Probabilistic Output Analysis by Program Manipulation

Mads Rosendahl Maja H. Kirkeby*
Computer Science, Roskilde University, Denmark
{madsr,majaht}@ruc.dk

The aim of a probabilistic output analysis is to derive a probability distribution of possible output values for a program from a probability distribution of its input. We present a method for performing the output analysis, based on program transformation techniques. It generates a probability function as a possibly uncomputable expression and transforms that into a closed form expression. The probability functions are viewed as programs in a separate language in which they may be analysed, transformed, and approximated. We focus on programs where the possible input follows a known probability distribution. Tests in programs are not assumed to satisfy the Markov property of having fixed branching probabilities independently of previous history.

1 Introduction

The aim of a probabilistic output analysis (POA) is to derive a probability distribution for output values from a probability distribution for input to a program. Internal properties of a program can also be analysed in this way by instrumenting programs with step-counters for complexity analysis [30] or energy consumption measures [23].

When analysing energy consumption, probability distributions may provide more useful information than boundaries. Wierman et al. states that *"global energy consumption is affected by the average case, rather than the worst case"* [38]. Also in scheduling *"an accurate measurement of a tasks average-case execution time (ACET) can assist in the calculation of more appropriate deadlines"* [18]. For a subset of programs a precise average case execution time can be found using static analysis [13, 33, 15]. In some cases the POA delivers not only an accurate output average but the more descriptive accurate output distribution. In other cases the POA must over-approximate the probability distribution and the expected value (average case result) will be approximated safely as a range. Another application area for POA is in temperature management, where worst-case bounds are important [34]. Because POA return distributions it can be used to calculate the probability of energy consumptions above a certain limit, and thereby indicating the risk of over-heating.

The main contribution in this paper is to present a technique for probabilistic analysis where the analysis is seen as a program-to-program translation. This means that the transformation to closed form is a source code program transformation problem and not specific to the analysis. Any necessary approximation in the analysis is also performed at the source code level. The technique also makes it possible to balance the precision of the analysis against the brevity of the result.

The method in this paper is inspired by the techniques used in automatic complexity analysis. Wegbreit's Metric system [37] laid the ground work for many later systems with an aim of deriving least, worst and average case complexity measures. Later works in this area have focused on worst case complexity [30, 2, 24] with advanced systems that can analyse realistic programs. The approach in this paper

*The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement no 318337, ENTRA - Whole-Systems Energy Transparency.

uses an approach similar to [30] in that we derive the probability function without approximations but only in the last phase introduce approximations. We transform the original program into a program that computes the probability distribution. The intermediate stage is then a potential subject of further analysis based on abstract interpretation. This program can be analysed, transformed, and approximated. It is thus an alternative to deriving cost relations directly from the program [2, 24] or expressing costs as abstract values in a semantics for the language.

As with automatic complexity analysis the aim of probabilistic output analysis is to express the result as a parameterized expression. The time complexity of a program should be expressed as a closed form expression in the input size and for probabilistic output analysis the aim is to express the probability of output values of the program as a function in output values and input size or range. As a small example let us consider the addition `add` of two independent integer values `x` and `y` evenly distributed from 1 to n. It is a tail-recursive program where the output distribution is well-known to be a pyramid shaped distribution. The input probability distributions should also be expressed in the language as they are part of the transformational approach to obtain the output probability distribution.

```
add(x,y) = if(x=0) then y else add(x-1,y+1)
px(x,n) = if(x >= 1 and x <= n) then 1/n else 0
py(y,n) = if(y >= 1 and y <= n) then 1/n else 0
```

Our probabilistic output analysis returns a function describing the probability distribution of the output:

```
padd(z,n) =
  1/(n*n)*max(min(n,z-1) - max(1,z-n) + 1,0)
```

The analysis can also be used for more complex input distributions and programs but it will not always be able to reduce it to closed form. If this is not possible we will approximate the distribution and thus get an over approximation of the extreme cases and a range for the expected value. If input values are not independent we can specify a joint distribution for the values. Values do not have to be restricted to a finite range but for infinite ranges the distribution would converge to zero towards the limit.

2 Probability distributions

The analysis presented here is based on using a discrete set of values for input and output. The set will be finite or countable and we will use discrete probability distributions. It is also possible to use an uncountable set of values or a combination of discrete and continuous random variables if one uses cumulative probability measures in the analysis. This will be discussed further later in the paper.

We consider the input to a program as a discrete random variable and the input probability distribution is then a probability measure that to an event of input having a given value assigns a value between 0 and 1. This is also often referred to as the *probability mass function* in the discrete case, and in the continuous case one will use a *probability density functions*. We will use the phrase *probability function* to denote mappings from single values (input or output) to a probability or number between 0 and 1, and we will use upper case P letters to denote such functions.

Definition 1 (input probability) *For a countable set X an input probability function is a mapping $P_x : X \to \{r \in \mathbb{R} \mid 0 \leq r \leq 1\}$, where*

$$\sum_{x \in X} P_x(x) = 1 \, .$$

We define the output probability distribution for a program p in a forward manner. It is the *weight* or sum of all probabilities of input values where the program returns the desired value z as output.

Definition 2 (output probability) *Given a program,* $\mathtt{p} : X \to Z$ *and a probability distribution for the input,* P_X*, the output probability function,* $P_{\mathtt{p}}(z)$*, is defined as:*

$$P_{\mathtt{p}}(z) = \sum_{x \in X \wedge \mathtt{p}(x)=z} P_X(x) \; .$$

Note that Kozen also uses a similar forward definition [21], whereas Monniaux constructs the inverse and expresses the relationship in a backwards style [25].

Lemma 1 *The output probability distribution,* $P_{\mathtt{p}}(z)$*, satisfies*

$$0 \leq \sum_z P_{\mathtt{p}}(z) \leq 1 \; .$$

The program may not terminate for all input and this means that the sum may be less than one. If we expand the domain Z with an element to denote non-termination, $Z_\perp$, the total sum of the output distribution $P_{\mathtt{p}}(z)$ would be 1.

Approximations of probability distributions. The output analysis cannot necessarily derive the precise probability distribution. Various approaches to approximations of probability distributions have been proposed and can be interpreted as *imprecise probabilities* [1, 11, 9]. Dempster-Shafer structures [17, 3] and P-boxes [12] can be used to capture and propagate uncertainties of probability distributions. There are several results on extending arithmetic operations to probability distributions for both known dependencies between random variables and when the dependency is unknown or only partially known [4, 6, 19, 35, 39]. Algorithms for lifting basic operations on numbers to basic operations on probability distributions can be used as abstractions in static analysis based on abstract interpretation. Our approach uses the P-boxes as bounds of probability distributions. P-boxes are normally expressed in terms of the cumulative probability function but we will here use the probability mass function. We do not, however, use the various basic operations on P-boxes but apply approximations to a probability program such that it forms a P-box.

Definition 3 (over and under approximation) *For a distribution* $P_{\mathtt{p}}$ *an over and under approximation (*$\overline{P}$ *and* $\underline{P}$*) of the distribution satisfies the conditions:*

$$\overline{P}_{\mathtt{p}} : \forall z . P_{\mathtt{p}}(z) \leq \overline{P}_{\mathtt{p}}(z) \leq 1$$
$$\underline{P}_{\mathtt{p}} : \forall z . 0 \leq \underline{P}_{\mathtt{p}}(z) \leq P_{\mathtt{p}}(z) \; .$$

The aim of the output analysis is to derive as tight approximations $\underline{P}$ and $\overline{P}$ as possible.

Lemma 2 *Given the definition for over and under approximation they will have boundaries for their total weights as*

$$0 \leq \sum_z \overline{P}_{\mathtt{p}}(z) \leq \infty \qquad 0 \leq \sum_z \underline{P}_{\mathtt{p}}(z) \leq 1 \; .$$

When $\underline{P}_{\mathtt{p}} = \overline{P}_{\mathtt{p}}$ the total weight for each function will be equal to the total weight of $P_{\mathtt{p}}$, according to definition 3. For terminating programs the total weight is 1.

Expected value. Provided that the output from the program is numerical, one may be interested in the average output value of the program. In this context this is *the expected value* of the output distribution. If the program does not terminate for all input it is not clear how to define the expected value because non-termination may indicate a possibly infinite output value. As part of the further analysis we need a guarantee that the program terminates. If the sum of the $\underline{P}_{\mathrm{p}}$ is one then we know that the program terminates for all possible input (*i.e.* input with probability greater than zero).

Lemma 3 *The under approximation of a probability distribution satisfies*

$$\sum_z \underline{P}_{\mathrm{p}}(z) = 1 \Rightarrow \sum_z P_{\mathrm{p}}(z) = 1 \ .$$

The expected value of the output distribution is defined as the weighted average of the distribution.

Definition 4 (expected value) *The expected value of the output distribution is defined as*

$$E_{\mathrm{p}} = \sum_z z \cdot P_{\mathrm{p}}(z) \ .$$

If we cannot analyse the program precisely, we can use the over approximation to compute an interval for the expected value. We cannot use the approximation $\overline{P}_{\mathrm{p}}$ directly as its weight is not necessarily 1. Using $\overline{P}_{\mathrm{p}}$ we can create two new probability distributions, each with a total weight of 1. One that favours the lower values, and one that favours the higher values. These two can then be used to calculate a lower and an upper bound for the expected values.

Definition 5 (expected value interval) *For an over approximation of a probability distribution $\overline{P}_{\mathrm{p}}$ we define an over and under accumulation ($F^{\uparrow}$ and $F^{\downarrow}$) and over and under expected value ($E^{\uparrow}$ and $E^{\downarrow}$).*

$$\begin{aligned}
F^{\uparrow}(z) &= \min(\sum_{v \leq z} \overline{P}_{\mathrm{p}}(v), 1) \\
F^{\downarrow}(z) &= \max(1 - \sum_{v \geq z} \overline{P}_{\mathrm{p}}(v), 0) \\
E^{\downarrow} &= \sum_z z \cdot (F^{\uparrow}(z) - F^{\uparrow}(dec(z))) \\
E^{\uparrow} &= \sum_z z \cdot (F^{\downarrow}(z) - F^{\downarrow}(dec(z))) \\
dec(z) &= \max\{v \in Z \mid v < z\} \qquad .
\end{aligned}$$

Notice that an expected value based on an over approximation of the accumulated probability gives an under approximation of the expected value. If the output space Z are integers then the *dec* function will just subtract one from its argument.

Lemma 4 (expected value interval) *For a terminating program the expected value can be approximated by an interval from the over approximation of the probability distribution.*

$$E^{\downarrow} \leq E_{\mathrm{p}} \leq E^{\uparrow} \ .$$

Externalise resource usage. The output analysis can be used to analyse internal properties of the program provided these properties are externalised. As in automatic worst case complexity analysis [30], this may be done by instrumenting the program with step counting information. Similarly we

might instrument programs with energy consumption based on low level energy models for operations [23] to be able to analyse programs for average energy consumption.

An operational or denotational semantics of a simple first order functional programming language would normally describe programs as mappings of input values to output values. The time, space or energy required to perform the computation would normally not be part of the semantics. The simplest form of a resource analysis is to count the number of basic operations that a computation would require. An automatic complexity analysis [30] is then based on a semantics that has been extended (or instrumented) with step-counting information so that the meaning of a program is a mapping of input values to a tuple of the output and the number of steps. If we write this semantics as an interpreter in the source language we can convert a program to a step-counting version of the program by partial evaluation. In this way the complexity analysis has been transformed into an output analysis of the program. The aim of the complexity analysis is then to generate an over-approximation of the (second component of the) possible output as a function of the size of the possible input. If we instrument the semantics with other types of resource information we can analyse programs with respect to these properties. Some automatic complexity analysis systems are based on translating programs into cost relations [2] or cost equations [24]. These approaches are then used as approximations of an instrumented semantics that captures the cost of computations.

The functional language we use here may be seen as a meta-language for analysis since we should extend source programs with resource information before the actual analysis. One can also use it as a meta-language for analysing programs in other languages. This can be achieved by having a step-counting interpreter for the other language written in the first order functional language. When analysing software for embedded systems the programs are often written in simple c-like languages which should then be translated into this meta-language.

The challenge of approximation. Analysis of probabilistic behaviour introduces some new challenges compared to worst case analysis. It is well known that a function of expected values is not necessarily the same as the expected value of the function. There are a number of other potential pitfalls when making approximations in a probabilistic setting. One might assume that conditions in a program can be assigned a fixed probability of being true independently of previous execution paths in the program. One might also assume that variables have independent probability distributions. An unfortunate effect of using independence as an approximation is that it tends to under approximate the extreme cases. In a throw of two dice the sum of 12 has probability $1/36$ if we can assume independence. If (by some magic) they always showed the same the probability increases to $1/6$. The situation is well-known in the insurance industry and for financial risk management (valuation of derivatives) where one may want to over approximate the risk of extreme event when events are not guaranteed to be independent. One approach to handle such situations is the use of copulas [5] and comonotonicity of probability measures [10].

3 Transformation Based Analysis

Our analysis is based on a small first order functional language with primitive recursion. The first step of the analysis is to translate programs into a new language of probability distribution programs. We will then use analysis and transformation techniques to transform the probability distributions into closed form. Failing that, we may approximate the distribution with an upper and lower approximation ($\overline{P}$ and $\underline{P}$).

Programs have the form of a collection of functions

$$\begin{aligned} f_1(x_1,\ldots,x_n) &= e_1 \\ &\vdots \\ f_n(x_1,\ldots,x_n) &= e_n \end{aligned}$$

The language uses a base set D of values for simple expressions, and functions in a program denotes mappings from tuples of values to values $D^* \to D$.

The base set of values will not be further restricted here, nor do we specify the exact set of basic operations in the language. The first function in the program is called externally and for that function we have an input probability distribution P_x specified as a symbolic expression e_x.

There are two forms of function: non-recursive and primitive recursive functions.

$$f(x_1,\ldots,x_n) = \textbf{ if } (b(x_1,\ldots,x_n)) \textbf{ then } g(x_1,\ldots,x_n) \textbf{ else } f(e_1\ldots,e_n)$$

Non-recursive functions have righhand-sides that are built from simple operations conditional expressions and function calls to non-recursive and primitive recursive functions.

3.1 Probability distribution program

When constructing the probability distribution program, in its raw form, we use two new language constructs: Sums over the (possibly) infinite set of all input values in D^* and a constraint function C. The constraint function eases the handling of boundaries and is defined as

$$C(condition) = \begin{cases} 1 & \text{if } condition = true \\ 0 & \text{otherwise} \end{cases}$$

This definition is related to the indicator function [26] or characteristic function for membership of sets. We also extend the language with a finite product construction which will be used for unfolding primitive recursion.

The output distribution program is expressed in a language similar to the original program but extended with two classes of functions. The orginal functions of type $D^* \to D$ and probability functions of type $D^* \to [0,1]$. One of these functions will be the output distribution function of type $D \to [0,1]$.

The basis for a probability distribution program is the program itself and a joint input distribution function. The input distribution is expressed as a function

$$P_x(x_1,\ldots,x_n) = e_x$$

that expresseses the probability of each possible input value. If the arguments are independent the function can be written as a product of the probability distribution of each parameter

$$P_x(x_1,\ldots,x_n) = P_{x1}(x_1)\cdots P_{xn}(x_n)$$

The raw form of the probability distribution program is defined as follows. Given an output value tuple, the distribution program sums the probabilities for all input value tuples that the original program

maps to the output value tuple. The probability distribution program is defined as follows.

$$
\begin{aligned}
&P_f(z) = \sum_{x_1} \cdots \sum_{x_n} P_x(x_1,\ldots,x_n) \cdot C(z = f_1(x_1,\ldots,x_n)) \\
&P_x(x_1,\ldots,x_n) = e_x \\
&f_1(x_1,\ldots,x_n) = e_1 \\
&\vdots \\
&f_n(x_1,\ldots,x_n) = e_n
\end{aligned}
$$

We interpret a probability distribution program as a program that can be transformed and analysed. The second phase is to unfold function calls and the following phase is to try to remove the infinite summations. The aim of the subsequent transformation stages is to remove the infinite summations from the program and in the process the functions from the original program.

3.2 Unfolding

In this phase we unfold function calls in the program. We will introduce the central transformation rules for unfolding calls to functions in the original program based on the syntactical structure.

Function calls. Simple calls to functions can be unfolded directly. Calls to primitive recursive function can be composed but each call can be analysed separately by constructing a joint input distribution function to the call. For such function calls we rewrite the program as follows

$$
\begin{aligned}
&\sum_{x_1} \cdots \sum_{x_n} P(x_1,\ldots,x_n) \cdot C(z = g(e_1,\ldots,e_n)) \\
&= \sum_{u_1} \cdots \sum_{u_n} P_c(u_1,\ldots,u_n) \cdot C(z = g(u_1,\ldots,u_n)) \cdot \\
&\quad P_c(u_1,\ldots,u_n) = \sum_{x_1} \cdots \sum_{x_n} P(x_1,\ldots,x_n) \cdot C(u_1 = e_1) \cdots C(u_n = e_n) \ .
\end{aligned}
$$

This rule extends the program with an extra probability function P_c. As we assume the programs do not have unrestricted recursion we will only generate a bounded number of extra probability functions.

Conditional expressions. For conditional expressions we use the following rule

$$
\begin{aligned}
&\sum_{x_1} \cdots \sum_{x_n} P(x_1,\ldots,x_n) \cdot C(z = \textbf{ if } (b(x_1,\ldots,x_n)) \textbf{ then } g(x_1,\ldots,x_n) \textbf{ else } h(x_1,\ldots,x_n)) \\
&= \sum_{x_1} \cdots \sum_{x_n} P(x_1,\ldots,x_n) \cdot \\
&\qquad (C(b(x_1,\ldots,x_n)) \cdot c(z = g(x_1,\ldots,x_n)) + C(\neg b(x_1,\ldots,x_n)) \cdot c(z = h(x_1,\ldots,x_n))) \ .
\end{aligned}
$$

Unfolding primitive recursion. For primitive recursion we collect the probability of a given result being returned for any number of recursive calls. The condition may never evaluate to true for a certain input (non-termination), and in that situation the sum of output probabilities will be less than 1.

The recursive functions have the form

$$
f(x_1,\ldots,x_n) = \textbf{ if } (b(x_1,\ldots,x_n)) \textbf{ then } g(x_1,\ldots,x_n) \textbf{ else } f(e_1 \ldots, e_n)
$$

and they should be analysed for all input probability distributions we detect at calls to these functions.

The transformation for the primitive recursion form is

$$\sum_{x_1}\cdots\sum_{x_n} P(x_1,\ldots,x_n)\cdot C(z = \textbf{ if } (b(x_1,\ldots,x_n)) \textbf{ then } g(x_1,\ldots,x_n) \textbf{ else } f(e_1\ldots,e_n))$$
$$= \sum_{x_1}\cdots\sum_{x_n} P(x_1,\ldots,x_n)\sum_{i=0}^{\infty}\prod_{j=0}^{i-1} C(\neg b(h(j,x_1,\ldots,x_n)))\cdot C(b(h(i,x_1,\ldots,x_n)))$$
$$\cdot C(z = g(h(i,x_1,\ldots,x_n)))$$

where

$$h(i,x_1,\ldots,x_n) = \textbf{ if } (i=0) \textbf{ then } \langle x_1,\ldots,x_n\rangle \textbf{ else } h(i-1,e_1,\ldots,e_n)\ .$$

In the transformed expression we introduce two variables: i that represents the number of recursive calls, and j that represents all previous recursions for the i under investigation (when i is 0 the term $\prod_{j=0}^{i-1} C(\neg b(h(j,x_1,\ldots,x_n)))$ evaluates to 1). The new function $h(i,x_1,\ldots,x_n)$ describes the evaluation of the expressions $\langle e_1,\ldots,e_n\rangle$, i times. Only when the ith condition is *true* and all previous conditions are *false* then can the expression evaluate to a probability above 0.

3.3 Symbolic summation

In the previous phase we unfolded calls to functions in the original program. The aim of the next phase is to use algebraic transformation techniques to remove summations. The methods we use are similar to the transformations used in worst case execution time system for solving recurrence equations [31, 24] or symbolic summation techniques in loop bound computations [20]. Some of the central transformation rules we use in this phase are listed below. In the following transformations the expressions e_1 and e_2 are assumed not to contain the summation variable x.

$$\sum_x C(x = e_1)\cdot f(x) = f(e_1)$$
$$\sum_x C(e_1 \le x \le e_2) = (e_2 - e_1 + 1)\cdot C(e_1 \le e_2)$$
$$\sum_x x\cdot C(e_1 \le x \le e_2) = \left(\frac{e_2\cdot(e_2+1)}{2} - \frac{e_1\cdot(e_1-1)}{2}\right)\cdot C(e_1 \le e_2)\ .$$

One could also use computer algebra systems in the reduction process but some of the rules are quite specific to the way we handle the boundaries of summations with the special constraint function. There are a number of rules to combine products of constraint functions and and split intervals into separate expressions.

$$C(e_1 \le x \le e_2)\cdot C(e_3 \le x \le e_4) = C(\max(e_1,e_3) \le x \le \min(e_2,e_4))$$

$$C(\max(e_1,e_2) \le e_3) = C(e_1 > e_2)\cdot C(e_1 \le e_3) + C(e_1 \le e_2)\cdot C(e_2 \le e_3)\ .$$

There are similar rules for removing the minimum function and for isolating variables in constraints.

There are also rules for symbolic summation of certain infinite summations. If a is an expression where $0 < a < 1$ then we can simplify the expression as follows:

$$\sum_x C(x \geq 0) \cdot a^x = \frac{1}{(1-a)}$$
$$\sum_x C(x \geq 0) \cdot x \cdot a^x = \frac{1}{(1-a)^2} - \frac{1}{(1-a)} .$$

This rule is useful when some of the input to the program follows a geometric distribution.

$$P_x(x,n) = C(x \geq 0) \cdot \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^x .$$

Max example. As a small example, let us look at the simple non-recursive program `max`, which given two values return the largest. It is chosen because it only makes use of the symbolic summation rules and that the output follows a non-uniform distribution even if the input variables are uniformly distributed. The program is defined as

```
max(x,y) = if (x>y) then x else y
```

The input probabilities are independent, each is a uniform distribution from 1 to n and can be defined as

$$P_x(x) = \frac{1}{n} \cdot C(1 \leq x \leq n) \quad \text{and} \quad P_y(y) = \frac{1}{n} \cdot C(1 \leq y \leq n) .$$

The following example uses the conditional transformation rule and the symbolic summation rules.

$$\begin{aligned} P_{\texttt{max}}(z) &= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z = \texttt{if } (x > y) \texttt{ then } x \texttt{ else } y) \\ &= \frac{1}{n^2} \cdot \Big(\sum_y \big(C(1 \leq z \leq n) \cdot C(1 \leq y \leq n) \cdot C(y \leq (z-1)) \big) \\ &\qquad + \sum_x \big(C(1 \leq x \leq n) \cdot C(1 \leq z \leq n) \cdot C(x \leq z) \big) \Big) \\ &= \frac{1}{n^2} \cdot (2z-1) \cdot C(1 \leq z \leq n) . \end{aligned}$$

Add example. The recursive addition function was used as an example in the introduction. We shall see how the original program is inserted into the probability formula, expanded and reduced to a closed form function expressing the probability distribution for the output. Recall the program:

```
add(x,y) = if(x=0) then y else add(x-1,y+1)
```

and that we assume independence between the input variables and for the sake of simplicity we let both input variables `x` and `y` have a uniform distribution from 1 to a number n.

$$P_{\texttt{add}}(z) = \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \sum_{i=0}^{i-1} \prod_{j=0} C(\neg b(h(j,x,y))) \cdot C(b(h(i,x,y))) \cdot C(z = g(h(i,x,y))$$

where

$$\begin{aligned} b(x,y) &= x = 0 \\ g(x,y) &= y \\ h(i,x,y) &= \textbf{ if } (i=0) \textbf{ then } \langle x,y \rangle \textbf{ else } h(i-1,x-1,y+1) \\ &= \langle x-i, y+i \rangle \end{aligned}$$

$$
\begin{aligned}
&P_{\texttt{add}}(z) = \\
&\sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \sum_{i=0}^{i-1} \prod_{j=0} C(\neg(x-j=0)) \cdot C(x-i=0) \cdot C(z=y+i) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot \sum_{i=0} C(x=i) \cdot C(z=y+i) \\
&= \sum_x \sum_y P_x(x) \cdot P_y(y) \cdot C(z=y+x) \\
&= \sum_y \frac{1}{n} \cdot C(z-n \le y \le z-1) \cdot \frac{1}{n} \cdot C(1 \le y \le n)\big) \\
&= \frac{1}{n^2} \cdot \max(\min(n, z-1) - \max(1, z-n) + 1, 0) \\
&= \frac{1}{n^2} \cdot \big(C(n < z \le 2n) \cdot (2n-z+1) \quad + \quad C(1 \le z \le n) \cdot (z-1)\big) \ .
\end{aligned}
$$

3.4 Expected value

If we have derived a probability program, we may also derive an expression that computes the expected value of the distribution.

$$E_p = \sum_x x \cdot P_p(x) \qquad .$$

For the `add` program this gives

$$E_{\texttt{add}} = \sum_{z=1}^{n} z \cdot \frac{1}{n^2} \cdot (z-1) + \sum_{z=n+1}^{2n} z \cdot \frac{1}{n^2} \cdot (2n-z+1)$$

which, of course, can be reduced further.

4 Composite Types

In the approach we have presented the base domain is a countable set and not necessarily just numbers. We only need to be able to define a probability distribution for values in the domain.

For lists of length $k > 0$ where elements are uniformly distributed over the interval 1 to n we can use the probability function

$$P_L(L) = \frac{1}{n^k} \cdot C(\mathit{length}(L) = k \wedge \forall j : 0 \le j \le k-1 \wedge 1 \le \mathit{hd}(\mathit{tl}^j(L)) \le n) \ .$$

We assign the probability $1/n^k$ to any list of length k where all elements are in the interval from 1 to n.

If we consider the member function for non-empty lists, it can be written as

```
member(X,L) =
  if(tl(L)=[] || hd(L)=X) then hd(L)=X
  else member(X,tl(L))
```

The function will follow the pattern of primitive recursion as described earlier and the output probability function for the member function is then

$$P_{\texttt{member}}(z) = \sum_X \sum_L P_X(X) \cdot P_L(L) \cdot C(z = \texttt{member}(X,L)) \ .$$

We can then use the unfolding rules to simplify the expression further.

The lists were here assumed to contain possibly repeating elements in the list. We could also use a different probability measure to restrict lists to non-repeating lists of values. The example is also analysed by Wegbreit [37] where he derives the probability as $1-(1-(1/n)^k)$. It is analysed under the assumption of non-repeating lists but is actually the correct result for repeating lists.

His technique is valid for programs where one can safely assume the Markov property (that probability of conditions are fixed). Wegbreit observes that this is not always true even in very simple cases, e.g. in nested conditionals where the outcome of the first condition influences the probability of the outcome for the subsequent condition.

It should be noted that conditions existing inside a recursive structure often invokes dependencies between variables. This occur when there is a *gain of knowledge*: For instance in the `union` function for two repeating lists; if the head of the list is not in the second list, the likelihood of the next element not being in the second list increases slightly.

5 Approximation Techniques

The probability distribution program expresses the probability distribution for output values. Our aim is to transform it into a closed form but this may not always be possible. Failing that, we can instead use approximation techniques to obtain an upper bound for the probability distribution. We have referred to this as the over approximation of the probability distribution, $\overline{P}$. The techniques we may apply here are similar to automatic worst case complexity analysis [30] where the aim is to obtain a closed form expression for the complexity of programs but failing that may obtain an over approximation.

Cumulative distribution functions. Cumulative probabilities will in some cases be more useful and expressive than probability distributions: Cumulative probabilities can be used in both the discrete and the continuous case, and in some cases approximations can be described more precisely using accumulated probabilities than with ordinary distributions. It tends, however, to be more complex to reduce to closed form and thus may require coarser approximations. The bounding of a cumulative distribution was introduced by Ferson [12] as a P-box and can be used to describe imprecise probability distributions.

Definition 6 (cumulative distribution) *Given a program output probability distribution, $P_{\mathrm{p}}(z)$, the cumulative program output probability distribution, $F_{\mathrm{p}}(z)$, is defined as*

$$F_{\mathrm{p}}(z) = \sum_{w \le z} P_{\mathrm{p}}(w) \ .$$

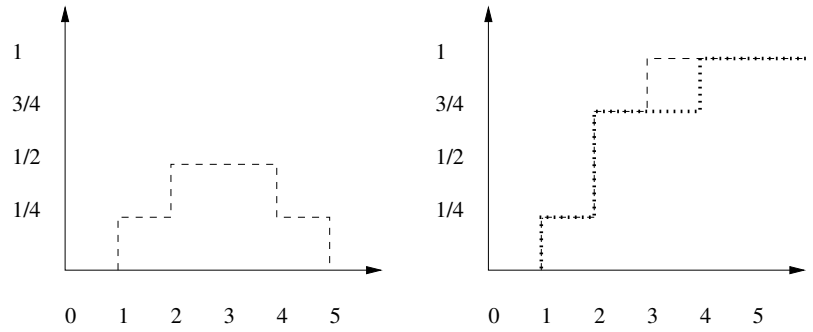
Definition 7 (over and under-approximation) *Given an accumulated output probability of a program P, F_{p}, the over-approximation, $\overline{F}_{\mathrm{p}}$, and the under-approximation, $\underline{F}_{\mathrm{p}}$, are defined as*

$$\overline{F}_{\mathrm{p}} : \forall z.F_{\mathrm{p}}(z) \le \overline{F}_{\mathrm{p}}(z) \qquad\qquad \underline{F}_{\mathrm{p}} : \forall z.\underline{F}_{\mathrm{p}}(z) \le F_{\mathrm{p}}(z)$$

where for each approximation it must always hold that

$$\forall z.0 \le \overline{F}_{\mathrm{p}}(z) \le 1 \qquad \forall z.0 \le \underline{F}_{\mathrm{p}}(z) \le 1 \ .$$

When we can deduce that a program may return one of two values, but not which one, then the cumulative probability can be used for a more precise description. Such a program could be `if x = 1 then 1 else (if x = 4 then 4 else (if (unanalysable) then 2 else 3))` and $1 \leq \mathtt{x} \leq 4$ with the probability distribution $P_x(x) = 1/4 \cdot C(1 \leq x \leq 4)$.



Here, the over-approximating distribution function will assign $1/2$ for both 2 and 3. In contrast, the over-approximating cumulative distribution can express that if the program-output is not 2 it must be 3.

$\underline{P}_{\mathtt{p}}$ and $\overline{P}_{\mathtt{p}}$ can be used to derive $\underline{F}_{\mathtt{p}}$ and $\overline{F}_{\mathtt{p}}$. However, these may not be as precise as cumulative distributions derived directly.

Approximations for accumulated probability functions.

When approximating accumulated probability functions the techniques are different from probability mass functions. Instead one may use copulas [5] to over and under approximate dependencies between subexpressions. Copulas are based on the theory of comonotonicity [10] for distributions that may depend on a common (possibly unknown) random variable.

6 Related Work

Probabilistic analysis is related to the analysis of probabilistic programs. Probabilistic analysis is analysis of programs with a normal semantics where the input variables are interpreted over probability distributions. Analysis of probabilistic programs analyse programs with probabilistic semantics where the values of the input variables are unknown (e.g. flow analysis [28]).

In probabilistic analysis it is important to determine how variables depend on each other, but already in 1976 Denning proposed a flow analysis for revealing whether variables depend on each other [8]. This was presented in the field of secure flow analysis. Denning introduced a lattice-based analysis where she, given the name of a variable, that should be kept secret, deducted which other variables those should be kept secret in order to avoid leaking information. In 1996, Denning's method was refined by Volpano et al. into a type system and for the first time, it was proven sound [36].

Reasoning about probabilistic semantics is a closely related area to probabilistic analysis, as they both work with nested probabilistic influence. The probabilistic analysis work on standard semantic and analyse it using input probability distributions, where a probabilistic semantics allow for random assignments and probabilistic choices [21] and is normally analysed using an expanded classical analysis or verification method [7].

Probabilistic model checking is an automated technique for formally verifying quantitative properties for systems with probabilistic behaviours. It is mainly focused on Markov decision processes, which can model both stochastic and non-deterministic behaviour [14, 22] It differs from probabilistic analysis as it assumes the Markov property.

In 2000, Monniaux applied abstract interpretation to programs with probabilistic semantics and gained safe bounds for worst case analysis [25]. Pierro et al. introduce a linear mapping structure, a Moore-Penrose pseudo-inverse, instead of a Galois connection. They use the linear structures to compare 'closeness' of approximations as an expression using the average approximation error. Pierro et al. further explores using probabilistic abstract interpretation to calculate the average case analysis [27]. In 2012, Cousot and Monerau gave a general probabilistic abstraction framework [7] and stated, in section 5.3, that Pierro et al.'s method and many other abstraction methods can be expressed in this new framework.

When analysing probabilities the main challenge is to maintain the dependencies throughout the program. Schellekens defines this as *Randomness preservation* [33] (or random bag preservation) which in his (and Gao's [15]) case enables tracking of certain data structures and their distributions. They use special data structures as they find these suitable to derive the average number of basic operations. In another approach [37, 29], tests in programs has been assumed to be independent of previous history, also known as the Markov property (the probability of true is fixed). As Wegbreit remarked, this is true only for some programs (e.g. linear search for repeating lists) and others, this is not the case (linear search for non-repeating lists). The Markov property is the foundation in Markov decision processes which is used in probabilistic model-checking [14]. Cousot et al. presents a probabilistic abstraction framework where they divide the program semantics into probabilistic behaviour and (non-)deterministic behaviour. They propose handling of tests when it is possible to assume the Markov property, and handle loops by using a probability function describing the probability of entering the loop in the ith iteration. Monniaux propose another approach for abstracting probabilistic semantics [25]; he first lifts a normal semantics to a probabilistic semantics where random generators are allowed and then uses an abstraction to reach a closed form. Monniaux's semantic approach uses a backward probabilistic semantics operating on measurable functions. This is closely related to the forward probabilistic semantics proposed earlier by Kozen [21].

An alterntive approach to probabilistic analysis is based on symbolic execution of programs with symbolic values [16]. Such techniques can also be used on programs with infinitely many execution paths by limiting the analysis to a finite sets of paths at the expense of tightness of probability intervals [32].

7 Conclusion

Probabilistic analysis of program has a renewed interest for analysing programs for energy consumptions. Numerous embedded systems and mobile applications are limited by restricted battery life on the hardware. In this paper we present a technique for extracting a probability distribution for programs from symbolic distributions of the input. It is a transformation based method, where we analyse a first order language with primitive recursion. From the original program we generate an equivalent probability distribution program, and transform this program into closed form. We present the essential transformation rules for unfolding calls to the original program and removing infinite sums. The transformed program may then be analysed and approximated using program analysis and transformation techniques known from automatic complexity analysis. The core elements of the analysis have been implemented in a prototype system with the aim of using it to improve energy efficiency of systems. The central challenges of approximating in a probabilistic setting are discussed and we describe some advantages of using cumulative distributions along with copulas to achieve a tighter approximation.

Acknowledgements. This work has benefitted from numerous discussions with Pedro López-García,

Alejandro Serrano Mena and other colleagues in Madrid, Bristol and Roskilde.

References

[1] A. Adje, O. Bouissou, J. Goubault-Larrecq, E. Goubault & S. Putot (2014): *Static analysis of programs with imprecise probabilistic inputs.* In: *In Verified Software: Theories, Tools, Experiments*, Springer Berlin Heidelberg., pp. 22–47.

[2] Elvira Albert, Puri Arenas, Samir Genaim & Germán Puebla (2009): *Cost Relation Systems: A Language-Independent Target Language for Cost Analysis. Electr. Notes Theor. Comput. Sci.* 248, pp. 31–46.

[3] Mathias Bauer (1996): *Approximations for Decision Making in the Dempster-Shafer Theory of Evidence.* In Eric Horvitz & Finn Verner Jensen, editors: *UAI*, Morgan Kaufmann, pp. 73–80.

[4] Daniel Berleant & Hang Cheng (1998): *A Software Tool for Automatically Verified Operations on Intervals and Probability Distributions. Reliable Computing* 4(1), pp. 71–82.

[5] Guillem Bernat, Alan Burns & Martin Newby (2005): *Probabilistic timing analysis: An approach using copulas. J. Embedded Computing* 1(2), pp. 179–194.

[6] Olivier Bouissou, Eric Goubault, Jean Goubault-Larrecq & Sylvie Putot (2012): *A generalization of p-boxes to affine arithmetic. Computing* 94(2-4), pp. 189–201.

[7] Patrick Cousot & Michael Monerau (2012): *Probabilistic Abstract Interpretation.* In Helmut Seidl, editor: *ESOP*, *LNCS* 7211, Springer, pp. 169–193.

[8] Dorothy E. Denning (1976): *A Lattice Model of Secure Information Flow. Commun. ACM* 19(5), pp. 236–243.

[9] S. Destercke & D. Dubois (2009): *The role of generalised p-boxes in imprecise probability models*. In: *6th International Symposium on Imprecise Probability: Theories and Applications.*

[10] Jan Dhaene, Michel Denuit, Marc J Goovaerts, R Kaas & David Vyncke (2002): *The Concept of Comonotonicity in Actuarial Science and Finance: Theory*. *Insurance, mathematics & economics* 31(2), pp. 133–161.

[11] Scott Ferson (2014): *Model uncertainty in risk analysis*. Tech. report, Centre de Recherches de Royallieu, Universite de Technologie de Compiegne.

[12] Scott Ferson, Vladik Kreinovich, Lev Ginzburg, Davis S. Myers, & Kari Sentz (2002): *Constructing Probability Boxes and Dempster-Shafer Structures*. SAND2002-4015, Sandia National Laboratories.

[13] Philippe Flajolet, Bruno Salvy & Paul Zimmermann (1991): *Automatic Average-Case Analysis of Algorithm. Theor. Comput. Sci.* 79(1), pp. 37–109.

[14] Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman & David Parker (2011): *Automated Verification Techniques for Probabilistic Systems.* In Marco Bernardo & Valérie Issarny, editors: *SFM*, *LNCS* 6659, Springer, pp. 53–113.

[15] Ang Gao (2013): *Modular average case analysis: Language implementation and extension*. Ph.d. thesis, University College Cork.

[16] Jaco Geldenhuys, Matthew B Dwyer & Willem Visser (2012): *Probabilistic symbolic execution*. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ACM, pp. 166–176.

[17] Jean Gordon & Edward H. Shortliffe (1984): *The Dempster-Shafer Theory of Evidence*. In: *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*, p. 21 pp.

[18] Xi Guo, Menouer Boubekeur, J. McEnery & David Hickey (2007): *ACET based scheduling of soft real-time systems: An approach to optimise resource budgeting*. *International Journal of Computers and Communications* 1(1), pp. 82–86.

[19] Rakowsky Uwe Kay (2007): *Fundamentals of the Dempster-Shafer theory and its applications to system safety and reliability modelling*. In: *RTA*, pp. 173–185.

[20] Jens Knoop, Laura Kovács & Jakob Zwirchmayr (2011): *Symbolic Loop Bound Computation for WCET Analysis.* In Edmund M. Clarke, Irina Virbitskaite & Andrei Voronkov, editors: *Ershov Memorial Conference*, *LNCS* 7162, Springer, pp. 227–242.

[21] Dexter Kozen (1981): *Semantics of Probabilistic Programs. J. Comput. Syst. Sci.* 22(3), pp. 328–350.

[22] M. Kwiatkowska, G. Norman & D. Parker (September 2010): *Advances and challenges of probabilistic model checking.* In: *48th Annual Allerton Conference on Communication, Control, and Computing*, IEEE, pp. 1691–1698.

[23] U. Liqat, S. Kerrison, A. Serrano, K. Georgiou, P. Lopez-Garcia, N. Grech, M. V. Hermenegildo & K. Eder (2013): *Energy Consumption Analysis of Programs based on XMOS ISA Level Models*. In: *23rd International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR*.

[24] Pedro López-García, Luthfi Darmawan & Francisco Bueno (2010): *A Framework for Verification and Debugging of Resource Usage Properties: Resource Usage Verification.* In Manuel V. Hermenegildo & Torsten Schaub, editors: *ICLP (Technical Communications)*, *LIPIcs* 7, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 104–113.

[25] David Monniaux (2000): *Abstract Interpretation of Probabilistic Semantics.* In Jens Palsberg, editor: *SAS*, *LNCS* 1824, Springer, pp. 322–339.

[26] Carroll Morgan, Annabelle McIver & Karen Seidel (1996): *Probabilistic Predicate Transformers. ACM Trans. Program. Lang. Syst.* 18(3), pp. 325–353.

[27] Alessandra Di Pierro, Chris Hankin & Herbert Wiklicky (2006): *Abstract Interpretation for Worst and Average Case Analysis.* In Thomas W. Reps, Mooly Sagiv & Jörg Bauer, editors: *Program Analysis and Compilation*, *LNCS* 4444, Springer, pp. 160–174.

[28] Alessandra Di Pierro, Herbert Wiklicky, Gabriele Puppis & Tiziano Villa (2013): *Probabilistic data flow analysis: a linear equational approach*. In: *Proceedings Fourth International Symposium*, 119, Open Publishing Association, pp. 150–165.

[29] Hector Soza Pollman, Manuel Carro & Pedro Lopez Garcia (2009): *Probabilistic Cost Analysis of Logic Programs: A First Case Study*. *INGENIARE - Revista Chilena de Ingeniera* 17(2), pp. 195–204.

[30] Mads Rosendahl (1989): *Automatic Complexity Analysis.* In: *FPCA*, pp. 144–156.

[31] Mads Rosendahl (2002): *Simple Driving Techniques.* In Torben Æ. Mogensen, David A. Schmidt & Ivan Hal Sudborough, editors: *The Essence of Computation*, *LNCS* 2566, Springer, pp. 404–419.

[32] S. Sankaranarayanan, A. Chakarov & S. Gulwani (June 2013): *Static analysis for probabilistic programs: inferring whole program properties from finitely many paths*. In: *In Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, ACM., pp. 447–458.

[33] Michel P. Schellekens (2008): *A modular calculus for the average cost of data structuring.* Springer.

[34] Lars Schor, Iuliana Bacivarov, Hoeseok Yang & Lothar Thiele (2012): *Worst-Case Temperature Guarantees for Real-Time Applications on Multi-core Systems.* In Marco Di Natale, editor: *IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE, pp. 87–96.

[35] A. Uwimbabazi (2013): *Extended probabilistic symbolic execution*. Master's thesis, University of Stellenbosch.

[36] Dennis M. Volpano, Cynthia E. Irvine & Geoffrey Smith (1996): *A Sound Type System for Secure Flow Analysis. Journal of Computer Security* 4(2/3), pp. 167–188.

[37] Ben Wegbreit (1975): *Mechanical Program Analysis. Commun. ACM* 18(9), pp. 528–539.

[38] Adam Wierman, Lachlan L. H. Andrew & Ao Tang (2008): *Stochastic Analysis of Power-Aware Scheduling*. In: *Proceedings of Allerton Conference on Communication, Control and Computing*, Urbana-Champaign, IL.

[39] Nic Wilson (2000): *Algorithms for Dempster-Shafer Theory*. In: *Handbook of defeasible reasoning and uncertainty management systems*, Springer Netherlands, pp. 421–475.