A Systematic Approach to Probabilistic Pointer Analysis

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

[1] University of Verona, Ca' Vignal 2 - Strada le Grazie 15 I-37134 Verona, Italy
[2] Imperial College London, 180 Queen's Gate London SW7 2AZ, UK

Abstract. We present a formal framework for syntax directed probabilistic program analysis. Our focus is on probabilistic pointer analysis. We show how to obtain probabilistic points-to matrices and their relational counterparts in a systematic way via Probabilistic Abstract Interpretation (PAI). The analysis is based on a non-standard semantics for a simple imperative language which corresponds to a Discrete-Time Markov Chain (DTMC). The generator of this DTMC is constructed by composing (via tensor product) the probabilistic control flow of the program and the data updates of the different variables at individual program points. The dimensionality of the concrete semantics is in general prohibitively large but abstraction (via PAI) allows for a drastic (exponential) reduction of size.

1 Introduction

We investigate a theoretical framework for the systematic construction of a *syntax directed* probabilistic program analysis. We will illustrate our approach based on a simple *imperative* language, **While**, and its probabilistic extension **pWhile**. Our focus is on the systematic derivation of a *probabilistic pointer analysis* within a static memory model, i.e. we do not have dynamically created objects (on a heap). These restrictions are mainly for presentational reasons and we are confident that our basic ideas also apply to other programming paradigms, e.g. object oriented, dynamic objects on a heap, etc.

Our aim is to introduce a *static* analysis, which could provide an alternative to experimental approaches like profiling. As the analysis is probabilistic we were lead to consider a probabilistic language from the start. However, this language subsumes the usual deterministic **While** and our approach thus applies as well to deterministic as to probabilistic programs. It is important to note that even for deterministic programs the analysis gives in general probabilistic results.

The main novel contributions of this study concern the following aspects. (i) We present for the first time a syntax directed construction of the generator of a Markov chain representing the concrete semantics of a **pWhile** program with static pointers. This exploits a tensor product representation which has been studied previously in areas like performance analysis. (ii) Although the concrete semantics is well-defined and based on highly sparse matrices, the use of the tensor product still leads to an exponential increase in its size, i.e. the dimension of

the generator matrices. In order to overcome this problem we apply Probabilistic Abstract Interpretation (PAI) – presented first in [1] (see also [2]) – to construct an abstract semantics, i.e. a static program analysis. The fact that PAI is compatible with the tensor product operation allows us to introduce a compositional construction not only of the concrete but also of the abstract semantics with a drastically reduced size. (iii) This leads to a systematic and formal analysis of (dynamic) *branching probabilities* – usually obtained experimentally via profiling or based on various heuristics (see e.g. [3, 4] and references therein) – on the basis of abstracted test operators. (iv) Within our framework, we are also able to construct so-called *points-to matrices* which are commonly employed in probabilistic pointer analysis. Furthermore, we will discuss the alternative concept of a *points-to tensor* which provides more precise relational information.

A possible application area of this type of program/pointer analysis could be *speculative optimisation* which recently is gaining importance – not least in modern multi-core environments. The idea here is (i) to execute code "speculatively" based on some *guess*; (ii) then to test at a later stage whether the *guess* was correct; (iii) if it was correct the computation continues, otherwise some repair code is executed. This guarantees that the computation is always correctly performed. The only issue concerns the (average) costs of "repairs". If the *guess* is correct sufficiently often, the execution of the repair code for the cases where it is wrong is amortised. Thus, in the speculative approach the compiler can take advantage of *statistical* (rather than only definite) information when choosing whether to perform an optimisation. The analysis we present would allow for a better exploitation of the 'maybe' case than with classical conservative compiler optimisation, replacing a *possibilistic* analysis by a *probabilistic* one.

2 Probabilistic While

We extend the probabilistic **While** language in [5, 6] with pointer expressions. In order to keep our treatment simple we allow only for pointers to (existing) variables, i.e. we will not deal with dynamically created objects on a heap.

A program P is made up from a possibly empty list of variable declarations D followed by a statement S. Formally, $P ::= D;\ S \mid S$ with $D ::= d;\ D \mid d$. A declaration d fixes the types of the program's variables. Variables can be either basic Boolean or integer variables or pointers of any order $r = 0, 1, \ldots$, i.e.:

$$d ::= \mathtt{x} : t \qquad t ::= \mathbf{int} \mid \mathbf{bool} \mid *^r t$$

(where we identify $*^0 t$ with t).

The syntax of statements is as follows:

$$\begin{aligned} S ::= {} & \mathbf{skip} \mid \mathbf{stop} \mid p \leftarrow e \mid S_1;\ S_2 \mid \mathbf{choose}\ p_1 : S_1\ \mathbf{or}\ p_2 : S_2 \\ & \mid \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end\ if} \mid \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{end\ while} \end{aligned}$$

In the **choose** statement we allow only for constant probabilities p_i and assume w.l.o.g. that they are normalised, i.e. add up to 1. A pointer expression is a

variable prefixed with r dereferencing $*$'s:

$$p \ ::= \ *^r\mathtt{x} \text{ with } \mathtt{x} \in \mathbf{Var}.$$

We allow for higher order pointers, i.e. for pointers to pointers to .... Expressions e are of different types, namely arithmetic expressions a, Boolean expressions b and locality expressions (or addresses) l. Formally: $e \ ::= \ a \mid b \mid l$. Arithmetic expressions are of the form $a \ ::= \ n \mid p \mid a_1 \odot a_2$ with $n \in \mathbb{Z}$, p a pointer to an integer, and '$\odot$' representing one of the usual arithmetic operations '$+$', '$-$', or '$\times$'. Boolean expressions are defined by $b \ ::= \ \text{TRUE} \mid \text{FALSE} \mid p \mid \neg b \mid b_1 \vee b_2 \mid b_1 \wedge b_2 \mid a_1 \lessapprox a_2$, where p is a pointer to a Boolean variable. The symbol '$\lessapprox$' denotes one of the standard comparison operators for arithmetic expressions, i.e. $<, \leq, =, \neq, \geq, >$. Addresses or locality expressions are: $l \ ::= \ \mathtt{nil} \mid \&p \mid p$.

The semantics of **pWhile** with pointers follows essentially the standard one for **While** as presented, e.g., in [7]. The only two differences concern (i) the probabilistic choice and (ii) the pointer expressions used in assignments. The operational semantics is given as usual via a transition system on configurations $\langle S, \sigma \rangle$, i.e. pairs of statements and states. To allow for probabilistic choices we label these transitions with probabilities. Except for the **choose** construct these probabilities will always be 1 as all other statements in **pWhile** are deterministic.

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

In the assignments we allow for general pointer expressions (not just basic variables) on the left as well as the right hand side. In order to give a semantics to assignments we therefore need to identify the actual variable a pointer expression on the left hand side of an assignment is referring to. This is achieved via the function $[\![.]\!]$ from $\mathbf{Pointer} \times \mathbf{State}$ into **Var**, where $\mathbf{Pointer} = \{*^r\mathtt{x} \mid \mathtt{x} \in \mathbf{Var}, r = 0, 1, \ldots\}$ denotes the set of all pointer expressions, defined as follows:

$$[\![\mathtt{x}]\!]\sigma = \mathtt{x} \qquad [\![*^r\mathtt{x}]\!]\sigma = [\![*^{r-1}\mathtt{y}]\!]\sigma \text{ if } \sigma(\mathtt{x}) = \&\mathtt{y}.$$

In other words, if we want to find out what variable a pointer expression refers to we either could have the case that $p = \mathtt{x}$ – in which case the reference is immediately to this variable – or $p = *^r\mathtt{x}$ – in which case we have to dereference the pointer to determine the variable $\mathtt{y}$ it points to in the current state σ (via $\sigma(\mathtt{x})$). If further dereferencing is needed we continue until we end up with a basic variable. The variable we finally reach this way might still be a pointer, i.e. contain a location rather than a constant, but we only dereference as often as required, i.e. r times. We also do not check whether further dereferencing is possible, i.e. whether $\mathtt{x}$ had been declared a pointer variable of a higher order than r – we assume that either a simple type system rejects malformed programs at compile time, or that the run-time environment raises an exception if there is a violation.

The expressions a, b, and l (on the right hand side of assignments or as tests in **if** and **while** statements) evaluate to values of type $\mathbb{Z}$, $\mathbb{B}$ and $\mathbb{L}$ in the usual

way. The only extension to the standard semantics is caused again when pointers p are part of the expressions. In order to treat this case correctly, we need to determine first the actual variable a pointer refers to and then obtain the value this variable contains in the current state σ. Again, we do not cover here any type checking, i.e. whether the variable contains ultimately a value of the correct type. If we denote by **Expr** the set of all expressions e then the evaluation function $\mathcal{E}(.)$. is a function from $\mathbf{Expr} \times \mathbf{State}$ into $\mathbb{Z}+\mathbb{B}+\mathbb{L}$. The semantics of arithmetic and Boolean expressions is standard. A new kind of expressions return locations in $\mathbb{L}$, either the constant NIL or the address contained in a variable a pointer p refers to, or the address of a variable which a pointer refers to. For these new expressions we define $\mathcal{E}(.)$. as follows:

$$\mathcal{E}(\texttt{nil})\sigma = \text{NIL} \qquad \mathcal{E}(p)\sigma = \sigma([\![p]\!]\sigma) \qquad \mathcal{E}(\&p)\sigma = \&([\![p]\!]\sigma).$$

Based on the functions $[\![.]\!]$. and $\mathcal{E}(.)$. the semantics of an assignment is given by

$$\langle p \leftarrow e, \sigma\rangle \longrightarrow_1 \langle \mathbf{stop}, \sigma[[\![p]\!]\sigma \mapsto \mathcal{E}(e)\sigma]\rangle.$$

The state σ stays unchanged except for the variable the pointer p is referring to (we obtain this information via $[\![.]\!]$.). The value of this variable is changed so that it now contains the value represented by the expression e. The rest of the SOS semantics of **pWhile** is quite standard and we will omit here a formal presentation.

3 Linear Operator Semantics

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

3.1 Probabilistic Control Flow

We base our construction on a probabilistic version of the *control flow* [7] or *abstract syntax* [8] of **pWhile** programs. The flow $\mathcal{F}(P)$ of a program P is based on a labelled version of P. Labelled programs follow the syntax:

$$\begin{aligned} S ::= {} & [\mathbf{skip}]^\ell \mid [\mathbf{stop}]^\ell \mid [p \leftarrow e]^\ell \mid S_1;\ S_2 \mid [\mathbf{choose}]^\ell\ p_1 : S_1\ \mathbf{or}\ p_2 : S_2 \\ & \mid \mathbf{if}\ [b]^\ell\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end\ if} \mid \mathbf{while}\ [b]^\ell\ \mathbf{do}\ S\ \mathbf{end\ while}. \end{aligned}$$

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

3.2 Concrete Semantics

The generator matrix of the DTMC which we will construct for any given **pWhile** program defines a linear operator – thus we refer to it as a *Linear Operator Semantics* (LOS) – on a vector space based on the labelled blocks and classical states of the program in question. In all generality, the (real) vector space $\mathcal{V}(S, \mathbb{R}) = \mathcal{V}(S)$ over a set S is defined as the formal linear combinations of elements in S which we can also see as tuples of real numbers x_s indexed by elements in S, i.e. $\mathcal{V}(X) = \{\langle x_s, s\rangle_{s\in S} \mid x_s \in R\} = \{(x_s)_{s\in S}\}$, with the usual (point-wise) algebraic operations, i.e. scalar multiplication and vector addition.

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

In order to keep the mathematical treatment as simple as possible we will exploit the fact that **Var** and thus $\mathbb{L}$ is finite for any given program. We furthermore restrict the actual range of integer variables to a finite sub-set $\underline{\mathbb{Z}}$ of $\mathbb{Z}$. Although such a finite restriction is somewhat unsatisfactory from a purely theoretical point of view, it appears to be justified in the context of static program analysis (one could argue that any "real world" program has to be executed on a computer with certain memory limitations). As a result, we can restrict our consideration to probability *distributions* on **State** rather than referring to the more general notion of probability *measures*. While in discrete, i.e. finite, probability spaces every measure can be defined via a distribution, the same does not hold any more for infinite state spaces, even for countable ones; it is, for example, impossible to define on the set of rationals in the interval $[0, 1]$ a kind of "uniform distribution" which would correspond to the Lebesgue measure.

State Space. As we consider only finitely many variables, $v = |\mathbf{Var}|$, we can represent the space of all possible states $\mathbf{Var} \to \underline{\mathbb{Z}} + \mathbb{B} + \mathbb{L}$ as the Cartesian product $(\underline{\mathbb{Z}} + \mathbb{B} + \mathbb{L})^v$, i.e. for every variable $\mathtt{x}_i \in \mathbf{Var}$ we specify its associated value in (a separate copy of) $\underline{\mathbb{Z}} + \mathbb{B} + \mathbb{L}$.

As the declarations of variables fix their types, in effect their possible range, we can exploit this information by presenting the (classical) state in a slightly more effective way: $\mathbf{State} = \mathbf{Value}_1 \times \mathbf{Value}_2 \ldots \times \mathbf{Value}_v$ with $\mathbf{Value}_i = \underline{\mathbb{Z}}$, or $\mathbb{B}$ or $\mathbb{L}$. We can go even a step further and exploit the fact that pointers have to refer to variables which represent pointers of a level lower than themselves, i.e. a simple first level pointer must refer to a simple variable, a second level pointer must refer to a first level pointer, etc.; only NIL can be used on all levels. Let us denote by $\mathbb{LZ}_0$ the set of integer variables (plus NIL), by $\mathbb{LB}_0$ the set of Boolean variables (plus NIL), and by $\mathbb{LZ}_l$ and $\mathbb{LB}_l$ the pointer variables which refer to variables in $\mathbb{LZ}_{l-1}$ and $\mathbb{LB}_{l-1}$ (plus NIL) respectively. Obviously, we have $\mathbb{L} = \bigcup_l \mathbb{LZ}_l \cup \bigcup_l \mathbb{LB}_l$.

As the declarations fix the level of a pointer (and its ultimate target type) we can represent the state in a slightly simpler way as $\mathbf{State} = \mathbf{Value}_1 \times \ldots \times \mathbf{Value}_v$ with $\mathbf{Value}_i = \underline{\mathbb{Z}}, \mathbb{B}, \mathbb{LZ}_l$ or $\mathbb{LB}_l$ with $l \geq 1$. We will use the following conventions for the representation of states and state vectors. Given v variables, we will enumerate them according to their pointer level, i.e. first the

basic variables, then the r_1 simple pointers, then the r_2 pointers to pointers, etc. We denote by $r = r_1 + r_2 + \ldots$ the number of all pointer variables. The last component of the state vector corresponds to the label ℓ.

The distributions which describe the probabilistic state of the execution of a program correspond to (normalised and positive) vectors in $\mathcal{V}(\mathbf{State})$. In terms of vector spaces, the above representation of the classical states can be expressed by means of the *tensor product* construction. We can construct the tensor product of two finite dimensional matrices (or vectors, seen as $1 \times n$ or $n \times 1$ matrices) via the so-called *Kronecker product*: Given an $n \times m$ matrix $\mathbf{A}$ and a $k \times l$ matrix $\mathbf{B}$ then $\mathbf{A} \otimes \mathbf{B}$ is the $nk \times ml$ matrix with entries $(\mathbf{A} \otimes \mathbf{B})_{(i_1-1)\cdot k+i_2,(j_1-1)\cdot l+j_2} = (\mathbf{A})_{i_1,j_1} \cdot (\mathbf{B})_{i_2,j_2}$. The representation of a state as a tuple in the Cartesian product of the sets of values for each variable can be re-formulated in our vector space setting by using the isomorphism $\mathcal{V}(\mathbf{State}) = \mathcal{V}(\mathbf{Value}_1 \times \ldots \times \mathbf{Value}_v) = \mathcal{V}(\mathbf{Value}_1) \otimes \ldots \otimes \mathcal{V}(\mathbf{Value}_v)$.

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

Consider a variable $\mathtt{x}$ together with the set of its possible values $\mathbf{Value} = \{v_1, v_2, \ldots\}$, and the vector space $\mathcal{V}(\mathbf{Value})$. The probabilistic state of the variable $\mathtt{x}$ can be described by a distribution over its possible values, i.e. a vector in $\mathcal{V}(\mathbf{Value})$. For example, if we know that $\mathtt{x}$ holds the value v_1 or v_3 with probabilities $\frac{1}{3}$ and $\frac{2}{3}$ respectively (and no other values) then this situation is represented by the vector $(\frac{1}{3}, 0, \frac{2}{3}, 0, \ldots)$. As we represent distributions by row vectors $\boldsymbol{x}$ the application of a linear map corresponds to a post-multiplication by the corresponding matrix $\mathbf{T}$, i.e. $\mathbf{T}(\boldsymbol{x}) = \boldsymbol{x} \cdot \mathbf{T}$.

We might need to apply a transformation $\mathbf{T}$ to the probabilistic state of the variable $\mathtt{x}_i$ only when a certain condition is fulfilled. We can express such a condition by a predicate q on $\mathbf{Value}_i$. Defining a diagonal matrix $\mathbf{P}$ with $(\mathbf{P})_{ii} = 1$ if $q(v_i)$ holds and 0 otherwise, allows us to "filter out" only those states which fulfill the condition q, i.e. $\mathbf{P} \cdot \mathbf{T}$ applies $\mathbf{T}$ only to those states.

Operators. The Linear Operator Semantics of **pWhile** is built using a number of basic operators which can be represented by the (sparse) square matrices: $(\mathbf{E}(m,n))_{ij} = 1$ if $m = i \wedge n = j$ and 0 otherwise, and $(\mathbf{I})_{ij} = 1$ if $i = j$ and 0 otherwise. The matrix units $\mathbf{E}(m,n)$ contains only one non-zero entry, and $\mathbf{I}$ is the identity operator. Using these basic building blocks we can define a number of "filters" $\mathbf{P}$ as depicted in Table 1. The operator $\mathbf{P}(c)$ has only one non-zero entry: the diagonal element $\mathbf{P}_{cc} = 1$, i.e. $\mathbf{P}(c) = \mathbf{E}(c,c)$. This operator extracts the probability corresponding to the c-th coordinate of a vector, i.e. for $\boldsymbol{x} = (x_i)_i$ the multiplication with $\mathbf{P}(c)$ results in a vector $\boldsymbol{x'} = \boldsymbol{x} \cdot \mathbf{P}(c)$ with only one non-zero coordinate, namely $x'_c = x_c$.

The operator $\mathbf{P}(\sigma)$ performs a similar test for a vector representing the probabilistic state of the computation. It filters the probability that the computation

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

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

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

$$\mathbf{U}(*^r \mathtt{x}_k \leftarrow e) = \sum_{\mathtt{x}_i} \mathbf{P}(\mathtt{x}_k = \&\mathtt{x}_i)\mathbf{U}(*^{r-1} \mathtt{x}_i \leftarrow e)$$

Table 1. Test and Update Operators for **pWhile**

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

The update operators (see Table 1) implement state changes. From an initial probabilistic state σ, i.e. a distribution over classical states, we get a new probabilistic state σ' via $\sigma \cdot \mathbf{U}$. The simple operator $\mathbf{U}(c)$ implements the deterministic update of a variable $\mathtt{x}_i$: Whatever the value(s) of $\mathtt{x}_i$ are, after applying $\mathbf{U}(c)$ to the state vector describing $\mathtt{x}_i$ we get a point distribution expressing the fact that the value of $\mathtt{x}_i$ is now certainly c. The operator $\mathbf{U}(\mathtt{x}_k \leftarrow c)$ puts $\mathbf{U}(c)$ into the context of other variables: Most factors in the tensor product are identities, i.e. most variables keep their previous values, only $\mathtt{x}_k$ is deterministically updated to its new value c using the previously defined $\mathbf{U}(c)$ operator. The operator $\mathbf{U}(\mathtt{x}_k \leftarrow e)$ updates a variable not to a constant but to the value of an expression e. This update is realised using the filter operator $\mathbf{P}(e = c)$: For all possible values c of e we select those states where e evaluates to c and then update $\mathtt{x}_k$ to this c. Finally, the update operator $\mathbf{U}(*^r \mathtt{x}_k \leftarrow e)$ is used for assignments where we have a pointer on the left hand side. In this case we select those states where $\mathtt{x}_k$ points to another variable $\mathtt{x}_i$ and then update $\mathtt{x}_i$ accordingly. This unfolding of references continues recursively until we end up with a basic variable where we can use the previous update operator $\mathbf{U}(\mathtt{x}_i \leftarrow e)$.

Semantics. The Linear Operator Semantics of a **pWhile** program P is defined as the operator $\mathbf{T} = \mathbf{T}(P)$ on $\mathcal{V}(\mathbf{State} \times \mathcal{B}(P))$. This can be seen as a collecting semantics for the program P as it is defined by

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

$\mathbf{T}(\ell_1, \ell_2) = \mathbf{I} \otimes \mathbf{E}_{\ell_1,\ell_2}$	for $[\mathbf{skip}]^{\ell_1}$
$\mathbf{T}(\ell_1, \ell_2) = \mathbf{U}(p \leftarrow e) \otimes \mathbf{E}_{\ell_1,\ell_2}$	for $[\mathtt{p} \leftarrow e]^{\ell_1}$
$\mathbf{T}(\ell, \underline{\ell_t}) = \mathbf{P}(b = \textsc{true}) \otimes \mathbf{E}_{\ell,\ell_t}$	for $[b]^{\ell}$
$\mathbf{T}(\ell, \ell_f) = \mathbf{P}(b = \textsc{false}) \otimes \mathbf{E}_{\ell,\ell_f}$	for $[b]^{\ell}$
$\mathbf{T}(\ell, \ell_k) = \mathbf{I} \otimes \mathbf{E}_{\ell,\ell_k}$	for $[\mathbf{choose}]^{\ell}$
$\mathbf{T}(\ell, \ell) = \mathbf{I} \otimes \mathbf{E}_{\ell,\ell}$	for $[\mathbf{stop}]^{\ell}$

Table 2. Linear Operator Semantics for **pWhile**

The meaning of $\mathbf{T}(P)$ is to collect for every triple in the probabilistic flow $\mathcal{F}(P)$ of P its effects, weighted according to the probability associated to this triple. The operators $\mathbf{T}(\ell_i, \ell_j)$ which implement the local state updates and control transfers from ℓ_i to ℓ_j are presented in Table 2.

Each local operator $\mathbf{T}(\ell_i, \ell_j)$ is of the form $\mathbf{N} \otimes \mathbf{E}(\ell_i, \ell_j)$ where the first factor $\mathbf{N}$ represents a state update or, in the case of tests, a filter operator while the second factor realises the transfer of control from label ℓ_i to label ℓ_j. For the **skip** and **stop** no changes to the state happen, we only transfer control (deterministically) to the next statement or loop on the current (terminal) statement using matrix units $\mathbf{E}$. Also in the case of a **choose** there is no change to the state but only a transfer of control, however the probabilities p_{ij} will in general be different from 1, unlike **skip**. With assignments we have both a state update, implemented using $\mathbf{U}(p \leftarrow e)$, and a control flow step. For tests b we use the filter operator $\mathbf{P}(b = \textsc{true})$ to select those states which pass the test, and $\mathbf{P}(b = \textsc{false})$ to select those states which fail it, in order to determine which label the control will pass to.

Note that $\mathbf{P}(b = \textsc{true}) + \mathbf{P}(b = \textsc{false}) = \mathbf{I}$, i.e. at any test b every state will cause exactly one (unambiguous) control transfer. We allow in **pWhile** only for constant probabilities p_i in the **choose** construct, which sum up to 1 and as with classical **While** we have no "blocked" configurations (even the terminal **stop** statements 'loop'). It is therefore not necessary to *re-normalise* dynamically the probabilities and it follows easily that:

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

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

Proposition 2. *For any* **pWhile** *program* P *and any classical state* $\sigma \in \mathbf{State}$*, we have:*

$$\langle S, \sigma \rangle \longrightarrow_p \langle S', \sigma' \rangle \quad \textit{iff} \quad (\mathbf{T}(P))_{\langle \sigma, \ell \rangle, \langle \sigma', \ell' \rangle} = p,$$

where ℓ *and* ℓ' *label the first block in the statement* S *and* S'*, respectively.*

4 Pointer Analysis

In principle, it is possible to construct the concrete linear operator semantics for any **pWhile** program with bounded value ranges for its variables and to analyse this way its properties. However, this remains – as in the classical case – only a hypothetical possibility. Even when using sparse matrix representations it is practically impossible to explicitly compute the semantics of all but very small toy examples. This is not least due to the unavoidable involvement of the tensor product which leads to an exponential growth in the dimension of the operator $\mathbf{T}$. Besides this, there might also be the desire to consider a semantics of a program with unbounded values for variables, in which case we are completely unable to explicitly construct the infinite dimensional operator $\mathbf{T}$. In order to analyse (probabilistic) properties of a program we therefore need to consider an abstract version of $\mathbf{T}$.

4.1 Probabilistic Abstract Interpretation

The general approach for constructing simplified versions of a concrete (collecting) semantics via *Abstract Interpretation*, which was introduced by Cousot & Cousot 30 years ago [9], is unfortunately based on order-theoretic and not on linear structures. One can define on a given vector space a number of orderings (lexicographic, etc.) as an additional structure. We could then use this order to compute over- or under-approximations using classical Abstract Interpretation. Though such approximations will always be safe, they might also be quite unrealistic, addressing a *worst case* scenario rather than the *average case* [2]. Furthermore, there is no *canonical* order on a vector space (e.g. the lexicographic order depends on the base). In order to provide probabilistic estimates we have previously introduced, cf. [1, 10], a quantitative version of the Cousot & Cousot framework, which we have called *Probabilistic Abstract Interpretation* (PAI).

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

Definition 1. *Let $\mathcal{C}$ and $\mathcal{D}$ be two finite dimensional vector spaces (or in general, Hilbert spaces) and $\mathbf{A} : \mathcal{C} \to \mathcal{D}$ a (bounded) linear map between them. The (bounded) linear map $\mathbf{A}^\dagger = \mathbf{G} : \mathcal{D} \to \mathcal{C}$ is the* Moore-Penrose pseudo-inverse *of $\mathbf{A}$ iff $\mathbf{A} \circ \mathbf{G} = \mathbf{P}_A$ and $\mathbf{G} \circ \mathbf{A} = \mathbf{P}_G$, 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 linear* adjoint *[11, Ch 10]) onto the ranges of $\mathbf{A}$ and $\mathbf{G}$.*

This allows us to construct the closest (i.e. least square) approximation $\mathbf{T}^\# : \mathcal{D} \to \mathcal{D}$ of $\mathbf{T} : \mathcal{C} \to \mathcal{C}$ as $\mathbf{T}^\# = \mathbf{G} \cdot \mathbf{T} \cdot \mathbf{A} = \mathbf{A}^\dagger \cdot \mathbf{T} \cdot \mathbf{A} = \alpha \circ \mathbf{T} \circ \gamma$. As our

concrete semantics is constructed using tensor products it is important that the Moore-Penrose pseudo-inverse of a tensor product can easily be computed as follows [12, 2.1,Ex 3]: $(\mathbf{A}_1 \otimes \mathbf{A}_2 \otimes \ldots \otimes \mathbf{A}_n)^\dagger = \mathbf{A}_1^\dagger \otimes \mathbf{A}_2^\dagger \otimes \ldots \otimes \mathbf{A}_n^\dagger$.

Example 1 (Parity). Let us consider as abstract and concrete domains $\mathcal{C} = \mathcal{V}(\{-n, \ldots, n\})$ and $\mathcal{D} = \mathcal{V}(\{\text{even}, \text{odd}\})$. The abstraction operator $\mathbf{A}_p$ and its concretisation operator $\mathbf{G}_p = \mathbf{A}_p^\dagger$ corresponding to a parity analysis are represented by the following $n \times 2$ and $2 \times n$ matrices (assuming w.l.o.g. that n is even):

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

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

Example 2 (Sign). With $\mathcal{C} = \mathcal{V}(\{-n, \ldots, 0, \ldots, n\})$ and $\mathcal{D} = \mathcal{V}(\{-, 0, +\})$ we can represent the usual sign abstraction by the following matrices:

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

Example 3 (Forget). We can also abstract all details of the concrete semantics. Although this is in general a rather unusual abstraction it is quite useful in the context of a tensor product state and/or abstraction. Let the concrete domain be the vector space over any range, i.e. $\mathcal{C} = \mathcal{V}(\{n, \ldots, 0, \ldots, m\})$, and the abstract domain a one dimensional space $\mathcal{D} = \mathcal{V}(\{\star\})$. Then the forgetful abstraction and concretisation can be defined by:

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

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

4.2 Abstract Semantics

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

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

where the transfer operator along a computational step from label ℓ_i to ℓ_j can be abstracted "locally". Abstracting each variable separately and using the concrete control flow we get the operator $\mathbf{A} = (\bigotimes_{i=1}^{v} \mathbf{A}_i) \otimes \mathbf{I} = \mathbf{A}_1 \otimes \mathbf{A}_2 \otimes \ldots \otimes \mathbf{A}_v \otimes \mathbf{I}$. Then the abstract transfer operator $\mathbf{T}^{\#}(\ell_i, \ell_j)$ can be defined as $\mathbf{T}^{\#}(\ell_i, \ell_j) = (\mathbf{A}_1^{\dagger}\mathbf{N}_{i1}\mathbf{A}_1) \otimes (\mathbf{A}_2^{\dagger}\mathbf{N}_{i2}\mathbf{A}_2) \otimes \ldots \otimes (\mathbf{A}_v^{\dagger}\mathbf{N}_{iv}\mathbf{A}_v) \otimes \mathbf{E}(\ell_i, \ell_j)$. This operator implements the (abstract) effect to each of the variables in the individual statement at ℓ_i and combines it with the concrete control flow.

It is of course also possible to abstract the control flow, or to use abstractions which abstract several variables at the same time, e.g. specifying the abstract state via the difference of two variables.

Example 4. Consider the following short program:

if $[(\mathtt{x} > 0)]^1$ **then** $[\mathtt{z} \leftarrow \&\mathtt{x}]^2$ **else** $[\mathtt{z} \leftarrow \&\mathtt{y}]^3$ **end if;** $[\mathbf{stop}]^4$

The LOS operator of this program can be straightforwardly constructed as

$$\begin{aligned}\mathbf{T}(P) = {} & \mathbf{P}(\mathtt{x} > 0) \otimes \mathbf{E}(1,2) + \mathbf{P}(\mathtt{x} \leq 0) \otimes \mathbf{E}(1,3) + \\ & + \mathbf{U}(\mathtt{z} \leftarrow \&\mathtt{x}) \otimes \mathbf{E}(2,4) + \mathbf{U}(\mathtt{z} \leftarrow \&\mathtt{y}) \otimes \mathbf{E}(3,4) + \mathbf{I} \otimes \mathbf{E}(4,4).\end{aligned}$$

Here we can see nicely the powerful reduction in size due to PAI. Assuming that `x` and `y` take, for example, values in the range $-100, \ldots, +100$ then the concrete semantics requires a $201 \times 201 \times 2 \times 4 = 323208$ dimensional space (as `z` can point to two variables and there are four program points). The concrete operator $\mathbf{T}(P)$ has about 10^{11} entries (although most of them are zero). If we abstract the concrete value of `x` and `y` using the sign or parity operators the operator $\mathbf{T}^{\#}(P)$ – constructed exactly in the same way as $\mathbf{T}(P)$ but using smaller, abstract value spaces, requires only a matrix of dimension $3 \times 3 \times 2 \times 4 = 72$ or $2 \times 2 \times 2 \times 4 = 32$, respectively. We can even go one step further and completely forget about the value of `y`, in which case we need simply a 24×24 or 16×16 matrix respectively to describe $\mathbf{T}^{\#}(P)$.

The dramatic reduction in size, i.e. dimensions, achieved via PAI and illustrated by the last example lets us hope that our approach could ultimately lead to scalable analyses, despite the fact that the concrete semantics is so large as to make its construction infeasible. However, further work in the form of practical implementations and experiments is needed in order to decide whether this is indeed the case.

The LOS represents the SOS via the generator of a DTMC. It describes the stepwise evolution of the (probabilistic) state of a computation and does not provide a fixed-point semantics. Therefore, neither in the concrete nor in the abstract case can we guarantee that $\lim_{n\to\infty}(\mathbf{T}(P))^n$ or $\lim_{n\to\infty}(\mathbf{T}(P)^{\#})^n$ always exists. The analysis of a program P based on the abstract operator $\mathbf{T}(P)^{\#}$ is considerably *simpler* than by considering the concrete one but still not entirely trivial. Various properties of $\mathbf{T}(P)^{\#}$ can be extracted by iterative methods (e.g. computing $\lim_{n\to\infty}(\mathbf{T}(P)^{\#})^n$ or some averages). As usual in numerical computation, these methods will converge only for $n \to \infty$ and any result obtained after only a finite number of steps will only be an approximation. However, one

can study *stopping criteria* which guarantee a certain quality of this approximation. The development or adaptation of iterative methods and formulation of appropriate stopping criteria might be seen as the numerical analog to the *widening* and *narrowing* techniques of the classical setting.

4.3 Abstract Branching Probabilities

The abstract, like the concrete, semantics is based on two types of basic operators, namely abstract update operators $\mathbf{U}^\#$ and abstract test operators $\mathbf{P}^\#$. As abstract tests introduce probabilistic choices which reflect the probabilities that a test is passed, the abstract semantics will always be probabilistic even if the considered program is deterministic.

Example 5. Obviously, the critical element in Example 4 for computing the probabilities of $\mathtt{z}$ pointing to $\mathtt{x}$ or $\mathtt{y}$ is given by the chances that the test $\mathtt{x} > 0$ in label 1 succeeds or fails. These chances depend on the initial (distribution of possible) values of $\mathtt{x}$. In the concrete semantics we can, for example, assume that $\mathtt{x}$ can take initially any value between $-N$ and $+N$ with the same probability, i.e. we could start with the uniform distribution $\boldsymbol{d}_0 = (\frac{1}{2N+1}, \frac{1}{2N+1}, \ldots, \frac{1}{2N+1})$ for $\mathtt{x}$ – or any other distribution. The probability of $\mathtt{z}$ pointing to $\mathtt{x}$ or $\mathtt{y}$ is then: $P(\mathtt{z} = \&\mathtt{x}) = \sum_i (\boldsymbol{d}_0 \cdot \mathbf{P}(\mathtt{x} > 0))_i = \frac{N+1}{2N+1}$ and $P(\mathtt{z} = \&\mathtt{y}) = \sum_i (\boldsymbol{d}_0 \cdot \mathbf{P}(\mathtt{x} \leq 0))_i = \frac{N}{2N+1}$. In other words, if we increase the range, i.e. for $N \to \infty$, the chances of $\mathtt{z}$ pointing to $\mathtt{x}$ or $\mathtt{y}$ are about $50:50$.

Even in this simple case, the involved matrices, i.e. $\mathbf{P}(\mathtt{x} > 0)$, can become rather large and one might therefore try to estimate the probabilities using PAI. Again the critical element is the abstract test $\mathbf{P}^\#(\mathtt{x} > 0) = \mathbf{A}^\dagger \cdot \mathbf{P}(\mathtt{x} > 0) \cdot \mathbf{A}$.The abstract test operator $\mathbf{P}^\#(\mathtt{x} \leq 0)$ can be computed in the same way or via the fact $\mathbf{P}^\#(\mathtt{x} \leq 0) = \mathbf{I} - \mathbf{P}^\#(\mathtt{x} > 0)$. For different ranges of $\mathtt{x}$ in $\{-N, \ldots, N\}$ we can construct the abstract test operators for the parity abstraction, i.e. $\mathbf{A} = \mathbf{A}_p$ in Example 1, and the sign abstraction, i.e. $\mathbf{A} = \mathbf{A}_s$ in Example 2. Using the `octave` system [13] we get for $\mathbf{P}_s^\# = \mathbf{A}_s^\dagger \cdot \mathbf{P}(\mathtt{x} > 0) \cdot \mathbf{A}_s$ and $\mathbf{P}_p^\# = \mathbf{A}_p^\dagger \cdot \mathbf{P}(\mathtt{x} > 0) \cdot \mathbf{A}_p$:

$$\mathbf{P}_s^\# = \begin{pmatrix} 0\,0\,0 \\ 0\,0\,0 \\ 0\,0\,1 \end{pmatrix}, \quad \begin{pmatrix} 0\,0\,0 \\ 0\,0\,0 \\ 0\,0\,1 \end{pmatrix} \text{ and } \begin{pmatrix} 0\,0\,0 \\ 0\,0\,0 \\ 0\,0\,1 \end{pmatrix} \text{ for } N = 1, 2 \text{ and } 10,$$

$$\mathbf{P}_p^\# = \begin{pmatrix} 0.00\ 0.00 \\ 0.00\ 0.50 \end{pmatrix}, \quad \begin{pmatrix} 0.50\ 0.00 \\ 0.00\ 0.33 \end{pmatrix} \text{ and } \begin{pmatrix} 0.50\ 0.00 \\ 0.00\ 0.45 \end{pmatrix} \text{ for } N = 1, 2 \text{ and } 10$$

These abstract test operators encode important information about the accuracy of the PAI estimates. Firstly, we observe that the sign abstraction $\mathbf{A}_s$ provides us with stable and correct estimates. As we would expect, we see that the abstract values '$-$' and 0 never pass the test, while '$+$' always does, independently of N. Secondly, we see that the parity analysis is rather pointless, for large N the test is passed with a 50% chance for **even** as well as **odd** – this expresses just the fact that parity has nothing to do with the sign of a variable. A bit more interesting are the values for $N = 1$ and $N = 2$. If we only consider the

concrete values $-1, 0, 1$ for $\mathtt{x}$ then only even numbers (0 in this set) always fail the test (thus we have a zero entry in the upper left entry) and that it succeeds for the odd numbers, -1 and 1, with an even chance; if we consider a larger concrete range, i.e. for $N > 1$, then the chances tend to become $50 : 50$ but due to 0 failing the test, there is a slightly lower chance for even numbers to succeed compared with odd numbers.

As we can see from the example, the abstract test operators $\mathbf{P}^{\#}$ contain information about the probabilities that test succeed for abstract values. This means that the abstract semantics contains estimates of *dynamic* branching probabilities, i.e. depending on the (abstract) state of the computation the probabilities to follow, for example, the **then** or **else** branch will change. One could utilise this information to distinguish between the branching probabilities in different phase of executions; during the initialisation phase of a program the branching probabilities could be completely different from later stages.

However, we can also obtain more conventional, i.e. *static*, branching probabilities for the whole execution of a program. In order to do this we have to provide an initial distribution over abstract values. Using the abstract semantics $\mathbf{T}^{\#}$, we can then compute distributions over abstract values for any program point which in particular provide estimates of the abstract state reaching any test. This amounts to performing a probabilistic forward analysis of the program. Based on the probability estimates for abstract values at a test we can then compute estimates for the branching probabilities in the same way as in the classical case.

With respect to a higher order analysis – such as a pointer analysis – we therefore propose a two-phase analysis: During the first phase the abstract semantics $\mathbf{T}^{\#}$ is used to provide estimates of the probability distributions over abstract values (and thus for the branching probabilities) for every program point; phase two then constructs the actual analysis, e.g. a probabilistic points-to matrix, for every program point. One could interpret phase one also as a kind of program transformation which replaces tests by probabilistic choices.

Example 6. For the program in Example 4 the corresponding probabilistic program we have to consider after performing a parity analysis in phase one is:

$$[\mathbf{choose}]^1 \;\; (p_\top : [\mathtt{z} \leftarrow \&\mathtt{x}]^2) \;\; \mathbf{or} \;\; (p_\bot : [\mathtt{z} \leftarrow \&\mathtt{y}]^3); \;\; [\mathbf{stop}]^4$$

where $p_\top$ and $p_\bot$ (which both depend on N) are the branching probabilities obtained in phase one.

4.4 Probabilistic Points-to Matrix vs State

The (probabilistic) state in the concrete semantics contains a complete description of the values of all variables as well as the current statement. We can extract information about where pointers (variables) point-to at a certain label by forgetting the value of the basic variables. The same is, of course, also possible with the abstract state. This is achieved via the abstraction $\mathbf{A}_r = \mathbf{A}_f^{\otimes(v-r)} \otimes \mathbf{I}^{\otimes r} \otimes \mathbf{I}$, where $\mathbf{A}_f$ is the "forgetful" abstraction in Example 3 while v and r denote the

number of all variables and of all pointer variables, respectively. If we are interested in the pointer structure at a certain program point we can also use the following abstraction:

$$\mathbf{A}_r(\ell) = \mathbf{A}_f^{\otimes(v-r)} \otimes \mathbf{I}^{\otimes r} \otimes \mathbf{A}_f(\ell),$$

where $\mathbf{A}_f(\ell)$ is represented by a column vector (i.e. a $n \times 1$ matrix) with a single non-zero entry $(\mathbf{A}_f(\ell))_{1,\ell} = 1$.

Note that $\mathbf{A}_f(\ell)^\dagger = \mathbf{A}_f(\ell)^T$ and that $\mathbf{A}_f(\ell) \cdot \mathbf{A}_f(\ell)^\dagger = \mathbf{P}(\ell)$ while $\mathbf{A}_f(\ell)^\dagger \cdot \mathbf{A}_f(\ell) = (1)$, i.e. the 1×1 (identity) matrix.

Given an initial distribution $\boldsymbol{d}_0$ or $\boldsymbol{d}_0^\#$, which represent the initial concrete or abstract values of all variables (and the initial label of the program in question), we can compute the computational state after n computational steps simply by iterating the concrete or abstract semantics $\mathbf{T}$ and $\mathbf{T}^\#$, i.e. $\boldsymbol{d}_n = \boldsymbol{d}_0 \cdot \mathbf{T}^n$ and $\boldsymbol{d}_n^\# = \boldsymbol{d}_0^\# \cdot \mathbf{T}^{\#n}$. Based on these distributions $\boldsymbol{d}_n$ or $\boldsymbol{d}_n^\#$ we can compute also statistical properties of a program, by averaging over a number of iterations, or the final situation, by considering the limit $n \to \infty$.

The typical result of a probabilistic pointer analysis, e.g. [4], is a so-called *points-to* matrix which records for every program point the probability that a pointer variable refers to a particular (other) variable. Using our systematic approach to pointer analysis we can construct such a points-to matrix, concretely or abstractly. However, we can also show that the points-to matrix contains – to a certain extent even in the concrete case – only partial information about the pointer structure of a program.

Example 7. Consider the following simple example program:

if $[(\mathtt{z}_0 \bmod 2 = 0)]^1$ **then** $[\mathtt{x} \leftarrow \&\mathtt{z}_1]^2$; $[\mathtt{y} \leftarrow \&\mathtt{z}_2]^3$
else $[\mathtt{x} \leftarrow \&\mathtt{z}_2]^4$; $[\mathtt{y} \leftarrow \&\mathtt{z}_1]^5$ **end if**; $[\mathbf{stop}]^6$

Any reasonable analysis of this program – assuming a uniform distribution over all possible values of $\mathtt{z}_0$ – will result in the following probabilistic points-to matrix at label 6, i.e. at the end of the program (we write the two rows corresponding to $\mathtt{x}$ and $\mathtt{y}$ as a direct sum):

$$(0,0,0,\frac{1}{2},\frac{1}{2}) \oplus (0,0,0,\frac{1}{2},\frac{1}{2}).$$

This probabilistic points-to matrix states that $\mathtt{x}$ and $\mathtt{y}$ point with probability $\frac{1}{2}$ to $\mathtt{z}_1$ and $\mathtt{z}_2$, but in fact there is a relational dependency between where $\mathtt{x}$ and $\mathtt{y}$ point to. This is detected if we construct the *points-to state* via $\boldsymbol{d}_0 \cdot (\lim_{n\to\infty} \mathbf{T}^{\#n}) \cdot \mathbf{A}_r(6) = \boldsymbol{d}_0 \cdot \lim_{n\to\infty} (\mathbf{A}_p^\dagger \mathbf{T} \mathbf{A}_p)^n \cdot \mathbf{A}_r(6)$. For our example program we get the following *points-to tensor*:

$$\frac{1}{2} \cdot (0,0,0,1,0) \otimes (0,0,0,0,1) + \frac{1}{2} \cdot (0,0,0,0,1) \otimes (0,0,0,1,0)$$

which expresses exactly the fact that (i) there is a 50% chance that $\mathtt{x}$ points to $\mathtt{z}_1$ and $\mathtt{y}$ points to $\mathtt{z}_2$, and that (ii) there is also a 50% chance that $\mathtt{x}$ and $\mathtt{y}$ point to $\mathtt{z}_2$ and $\mathtt{z}_1$, respectively.

For every pointer variable $\mathtt{x}_i$ we can compute the corresponding row in the *points-to matrix* using instead of $\mathbf{A}_r(6)$ the abstraction

$$\mathbf{A}_r(\ell, \mathtt{x}_i) = \mathbf{A}_f^{\otimes(i-1)} \otimes \mathbf{I} \otimes \mathbf{A}_f^{\otimes(v-i+1)} \otimes \mathbf{A}_f(\ell).$$

However, this way we get less information than with the *points-to tensor* above. By a simple dimension argument it's easy to see that, for instance, in our example the points-to matrix has $2 \times 5 = 10$ entries while the points-to state is given by a $5 \times 5 = 25$ dimensional vector.

In fact, it is sufficient to consider the points-to matrix to describe the common state (i.e. distribution) of two pointer variables (seen as random variables) if and only if they are (probabilistically) *independent*, cf e.g. [14, Sect 20]. If two random (pointer) variables are not independent but somehow *correlated*, then we need a points-to tensor to describe the situation precisely. In the classical framework this corresponds exactly to the distinction between *independent* and *relational* analysis. We can combine in our framework both approaches. However – as always – it will depend on the concrete application how much *precision* (provided by the points-to tensor) one is willing to trade in for lower *computational complexity* (the points-to matrix allows for).

5 Conclusions and Further Work

We presented a compositional semantics of a simple imperative programming language with a probabilistic choice construct. The executions of a program in this language correspond to a discrete time Markov chain. Important for the syntax directed construction of the generator matrix of this DTMC is the tensor product representation of the probabilistic state. Using a small number of basic filter and update operators we were also able to provide the semantics of pointers (to static variables). Probabilistic Abstract Interpretation, a quantitative generalisation of the classical Cousot & Cousot approach, provided the framework for constructing a "simplified" abstract semantics. Linearity, distributivity and the tensor product enabled us to construct this abstract semantics in the same syntax directed way as the for concrete semantics. Our approach allows for a systematic development and study of various probabilistic pointer analyses. We could, for example, argue that the traditional points-to matrix is not sufficient for providing relational information about the pointer structure of a program.

We used static techniques for estimating execution frequencies. A more common approach is the use of profiles which are derived by running the program on a selection of sample inputs. Our static estimation does not require this separate compilation and is not dependent on the choice of the representative inputs which is a crucial and often very difficult part of the profiling process. Several authors have argued about the advantages of static estimators or program-based branch prediction as opposed to the time-consuming profiling process as a base for program optimisation [15–17]. However, since estimates derived from runtime profile information are generally regarded as the most accurate source of

information, it is necessary to measure the utility of an estimate provided by static techniques by comparing them with the actual measurements in order to assess their accuracy. We plan to further develop this point in future work; in particular, we plan to exploit the metric intrinsic in the PAI framework for the purpose of measuring the precision of our analyses and to use the mathematical theory of testing and its well-known results (cf. [18]) in order to provide the outcomes of our analyses with a statistical interpretation.

Further work will address various extensions of the current approach: (i) an extension to unbounded value ranges (this will require the reformulation of our framework based on more advanced mathematical structures like measures and Banach/Hilbert spaces), (ii) the introduction of dynamical pointer structures using a heap and a memory allocation function, and (iii) a practical implementation, e.g. by investigating some forms of probabilistic *widening*, of our analysis in order to establish whether it scales, i.e. if it can also be applied to "real world" programs, and provides enough useful information for speculative optimisation.

References

1. Di Pierro, A., Wiklicky, H.: Concurrent Constraint Programming: Towards Probabilistic Abstract Interpretation. In: PPDP'00. (2000) 127–138
2. Di Pierro, A., Hankin, C., Wiklicky, H.: Abstract interpretation for worst and average case analysis. In: Program Analysis and Compilation, Theory and Practice. Volume 4444 of LNCS. Springer Verlag (2007) 160–174
3. Chen, P.S., Hwang, Y.S., Ju, R.D.C., Lee, J.K.: Interprocedural probabilistic pointer analysis. IEEE Trans. Parallel and Distributed Systems **15** (2004) 893–907
4. Da Silva, J., Steffan, J.G.: A probabilistic pointer analysis for speculative optimizations. In: ASPLOS-XII, ACM Press (2006) 416–425
5. den Hartog, J., de Vink, E.: Verifying probabilistic programs using a Hoare-like logic. International Journal of Foundations of Computer Science **13** (2002) 315–340
6. Di Pierro, A., Hankin, C., Wiklicky, H.: On probabilistic techniques for data flow analysis. In: QAPL'07. (2007) to appear in ENTCS.
7. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag, Berlin – Heidelberg (1999)
8. Cousot, P., Cousot, R.: Systematic design of program transformation frameworks by abstract interpretation. In: POPL'02. (2002) 178–190
9. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL'77. (1977) 238–252
10. Di Pierro, A., Wiklicky, H.: Measuring the precision of abstract interpretations. In: LOPSTR'00. Volume 2042 of LNCS., Springer Verlag (2001) 147–164
11. Roman, S.: Advanced Linear Algebra. 2nd ed. Springer Verlag (2005)
12. Ben-Israel, A., Greville, T.: Generalised Inverses. 2nd ed. Springer Verlag (2003)
13. Eaton, J.: Gnu Octave Manual. www.octave.org (2002)
14. Billingsley, P.: Probability and Measure. Wiley & Sons, New York (1978)
15. Wagner, T.A., Maverick, V., Graham, S.L., Harrison, M.A.: Accurate static estimators for program optimization. SIGPLAN Not. **29**(6) (1994) 85–96
16. Ramalingam, G.: Data flow frequency analysis. In: PLDI'96. (1996) 267–277
17. Ball, T., Larus, J.R.: Branch prediction for free. SIGPLAN Not. **28** (1993) 300–313
18. Ferguson, T.S.: Mathematical Statistics. Academic Press (1967)