal{V}(\{1, \ldots, 5\})$, i.e. a $5 \cdot N \cdot N!$ dimensional vector. The concrete (finite) semantics is thus represented by a matrix of the same dimensions. Though sparse, it is for $N = 10$ already an about 18 million $\times$ 18 million matrix.

Abstract Semantics

In order to obtain an abstract semantics we can abstract both variables, i.e. "execute" the program with m and n being classified as *even* or *odd*. Alternatively, we can abstract only m and use the concrete values for the control variable n (the **while**-loop here is actually a **for**-loop). In both cases the dimension of $\mathbf{T}$ is drastically reduced. It is $2 \cdot 2 \cdot 5 = 20$ in the first case, and $2 \cdot N \cdot 5 = 10N$ in the second case – for $N = 10$ we get a 100×100 matrix to represent $\mathbf{T}^\#$ instead of the 18 million in the concrete case.

To be more precise: We construct the abstract semantics $\mathbf{T}^\#$ simply as

$$\begin{aligned}\mathbf{T}^\# = &\, \mathbf{M}_i^\# \otimes \mathbf{I} \otimes \mathbf{E}_{1,2} \\ &+ \mathbf{T}_\top^\#(\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{E}_{2,3}) \\ &+ \mathbf{T}_\bot^\#(\mathbf{I} \otimes \mathbf{I} \otimes \mathbf{E}_{2,5}) \\ &+ \mathbf{A}^\# \otimes \mathbf{E}_{3,4} \\ &+ \mathbf{I} \otimes \mathbf{N}^\# \otimes \mathbf{E}_{4,2} \\ &+ \mathbf{I} \otimes \mathbf{I} \otimes \mathbf{E}_{5,5}\end{aligned}$$

i.e. exactly in the same way as the concrete one. The difference is exhibited by only few of the operators (and the fact that the identities have a much reduced dimension). The control-flow steps are not changed at all as we are interested here only in a data-flow analysis. Each of the operators is constructed using the *parity* *abstraction* operator $\mathbf{P}$:

$$\mathbf{P}^T = \begin{pmatrix} 0\ 1\ 0\ 1\ 0\ \dots \\ 1\ 0\ 1\ 0\ 1\ \dots \end{pmatrix}$$

and its Moore-Penrose Pseudo Inverse $\mathbf{P}^\dagger$. For example: $\mathbf{M}_i^\# = \mathbf{P}^\dagger \mathbf{M}\mathbf{P}$, $\mathbf{A}^\# = (\mathbf{P}^\dagger \otimes \mathbf{P}^\dagger)\mathbf{A}(\mathbf{P} \otimes \mathbf{P})$, etc. We thus get (if we abstract both variables):

$$\mathbf{M}_1^\# = \begin{pmatrix} 0\ 1 \\ 0\ 1 \end{pmatrix} \quad \mathbf{M}_2^\# = \begin{pmatrix} 1\ 0 \\ 1\ 0 \end{pmatrix} \quad \mathbf{N}^\# = \begin{pmatrix} 0\ 1 \\ 1\ 0 \end{pmatrix} \quad \mathbf{A}^\# = \begin{pmatrix} 0\ 1 \\ 1\ 0 \end{pmatrix}$$

The abstract test operators $\mathbf{T}_\top^\#$ and $\mathbf{T}_\perp^\#$ depend on the finite approximation we consider. If we do not abstract the control variable n we need a more complicated $\mathbf{A}^\#$ but the test operators $\mathbf{T}_\top^\#$ and $\mathbf{T}_\perp^\#$ as well as $\mathbf{N}^\#$ are the same as for the concrete semantics.

Numerical Results

It is rather straightforward to implement the finite approximations of the concrete as well as the abstract semantics using a numerical program like `octave` [12]. As to be expected the size, even using sparse matrix representations, is prohibitively large; while for $N = 5$ things still work out it is effectively impossible to deal with case that $N = 10$. The abstract semantics creates no such problem.

If we abstract m to *even* and *odd* but leave n a concrete control variable, we can "execute" the (abstract) program for example in an initial configuration where n is represented by a uniform distribution over all possible input values for $n < N$ and m has a 50 : 50 chance of being *even* and *odd*. Depending on the cut-off point N we obtain for $N = 10$ a 20%, for $N = 100$ a 2%, and for $N = 1000$ a 0.2% chance that m is *odd* at the end of the program. As expected, this reflects exactly the fact that m "most likely" will be even in the end, the more so the larger the maximal input for n is.

5 Live Variable Analysis

A more interesting static program analysis is the classical **Live Variable** (*LV*) Analysis, see e.g. [16]. It is an example of a backward analysis, i.e. the control flow has to be reversed. In order to do this we use the transposes of the original control-transfer matrices.

The property we are interested in is whether a variable at every program point is *live* or *dead* (at the exit of a program point), i.e. if there is a possible use for a variable at a later stage of the program's execution. If, for example, there is an assignment to a program variable which is never used then we can eliminate this assignment, i.e. perform a *dead code elimination*. The aim of a probabilistic *LV* Analysis is to provide estimates for the probabilities that a certain variable is later used. A possible application could be to support *caching*, variables which are more likely to be used could be kept in a 'faster' cache, while variables which have a very low probability of being *live* would be kept in a 'slower' cache or the main memory.

The idea is, as before, to replace the operators which define the concrete semantics (describing the precise changes of the values of variables) with operators which only record whether a variable is used (in a test b or an expression a on the right hand side of an assignment) or being redefined (it appears on the left hand side of an assignment). This is essentially what also happens in the classical *LV* Analysis: if `x` is on the right hand side of an assignment it gets "killed", if it is a (free) variable in a or b then it becomes alive (gets "generated"). The semantics of all other statements does not change the "liveliness" of a variable.

5.1 Intra-Statement Updates

The information we record about every variable is just if it is alive or not, i.e. we need to consider for every variable an abstract state in $\mathcal{V}(\{live, dead\})$. The update can be specified in a very similar way as in the case of the classical analysis (following e.g. [16]), where the local transfer function is usually defined by means of two auxiliary functions *kill* and *gen* returning for each label the set of variables which are dead and alive, respectively. We can define kill and gen operators as:

$$\mathbf{K} = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \qquad \mathbf{G} = \begin{pmatrix} 1 & 0 \\ 1 & 0 \end{pmatrix}.$$

By using these operators we can represent tests and assignments by means of the following abstract operators:

$$[\![[b]^\ell]\!] = \mathbf{V}(b) \otimes \mathbf{N}^T \quad \text{and}$$

$$[\![[x_i := a]^\ell]\!] = \left((\bigotimes_{j=1}^{i-1} \mathbf{I} \otimes \mathbf{K} \otimes \bigotimes_{j=i+1}^{n} \mathbf{I}) \cdot \mathbf{V}(a) \right) \otimes \mathbf{N}^T,$$

where $\mathbf{N}^T$ is the transposed of the control-flow matrix, and $\mathbf{V}(e)$ identifies the (free) variables in a Boolean or arithmetic expression e:

$$\mathbf{V}(e) = \bigotimes_{j=1}^{n} \mathbf{V}_i(e) \quad \text{with} \quad \mathbf{V}_i(e) = \begin{cases} \mathbf{G} \text{ if } x_i \in FV(e) \\ \mathbf{I} \text{ otherwise.} \end{cases}$$

Note, that we could obtain these operators by abstracting the concrete ones but for brevity's sake we based our definitions directly on the classical analysis (as presented in [16]).

5.2 Inter-Statement Updates

The problem which we still have to resolve is what happens at "confluence points", i.e. when several (backward) control-flows come together as at a test in an **if-** or **while**-statement. To do this we need additional information about the branching probabilities. This could be obtained (i) experimentally from profiling or (ii) from the concrete semantics, by not abstracting at least those variables which determine the control-flow (as in the previous example) or (iii) via another abstract semantics which estimates these probabilities. In the following we will sketch the basic elements of an analysis which is based on the last option.

To illustrate the basic aims and structure of a probabilistic version of the *LV* Analysis consider the following two programs:

```
[skip]^1                        [y ← 2 × x]^1
if [odd(y)]^2 then              if [odd(y)]^2 then
    [x ← 1]^3                       [x ← 1]^3
else                            else
    [y ← 1]^4                       [y ← 1]^4
end if                          end if
[y ← x]^5                       [y ← x]^5
```

The classical *LV* Analysis of both programs, starting with $\mathsf{LV}_{exit}(5) = \emptyset$, results, among other things, in the following description of the live variables at the beginning of the test in the second statement:

$$\mathsf{LV}_{entry}(2) = \{x, y\}.$$

This indicates the fact that both variables x and y might be alive at the test point. This is a conservative result. It is possible that in a concrete execution x is actually not alive: although it will be needed at label 5, it will be 'killed' (by redefining its value) if the test results fails, i.e. if y is even. In the first program, we can't make too many assumptions about the probability that the test succeeds, but a closer inspection of the second program easily confirms that indeed y is always even, although the lack of knowledge about x makes it impossible to know the concrete value of y.

A reasonable result of a probabilistic *LV* analysis we are aiming for could be:

$$\mathsf{LV}_{entry}(2) = \{\langle x, \frac{1}{2}\rangle, \langle y, 1\rangle\}$$

for the first program and

$$\mathsf{LV}_{entry}(2) = \{\langle y, 1\rangle\}$$

for the second one. In the first case we would expect that a good estimate for the branching probabilities is given by 50 : 50 chance that the test succeeds or fails. This is a consequence that (without any further assumptions) the chances that y is *even* or *odd* are the same. In the second program we can guarantee via a reasonable parity analysis that y is always even, despite the fact that the exact value of y is completely unknown.

As with all probabilistic versions of classical analyses we could (re)construct the classical result from the probabilistic version by just recording the *possibilities* instead of *probabilities*. For this we only have to define a forgetful map from $\mathcal{V}(X)$ to $\mathcal{P}(X)$ which just considers the support of a probability distribution, i.e. those elements in X which are associated with a non-zero probability.

5.3 Estimating the Branching Probabilities

A formal estimation of the branching probabilities for **if**-statements (and similarly for **while**-loops) follows the following scheme: Perform a first phase analysis (e.g. a parity analysis) to determine the branching probabilities, then use these estimates to perform the actual analysis. This means in effect that we replace tests b with probabilistic choices where the choice probabilities $p_\top$ and $p_\bot$ are determined by a first phase analysis.

```
[skip]^1
[choose]^2
    $p_\top$ : [$x \leftarrow 1$]^3
or
    $p_\bot$ : [$y \leftarrow 1$]^4
[$y \leftarrow x$]^5
```

```
[$y \leftarrow 2 \times x$]^1
[choose]^2
    $p_\top$ : [$x \leftarrow 1$]^3
or
    $p_\bot$ : [$y \leftarrow 1$]^4
[$y \leftarrow x$]^5
```

As can be seen in this example our approach makes it necessary that “abstract” programs are eventually probabilistic ones, even when the “concrete” programs were deterministic.

Once we have determined the choice probabilities $p_\top$ and $p_\bot$ the actual *LV* Analysis is performed in essentially the same way as in the classical case, with the exception that we have to deal with probability distributions over variables (indicating the chances that a variable is live at a certain program point) instead of sets of potentially live variables. In order to combine information about live variables at “confluence” points we then use weighted sums or linear combinations utilising the choice probabilities $p_\top$ and $p_\bot$ instead of set union as in the classical case.

6 Conclusions

In this paper we presented the basic elements of syntax-based probabilistic data-flow analyses constructed via Probabilistic Abstract Interpretation which replaces the standard Cousot & Cousot approach for a possibilistic analysis.

We illustrated this framework using a small imperative language with a probabilistic choice construct, namely **pWhile**. For this language we presented a *collecting semantics* which essentially constructs the generator of a Discrete Time Markov Chain implementing the *operational semantics* of a **pWhile** program. This collecting semantics was defined “compositionally” using linear combinations (sums) and the tensor product in order to represent states and state transformations. Exploiting this structure we could then define an abstract semantics as the basis of our analyses. This kind of “compositional” analysis was possible because sum and tensor product distribute over probabilistic abstractions.

While it is straightforward to lift analyses like the **Parity** Analysis to a probabilistic version, other analyses like the **Live Variable** Analysis require a two-phase analysis where we first estimate the branching probabilities before the actual analysis takes place. Further work will look into the issue of optimal abstractions, i.e.

ways to construct abstractions of in particular control variables in order to obtain optimal estimates for these branching probabilities. Alternatively, for sub-optimal abstractions, we are also interested in determining the error margins of these estimates. Finally, we plan to extend this approach also to higher order, e.g. functional, languages and to implement a prototype of an automatic analyser.

References

[1] Bause, F. and P. Kritzinger, "Stochastic Petri Nets – An Introduction to the Theory," Vieweg Verlag, 2002, second edition.

[2] Bergstra, J., A. Ponse and S. Smolka, editors, "Handbook of Process Algebra," Elsevier Science, Amsterdam, 2001.

[3] Cousot, P. and R. Cousot, *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*, in: *Proceedings of POPL'77*, Los Angeles, 1977, pp. 238–252.

[4] Cousot, P. and R. Cousot, *Systematic Design of Program Analysis Frameworks*, in: *Proceedings of POPL'79*, San Antonio, Texas, 1979, pp. 269–282.

[5] Cousot, P. and R. Cousot, *Systematic design of program transformation frameworks by abstract interpretation*, in: *Proceedings of POPL'02* (2002), pp. 178–190.

[6] den Hartog, J. and E. de Vink, *Verifying probabilistic programs using a Hoare-like logic*, International Journal of Foundations of Computer Science **13** (2002), pp. 315–340.

[7] Di Pierro, A., C. Hankin and H. Wiklicky, *Quantitative relations and approximate process equivalences*, in: R. Amadio and D. Lugiez, editors, *Proceedings of CONCUR'03*, Lecture Notes in Computer Science **2761** (2003), pp. 508–522.

[8] Di Pierro, A., C. Hankin and H. Wiklicky, *Measuring the confinement of probabilistic systems*, Theoretical Computer Science **340** (2005), pp. 3–56.

[9] Di Pierro, A., C. Hankin and H. Wiklicky, *Abstract interpretation for worst and average case analysis*, in: T. Reps, M. Sagiv and J. Bauer, editors, *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, LNCS 4444, Springer-Verlag, 2007 pp. 160–174.

[10] Di Pierro, A. and H. Wiklicky, *Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation*, in: *Proceedings of PPDP'00* (2000), pp. 127–138.

[11] Di Pierro, A. and H. Wiklicky, *Measuring the precision of abstract interpretations*, in: *Proceedings of LOPSTR'00*, Lecture Notes in Computer Science **2042** (2001), pp. 147–164.

[12] Eaton, J., "Gnu Octave Manual," www.octave.org, 2002.

[13] Jonsson, B., W. Yi and K. Larsen, "Probabilistic Extensions of Process Algebras," Elsevier Science, Amsterdam, 2001 pp. 685–710, see [2].

[14] Kozen, D., *Semantics for probabilistic programs*, Journal of Computer and System Sciences **22** (1981), pp. 328–350.

[15] Monniaux, D., *Abstract interpretation of probabilistic semantics*, in: *Proceedings of SAS'00*, Lecture Notes in Computer Science **1824** (2000), pp. 322–339.

[16] Nielson, F., H. R. Nielson and C. Hankin, "Principles of Program Analysis," Springer Verlag, Berlin – Heidelberg, 1999.

[17] Plotkin, G., *A structural approach to operational semantics*, Journal of Logic and Algebraic Programming **60-61** (2004), pp. 17–139.

[18] Roman, S., "Advanced Linear Algebra," Graduate Texts in Mathematics **135**, Springer Verlag, 2005, second edition.

[19] Tijms, H., "Stochastic Models – An Algorithmic Approach," John Wiley & Sons, Chichester, 1994.

[20] van Glabbeek, R., S. Smolka and B. Steffen, *Reactive, generative and stratified models of probabilistic processes*, Information and Computation **121** (1995), pp. 59–80.