Analyzing Probabilistic Programs with Dynamic Logic

Tong Cheng *
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan, Hubei 430074, China
Email: tcheng@hust.edu.cn

***Abstract*—We present a unified framework for the analysis of probabilistic programs based on Dynamic Logic in which both iterative and recursive programs can be analyzed. We augment the traditional rules of probabilistic Dynamic Logic with rules for assignment based on schematic Kleene algebra, as well as proof rules for recursive programs with call-by-value parameters. We show soundness of our system with respect to the standard Markov kernel semantics. We give an example of their use in the analysis of the Coupon Collector's Problem.**

1. Introduction

The verification of probabilistic programs has been an active topic of research for almost four decades [1]. Recent applications in artificial intelligence and machine learning have generated renewed interest, but much recent research has ignored the early classical results in the field. In this note, we show how these classical systems can be easily adapted to provide tools that match the deductive power of recently proposed systems. In particular, we show that there is no need for specialized logical syntax for reasoning about expected behavior as advocated in [2], [3].

Our approach is based on Probabilistic Propositional Dynamic Logic (PPDL), a system for reasoning about partial correctness of probabilistic programs and deriving expected values of measurable function and other moments. We provide several enhancements to streamline the reasoning process; in particular, we illustrate the use of a new construct that treats an arithmetic expression as a proposition, thereby allowing expected values to be calculated by simple substitutions. In particular, this provides a substantially simpler method for reasoning about nested loops. We illustrate the construct by giving a relatively straightforward analysis of the Coupon Collector's Problem, a problem that has been recognized as notoriously difficult to analyze formally [2].

We also augment the logic with syntactic constructs for representing general recursion, not just iteration or tail-recursion. We provide proof rules for recursion with call-by-value parameters and prove their soundness. This has been done by [4], but our point here is to show how it is a simple extension of much more classical work.

* *This work performed at Cornell University.*

For parameter passing, we provide a mechanism to define and reason about local variables in the form of a let-in construct. We give a semantics based on Markov kernels [5] and give sound proof rules for this construct.

1.1. Related Work

Recent work on deductive systems for probabilistic programs has focussed on deductive systems for iterative probabilistic programs [2] and recursive probabilistic programs [3]. In that work, calculi for iterative and recursive programs are defined separately, and specialized syntax for reasoning about expectations is used. Here we provide a unified system for termination, correctness and expected time analysis. In our approach, there is no need for specialized syntax for expectations; mechanisms already available from much earlier work not only subsume these results but are more flexible.

McIver and Morgan [6], [7] handle probabilistic programs with nondeterminism, but as nondeterminism presents certain complications, here we focus on probabilistic programs without nondeterminism.

Our semantics is based on the denotational semantics of probabilistic programs as presented in [1].

2. Syntax and Semantics of Probabilistic DL

In PPDL, terms are of two types: programs and measurable functions. Here we use assignments $x \leftarrow e$ ($x \in$ *Var*, $e \in$ *expr*) as primitive programs and expressions e ($e \in$ *expr*) as primitive measurable functions.

We assume a set of variables *Var*, and an inductively defined set of expressions *expr*. Elements in *expr* can be arithmetic expressions or Boolean expressions. In PPDL, Boolean expressions are called *propositions*, and usually used in tests. Here we generalize the concept of *propositions* to all measurable functions, allowing tests to hold with respect to some probability. This allows simpler evaluation for expected values, as shown in Section 4. We are changing our perspectives: propositions not only can represent some truth statement over a state, but also can represent a value over a state, due to the arithmetic nature of the probabilistic version of Dynamic Logic.

The grammar for propositions and programs are as follows.

prop	::=	e	$(e \in \text{expr})$	(expression)
	$\mid$	$k * f$	$(k \in \mathbb{R})$	(scalar)
	$\mid$	$f_1 + f_2$		(sum)
	$\mid$	$f_1 * f_2$		(multiplication)
	$\mid$	$\langle p \rangle f$		(eventuality)

prog	::=	$x \leftarrow e$	$(e \in \text{expr})$	(assignment)
	$\mid$	$k * p$	$(k \in \mathbb{R})$	(scalar)
	$\mid$	$p_1 + p_2$		(sum)
	$\mid$	$p_1; p_2$		(composition)
	$\mid$	$f?$		(test)
	$\mid$	p^n		(power)
	$\mid$	p^*		(Kleene star)

Denote the set of possible states (variable valuation) as $S = \{(x_1, x_2, \ldots, x_n) : x_i \in \mathbb{N}$ as the value of i-th variable$\}$. A state is a vector of values of variables [5]. Let Σ be a σ-algebra of measurable sets on S.

Expressions are interpreted as measurable functions $S \to \mathbb{R}_{\geq 0} \cup \{+\infty\}$, denoted by $\mathcal{A}[\![e]\!]$. Those are naturally given by a inductive definition on the structure of *expr*.

Propositions are interpreted as measurable functions $S \to \mathbb{R}_{\geq 0} \cup \{+\infty\}$:

$$\mathcal{B}[\![e]\!] = \mathcal{A}[\![e]\!] \tag{1}$$

$$\mathcal{B}[\![k * f]\!] = k * \mathcal{B}[\![f]\!] \tag{2}$$

$$\mathcal{B}[\![f_1 + f_2]\!] = \mathcal{B}[\![f_1]\!] + \mathcal{B}[\![f_2]\!] \tag{3}$$

$$\mathcal{B}[\![f_1 * f_2]\!] = \mathcal{B}[\![f_1]\!] * \mathcal{B}[\![f_2]\!] \tag{4}$$

$$\mathcal{B}[\![\langle p \rangle f]\!] = \lambda s. \int_{t \in S} \mathcal{B}[\![f]\!](t) \cdot \mathcal{C}[\![p]\!](s, dt) \tag{5}$$

Programs are interpreted as Markov kernels $S \to \Sigma \to \mathbb{R}_{\geq 0} \cup \{+\infty\}$:

$$\mathcal{C}[\![x \leftarrow e]\!] = \lambda s, A.\ \delta_{s[x/\mathcal{A}[\![e]\!](s)]}(A) \tag{6}$$

$$\mathcal{C}[\![k * p]\!] = k * \mathcal{C}[\![p]\!] \tag{7}$$

$$\mathcal{C}[\![p_1 + p_2]\!] = \mathcal{C}[\![p_1]\!] + \mathcal{C}[\![p_2]\!] \tag{8}$$

$$\mathcal{C}[\![p_1; p_2]\!] = \lambda s, A. \int_{t \in S} \mathcal{C}[\![p_2]\!](t, A) \cdot \mathcal{C}[\![p_1]\!](s, dt) \tag{9}$$

$$\mathcal{C}[\![f?]\!] = \lambda s, A.\ \delta_s(A) \cdot \mathcal{B}[\![f]\!](s) \tag{10}$$

$$\mathcal{C}[\![p^n]\!] = \begin{cases} 1? & \text{if } n = 0 \\ \mathcal{C}[\![p^{n-1}; p]\!] & \text{if } n > 0 \end{cases} \tag{11}$$

$$\mathcal{C}[\![p^*]\!] = \sum_{n=0}^{\infty} \mathcal{C}[\![p^n]\!] \tag{12}$$

where δ_s is the Dirac measure centered on state s.

3. Proof Rules for First-order Terms

We present some proof rules for assignment in first-order terms. These rules are based on rules for Schematic Kleene algebra with tests (SKAT) as presented in [8].

$$x \leftarrow e;\ y \leftarrow a = y \leftarrow a[x/e];\ x \leftarrow e, \tag{13}$$

$$x \leftarrow e;\ y \leftarrow a = x \leftarrow e;\ y \leftarrow a[x/e], \tag{14}$$

$$x \leftarrow e;\ x \leftarrow a = x \leftarrow a[x/e], \tag{15}$$

$$x \leftarrow e;\ f? = f[x/e]?;\ x \leftarrow e, \tag{16}$$

$$\langle x \leftarrow e \rangle f = f, \tag{17}$$

where equation (13) holds when $x \neq y$, $y \notin FV(e)$, equation (14) holds when $x \neq y$, $x \notin FV(e)$, equation (17) holds when $x \notin FV(f)$.

Theorem 1. Equations (13)–(17) are sound with respect to the Markov kernel semantics.

4. Well-Structured Iterative Programs

Our extension to admit arithmetic expressions as propositions allows a simpler representation for expected value evaluation, which is often used in analysis based on invariants. In this section, we will introduce this representation and present an example using invariant analysis.

In [5], a PPDL proposition used to compute the expected value of variable c in program p was proposed as follows:

$$\langle (c \leftarrow c - 1)^*; c \geq 1? \rangle 1. \tag{18}$$

This proposition is the stair function on c-axis, whose integration with program p will be the expectation of c, if $\mathcal{C}[\![p]\!](s)$ is always a probability measure (which is always true for well-structured programs).

With our extension, we can directly represent the identity function on c-axis (as single letter c), whose integration with program p will also be the expectation of c:

$$\begin{aligned} \langle p \rangle c &= \int_{t \in S} \mathcal{B}[\![c]\!](t) \cdot \mathcal{C}[\![p]\!](s, dt) \\ &= \int_{t \in S} \mathcal{A}[\![c]\!](t) \cdot \mathcal{C}[\![p]\!](s, dt). \end{aligned}$$

Furthermore, we can use rules introduced in Section 3 to directly do transformations. For instance, we can transform $\langle c \leftarrow c + 1 \rangle c$ to $c + 1$.

$$\begin{aligned} & \langle c \leftarrow c + 1 \rangle c \\ &= \langle c \leftarrow c + 1 \rangle c \cdot 1 \\ &= \langle c \leftarrow c + 1 \rangle \langle c? \rangle 1 \\ &= \langle c \leftarrow c + 1;\ c? \rangle 1 \\ &= \langle c + 1?;\ c \leftarrow c + 1 \rangle 1 && \textit{by (16)} \\ &= \langle c + 1? \rangle \langle c \leftarrow c + 1 \rangle 1 \\ &= (c + 1) \cdot \langle c \leftarrow c + 1 \rangle 1 \\ &= (c + 1) \cdot 1 && \textit{by (17)} \\ &= c + 1. \end{aligned}$$

PPDL has an induction rule

$$f + \langle p \rangle g \leq g \quad \rightarrow \quad \langle p^* \rangle f \leq g$$

that can be used to obtain the upper bound of expected value of a variable in a loop.

```
c <- 0;
y[0...n-1] <- [0...0];
i <- rand(n);                                  p_i
c <- c + 1;                                    p_c
x <- n;
while (x > 0) {                                b_x
    while (y[i] ≠ 0) {                         b_y
        i <- rand(n);
        c <- c + 1;
    };
    y[i] <- 1;                                 p_y
    x <- x - 1                                 p_x
}
```

Figure 1. The Coupon Collector's Problem

For example, for a loop with an uncommon form $(pbu)^*p\bar{b}u$, we can instantiate an induction rule

$$\langle p\bar{b}u\rangle c + \langle pbu\rangle g \le g \quad \to \quad \langle (pbu)^*\rangle\langle p\bar{b}u\rangle c \le g$$

to prove g as the upper bound of expected value of variable c.

An important instantiation of induction rule is the while rule

$$\langle \bar{b}\rangle f + \langle bp\rangle g \le g \quad \to \quad \langle (bp)^*\rangle\langle \bar{b}\rangle f \le g,$$

i.e.

$$\langle \bar{b}\rangle f + \langle bp\rangle g \le g \quad \to \quad \langle while\ b\ do\ p\rangle f \le g.$$

In many circumstances, an upper bound is good enough. For example, when we analyze the expected running time of an algorithm and are going to represent the result in terms of Big-O notation, we just need an upper bound for the expected value. Nevertheless, we can get an lower bound by a strengthened version of induction rule

$$\frac{g_0 = 0, \quad f + \langle p\rangle g_n \ge g_{n+1}}{\langle p^*\rangle f \ge sup_n\, g_n} \tag{19}$$

$$\frac{g_0 = 0, \quad f + \langle p\rangle g_n \le g_{n+1}}{\langle p^*\rangle f \le sup_n\, g_n} \tag{20}$$

We can use PPDL to analyze the Coupon Collector's Problem (Figure 1). Suppose we are collecting coupons of n sorts (assume $n > 0$). One coupon of a random sort can be acquired in each round (equally likely for all sorts). Our goal is to evaluate the expected number of coupons we need to get coupons of all sorts, which is also the expected running time for the collecting procedure.

Let

$$\begin{array}{rcl} p_{in} & = & while\ (b_y)\ \{\ p_i;\ p_c\ \} \\ & = & (b_y?; p_i; p_c)^*; \bar{b_y}? \\ p_{out} & = & while\ (b_x)\ \{\ p_{in};\ p_y;\ p_x\ \} \\ & = & (b_x?; p_{in}; p_y; p_x)^*; \bar{b_x}? \end{array}$$

To analyze the inner loop, we use PPDL induction rule $(f + \langle p\rangle g \le g \ \to \ \langle p^*\rangle f \le g)$, which can be instantiated as:

$$\langle \bar{b_y}?\rangle c + \langle b_y?; p_i; p_c\rangle I \le I \quad \to \quad \langle (b_y?; p_i; p_c)^*\rangle\langle \bar{b_y}?\rangle c \le I. \tag{21}$$

We propose an invariant

$$I = c + \langle b_y?\rangle \frac{n}{n-k},\ where\ k = \sum_{i=0}^{n-1} \langle y[i] \ne 0?\rangle 1 \tag{22}$$

which can be verified in Section 7.1 to satisfy the left-hand condition in (21). Then we have

$$\langle p_{in}\rangle c \quad \le \quad c + \langle b_y?\rangle \frac{n}{n-k}.$$

For the outer loop, we instantiate the induction rule as:

$$\begin{array}{c} \langle \bar{b_x}?\rangle c + \langle b_x?; p_{in}; p_y; p_x\rangle J \le J \\ \to \quad \langle (b_x?; p_{in}; p_y; p_x)^*\rangle\langle \bar{b_x}?\rangle c \le J. \end{array} \tag{23}$$

For $x > 0$, we propose an invariant

$$J = c + \langle b_y?\rangle \frac{n}{n-k} + \sum_{j=n-k-x+1}^{n-k-1} \frac{n}{j}, \tag{24}$$

which can also be verified in Section 7.2. Then we have

$$\langle p_{out}\rangle c \quad \le \quad J.$$

For the program overall,

$$\begin{array}{rcl} & & \langle c \leftarrow 0; y \leftarrow [0 \ldots 0]; p_i; p_c; x \leftarrow n\rangle\langle p_{out}\rangle c \\ & \le & J[x/n,\ c/c+1,\ i/rand(n),\ y[0]/0,\ \ldots,\ y[n-1]/0,\ c/0] \\ & = & 1 + \sum_{j=n-\sum\langle 0\ne 0?\rangle 1-n-1}^{n-\sum\langle 0\ne 0?\rangle 1-1} \frac{n}{j} \quad = \quad \sum_{j=1}^{n} \frac{n}{j} \\ & = & n \cdot \mathcal{H}(n) \quad = \quad \mathcal{O}(n \log n). \end{array}$$

5. Recursion

Our language can be easily extended to support recursion. We define a recursive procedure by writing down a program operator $T : prog \to prog$ and use a new program construct $fix\ T(\cdot)$ for the least fixpoint of the operator, i.e. the recursion procedure.

We define the semantics of the least fixpoint as follows:

$$\mathcal{C}[\![fix\ T(q)]\!] = \sup_n\ \mathcal{C}[\![T^n(0?)]\!]. \tag{25}$$

For example, the loop **while** (true) **do** {skip} can be written as the least fixpoint of a program operator:

$$\begin{array}{rcl} p & = & fix\ T(q), \\ T(q) & = & 0? + 1?; 1?; q. \end{array}$$

Since $T^n(0?) = 0?$ for all n, we have $p = fix\ T(q) = 0?$.

We also present the induction rules for recursions:

$$\frac{\langle p\rangle f \le g \quad \to \quad \langle T(p)\rangle f \le g}{\langle fix\ T(p)\rangle f \le g} \tag{26}$$

$$\frac{g_0 = 0, \quad \langle p\rangle f \le g_n \quad \to \quad \langle T(p)\rangle f \le g_{n+1}}{\langle fix\ T(p)\rangle f \le sup_n\, g_n} \tag{27}$$

$$\frac{g_0 = 0, \quad \langle p\rangle f \le g_n \quad \to \quad \langle T(p)\rangle f \ge g_{n+1}}{\langle fix\ T(p)\rangle f \ge sup_n\, g_n} \tag{28}$$

where $p = T^i(0?)$ for some i in each rule.

We can illustrate the usage of those rules with the following program:

$$\begin{aligned} p &= fix\ T(q), \\ T(q) &= \frac{1}{2}\cdot \text{skip} + \frac{1}{2}(q; q; q), \end{aligned}$$

whose upper bound for probability of termination can be calculated as follows:

Assume $\langle q\rangle 1 = \phi = \frac{\sqrt{5}-1}{2}$, we have

$$\begin{aligned} \langle T(q)\rangle 1 &= \left\langle \frac{1}{2}\cdot \text{skip} + \frac{1}{2}(q; q; q)\right\rangle 1 \\ &= \frac{1}{2} + \frac{1}{2}\langle q\rangle\langle q\rangle\langle q\rangle 1 \\ &= \frac{1}{2} + \frac{1}{2}\langle q\rangle\langle q\rangle\phi \\ &= \frac{1}{2} + \frac{1}{2}\phi\langle q\rangle\langle q\rangle 1 \\ &= \frac{1}{2} + \frac{1}{2}\phi\langle q\rangle\phi \\ &= \frac{1}{2} + \frac{1}{2}\phi\cdot\phi\langle q\rangle 1 \\ &= \frac{1}{2} + \frac{1}{2}\phi\cdot\phi\cdot\phi \\ &= \phi. \end{aligned}$$

Then we can conclude that

$$\langle p\rangle 1 \quad = \quad \langle fix\ T\rangle 1 \quad \le \quad \phi.$$

6. Local Variables

In this section we show how to add local variables to the language by providing semantics for a let-in construct. We add a new program term to the language:

$$let\ x \leftarrow e\ in\ p\ end$$

whose semantics is defined as

$$\begin{aligned} &\mathcal{C}[\![let\ x \leftarrow e\ in\ p\ end]\!](s, A) \\ = &\int_{t\in S} \delta_{t[x/\mathcal{A}[\![x]\!](s)]}(A)\cdot \mathcal{C}[\![p]\!](s[x/\mathcal{A}[\![e]\!](s)], dt). \end{aligned}$$

Then we present some proof rules for let-expessions:

$$\langle let\ x \leftarrow e\ in\ p\ end\rangle f \quad = \quad \langle x \leftarrow e; p\rangle f, \tag{29}$$

$$\langle let\ x \leftarrow e\ in\ p\ end\rangle f \quad = \quad f. \tag{30}$$

where equation (29) holds for $x \notin FV(f)$, equation (30) holds for $FV(f) = \{x\}$.

Theorem 2. Equations (29) and (30) are sound with respect to the Markov kernel semantics.

7. The Coupon Collector's Problem

In this section we provide a complete proof of the expected running time of the Coupon Collector's Problem.

7.1. Inner Loop Verification

$$\begin{aligned} &\langle \bar{b_y}?\rangle c + \langle b_y?; p_i; p_c\rangle I \\ = &\langle \bar{b_y}?\rangle c + \langle b_y?; p_i; p_c\rangle c + \langle b_y?; p_i; p_c\rangle\langle b_y?\rangle\frac{n}{n-k} \\ = &\langle \bar{b_y}?\rangle c + \langle b_y?\rangle(c+1) + \langle b_y?; p_i; b_y?\rangle\frac{n}{n-k} \\ = &\ c + \langle b_y?\rangle\left(1 + \langle p_i; b_y?\rangle\frac{n}{n-k}\right) \\ = &\ c + \langle b_y?\rangle\left(1 + \frac{1}{n}\sum_{j=0}^{n-1}\langle i \leftarrow j; b_y?\rangle\frac{n}{n-k}\right) \\ = &\ c + \langle b_y?\rangle\left(1 + \frac{1}{n}\sum_{j=0}^{n-1}\langle y[j] \neq 0?\rangle\frac{n}{n-k}\right) \\ = &\ c + \langle b_y?\rangle\left(1 + \frac{1}{n}\cdot\frac{n}{n-k}\sum_{j=0}^{n-1}\langle y[j] \neq 0?\rangle 1\right) \\ = &\ c + \langle b_y?\rangle\left(1 + \frac{1}{n}\cdot\frac{n}{n-k}\cdot k\right) \\ = &\ c + \langle b_y?\rangle\frac{n}{n-k} \quad = \quad I. \end{aligned}$$

7.2. Outer Loop Invariant

$$\begin{aligned} &\langle \bar{b_x}?\rangle c + \langle b_x?; p_{in}; p_y; p_x\rangle J \\ = &\langle p_{in}; p_y; p_x\rangle J \quad \text{(assumed initial state where } x > 0) \\ = &\langle p_{in}; p_y; p_x\rangle c + \langle p_{in}; p_y; p_x\rangle\langle b_y?\rangle\frac{n}{n-k} \\ &+ \langle p_{in}; p_y; p_x\rangle\sum\nolimits_{j=n-k-x+1}^{n-k-1}\frac{n}{j} \\ = &\langle p_{in}\rangle c + \langle p_{in}; p_y\rangle\langle b_y?\rangle\frac{n}{n-k} \\ &+ \langle p_{in}; p_y\rangle\sum\nolimits_{j=n-k-x+2}^{n-k-1}\frac{n}{j} \\ = &\langle p_{in}\rangle c + \langle p_{in}; p_y\rangle\frac{n}{n-k} + \langle p_{in}; p_y\rangle\sum\nolimits_{j=n-k-x+2}^{n-k-1}\frac{n}{j} \\ = &\langle p_{in}\rangle c + \langle p_{in}\rangle\frac{n}{n-k-1} + \langle p_{in}\rangle\sum\nolimits_{j=n-k-x+1}^{n-k-2}\frac{n}{j} \\ = &\langle p_{in}\rangle c + \sum\nolimits_{j=n-k-x+1}^{n-k-1}\frac{n}{j} \\ &\text{(recall that LValue}(p_{in}) = \{i, c\}) \\ \le &\ c + \langle b_y?\rangle\frac{n}{n-k} + \sum\nolimits_{j=n-k-x+1}^{n-k-1}\frac{n}{j} \quad = \quad J. \end{aligned}$$

Some first-order formula used above can be verified as

$$p_y; b_y? \quad = \quad 1 \neq 0?; p_y \quad = \quad p_y,$$

$$
\begin{aligned}
& \langle p_{in}; p_y \rangle \frac{n}{n-k} \\
= \quad & \langle p_{in}; \bar{b_y}?; p_y \rangle \frac{n}{n-k} \\
= \quad & \langle p_{in}; y[i] = 0?; y[i] \leftarrow 1 \rangle \frac{n}{n - \left(\sum_j \langle y[j] \neq 0? \rangle 1 \right)} \\
= \quad & \langle p_{in}; y[i] = 0?; y[i] \leftarrow 1 \rangle \frac{n}{n - \left(\sum_{j \neq i} \langle y[j] \neq 0? \rangle 1 + \langle y[i] \neq 0? \rangle 1 \right)} \\
= \quad & \langle p_{in}; y[i] = 0? \rangle \frac{n}{n - \left(\sum_{j \neq i} \langle y[j] \neq 0? \rangle 1 + \langle 1 \neq 0? \rangle 1 \right)} \\
= \quad & \langle p_{in}; y[i] = 0? \rangle \frac{n}{n - \left(\sum_{j \neq i} \langle y[j] \neq 0? \rangle 1 + 1 + 0 \right)} \\
= \quad & \langle p_{in}; y[i] = 0? \rangle \frac{n}{n - \left(\sum_{j \neq i} \langle y[j] \neq 0? \rangle 1 + 1 + \langle y[i] \neq 0 \rangle 1 \right)} \\
= \quad & \langle p_{in} \rangle \frac{n}{n - (k+1)} \\
= \quad & \langle p_{in} \rangle \frac{n}{n-k-1},
\end{aligned}
$$

and similiarly we have,

$$
\langle p_{in}; p_y \rangle \sum\nolimits_{j=n-k-x+2}^{n-k-1} \frac{n}{j} \quad = \quad \langle p_{in} \rangle \sum\nolimits_{j=n-k-x+1}^{n-k-2} \frac{n}{j}.
$$

Disclaimer: This work has been submitted to MFCS 2017.

References

[1] D. Kozen, "Semantics of probabilistic programs," *J. Comput. Syst. Sci.*, vol. 22, pp. 328–350, 1981.

[2] B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo, "Weakest Precondition Reasoning for Expected Run–Times of Probabilistic Programs," in *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2–8, 2016, Proceedings*, P. Thiemann, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 364–389.

[3] F. Olmedo, B. L. Kaminski, J.-P. Katoen, and C. Matheja, "Reasoning About Recursive Probabilistic Programs," in *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '16. New York, NY, USA: ACM, 2016, pp. 672–681.

[4] K. Aboul-Hosn and D. Kozen, "Local Variable Scoping and Kleene Algebra with Tests," *J. Log. Algebr. Program.*, 2007.

[5] D. Kozen, "A probabilistic PDL," *J. Comput. Syst. Sci.*, vol. 30, no. 2, pp. 162–178, Apr. 1985.

[6] C. Morgan, A. McIver, and K. Seidel, "Probabilistic Predicate Transformers," *ACM Trans. Program. Lang. Syst.*, vol. 18, no. 3, pp. 325–353, May 1996.

[7] A. K. McIver and C. Morgan, "Partial correctness for probabilistic demonic programs," *Theoretical Computer Science*, vol. 266, no. 1, pp. 513 – 541, 2001.

[8] A. Angus and D. Kozen, "Kleene Algebra with Tests and Program Schematology," Computer Science Department, Cornell University, Tech. Rep., Jul. 2001.