Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 220 (2008) 43–59

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Probabilistic Abstract Interpretation of Imperative Programs using Truncated Normal Distributions

Michael J. A. Smith[1] ,[2]

Laboratory for Foundations of Computer Science
University of Edinburgh
Edinburgh, United Kingdom

Abstract

When modelling a complex system, such as one with distributed functionality, we need to choose an appropriate level of abstraction. When analysing quantitative properties of the system, this abstraction is typically probabilistic, since we introduce uncertainty about its state and therefore its behaviour. In particular, when we aggregate several concrete states into a single abstract state we would like to know the distribution over these states. In reality, any probability distribution may be possible, but this leads to an intractable analysis. Therefore, we must find a way to approximate these distributions in a safe manner. We present an abstract interpretation for a simple imperative language with message passing, where truncated multivariate normal distributions are used as the abstraction. This allows the probabilities of transient properties to be bounded, without needing to calculate the exact distribution. We describe the semantics of programs in terms of automata, whose transitions are linear operators on measures. Given an input measure, we generate a probabilistic trace whose states are labelled by measures, describing the distribution of the values of variables at that point. By the use of appropriate widening operators, we are able to abstract the behaviour of loops to various degrees of precision.

Keywords: Probabilistic abstract interpretation, Probabilistic semantics, Truncated normal distributions, Widening operators

1 Introduction

In classical program analysis, abstract interpretation [1] is a powerful framework for computing *safe approximations* to properties of interest. These properties may be undecidable in general, but we can compute them efficiently if we are prepared for the answer to be uncertain some of the time, in that impossible behaviours are thought to be possible. Importantly though, the safety of our abstraction ensures that the converse is false – no possible behaviour is ever thought to be impossible.

[1] This work was funded by a Microsoft Research European Scholarship

[2] Email:M.J.A.Smith@sms.ed.ac.uk

doi:10.1016/j.entcs.2008.11.018

In 1981, Kozen [3] described a semantics for *probabilistic* programming languages in terms of linear operators on Banach spaces. A program is then a linear map from an input probability measure (the joint distribution of the initial values of its variables) to an output sub-probability measure (since there may be some probability of non-termination). Monniaux [4] and Di Pierro and Wiklicky [6] use two different ideas to extend abstract interpretation to these domains, resulting in a *probabilistic abstract interpretation*.

The approach taken by Wiklicky and Di Pierro defines the notion of 'closeness' of approximation in terms of an inner-product metric on measures. Monniaux, on the other hand, applies the more classical order-theoretic approach to Banach spaces, by ordering measures based on their total measure. Both approaches, whilst theoretically applicable to continuous measures, have difficulty in practice – in the former, the abstraction and concretisation functions need to actually be constructed, and in the latter the distributions must be discretised to the required precision.

In this paper, we present an alternative approach, which is less general than the two aforementioned, but provides an abstract domain in which continuous measures can be operated on in an efficient and scalable manner. This is not to say that the aforementioned have not been successfully applied to other domains, such as approximation of finite probabilistic automata in the case of Wiklicky and Di Pierro, but we wish to avoid discretising distributions, which can lead to a state space explosion.

Our motivation is to provide a probabilistic abstraction of deterministic programs, in that we are interested in the behaviour of a program when its input follows some distribution. This is particularly applicable to distributed systems, where we are interested in the behaviour of the system given some environment (such as a distribution of packet lengths, or over varieties of client behaviour), which is often not deterministic. In this case, knowing how the system behaves on a particular input cannot necessarily help us to determine more global performance properties.

In building an abstract interpretation we need to have some property that we want to approximate, and in our case this is the probability of being in a certain state of the system, and also of taking a particular path through the system. As a consequence, we look for abstract measures that are strict upper bounds of the concrete, or actual measures. For our abstract domain, we choose the truncated multivariate normal distributions (or more precisely, measures), because they can be easily operated upon, since they have only a small number of parameters, and can approximate a variety of distributions (for example, an exponential distribution can be approximated by truncating a normal distribution to just its tail).

This paper is structured as follows. In Section 2 we describe a simple imperative language with message passing, and give a concrete probabilistic semantics in Section 3. Following this, we review the framework of abstract interpretation in Section 4 before presenting our abstraction and abstract semantics in Section 5. We conclude with Section 6.

2 Syntax of Imperative Programs

We consider simple imperative programs with the following syntax. Variables X are real-valued, which can be viewed as a continuous abstraction of integer-valued variables. Arithmetic expressions E have the following syntax (where $c \in \mathbb{R}$ are constants):

$$E \quad \texttt{::=} \quad c \mid X \mid -E \mid E + E \mid c \times E$$

Note that these expressions are always linear. This does not reduce expressivity, since non-linear operations can be encoded using loops. By defining our language in this way we will need only to abstract loops in order to abstract non-linear behaviour, hence two separate abstractions are not necessary.

For Boolean expressions B:

$$B \quad \texttt{::=} \quad \texttt{true} \mid \texttt{false} \mid X < c \mid X \leq c \mid \neg B$$

Note that without loss of generality, we only allow a variable to be compared with a constant. Since we can construct more complex comparisons by first defining a new variable, this serves to simplify our abstraction as we shall see later. Furthermore, conjunctions and disjunctions of conditions can be expressed (albeit inefficiently) by nesting `if`-statements, hence are not necessary as primitives.

Finally, a command C is of the following form:

$$\begin{array}{lll} C & \texttt{::=} & \texttt{skip} \mid \texttt{return}\ E \mid X\ \texttt{:=}\ E \mid X\ \texttt{:=}\ f(X, \ldots, X) \\ & \mid & C\ \texttt{;}\ C \mid \texttt{if}\ B\ \texttt{then}\ C\ \texttt{else}\ C \mid \texttt{while}\ B\ \texttt{do}\ C \\ & \mid & \texttt{send}_\alpha(X) \mid \texttt{recv}_\alpha(X, X) \end{array}$$

To simplify the presentation, we do not allow recursive function calls. The only unusual feature of this language is *asynchronous message passing*. This is implemented by the `send` and `recv` actions, which are parameterised by a channel name, α. Both of these commands are *asynchronous*, in that the program will never block on their execution. The $\texttt{send}_\alpha(X)$ command writes the value of variable X to channel α (overwriting any value that is already on the channel), and $\texttt{recv}_\alpha(X, Y)$ attempts to place the content of α into X. If the `recv` is successful, then Y is assigned the value 1, otherwise $Y\ \texttt{:=}\ 0$. A channel can be thought of as a global variable with two states – either it is empty, or it contains a real value.

A system consists of a number of function definitions, and an initial state. Function definitions are of the form $f(X, \ldots, X)\ \{\ C\ \}$. The initial state S is of the form:

$$S \quad \texttt{::=} \quad f(X, \ldots, X) \mid S \mid S$$

This can be thought of as the parallel composition of a number of function calls, with the initial values of the arguments described by a joint distribution. These function calls, or threads, can communicate with one another via message passing, and will execute in parallel (we use an interleaved semantics). Since we do not allow the explicit creation or deletion of threads, the number of threads in the system is constant.

3 Concrete Semantics

Before we describe our semantics in terms of probabilistic automata, we first need to describe the data environment of a program; in other words, the domain of its variables. Rather than considering individual values that a variable can take, we consider it to take a range of values according to some *probability distribution.* In essence, variables are viewed as *random variables* and the operation of the program is to transform them. However, since we are only concerned with the *distribution* of these random variables (specifically, their joint distribution), we can treat the program as operating on this distribution directly.

Before we proceed, let us remind ourselves of the following definitions from measure theory:

- A *σ-algebra* of a set X is a subset of $\mathcal{P}(X)$ that contains $\emptyset$ and X, and is closed under countable union and intersection.
- A *measurable space* (X, σ_X) is a set X with a σ-algebra σ_X. The elements of σ_X are the *measurable subsets* of X.
- A *measurable function* is a well-behaved function between two measurable spaces.
- A *measure* is a countably additive function $\mu : \sigma_X \rightarrow \mathcal{R}$.
- A *positive measure* is a countably additive function $\mu : \sigma_X \rightarrow \mathcal{R}^+$.
- The *total weight* of a measure μ on a measurable space (X, σ_X) is given by $\mu(X)$.
- A *probability measure* is a positive measure with total weight 1.
- A *measure space* (X, σ_X, μ) is a measurable space (X, σ_X) equipped with a measure μ.
- A *probability space* is a measurable space equipped with a probability measure.
- A *random variable* is a measurable function whose domain is a probability space.

As per Kozen [3], we consider the state of a program's variables to be described by a *probability measure.* That is to say, we can assign a probability to each (measurable) set of values that the variables could range over. For a given measurable space, the set of measures over that space induces a *Banach space.* In other words, it induces a normed vector space that is complete with respect to the metric induced by its norm.

The operation of a program is to transform one measure into another, and so is a linear operator between two measure spaces, or a continuous linear operator on the induced Banach space. Kozen describes a denotational semantics in which a single linear operator describes the entire program. We can think of such an operator as a map from a probability to a *sub-probability* space in general, since the program may not terminate on all inputs. Under this approach, loops are denoted using a least fixed point operator, in the style of conventional Scott-Strachey semantics. In our semantics we represent loops by cyclic transitions in an automaton, but for sequential code that does not involve message passing, our semantics are essentially the same.

We will refer in the following to three distinct semantics. $[\![\cdot]\!]$ is the conventional deterministic semantics of our programs, and we use this to define what we mean by assignment, and boolean conditions on our variables. $[\![\cdot]\!]_p$ is the probabilistic semantics described by Kozen, which we will use for the sequential fragments of our programs. Finally, $[\![\cdot]\!]_{pa}$ is the probabilistic automaton semantics, which we will define shortly.

A program consists of a vector of N variables, $X_1, \ldots, X_N$, which may be assigned to linear expressions of one another. In the deterministic semantics, our state is a vector of size N, storing the value of each variable, and so we view an assignment as follows:

$$[\![X_i := E]\!] = \langle x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n \rangle \mapsto \langle x_1, \ldots, x_{i-1}, [\![E]\!], x_{i+1}, \ldots, x_n \rangle$$

As usual, we replace the old value of the variable by the new, which is given by the expression E. Extending this to the probabilistic setting, we want instead a map between two measures. We have an initial measure μ, and so the new measure μ' after executing the assignment can be found by 'reversing' the assignment, then applying μ. In other words, the probability of being in a certain state s *after* the assignment is the probability of being in any of the states that could lead to s *before* the assignment. Formally, the semantics is as follows:

$$[\![X_i := E]\!]_p(\mu) = \mu \circ [\![X_i := E]\!]^{-1}$$

To describe our semantics $[\![\cdot]\!]_{pa}$, we need a little more than just linear operators. In particular we have a set of *states*, and a *transition relation* between these states. Since a transition represents a sequence of sequential operations, we associate a linear operator with it, in the same way as above. We only introduce states when there is a branch in control-flow due to a loop, or due to message passing (which is encoded by a label on the state). Forward branches due to conditional statements do not require additional states, and can be represented by multiple transitions out of a state (analogous to addition in $[\![\cdot]\!]_p$).

More formally, the semantics of a program consists of the following elements:

(i) A vector $\boldsymbol{X}$ of variables, $X_1, \ldots, X_N$ (where N is the number of variables).

(ii) A set $\mathcal{C}$ of channels.

(iii) A set $\mathcal{S}$ of internal states (which may be empty). States have an optional label $L \in \mathcal{L} = \{send^i_\alpha, recv^{i,j}_\alpha \mid \alpha \in \mathcal{C}, 1 \leq i \leq N\}$, which specify sending or receiving the value of a variable X_i over a channel α (and in the case of a `recv` recording the success in variable X_j).

(iv) A transition relation $\mathcal{T} \subseteq (\{\bullet\} \cup \mathcal{S})^2 \times \mathcal{M}$, where $\mathcal{M}$ is the set of linear operators $M : \mu \rightarrow \mu$ on measures μ. For convenience, we will write a transition (s, s', M) as $s \xrightarrow{M} s'$.

The special state $\bullet$ denotes the entry and exit points of the automaton, depending on whether it occurs on the left or right hand side of a transition. For simplicity, we will assume that $\boldsymbol{X}$ and $\mathcal{C}$ are fixed, and are known before applying the semantics to our programs. In particular, this means that there are no name conflicts between variables in different functions; a condition easily met by replacing a function with an α-equivalent version. The remaining two factors, the states and the transitions, may vary between two programs, and we use $[\![\cdot]\!]^S_{pa}$ and $[\![\cdot]\!]^T_{pa}$ to refer to these respectively.

We can now describe our probabilistic automaton semantics. We begin with the `skip` command, which is trivially the identity map ($I(\mu) = \mu$) on the input measure, and has no states:

$$[\![\texttt{skip}]\!]^S_{pa} = \{\} \qquad [\![\texttt{skip}]\!]^T_{pa} = \{\bullet \xrightarrow{I} \bullet\}$$

Perhaps surprisingly, `return` statements are treated in the same way – since we do not modify any variables, it has no effect on their joint distribution:

$$[\![\texttt{return}\ E]\!]^S_{pa} = \{\} \qquad [\![\texttt{return}\ E]\!]^T_{pa} = \{\bullet \xrightarrow{I} \bullet\}$$

Basic assignments do not introduce any state, and are denoted by the same operator as in Kozen's semantics:

$$[\![X_i := E]\!]^S_{pa} = \{\}$$
$$[\![X_i := E]\!]^T_{pa} = \{\bullet \xrightarrow{M} \bullet\} \text{ where } M = [\![X_i := E]\!]_p$$

Calling a function f, defined $f(X_{j_1}, \ldots, X_{j_n})\ \{\ C\ \}$, can be thought of as modifying the denotation of the body of the function so that the argument variables are replaced by the actual arguments, and the `return` call is replaced by the appropriate variable assignment. More formally:

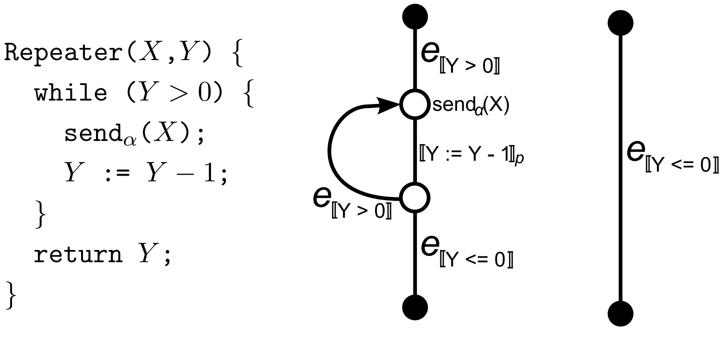
$$[\![X_i := f(X_{i_1}, \ldots, X_{i_n})]\!]_{pa} = [\![C\{X_{i_1}/X_{j_1}, \ldots, X_{i_n}/X_{j_n}, X := E/\texttt{return}\ E\}]\!]_{pa}$$

Sequencing involves composing two automata, so that the exit transitions of the first are merged with the entry transitions of the second. The resulting linear operator is the composition of the originals. If there is more than one start or exit transition, we must take all possible combinations. Hence in the worst case, the size of the automaton may grow exponentially in the number of branching instructions.

Formally, the semantics is as follows:

$$[\![C_1\ ;\ C_2]\!]^S_{pa} = [\![C_1]\!]^S_{pa} \cup [\![C_2]\!]^S_{pa}$$
$$[\![C_1\ ;\ C_2]\!]^T_{pa} = \{s \xrightarrow{M_1 \circ M_2} s' \mid s \xrightarrow{M_1} \bullet \in [\![C_1]\!]^T_{pa} \wedge \bullet \xrightarrow{M_2} s' \in [\![C_2]\!]^T_{pa}\} \cup$$
$$\{s \xrightarrow{M} s' \in [\![C_1]\!]^T_{pa} \mid s' \neq \bullet\} \cup \{s \xrightarrow{M} s' \in [\![C_2]\!]^T_{pa} \mid s \neq \bullet\}$$

To describe the semantics of `if`-statements, we first need to describe the semantics of conditions. A condition B denotes the set $[\![B]\!]$ of valuations of variables that satisfy B. Let the measure μ_Y be such that $\mu_Y(Z) = \mu(Y \cap Z)$. Then we define $e_{[\![B]\!]}$ to be the linear operator $\mu \mapsto \mu_{[\![B]\!]}$. Intuitively, we throw away all the unreachable environments (that do not satisfy the condition B) before applying the measure μ:



```
Repeater(X,Y) {
  while (Y > 0) {
    send_α(X);
    Y := Y - 1;
  }
  return Y;
}
```

$e_{\llbracket Y > 0 \rrbracket}$
send$_\alpha$(X)
$\llbracket Y := Y - 1 \rrbracket_p$
$e_{\llbracket Y > 0 \rrbracket}$
$e_{\llbracket Y <= 0 \rrbracket}$
$e_{\llbracket Y <= 0 \rrbracket}$

Fig. 1. An example program and its concrete semantics

$$\llbracket \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \rrbracket^S_{pa} = \llbracket C_1 \rrbracket^S_{pa} \cup \llbracket C_2 \rrbracket^S_{pa}$$

$$\begin{aligned}\llbracket \texttt{if } B \texttt{ then } C_1 \texttt{ else } C_2 \rrbracket^T_{pa} = & \{\bullet \xrightarrow{e_{\llbracket B \rrbracket} \circ M} s \mid \bullet \xrightarrow{M} s \in \llbracket C_1 \rrbracket^T_{pa}\} \ \cup \\ & \{s \xrightarrow{M} s' \in \llbracket C_1 \rrbracket^T_{pa} \mid s \neq \bullet\} \ \cup \\ & \{\bullet \xrightarrow{e_{\llbracket \neg B \rrbracket} \circ M} s \mid \bullet \xrightarrow{M} s \in \llbracket C_2 \rrbracket^T_{pa}\} \ \cup \\ & \{s \xrightarrow{M} s' \in \llbracket C_2 \rrbracket^T_{pa} \mid s \neq \bullet\}\end{aligned}$$

The behaviour of `while` loops requires us to introduce an additional state. This corresponds to the beginning of the body of the loop, so that reentering the loop equates to returning to this state. If we were to unroll the loop, we would effectively compute the fixed point, as done explicitly by Kozen. The formal statement of this semantics is as follows, where s' is a fresh state:

$$\llbracket \texttt{while } B \texttt{ do } C \rrbracket^S_{pa} = \llbracket C \rrbracket^S_{pa} \cup \{s'\}$$

$$\begin{aligned}\llbracket \texttt{while } B \texttt{ do } C \rrbracket^T_{pa} = & \{\bullet \xrightarrow{e_{\llbracket B \rrbracket}} s'\} \ \cup \ \{\bullet \xrightarrow{e_{\llbracket \neg B \rrbracket}} \bullet\} \ \cup \\ & \{s' \xrightarrow{M} s \mid \bullet \xrightarrow{M} s \in \llbracket C \rrbracket^T_{pa}\} \ \cup \\ & \{s \xrightarrow{M \circ e_{\llbracket B \rrbracket}} s' \mid s \xrightarrow{M} \bullet \in \llbracket C \rrbracket^T_{pa}\} \ \cup \\ & \{s \xrightarrow{M \circ e_{\llbracket \neg B \rrbracket}} \bullet \mid s \xrightarrow{M} \bullet \in \llbracket C \rrbracket^T_{pa}\} \ \cup \\ & \{s \xrightarrow{M} s'' \in \llbracket C_1 \rrbracket^T_{pa} \mid s \neq \bullet \wedge s'' \neq \bullet\}\end{aligned}$$

Finally, we consider the semantics of message passing. For both `send` and `recv` calls, the only action is to introduce an additional state, with the corresponding label. In the following, s is a fresh state:

$$\llbracket \texttt{send}_\alpha(X_i) \rrbracket^S_{pa} = \{s[send^i_\alpha]\} \qquad \llbracket \texttt{send}_\alpha(X_i) \rrbracket^T_{pa} = \{\bullet \xrightarrow{I} s, s \xrightarrow{I} \bullet\}$$

$$\llbracket \texttt{recv}_\alpha(X_i, X_j) \rrbracket^S_{pa} = \{s[recv^{i,j}_\alpha]\} \qquad \llbracket \texttt{recv}_\alpha(X_i, X_j) \rrbracket^T_{pa} = \{\bullet \xrightarrow{I} s, s \xrightarrow{I} \bullet\}$$

The above semantics gives an automaton describing the behaviour of the program as a function of its input distribution. An example program and its semantics are shown in Figure 1 for illustration. If we consider a *particular* input distribution, the *concrete interpretation* of this automaton yields a probabilistic trace. If there are n variables in a program, the states of this trace are of the form $\mu \vdash s$, where μ is a sub-probability measure over the measurable space $(\mathbb{R}^n, \sigma_{\mathbb{R}}^{(n)})$, and s is a state in the automaton. A transition $\mu \vdash s \to \mu' \vdash s'$ in the trace exists iff there is a transition $s \xrightarrow{M} s'$ in the automaton, and $\mu' = M(\mu)$. This transition occurs with probability $\frac{\mu'(\mathbb{R}^n)}{\mu(\mathbb{R}^n)}$.

Due to lack of space, and since it is not essential to the probabilistic abstract interpretation, we will not formally describe the trace semantics of labelled states. Intuitively, a state labelled with $\mathtt{send}_\alpha$ replaces the measure associated with channel α (over $(\mathbb{R} \cup \{\bot\}, \sigma_{\mathbb{R}\cup\{\bot\}})$) with the measure obtained by projecting the variable we wish to send. Similarly, a state labelled with $\mathtt{recv}_\alpha$ replaces the measures on the two variables we receive into (the data and the success indicator) based on the measure on α. Essentially, we treat channels just like any other variable, except that the interleaving semantics introduces non-determinism.

Since we are interested not in the concrete interpretation itself, but in some property of it, we define a *collecting semantics* to describe this. In our case, we choose to look at the total measure of each state in the automaton (corresponding roughly to points in the program). This is the sum of all the measures that occur at a particular state in the automaton throughout the trace. The collecting semantics $\mathit{Coll}[[\![P]\!]_{\mathrm{pa}}(\mu)]$ of the concrete interpretation of the semantics $[\![\cdot]\!]_{\mathrm{pa}}$ of a program P with input μ is defined as:

$$\mathit{Coll}[[\![P]\!]_{\mathrm{pa}}(\mu)](s) = \sum_{\mu' \vdash s \in [\![P]\!]_{pa}(\mu)} \mu'$$

This can be thought of as the semantics of profiling, in that collected measures $\mu(X)$ give the expected number of times that we will see the values X in the given state. We can obtain an average measure for the state by dividing by $\mu(\mathbb{R}^n)$, the total measure.

4 Abstract Interpretation

Classical abstract interpretation [1] is a mathematical framework that relates a concrete domain to an abstract one. Properties in the abstract domain are *safe approximations* (supersets) of their concrete counterparts. By constructing a suitable abstract domain and abstract semantics, we can reason about properties of a program that would otherwise be undecidable in general, at the cost of some precision.

Abstract interpretation can be applied to many different semantic frameworks, but since we are working with a transition system style of semantics, we will describe it just in this setting. Consider two preordered sets, the concrete domain $(X, \leq)$ and the abstract domain $(X^\sharp, \leq^\sharp)$. To relate the two domains, we have an *abstraction function*, $\alpha : X \to X^\sharp$, and a *concretisation function*, $\gamma : X^\sharp \to X$. A *safe abstraction* will be one that satisfies, for all $x \in X$, $x \leq \gamma(\alpha(x))$. If $X^\sharp \subset X$, we can let the concretisation function be the identity map so that we can concentrate solely on the definition of α.

The usefulness of this framework comes when we apply it to our transition semantics. If a property $x \in X$ (e.g. a valuation of variables) holds at a state s (e.g. a program point), then we write the statement $x \vdash s$. The concrete semantics then induces a transition relation $\rightarrow$ between these statements. An abstract semantics, inducing $\rightarrow^\sharp$ for $x^\sharp \in X^\sharp$, is then safe if the following property holds:

Definition 4.1 A concrete and an abstract transition relation, $\rightarrow$ and $\rightarrow^\sharp$, satisfy the *relational homomorphism* property if $x_1 \vdash s_1 \rightarrow x_2 \vdash s_2$ and $\alpha(x_1) \leq^\sharp x_1^\sharp$ imply that there is an abstract transition $x_1^\sharp \vdash s_1 \rightarrow^\sharp x_2^\sharp \vdash s_2$ such that $\alpha(x_2) \leq^\sharp x_2^\sharp$.

For the proof of this, see [7]. Note that a very common way of constructing an abstract interpretation is to find monotone functions α and γ that form a *Galois connection* [1]. To do this, we need a notion of 'best' approximation, which does not always exist, and indeed does not exist for our domains. This approach has the advantage of telling us *how* to construct our abstract semantics, as opposed to constructing it first and then proving that it is safe.

In the probabilistic setting, our domains are Banach spaces rather than preordered sets, but the above approach still applies. By applying classical abstract interpretation to the probabilistic setting, we take an approach similar to Monniaux [4]. Rather than comparing measures by their *total measure*, however, we choose a much stronger comparison, which we call the *strict ordering* on measures.

Definition 4.2 Two measures μ and μ' over the same measurable space (X, σ_X) are comparable under the *strict ordering* on measures so that $\mu \leq_{\mathrm{str}} \mu'$, if:

$$\forall x \in \sigma_X . \mu(x) \leq \mu'(x)$$

Our motivation for this ordering is that it allows the measures on any set to be compared. For example, an approximation of the probability of taking a control-flow decision can be obtained by looking at the measure on the set of values that satisfy the condition. Note that any measure that approximates a probability measure (other than itself) will be a super-probability measure – if the abstraction gives a probability of 0.5 then we know that the actual probability is less than or equal to this, but can never be greater.

An alternative approach taken by Di Pierro, Wiklicky et al [6,5] is to look for a probabilistic analogue of the Galois connection. This, the Moore-Penrose pseudo inverse, gives the closest approximation to the inverse of a function, leading to a *probabilistic* notion of safety. While this approach has had much success, it is difficult to use in practice for infinite Banach spaces (i.e. continuous measures), such as the ones we consider.

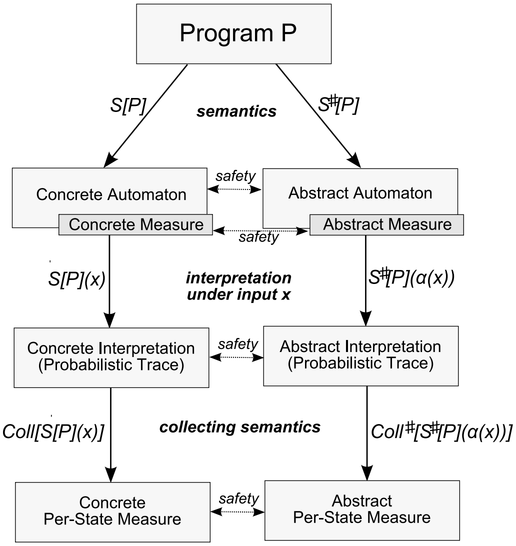


Fig. 2. Overview of our probabilistic abstract interpretation

5 Abstract Semantics

So far, we have considered the concrete domain, where any probability measure is allowed. In a practical sense, however, it is infeasible to deal with such arbitrary distributions, as we need to represent them somehow. The approach taken by Monniaux [4] is to discretise distributions to the required degree of precision and operate on these. We take a different approach, by looking for a suitable class of *continuous* distributions that are easily parameterised, and therefore lend themselves to efficient manipulation in the abstract domain.

Such a class of distributions are the *multivariate normal distributions* [9], and more generally the *truncated multivariate normal distributions*. The appeal of normal distributions is that they are preserved under linear operations, and are commonly observed in practice, due to the central limit theorem. By using a multivariate distribution, we can represent dependencies between variables in a compact way. Truncations represent control-flow constraints, which restrict the range of values that the variables can take. An overview of our abstract interpretation is shown in Figure 2. Note that at every stage we have a safety relation between the concrete and abstract domains, so that the safety of our abstraction is preserved throughout. Just like the concrete semantics, there are three stages to deriving an *abstract* property of a program:

(i) *Abstract semantics* – we compute an abstract probabilistic automaton that is a safe approximation of the concrete semantics of the program. In other words, we satisfy the relational homomorphism property (Definition 4.1).

(ii) *Abstract interpretation* – we “execute” the abstract automaton with a particular input measure. This generates a probabilistic trace that approximates the concrete one, in that the set of measures associated to each state bounds that of the concrete trace. It follows that the abstract transition probabilities are also upper bounds of the concrete transition probabilities.

The main issue here is ensuring that the abstract interpretation terminates, and to do so we employ *memoisation*, or *widening operators*, as we shall see later.

(iii) *Abstract collecting semantics* – we sum the measures associated with each state in the automaton (which may appear more than once in the trace). Since the actual measure may be difficult to calculate, we compute a safe over-approximation.

We will present each of these stages in more detail, but first we need to formally describe the abstract domain.

Since we will be over-approximating the measures in the concrete domain, the abstract measures will not in general be probability measures. We will therefore work with truncated multivariate normal *measures*, which can be thought of as truncated multivariate normal distributions, with an additional parameter denoting its total measure. For our purposes, we will describe a measure μ by the pair (μ_T, f), where f is a probability density function, and μ_T is the total measure of μ ($\mu(\mathbb{R}^n)$). Hence $\mu(X) = \mu_T \int_{x \in X} f(x)dx$. If μ has the probability density function f, then we write $\mu \sim f$.

Definition 5.1 A *multivariate normal measure*, $\mu \sim N_n(\boldsymbol{\mu}, \boldsymbol{\Sigma})$, is a measure over $(\mathbb{R}^n, \sigma_{\mathbb{R}}^{(n)})$ with the following probability density function:

$$f(\boldsymbol{x}) = \frac{1}{|\Sigma|^{\frac{1}{2}}\sqrt{2\pi}} e^{-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu})^T \Sigma^{-1} (\boldsymbol{x}-\boldsymbol{\mu})}$$

where n is the number of variables, $\boldsymbol{\mu}$ is the mean vector of length n, and $\boldsymbol{\Sigma}$ is the $n \times n$ covariance matrix. If we think of this as the joint distribution of values of a vector $\boldsymbol{X}$ of n variables, then the elements of $\boldsymbol{\Sigma}$ are such that $\sigma_{ii} = \mathrm{Var}(\boldsymbol{X}_i)$, and $\sigma_{ij} = \mathrm{Cov}(\boldsymbol{X}_i, \boldsymbol{X}_j) = \sigma_{ji}$.

It is unfortunate that the Greek letter μ is conventionally used for both measures and means, however since we are dealing with multivariate distributions, we will always use boldface $\boldsymbol{\mu}$ to refer to the mean, and lightface μ for measures.

In order to allow *truncated* multivariate normal measures, we define a truncation function $T[\boldsymbol{a}, \boldsymbol{b}]$ over measures:

Definition 5.2 The truncation function $T[\boldsymbol{a}, \boldsymbol{b}]$, where $\boldsymbol{a}$ and $\boldsymbol{b}$ are column vectors of length n, is defined for measures μ over $(\mathbb{R}^n, \sigma_{\mathbb{R}}^{(n)})$, such that:

$$T[\boldsymbol{a}, \boldsymbol{b}](\mu)(X) = \mu(X \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a} \leq \boldsymbol{x} \leq \boldsymbol{b}\})$$

The elements of $\boldsymbol{a}$ and $\boldsymbol{b}$ are from the set $\mathbb{R} \cup \{\bot, \top\}$, where $\forall x \in \mathbb{R}.\ \bot < x < \top$.

Intuitively, the truncation $T[\boldsymbol{a}, \boldsymbol{b}]$ confines measures to the rectangular region $[\boldsymbol{a}, \boldsymbol{b}]$, such that the measure on any set outside this region is zero. We can now define the class of truncated multivariate normal measures:

Definition 5.3 A measure μ is a *truncated multivariate normal measure* if it can be written in the form $T[\boldsymbol{a},\boldsymbol{b}](\mu')$, where μ' is a multivariate normal measure. When $\mu'_T = 1$ (i.e. μ' is a probability measure), we will use the shorthand $T[\boldsymbol{a},\boldsymbol{b}]N_n(\boldsymbol{\mu},\boldsymbol{\Sigma})$ to completely define μ.

Before we describe our abstraction function α and our abstract semantics, we will recall an important property of the multivariate normal distribution. For a vector of random variables $\boldsymbol{X} \sim N_n(\boldsymbol{\mu},\boldsymbol{\Sigma})$, if we apply a linear operation such that $\boldsymbol{B}$ is an $n \times n$ matrix and $\boldsymbol{c}$ is a column vector of size n, the following standard result [9] holds:

$$\boldsymbol{Y} = \boldsymbol{B}\boldsymbol{X} + \boldsymbol{c} \sim N_n(\boldsymbol{B}\boldsymbol{\mu} + \boldsymbol{c}, \boldsymbol{B}\boldsymbol{\Sigma}\boldsymbol{B}^T)$$

So that we can directly talk about an operation on measures, rather than on random variables, we introduce the operator $L[\boldsymbol{B},\boldsymbol{c}]$, which is defined as follows:

Definition 5.4 The linear operator function $L[\boldsymbol{B},\boldsymbol{c}]$, where $\boldsymbol{B}$ is an $n \times n$ matrix and $\boldsymbol{c}$ is a column vector of length n, is defined for measures μ over $(\mathbb{R}^n, \sigma_{\mathbb{R}}^{(n)})$, such that:

$$L[\boldsymbol{B},\boldsymbol{c}](\mu)(X) = \mu(\{\boldsymbol{x} \mid \boldsymbol{B}x + \boldsymbol{c} \in X\})$$

A consequence of the standard properties of multivariate normal distributions is that if $\mu \sim N_n(\boldsymbol{\mu},\boldsymbol{\Sigma})$ then $L[\boldsymbol{B},\boldsymbol{c}](\mu) \sim N_n(\boldsymbol{B}\boldsymbol{\mu} + \boldsymbol{c}, \boldsymbol{B}\boldsymbol{\Sigma}\boldsymbol{B}^T)$.

Notice that we can easily apply such linear operations precisely to multivariate normal measures, but not to truncated multivariate normal measures (which will in general not remain truncated multivariate normal after the operation [2]). To combat this, we introduce an *abstract* linear operator function $L^\sharp[\boldsymbol{B},\boldsymbol{c}]$:

Definition 5.5 The abstract linear operator function $L^\sharp[\boldsymbol{B},\boldsymbol{c}]$ is defined over truncated multivariate normal measures $T[\boldsymbol{a},\boldsymbol{b}](\mu)$, such that:

$$L^\sharp[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu) = T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}] + \boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}](\mu)$$

where $\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}] + \boldsymbol{c}$ is defined as per interval analysis.

The safety of this abstraction is established in the following theorem, which allows us to over-approximate the actual answer by reversing the order of the linear operation and the truncation.

Theorem 5.6 *For all measures μ, $L[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu) \leq_{\text{str}} L^\sharp[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu)$.*

Proof. By definition of the operators, and since we can safely apply the new truncation interval $T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}] + \boldsymbol{c})$ first (values outside this region are impossible to obtain), we have:

$$\begin{aligned}
L[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu)(X) &= \\
& T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu)(X) \\
=\ & T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}](\mu(X \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a} \le \boldsymbol{x} \le \boldsymbol{b}\})) \\
=\ & T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ \mu(\{\boldsymbol{x} \mid \boldsymbol{B}\boldsymbol{x}+\boldsymbol{c} \in X \cap \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{a} \le \boldsymbol{x} \le \boldsymbol{b}\}\}) \\
\le_{\text{str}}\ & T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ \mu(\{\boldsymbol{x} \mid \boldsymbol{B}\boldsymbol{x}+\boldsymbol{c} \in X\}) \\
=\ & T(\boldsymbol{B}[\boldsymbol{a},\boldsymbol{b}]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}](\mu)(X) = L^\sharp[\boldsymbol{B},\boldsymbol{c}] \circ T[\boldsymbol{a},\boldsymbol{b}](\mu)(X)
\end{aligned}$$

□

5.1 Relating the Concrete and Abstract Domains

We would like our concrete domain to consist of all possible measures, and our abstract domain to be the truncated multivariate normal measures, as described. Unfortunately, constructing an abstraction function from such a domain is not a simple task. Not only does it contain measures that we cannot write down, but it is difficult to satisfy the relational homomorphism property (Definition 4.1). Instead we restrict our concrete domain to those measures that can be computed by a series of linear operations and truncations applied to a multivariate normal measure. Whilst this is restrictive, it still allows us to represent a useful class of measures, and loosening this is the subject of future work. More formally:

- The *concrete domain* $\mathcal{D}$ consists of measures of the form $L[\boldsymbol{B}_n,\boldsymbol{c}_n] \circ T[\boldsymbol{a}_n,\boldsymbol{b}_n] \circ \ldots \circ L[\boldsymbol{B}_1,\boldsymbol{c}_1] \circ T[\boldsymbol{a}_1,\boldsymbol{b}_1](\mu)$, and is ordered by the strict ordering on measures.
- The *abstract domain* $\mathcal{D}^\sharp$ consists of measures of the form $T[\boldsymbol{a},\boldsymbol{b}](\mu)$, and is ordered by $\le^\sharp_{\text{str}}$. This is defined such that $T[\boldsymbol{a}_1,\boldsymbol{b}_1](\mu_1) \le^\sharp_{\text{str}} T[\boldsymbol{a}_2,\boldsymbol{b}_2](\mu_2)$ if $\mu_1 \le_{\text{str}} \mu_2$ and $[\boldsymbol{a}_1,\boldsymbol{b}_1] \subseteq [\boldsymbol{a}_2,\boldsymbol{b}_2]$.

The ordering of the abstract domain is necessarily stronger than that of the concrete domain, in order for the abstract semantics to be monotone. We can now define our abstraction function as follows:

Definition 5.7 The abstraction function $\alpha : \mathcal{D} \to \mathcal{D}^\sharp$ of a measure $\mu \in \mathcal{D}$ is defined inductively as follows:

$$\begin{aligned}
\alpha(\mu) &= \mu \text{ if } \mu \text{ is a multivariate normal measure} \\
\alpha(T[\boldsymbol{a},\boldsymbol{b}](\mu)) &= T[\boldsymbol{a},\boldsymbol{b}](\alpha(\mu)) \\
\alpha(L[\boldsymbol{B},\boldsymbol{c}](\mu)) &= L^\sharp[\boldsymbol{B},\boldsymbol{c}](\alpha(\mu))
\end{aligned}$$

Note that $T[\boldsymbol{a}_2,\boldsymbol{b}_2] \circ T[\boldsymbol{a}_1,\boldsymbol{b}_1] = T([\boldsymbol{a}_2,\boldsymbol{b}_2] \cap [\boldsymbol{a}_1,\boldsymbol{b}_1])$ if the intersection of the intervals is non-empty, and $\lambda x.0$ (the zero measure) otherwise. For linear operators, $L[\boldsymbol{B}_2,\boldsymbol{c}_2] \circ L[\boldsymbol{B}_1,\boldsymbol{c}_1] = L[\boldsymbol{B}_2\boldsymbol{B}_1, \boldsymbol{B}_2\boldsymbol{c}_1 + \boldsymbol{c}_2]$.

5.2 Stage 1 – Abstract Semantics

The abstract semantics of a program is an automaton with the same states as its concrete semantics, but with transitions that operate on truncated multivariate normal measures, rather than arbitrary measures. Since the structure of the automaton is the same as for the concrete semantics, we need only describe the abstraction of the functions on transitions. Hence we need an abstraction for the probabilistic semantics of assignment, $[\![\cdot]\!]^\sharp_p$, and for the semantics of conditional operators, $e^\sharp_{[\![\cdot]\!]}$. Given these, $[\![C]\!]^\sharp_{pa} = [\![C]\!]_{pa}\{[\![\cdot]\!]^\sharp_p \;/\; [\![\cdot]\!]_p,\; e^\sharp_{[\![\cdot]\!]} \;/\; e_{[\![\cdot]\!]}\}$.

We begin with the abstract semantics of assignment:

$$[\![X_i{:=}E]\!]^\sharp_p(\mu) = L^\sharp[\boldsymbol{B}, \boldsymbol{c}](\mu)$$

where $\boldsymbol{B}$ and $\boldsymbol{c}$ describe the operation of E, such that if the program's variables have state $\boldsymbol{x}$ (a column vector of length n), then $[\![X_i{:=}E]\!](\boldsymbol{x}) = \boldsymbol{B}\boldsymbol{x} + \boldsymbol{c}$.

For the abstract semantics of conditional operators, we have the following:

$$\begin{array}{llll}
e^\sharp_{[\![\texttt{true}]\!]}(\mu) & = \mu & e^\sharp_{[\![\texttt{false}]\!]}(\mu) & = \lambda x.0 \\
e^\sharp_{[\![X_i \leq c]\!]}(\mu) & = T[\bot, \boldsymbol{b}](\mu) & e^\sharp_{[\![\neg(X_i < c)]\!]}(\mu) & = T[\boldsymbol{a}, \top](\mu) \\
e^\sharp_{[\![X_i \leq c]\!]}(\mu) & = T[\bot, \boldsymbol{b}](\mu) & e^\sharp_{[\![\neg(X_i < c)]\!]}(\mu) & = T[\boldsymbol{a}, \top](\mu)
\end{array}$$

where $\boldsymbol{b}_i = c$, $\boldsymbol{b}_j = \top$ (for $j \neq i$), and $\boldsymbol{a}_i = c$, $\boldsymbol{a}_j = \bot$ (for $j \neq i$). Note that there is a slight over-approximation for the $\{<, >\}$ comparisons, to avoid distinguishing between open and closed truncation intervals in the abstract domain.

It remains to prove that the abstract semantics is safe; that is to say, that it satisfies the relational homomorphism property (Definition 4.1).

Theorem 5.8 *For all programs P, for all transitions $s_1 \xrightarrow{M} s_2 \in [\![P]\!]_{\mathrm{pa}}$ there exists an abstract transition $s_1 \xrightarrow{M^\sharp} s_2 \in [\![P]\!]^\sharp_{\mathrm{pa}}$ such that for all measures $\mu \in \mathcal{D}$, if $\alpha(\mu) \leq^\sharp_{\mathrm{str}} \mu^\sharp \in D^\sharp$ then $\alpha(M(\mu)) \leq^\sharp_{\mathrm{str}} M^\sharp(\mu^\sharp)$.*

Proof. Firstly, we note that there is a bijection between concrete and abstract transitions, which ensures a unique $M^\sharp$ for each M. The structure of both M and $M^\sharp$ consists of a sequence of truncation operators and linear operators (ignoring the identity operator as trivial). We prove the theorem by induction on this structure, starting with the two base cases.

For a truncation operator, we note that the abstract semantics generates an interval that over-approximates the actual set of values that satisfy the condition. Hence if $\alpha(M(\mu)) = \alpha(T[\boldsymbol{a}, \boldsymbol{b}](\mu)) = T[\boldsymbol{a}, \boldsymbol{b}](\alpha(\mu))$ (using the definition of α), then $M^\sharp(\mu^\sharp) = T[\boldsymbol{a}', \boldsymbol{b}'](\mu^\sharp)$, where $[\boldsymbol{a}, \boldsymbol{b}] \subseteq [\boldsymbol{a}', \boldsymbol{b}']$. Hence $\alpha(M(\mu)) \leq^\sharp_{\mathrm{str}} M^\sharp(\mu^\sharp)$ since $\alpha(\mu) \leq^\sharp_{\mathrm{str}} \mu^\sharp$.

If M and $M^\sharp$ are linear operators then they have the forms $L[\boldsymbol{B}, \boldsymbol{c}]$ and $L^\sharp[\boldsymbol{B}, \boldsymbol{c}]$ respectively. Let $\alpha(\mu) = T[\boldsymbol{a}_1, \boldsymbol{b}_1](\mu_1)$ and $\mu^\sharp = T[\boldsymbol{a}_2, \boldsymbol{b}_2](\mu_2)$. Then:

$$\begin{aligned}
\alpha(L[\boldsymbol{B},\boldsymbol{c}](\mu)) &= L^\sharp[\boldsymbol{B},\boldsymbol{c}](\alpha(\mu)) \text{ from the definition of } \alpha \\
&= L^\sharp[\boldsymbol{B},\boldsymbol{c}](T[\boldsymbol{a}_1,\boldsymbol{b}_1](\mu_1)) \\
&= T(\boldsymbol{B}[\boldsymbol{a}_1,\boldsymbol{b}_1]+\boldsymbol{c}) \circ L[\boldsymbol{B},\boldsymbol{c}](\mu_1) \\
&\leq^\sharp_{\text{str}} T(\boldsymbol{B}[\boldsymbol{a}_2,\boldsymbol{b}_2]+\boldsymbol{c})L[\boldsymbol{B},\boldsymbol{c}](\mu_2) \text{ since } \alpha(\mu) \leq^\sharp_{\text{str}} \mu^\sharp \\
&= L^\sharp[\boldsymbol{B},\boldsymbol{c}](T[\boldsymbol{a}_2,\boldsymbol{b}_2](\mu_2)) = L^\sharp[\boldsymbol{B},\boldsymbol{c}](\mu^\sharp)
\end{aligned}$$

This uses the fact that if $\mu_1 \leq_{\text{str}} \mu_2$ then $L[\boldsymbol{B},\boldsymbol{c}](\mu_1) \leq_{\text{str}} L[\boldsymbol{B},\boldsymbol{c}](\mu_2)$.

Finally, the inductive step completes the proof. If $M = M_2 \circ M_1$ and $M^\sharp = M_2^\sharp \circ M_1^\sharp$, such that M_2 and $M_2^\sharp$ are base operators, then by the induction hypothesis the theorem holds for M_1 and $M_1^\sharp$. Hence $\alpha(M_1(\mu)) \leq^\sharp_{\text{str}} M_1(\mu^\sharp)$, and since M_2 and $M_2^\sharp$ are both either truncation or linear operators, it follows that $\alpha(M_2 \circ M_1(\mu)) \leq^\sharp_{\text{str}} M_2 \circ M_1(\mu^\sharp)$ holds by the above cases. □

5.3 Stage 2 – Abstract Interpretation

The abstract interpretation of our abstract automaton is defined in the same way as for the concrete automaton, inducing an abstract transition relation $\rightarrow^\sharp$ when given an input distribution. This has states $\mu^\sharp \vdash s$, where $\mu^\sharp$ is a truncated multivariate normal measure, and s is a state in the abstract automaton. The safety of the abstract interpretation is a consequence of Theorem 5.8, which ensures that the concrete trace generated from the measure μ is simulated by the abstract traces generated from all $\mu^\sharp \geq^\sharp_{\text{str}} \alpha(\mu)$.

As it stands, the abstract interpretation may not terminate, and even if it does, it may take a long time to do so, due to the presence of loops. To avoid this problem, we *memoise* it, using a *widening operator* ∇, which allows us to jump to an over-approximation of all the measures that can ever occur at a given state. This requires a slight modification to our abstract semantics, so that the states in the trace are assigned a set of measures, $\mathcal{M}$, rather than a single measure μ. We say that $\mathcal{M} \sqsubseteq \mathcal{M}'$ if $\forall m \in \mathcal{M}.\ \exists m' \in \mathcal{M}'.\ m \leq^\sharp_{\text{str}} m'$. We denote by $\top$ the set of all truncated multivariate normal measures, which is the worst case approximation when we lack precision.

To perform the memoised abstract interpretation, we keep a lookup table of the set of measures $\mathit{lookup}(s)$ that have so far been assigned to each state s. We allow such sets to either be empty (when we have yet to visit the state), $\top$ (when we cannot say anything about the measures that can occur), contain just one measure, or contain a parameterised set of measures. For the purposes of this paper, the latter must be of the form $\mathcal{M} = \{T[\boldsymbol{a},\boldsymbol{b}] \circ N_n(\boldsymbol{\mu} + c\boldsymbol{x}, \boldsymbol{\Sigma}) \mid c \in \mathbb{N}\}$. This corresponds to a set of multivariate normal measures with linearly varying means that are truncated to the interval $[\boldsymbol{a},\boldsymbol{b}]$. This would occur in a state within a loop where the variables are only incremented or decremented by a constant value.

To execute the abstract semantics, we do the following. If we are currently at state $\mathcal{M} \vdash s$ then for each transition $s \xrightarrow{M} s'$ in the automaton, we generate a transition $\mathcal{M} \vdash s \rightarrow^\sharp M(\mathcal{M}) \nabla \mathit{lookup}(s') \vdash s'$ in the abstract trace. Finally, we update the lookup table so that $\mathit{lookup}(s') = M(\mathcal{M}) \nabla \mathit{lookup}(s')$. The map

$M(\mathcal{M})$ is defined such that $M(\{\}) = \{\}$, $M(\top) = \top$, $M(\{\mu\}) = \{M(\mu)\}$, and $M(\{T[\boldsymbol{a},\boldsymbol{b}] \circ N_n(\boldsymbol{\mu} + c\boldsymbol{x}, \boldsymbol{\Sigma})\}) = \{M \circ T[\boldsymbol{a},\boldsymbol{b}] \circ N_n(\boldsymbol{\mu} + c\boldsymbol{x}, \boldsymbol{\Sigma})\}$ (for $c \in \mathbb{N}$).

The widening operator is defined as follows, with $\{\}\nabla\mathcal{M} = \mathcal{M}$, $\top\nabla\mathcal{M} = \top$, and $\mathcal{M}_1\nabla\mathcal{M}_2 = \mathcal{M}_2\nabla\mathcal{M}_1$:

$$\{\mu\}\nabla\{T[\boldsymbol{a},\boldsymbol{b}] \circ N_n(\boldsymbol{\mu} + c\boldsymbol{x}, \boldsymbol{\Sigma})\}$$
$$= \begin{cases} \{T[\boldsymbol{a},\boldsymbol{b}]N_n(\boldsymbol{\mu} + c\boldsymbol{x}, \boldsymbol{\Sigma})\} & \text{if } \mu \in \{T[\boldsymbol{a},\boldsymbol{b}]N_n(\boldsymbol{\mu} + c\boldsymbol{x}, \boldsymbol{\Sigma})\} \\ \top & \text{otherwise} \end{cases}$$

$$\{T[\boldsymbol{a}_1,\boldsymbol{b}_1] \circ N_n(\boldsymbol{\mu}_1 + c\boldsymbol{x}_1, \boldsymbol{\Sigma}_1)\}\nabla\{T[\boldsymbol{a}_2,\boldsymbol{b}_2] \circ N_n(\boldsymbol{\mu}_2 + c\boldsymbol{x}_2, \boldsymbol{\Sigma}_2)\}$$
$$= \begin{cases} \{T([\boldsymbol{a}_1,\boldsymbol{b}_1] \cap [\boldsymbol{a}_2,\boldsymbol{b}_2])N_n(\boldsymbol{\mu}_1 + c\boldsymbol{x}_1, \boldsymbol{\Sigma}_1)\} & \text{if } \boldsymbol{\mu}_1 = \boldsymbol{\mu}_2 \wedge \boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 \wedge \boldsymbol{x}_1 = \boldsymbol{x}_2 \\ \top & \text{otherwise} \end{cases}$$

$$\{T[\boldsymbol{a}_1,\boldsymbol{b}_1]N_n(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)\}\nabla\{T[\boldsymbol{a}_2,\boldsymbol{b}_2]N_n(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)\}$$
$$= \begin{cases} \{T[\boldsymbol{a}_1,\boldsymbol{b}_1]N_n(\boldsymbol{\mu}_1 + c\boldsymbol{x}, \boldsymbol{\Sigma}_1)\} & \text{if } [\boldsymbol{a}_2,\boldsymbol{b}_2] \subseteq [\boldsymbol{a}_1,\boldsymbol{b}_1] \wedge \boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2 \wedge \boldsymbol{x} = \boldsymbol{\mu}_2 - \boldsymbol{\mu}_1 \\ \top & \text{otherwise} \end{cases}$$

where c ranges over $\mathbb{N}$.

The interesting case is the last one, which identifies loops whose variables are only incremented or decremented by a constant value. This occurs when the covariance matrix is unmodified, meaning that the variables have only been shifted by a constant amount. We can extend this to detect other types of loop activity (for example, a multiplicative update), but that is beyond the scope of this paper.

5.4 Stage 3 – Abstract Collecting Semantics

We conclude this paper with a brief discussion of the abstract collecting semantics. As we have seen, our memoised abstract interpretation associates a set of measures to each state in the abstract semantics. We are not interested in the set itself, however, but in the *sum* of the measures it contains. Thus our collecting semantics can be seen as a way to safely approximate this sum to a measure that is easier to compute. This need not be a truncated multivariate normal measure as the result of this final stage of analysis is not needed for further computation.

At present, the only general solution we have is a numerical one. To compute an upper bound of the measure on a particular interval $[\boldsymbol{a}, \boldsymbol{b}]$, we iteratively sum all the measures in the set, applied to this interval. Although the set of measures will in general be infinite, we note that the iterated sum will quickly converge as the residual probability mass exponentially decreases. It is easy to calculate such measures from a truncated multivariate normal measure, by performing eigenvalue decomposition to separate it into independent truncated normal measures. This loses some precision, since we over-approximate the truncation interval, but it is a safe approximation.

Analytical solutions certainly exist, where we compute a single measure as an upper approximation, but we have yet to find one that is both computationally easy to compute, and gives a satisfactory precision in general.

6 Conclusions

In this paper, we presented an abstract interpretation of a probabilistic automaton semantics for a simple imperative language. We believe that this work is complementary to other approaches to probabilistic abstract interpretation, allowing an efficient approximation to the behaviour of programs whose input is governed by a probability distribution. There is clearly some way to go in terms of improving this approach, for example finding a better abstract collecting semantics and a greater range of widening operators. We also need to carry out some larger case studies, to investigate the precision of the bounds in comparison to other methodologies.

The ultimate aim of this work is to provide a formal framework for the ideas presented in [8], where we attempt to derive stochastic models of communication protocols directly from source code. The advantage of abstract interpretation is that it can be easily automated, and therefore fits in well with the aim of providing tools that can be used by real developers. We feel that the abstract interpretation presented in this paper is an important milestone towards this goal.

References

[1] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[2] W. C. Horrace. Some results on the multivariate truncated normal distribution. In *Journal of Multivariate Analysis*, volume 94, pages 209–221, 2005.

[3] D. Kozen. Semantics of probabilistic programs. *Journal of Computer and System Sciences*, 22(3):328–350, 1981.

[4] D. Monniaux. Abstract interpretation of probabilistic semantics. In *Seventh International Static Analysis Symposium (SAS'00)*, number 1824 in Lecture Notes in Computer Science, pages 322–339. Springer-Verlag, 2000.

[5] A. Di Pierro, C. Hankin, and H. Wiklicky. Quantitative static analysis of distributed systems. *Journal of Functional Programming*, 15(5):703–749, 2005.

[6] A. Di Pierro and H. Wiklicky. Probabilistic abstract interpretation and statistical testing. In *PAPM-PROBMIV*, pages 211–212, 2002.

[7] D. Schmidt. Abstract interpretation in the operational semantics hierarchy. Technical report, Kansas State University, 1997.

[8] M. J. A. Smith. Stochastic modelling of communication protocols from source code. In *Proceedings of the 5th Workshop on Quantitative Aspects of Programming Languages (QAPL)*, 2007.

[9] Y. L. Tong. *The Multivariate Normal Distribution.* Springer-Verlag, 1990.