Understanding Probabilistic Programs*

Joost-Pieter Katoen, Friedrich Gretz, Nils Jansen,
Benjamin Lucien Kaminski, and Federico Olmedo

{katoen, friedrich.gretz, nils.jansen,
benjamin.kaminski, federico.olmedo}@cs.rwth-aachen.de

RWTH Aachen University
Aachen, Germany

Abstract. We present two views of probabilistic programs and their relationship. An operational interpretation as well as a weakest pre-condition semantics are provided for an elementary probabilistic guarded command language. Our study treats important features such as sampling, conditioning, loop divergence, and non-determinism.

1 Introduction

Probabilistic programs are sequential programs with the ability to draw values at random from probability distributions. Probabilistic programs are not new at all. Seminal papers from the mid–eighties consider their formal semantics [16] as well as their formal verification [25]. Variations of probabilistic propositional dynamic logic [5] have been defined to enable reasoning about probabilistic programs. McIver and Morgan [17] generalized Dijkstra's weakest pre–conditions to weakest pre–expectations (wp) so as to formally analyze pGCL—the probabilistic guarded command language. Mechanized wp–reasoning has been realized [13, 3].

In the last years the interest in probabilistic programs is rapidly growing [8]. This is mainly due to their wide applicability. Probabilistic programs are used in security to describe cryptographic constructions (such as randomized encryption) and security experiments [1], in machine learning to describe distribution functions that are analyzed using Bayesian inference, and naturally occur in randomized algorithms [18]. Other applications include [6] scientific modeling, information retrieval, bio–informatics, epidemiology, vision, seismic analysis, semantic web, business intelligence, human cognition, and more. The variety of probabilistic programming languages is immense. Almost each programming language, being it imperative, declarative, object–oriented or logical, has a probabilistic counterpart. Probabilistic C [21] extends C with sampling, Church is based on the λ–calculus, Figaro [22] is fully integrated in the Scala object–oriented language, and CHRiSM is a probabilistic version of Prolog. Probabilistic programs are not just of academic interest; they are highly relevant to industry; DARPA invests 48 million US dollar on probabilistic programming for advanced machine learning because:

* This work was supported by the Excellence Initiative of the German federal and state government.

> "probabilistic programming is a new programming paradigm for managing uncertain information. By incorporating it into machine learning, we seek to greatly increase the number of people who can successfully build machine learning applications, and make machine learning experts radically more effective".

Microsoft has recently started a large initiative to improve the usability of probabilistic programming. New languages and approaches such as Infer.NET (akin to C#), R2 [19] and Tabular [7] emerged.

What is special about probabilistic programs? They are typically just a few number of lines, but hard to understand and analyze, let alone algorithmically. For instance, the elementary question of almost–sure termination—for a given input, does a probabilistic program terminate with probability one?—is as hard as [14] the universal halting problem—does an ordinary program halt on *all* possible inputs? Loop invariants of probabilistic programs typically involve quantitative statements and synthesizing them requires more involved techniques than for ordinary programs [15, 2]. Modern probabilistic programming languages do not just support sampling, but also have the ability to condition values of variables in a program through *observations*. Conditioning blocks all program runs violating its Boolean condition and prevents those runs from happening. Consequently, the likelihood of the remaining runs is normalized. The latter effect makes observations differ from program annotations like probabilistic assertions [24].

Conditioning of variables through observations is less well–understood and raises various semantic difficulties, in particular in the presence of possibly non–terminating loops and non–determinism[1]. Previous works on semantics for probabilistic programs with observations [19, 12] do not consider these important features. In fact, many works on probabilistic programs ignore the notion of non–termination and assume that loops always terminate—a property that is unrealistic in practice and highly undecidable to establish. This paper sketches the semantic intricacies, and presents ideas of providing a formal semantics of pGCL treating conditioning in presence of possibly diverging loops and non–determinism.

Much in the vein of Olderog's view [20] that multiple semantic perspectives are useful for a full understanding of programs and systems, we provide *two* semantic views and study their relationship. We present an operational semantics in terms of infinite–state parametric Markov decision processes [23] as well as a weakest (liberal) precondition semantics à la McIver and Morgan [17] and Dijkstra [4]. The main result is a transfer theorem that establishes the relationship between the two semantics. A program transformation is described to remove conditioning and its correctness is established. The presentation is kept informal; full technical details can be found in [11, 9, 10].

---

[1] As stated in [8], "representing and inferring sets of distributions is more complicated than dealing with a single distribution, and hence there are several technical challenges in adding non–determinism to probabilistic programs".

2 Probabilistic Programs

This section introduces our programming language. Probabilistic programs are presented by means of examples that elucidate the key insights behind them.

Main features. Roughly speaking, probabilistic programs are ordinary sequential programs with two additional features:

(i) The ability to *draw samples* from a probability distribution. For simplicity, we consider discrete probability distributions only, and model sampling by means of a probabilistic choice [2] of the form:

$$\{P_1\}\ [p]\ \{P_2\}\ .$$

Here, P_1 and P_2 are programs and p is a probability value in $[0,\ 1]$. Intuitively, this construct behaves as P_1 with probability p and as P_2 with probability $1{-}p$.

(ii) The ability to *condition* the distribution of program states with respect to an observation. This is done using statements of the form:

$$\texttt{observe}\ (G)\ ,$$

where G is a Boolean expression over the program variables. The effect of such an instruction is to block all program executions violating G and rescale the probability of the remaining executions so that they sum up to one. In other words, `observe` (G) transforms the current distribution μ over states into the conditional distribution $\mu|_G$.

To clarify these features consider the two simple sample programs given below:

```
1: {x := 0} [1/2] {x := 1};
2: {y := 0} [1/2] {y := −1}
```

```
1: {x := 0} [1/2] {x := 1};
2: {y := 0} [1/2] {y := −1};
3: observe (x + y = 0)
```

The left program flips two fair (and independent) coins and assigns different values to variables x and y depending on the result of the coin flips. This program admits four executions and yields the outcome

$$\Pr[x{=}0, y{=}0]\ =\ \Pr[x{=}0, y{=}{-1}]\ =\ \Pr[x{=}1, y{=}0]\ =\ \Pr[x{=}1, y{=}{-1}]\ =\ \tfrac{1}{4}\ .$$

The program on the right blocks two of these four executions as they violate the observation $x{+}y$ equals zero in the last line. The probabilities of the remaining two executions are normalized. This leads to the outcome

$$\Pr[x{=}0, y{=}0]\ =\ \Pr[x{=}1, y{=}{-1}]\ =\ \tfrac{1}{2}\ .$$

[2] Alternatively, one can use *random assignments* which sample a value from a distribution and assign it to a program variable; see e.g. [8].

Remarks on conditioning. The `observe` statement is related to the well–known `assert` statement: both statements `observe` (G) and `assert` (G) block all runs violating the Boolean condition G. The crucial difference, however, is that `observe` (G) normalizes the probability of the remaining runs while `assert` (G) does not. This yields a sub–probability distribution of total mass possibly less than one [1].

We also like to point out that an observation may block *all* program runs. In this case the normalization process is not well–defined and the program admits no feasible run. This is similar to the situation that conditional probabilities are ill–defined when conditioning to an event of probability zero. Section 3 sheds more light on this phenomenon. A possible way out is to only allow conditioning at the end of the program, in particular not inside loops. Whereas this view indeed simplifies matters, modern probabilistic programming languages [7, 19, 21] do not impose this restriction for good reasons. Instead, they allow the use of `observe` statements at any place in a program, e.g. in loops. Section 4 presents two program semantics that adequately handle such (infeasible) programs.

Loops. Let us now consider loops. Consider the following two loopy programs:

```
1: i := 0;
2: repeat
3:     {b := heads} [p] {b := tails};
4:     i := i + 1
5: until (b = heads)
```

```
1: i := 0;
2: repeat
3:     {b := heads} [p] {b := tails};
4:     i := i + 1
5: until (b = heads);
6: observe (odd(i))
```

The left program tosses a (possibly biased) coin until it lands heads and tracks the number of necessary trials. It basically simulates a *geometric distribution* with success probability p and upon program termination we have

$$\Pr[i = N] \;=\; (1-p)^{N-1}\, p \qquad \text{for } N \geq 1 \;.$$

The program on the right is as the left program but models the situation where on termination we observe that the number of trials until the first heads is odd. The set of program executions complying this observation has an overall probability of $\sum_{N\geq 0} (1-p)^{2N} p = {}^{1}/_{(2-p)}$. This follows from considering a geometric series on even indices. Accordingly, the distribution of variable i is now governed by

$$\begin{array}{lcl} \Pr[i = 2N{+}1] & = & (1-p)^{2N} p\big(2-p\big) \\ \Pr[i = 2N] & = & 0 \end{array} \qquad \text{for } N \geq 0 \;.$$

As a final remark regarding the previous pair of loopy programs, observe that we allow the probability value of probabilistic choices to remain unspecified. This allows us to deal with *parametric* programs in which the exact values of the probabilities are not known.

Non–determinism. Our programming model also accounts for the possibility of non–determinism. Let $\{P_1\} \Box \{P_2\}$ represent the non–deterministic choice between the programs P_1 and P_2. Non–deterministic choices are resolved by means of a so–called *scheduler* (akin: adversary). On the occurrence of the non–deterministic choice $\{P_1\} \Box \{P_2\}$ during a program run, a scheduler decides whether to execute P_1 or P_2. This choice can in principle depend on the sequence of program states encountered so far in the run. Consider, for instance

$$\begin{aligned}&{\scriptstyle 1:}\ \{i \coloneqq 2j\} \Box \{i \coloneqq 2j{+}1\};\\&{\scriptstyle 2:}\ \{i \coloneqq i{+}1\}\ [1/3]\ \{i \coloneqq i{+}2\}\ .\end{aligned}$$

It admits the schedulers $\mathcal{L}$ and $\mathcal{R}$, say. Scheduler $\mathcal{L}$ resolves the non–deterministic choice in favor of the assignment $i \coloneqq 2j$, whereas scheduler $\mathcal{R}$ selects the assignment $i \coloneqq 2j + 1$. Evidently, imposing either the scheduler $\mathcal{L}$ or $\mathcal{R}$ on this program yields a purely probabilistic program.

As in [17], we consider a *demonic* model to determine the probability of an event in the presence of non–determinism. This amounts to resolving all non–deterministic choices in a way that minimizes the probability of the event at hand. In other words, we assume a scheduler that leads to the event occurring with the least probability. For instance, the probability that i is odd in the above program is computed as follows

$$\Pr[\mathsf{odd}(i)] = \min\left\{\Pr^{\mathcal{L}}[\mathsf{odd}(i)],\, \Pr^{\mathcal{R}}[\mathsf{odd}(i)]\right\} = \min\left\{\tfrac{1}{3},\, \tfrac{2}{3}\right\} = \tfrac{1}{3}\ .$$

By a similar reasoning it follows that the probability that i is even is also $1/3$. This shows that in the presence of non–determinism the law of total probability, namely $\Pr[A] + \Pr[\neg A] = 1$, does not hold.

Observe that our demonic model of non-determinism impacts directly on the termination behavior of programs. This is because in the probabilistic setting, the termination behaviour of a program is given by the probability of establishing true, which—like the probability of any other event—is to be minimized. To clarify this consider the following example. Assume that P is a program which admits a scheduler that leads to a probability of termination zero, while all other schedulers induce a probability of termination that is strictly positive. We will then say that P is non-terminating, or more formally, that it *diverges almost surely*, since according to our demonic model of non-determinism, the probability of establishing true, i.e., termination, will be zero.

3 Semantic Intricacies

In this section, we investigate semantic difficulties that arise in the context of non–deterministic and probabilistic uncertainty in probabilistic programs, in particular in combination with conditioning. We do this by means of examples. Consider as a first example the following two ordinary (i.e. deterministic

and non–probabilistic) programs P_{div} (left) and P_{term} (right):

```
1: repeat                 1: repeat
2:     x := 1             2:     x := 0
3: until (x = 0)          3: until (x = 0)
```

While the left program never terminates as the variable x is always set to one, the right program performs only one loop iteration. The right program is said to *certainly terminate*.

Non–deterministic uncertainty. The first type of uncertainty we take a look at is non–determinism. For that, consider the following program P_{nd}:

```
1: repeat
2:     {x := 1} □ {x := 0}
3: until (x = 0)
```

In each loop iteration, the variable x is set non–deterministically either to 1 or to 0. A natural question is whether this program terminates or not. Obviously, this depends on the resolution of the non–deterministic choice inside the loop body. For the scheduler that chooses the left branch $x \coloneqq 1$ in each loop iteration, the probability of termination is zero, while for any other scheduler the probability of termination is one. (As P_{nd} contains no probabilistic choice, any event will occur with probability either zero or one). In view of our demonic model of non-determinism, the program presents a certain behavior: non-termination.

Probabilistic uncertainty. Consider now the following program P_{pr}, which is obtained from the previous program P_{nd} by replacing the non-deterministic choice by a random choice:

```
1: repeat
2:     {x := 1} [1/3] {x := 0}
3: until (x = 0)
```

In each loop iteration, the variable x is set to 1 with probability $1/3$ and to 0 with probability $2/3$. Again we pose the question: does this program terminate? The answer to that requires a differentiated view: there does exist *a single non–terminating program run*, namely the one in which x is set to 1 in each loop iteration. This infinite run, however, has probability $1/3 \cdot 1/3 \cdot 1/3 \cdots = 0$. Thus, the terminating runs have probability $1 - 0 = 1$. In this case, the program is said to terminate *almost surely*. Note that it does not terminate certainly though, as it admits an infinite run.

Combining non–deterministic and probabilistic uncertainty. Let us consider the two notions of uncertainty in a single program P_{nd+pr}:

```
1: repeat
2:     {{x := 1} [8/9] {x := 0}} □ {{x := 1} [1/9] {x := 0}}
3: until (x = 0)
```

In each loop iteration, the variable x is set to 0 with a certain probability, but this probability is chosen non–deterministically to be $1/9$ or $8/9$. Again we pose the question: does this program terminate almost–surely? As a matter of fact, the scheduler cannot prevent this program from terminating almost–surely. In fact the two programs

```
1: repeat                          1: repeat
2:     {x := 1} [1/9] {x := 0}     2:     {x := 1} [8/9] {x := 0}
3: until (x = 0)                   3: until (x = 0)
```

are semantically equivalent in both our semantic views [17, 11].

Still it seems natural to ask whether choosing $1/9$ over $8/9$ as the probability of setting x to 0 would not be—so to say—*more demonic* as this would increase the expected time until termination and therefore the right program converges slower. To the best of our knowledge, however, existing semantics for probabilistic programs with non–determinism do not take this convergence rate into account (and neither do our two semantic views).

Observations. Next, we turn towards the second characteristic feature of probabilistic programs—conditioning—and take a look at termination in this context. Consider the following two programs P_{div} (left) and P_{obs} (right):

```
1: repeat                 1: repeat
2:     x := 1             2:     {x := 1} [1/2] {x := 0};
3: until (x = 0)          3:     observe (x = 1)
                          4:  until (x = 0)
```

As noted earlier, the left program certainly diverges. For the right program, things are not so clear any more: On the one hand, the only non–terminating run is the one in which in every iteration x is set to 1. This event of setting x infinitely often to 1, however, has probability 0. So the probability of non–termination would be 0. On the other hand, the global effect of the observe statement within the loop is to condition on exactly this event, which occurs with probability 0. Hence, the conditional termination probability is 0 divided by 0, i.e. *undefined.*

Remark 1. Notice that while in this sample program it is immediate to see that the event to which we condition has probability 0, in general it might be highly non–trivial to identify this. Demanding from a “probabilistic programmer” to

condition only to events with non–zero probability would thus be just as (if not even more) far–fetched as requiring an "ordinary programmer" to write only terminating programs. Therefore, a rigorous semantics for probabilistic programs with conditioning has to take the possibility of conditioning to zero–probability events into account: To the program on the right such a semantics should assign a dedicated denotation which represents undefined due to conditioning to a zero–probability event.

Conditioning in presence of uncertainty. Our final example in this section blurs the situation even further by incorporating both notions of uncertainty and conditioning into the single program P_{all}:

```
1: repeat
2:     {x := 1} [1/2] {x := 0};
3:     {x := 1} □ {observe (x = 1)}
4: until (x = 0)
```

This program first randomly sets x to 1 or 0. Then it either sets x to 1 or conditions to the event that x was set to 1 in the previous probabilistic choice. The latter choice is made non–deterministically and therefore the semantics of the entire program is certainly not clear: If in line 3, the scheduler always chooses $x \coloneqq 1$, then this results in certain non–termination. If, on the other hand, the scheduler always chooses observe $(x = 1)$, then the global effect of the observe statement is a conditioning to this zero–probability event. Which behavior of the scheduler is more demonic? We take the point of view that certain non–termination is a more well–behaved phenomenon than conditioning to a zero–probability event. Therefore a demonic scheduler should prefer the latter.

4 Expectation Transformer and Operational Semantics

This section presents the two semantic views and their relationship. The first perspective is a semantics in terms of weakest pre–expectations, the quantitative analogue of Dijkstra's weakest pre–conditions [4]. The second view is an operational semantics in terms of Markov decision processes (MDPs) [23]. The relationship between the semantics is established by linking weakest pre–expectations to (conditional) rewards in the MDPs associated to the programs.

4.1 Weakest pre–expectation Semantics

The semantics of Dijkstra's seminal guarded command language [4] has been given in terms of weakest preconditions. It is in fact a predicate transformer semantics, i.e. a total function between two predicates on the state of a program. The predicate transformer $E = \mathsf{wp}(P, F)$ for program P and postcondition F yields the weakest precondition E on the initial state of P ensuring that the execution of P terminates in a final state satisfying F. There is a direct relation

with axiomatic semantics: the Hoare triple $\langle E \rangle\, P\, \langle F \rangle$ holds for total correctness if and only if $E \Rightarrow \mathsf{wp}(P, F)$. The weakest *liberal* precondition $\mathsf{wlp}(P, F)$ yields the weakest precondition for which P either does not terminate or establishes F. It does not ensure termination and corresponds to Hoare logic for partial correctness.

Weakest pre–expectations. Qualitative annotations in predicate calculus are often insufficient for probabilistic programs as they cannot express quantities such as expectations over program variables. To that end, we adopt the approach by McIver and Morgan [17] and consider *expectations* over program variable valuations. They are the quantitative analogue of predicates and are in fact just random variables (over variable valuations). An *expectation transformer* is a total function between expectations on the state of a program. Stated colloquially, the expectation transformer $e = \mathsf{wp}(P, f)$ for pGCL–program P and post–expectation f over final states yields the least expected "value" e on P's initial state ensuring that P's execution terminates with a "value" f. That is to say, $e(\sigma) = \mathsf{wp}(P, f)(\sigma)$ represents the expected value of f with respect to the distribution of final states obtained from executing program P in state σ, where σ is a valuation of the program variables. The annotation $\langle e \rangle\, P\, \langle f \rangle$ holds for total correctness if and only if $e \leq \mathsf{wp}(P, f)$, where $\leq$ is to be interpreted in a point–wise manner. The weakest *liberal* pre–expectation $\mathsf{wlp}(P, f)$ yields the least expectation for which P either does not terminate or establishes f. It does not ensure termination and corresponds to partial correctness.

Determining weakest pre–expectations. We explain the transformation of expectations by means of an example. Consider the program P:

$$\{\{x \coloneqq 5\} \;\square\; \{x \coloneqq 2\}\}\; [p]\; \{x \coloneqq 2\}$$

We would like to find the (least) average value of x produced by this program. This quantity is given by

$$\mathsf{wp}(P, x) = \mathsf{wp}(\{\{x \coloneqq 5\} \;\square\; \{x \coloneqq 2\}\}\; [p]\; \{x \coloneqq 2\}, x) \ .$$

The expectation of the probabilistic choice is given by the weighted average of the expectations of its sub–programs, thus we obtain

$$p \cdot \mathsf{wp}(\{x \coloneqq 5\} \;\square\; \{x \coloneqq 2\}, x) + (1-p) \cdot \mathsf{wp}(x \coloneqq 2, x) \ .$$

As non–determinism is resolved in a demonic manner, it yields the expectation given by the minimum between the expectations of the sub–programs

$$p \cdot \min\{\mathsf{wp}(x \coloneqq 5, x), \mathsf{wp}(x \coloneqq 2, x)\} + (1-p) \cdot \mathsf{wp}(x \coloneqq 2, x) \ .$$

In the last step we apply the assignments and evaluate the expression

$$p \cdot \min\{5, 2\} + (1-p) \cdot 2 \;=\; p \cdot 2 + (1-p) \cdot 2 \;=\; 2 \ .$$

For loops, the semantics is as usual defined by a least fixed point; in our case, over the domain of expectations with partial order the point–wise ordering $\leq$ on expectations.

Conditioning. Let $\mathsf{wp}(\mathtt{observe}(G),\ f) \;=\; \mathsf{wlp}(\mathtt{observe}(G),\ f) = [G]\cdot f$, where $[G]$ stands for the characteristic function of the Boolean expression G over the program variables. For probabilistic programs with observations we define a transformer to determine the conditional expectation $\underline{\mathsf{cw}}\mathsf{p}(P, f)$. Intuitively, the conditioning takes place on the probability that all observations in the program are successfully passed. The *conditional expectation* of program P with respect to post–expectation f is given as a pair:

$$\underline{\mathsf{cw}}\mathsf{p}(P, f) \;=\; \big(\mathsf{wp}(P,\ f),\, \mathsf{wlp}(P,\ 1)\big)\ .$$

The first component gives the expectation of the random variable f, whereas $\mathsf{wlp}(P,\ 1)$ is the probability that no observation has been violated (this includes non–terminating runs). The pair $\big(\mathsf{wp}(P,\ f),\, \mathsf{wlp}(P,\ 1)\big)$ is to commonly be interpreted as the quotient

$$\frac{\mathsf{wp}(P, f)}{\mathsf{wlp}(P, 1)}\ .$$

It is possible though that both $\mathsf{wp}(P, f)$ and $\mathsf{wlp}(P, 1)$ evaluate to 0. In that case, the quotient $\frac{0}{0}$ is undefined due to division by zero. The pair (0, 0), however, is well–defined. Let us give an example. Consider the program P from Section 2:

$$
\begin{aligned}
&{\scriptstyle 1:}\ \{x \coloneqq 0\}\ [1/2]\ \{x \coloneqq 1\};\\
&{\scriptstyle 2:}\ \{y \coloneqq 0\}\ [1/2]\ \{y \coloneqq -1\};\\
&{\scriptstyle 3:}\ \mathtt{observe}\ (x + y = 0)
\end{aligned}
$$

Assume we want to compute the conditional expected value of expression x given that observation $x + y = 0$ is passed. This expected value is given by $\underline{\mathsf{cw}}\mathsf{p}(P, x)$ and its computation is sketched below. During the computation we use P_{i-j} to denote the fragment of program P from line i to line j. For the first component of $\underline{\mathsf{cw}}\mathsf{p}(P, x)$ we have:

$$
\begin{aligned}
&\mathsf{wp}(P, x)\\
={}& \mathsf{wp}(P_{1-2}, [x+y=0]\cdot x)\\
={}& 1/2\cdot \mathsf{wp}(P_{1-1}; y \coloneqq 0, [x+y=0]\cdot x) + 1/2\cdot \mathsf{wp}(P_{1-1}; y \coloneqq -1, [x+y=0]\cdot x)\\
={}& 1/2\cdot \mathsf{wp}(P_{1-1}, [x=0]\cdot x) + 1/2\cdot \mathsf{wp}(P_{1-1}, [x=1]\cdot x)\\
={}& 1/2\cdot (1/2\cdot 1\cdot 0 + 1/2\cdot 0\cdot 1) + 1/2\cdot (1/2\cdot 0\cdot 0 + 1/2\cdot 1\cdot 1)\\
={}& 1/4
\end{aligned}
$$

For the second component of $\underline{\mathsf{cw}}\mathsf{p}(P, x)$ we derive:

$$
\begin{aligned}
&\mathsf{wlp}(P, 1)\\
={}& \mathsf{wlp}(P_{1-2}, [x+y=0]\cdot 1)\\
={}& 1/2\cdot \mathsf{wlp}(P_{1-1}; y \coloneqq 0, [x+y=0]) + 1/2\cdot \mathsf{wlp}(P_{1-1}; y \coloneqq -1, [x+y=0])\\
={}& 1/2\cdot \mathsf{wlp}(P_{1-1}, [x=0]) + 1/2\cdot \mathsf{wlp}(P_{1-1}, [x=1])\\
={}& 1/2\cdot (1/2\cdot 1 + 1/2\cdot 0) + 1/2\cdot (1/2\cdot 0 + 1/2\cdot 1)\\
={}& 1/2
\end{aligned}
$$

Thus the conditional expected value of x is

$$\frac{\mathsf{wp}(P, x)}{\mathsf{wlp}(P, 1)} = \frac{1/4}{1/2} = \frac{1}{2} .$$

Revisiting the purely probabilistic example programs of Section 3 (i.e. those not containing any non–deterministic choices), with respect to post–expectation $x+5$ we would obtain the following conditional expectations and according quotients:

P_{div}	(0, 1)	$\frac{0}{1} = 0$
P_{term}	(5, 1)	$\frac{5}{1} = 5$
P_{pr}	(5, 1)	$\frac{5}{1} = 5$
P_{obs}	(0, 0)	$\frac{0}{0} = \mathit{undefined}$

In particular notice that P_{div} and P_{obs} diverge due to different reasons and that our semantics discriminates these two programs by assigning different denotations to them.

Remark 2. Note that the example for the weakest pre–expectation semantics for programs with conditioning does not contain non–determinism. This is deliberate as it is *impossible* to treat non–determinism in a compositional manner [9]. The problem is that determining the conditional expectation in a compositional fashion is not feasible.

4.2 Operational Semantics

MDPs. Markov decision processes (MDPs [23]) serve as a model for probabilistic systems that involve *non–determinism*. An MDP is a state–transition system in which the target of a transition is a discrete probability distribution over states. As in state–transition systems, several transitions may emanate from a state. An MDP thus reduces to an ordinary state–transition system in case all transitions are equipped with a Dirac distribution. In the sample MDP in Figure 1 there is a choice in state s_0 between distributions (or: transitions) α and β. Choosing α results in a *probabilistic choice* of moving either to state s_1 or to state s_2 with probability $1/2$ in each case. Choosing β results in going to s_3 with probability $9/10$ and to s_1 with probability $1/10$. Additionally, in state s_1 a *reward* (also referred to as cost) of 10 is earned; all other states have reward zero, which is omitted from the figure. The *expected reward* of reaching s_1 from state s_0 equals the reward that on average will be earned with respect to the overall probability of reaching state s_1.

These MDPs serve as an operational model for our probabilistic programs. The MDP states are tuples of the form $\langle P, \sigma \rangle$ where P denotes the remaining program to be executed (or equals $\langle \mathit{sink} \rangle$ if the program successfully terminated), and σ is the current valuation of the program variables. Executing a program statement is mimicked by a state change in the MDP. By equipping the MDP states with rewards it is possible to express the expected outcome of a program

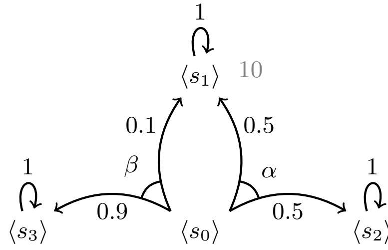
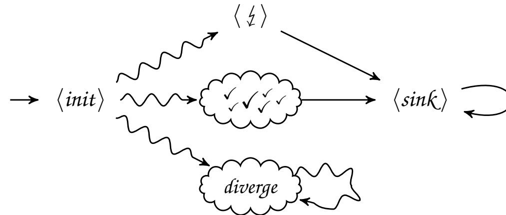


Fig. 1. Sample MDP with four states and a non–deterministic choice between α and β

variable as an expected reward on the MDP. This will become more explicit when discussing the relationship to the weakest pre-expectation semantics at the end of this section. Note that the resulting MDP of a probabilistic program is in general countably infinite (as the variable domains can be infinitely large) and parametric (as probabilistic choices can be parametric).

The structure of MDPs for probabilistic programs. Let us examine the different kinds of *runs* a program can have. First, we have *terminating runs* where—in presence of conditioning—one has to distinguish between runs that satisfy the condition and those that do not. In addition, a program may have *diverging runs*, i.e. runs that do not terminate. Schematically, the MDP of a probabilistic program has the following structure:



For terminating runs of the program, we use a dedicated $\langle sink \rangle$ state where all terminating runs will end. All diverging runs never reach $\langle sink \rangle$. A program terminates either successfully, i.e. a run passes a ✓–labeled state, or terminates due to violating an observation, i.e. a run passes $\langle \lightning \rangle$. Squiggly arrows indicate reachability via possibly multiple paths and states; the clouds indicate that there might be several or even infinitely many states of the particular kind. The ✓–labeled states are the *only ones* where one is allowed to assign positive reward as this corresponds to a desired outcome of the program when subsequently terminating. Note that the sets of paths that eventually reach $\langle \lightning \rangle$, eventually reach ✓, or diverge, are pairwise disjoint.

As an example, consider the following program:

$$\{\{x \coloneqq 5\} \square \{x \coloneqq 2\}\} \; [q] \; \{x \coloneqq 2\};$$
$$\texttt{observe}\ (x > 3)$$

With parametrized probability q, a non–deterministic choice either assigns x with 2 or 5. With probability $1-q$, x is directly assigned 2, so in this program branch no non–deterministic choice occurs. The event that x exceeds 3 is observed. For the sake of readability, let: $P_1 = \{x := 5\} \,\Box\, \{x := 2\}$, $P_2 = x := 2$, $P_3 =$ `observe` $(x > 3)$, and $P_4 = x := 5$. Figure 2 shows the resulting MDP, where σ_I denotes some initial variable valuation for x. Let $\sigma_I[x/y]$ denote the variable valuation that is obtained from σ_I by replacing x by y. Starting in

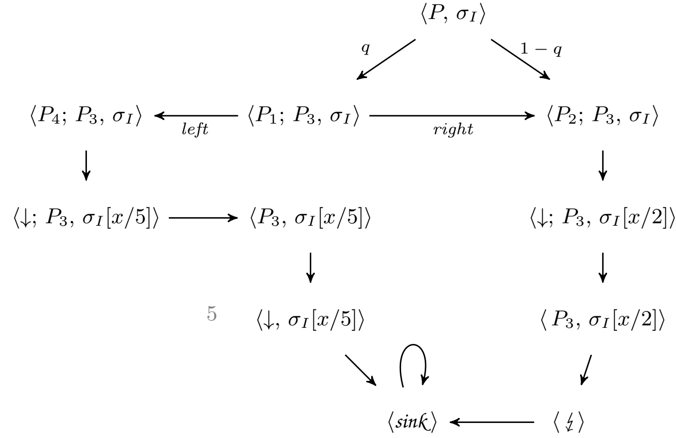


Fig. 2. Reward MDP for the example program

the initial state $\langle P, \sigma_I \rangle$, the probabilistic choice takes place. With probability q, the next state is $\langle P_1;\, P_3,\, \sigma_I \rangle$ while with probability $1-q$, the next state is $\langle P_2;\, P_3,\, \sigma_I \rangle$. The non–deterministic choice in state $\langle P_1;\, P_3,\, \sigma_I \rangle$ is indicated by *left* and *right*. Note that non–deterministic choices yield a choice in the MDP between Dirac distributions.

Conditional expected rewards. The *operational semantics* of a probabilistic program P, a program state σ and an expectation (i.e. random variable) f is the reward MDP $\mathfrak{R}^f_\sigma[\![P]\!]$ constructed as described in the paragraph above. Note that in the context of MDPs, the random variable f can also be seen as a *reward function* which adds a positive real–valued reward to certain states of the MDP. In our previous example, the only state with positive reward (5) is $s' := \langle \downarrow,\, \sigma_I[x/5] \rangle$; all other states have reward zero. In absence of conditioning, we are interested in the *expected reward* to reach a $\langle sink \rangle$–state from the MDP's initial state σ_I:

$$\mathsf{er}(P,\, f)(\sigma_I) = \mathsf{ExpRew}^{\mathfrak{R}^f_{\sigma_I}[\![P]\!]}(\Diamond\, sink)\,.$$

The right-hand side denotes the sum over all (countably many) paths in the reward MDP $\mathfrak{R}^f_{\sigma_I}[\![P]\!]$ where for each path its likelihood is weighed with its

reward. The reward of a path is simply the sum of the rewards of the states it contains.

In the presence of conditioning (i.e. for programs having `observe`–statements), we consider the *conditional expected reward* to reach a $\langle \mathit{sink} \rangle$–state without intermediately passing the $\langle \lightning \rangle$–states:

$$\mathsf{cer}(P,\, f)(\sigma_I) = \frac{\mathsf{ExpRew}^{\mathfrak{R}^{f}_{\sigma_I}[\![P]\!]}\,(\Diamond\, \mathit{sink} \cap \neg \Diamond \lightning\,)}{\Pr(\neg \Diamond \lightning\,)}.$$

Let us illustrate these two notions by our example reward MDP in Figure 2. Consider a scheduler choosing action *left* in the state $\langle P_1;\, P_3,\, \sigma_I \rangle$. Then, the only path accumulating positive reward is the path π going from $\langle P,\, \sigma_I \rangle$ via s' to $\langle \mathit{sink} \rangle$; it has reward 5 and occurs with probability q. This gives an expected reward

$$\mathsf{er}(P,\, f)(\sigma_I) \;=\; 5 \cdot q\ .$$

The overall probability of not reaching $\langle \lightning \rangle$ is q. The conditional expected reward of eventually reaching $\langle \mathit{sink} \rangle$ given that $\langle \lightning \rangle$ is not reached is hence

$$\mathsf{cer}(P,\, f)(\sigma_I) \;=\; \frac{5 \cdot q}{q} \;=\; 5\ .$$

Consider now the scheduler choosing *right* at state $\langle P_1;\, P_3,\, \sigma_I \rangle$. In this case, there is no path with positive accumulated reward, yielding an expected reward of 0. The probability of not reaching $\langle \lightning \rangle$ is also 0. The conditional expected reward in this case is undefined ($0/0$). Thus, the *right* branch is preferred over the *left* branch by a demonic scheduler, as discussed in Section 3.

4.3 Relating the Two Semantic Views

A key insight is that the operational program semantics in terms of MDPs and the semantics in terms of expectation transformers, as explained in the previous section, correspond in the following sense:

Theorem 1 (Transfer theorem [11]). *For a probabilistic program P without observations, a random variable f, and some initial state σ_I:*

$$\mathsf{wp}(P,\, f)(\sigma_I) = \mathsf{er}(P,\, f)(\sigma_I)\ .$$

Stated in words, this result asserts that the weakest-pre-expectation of program P in initial state σ_I wrt. post-expectation f coincides with the expected reward in the MDP of P where reward f is assigned to successfully terminating states. For probabilistic programs with observations but without non–determinism we can establish a correspondence between the conditional expected reward on the MDP of a program and its conditional pre–expectation:

Theorem 2 (Transfer theorem for conditional expectations [9]). *For a purely probabilistic program P (with observations), a random variable f, and some initial state σ_I, let* $\underline{\mathsf{cw}}\mathsf{p}(P, f) = (g, h)$. *Then*

$$\frac{g(\sigma_I)}{h(\sigma_I)} \simeq \mathsf{cer}(P,\, f)(\sigma_I) ,$$

where $x \simeq y$ holds iff either $x = y$ or both sides of the equation are undefined.

For weakest *liberal* pre-expectations, we obtain a similar pair of theorems, where the notions of (conditional) *liberal* expected reward also takes the mere probability of not reaching the target states into account. For further details, the reader is referred to [11, 9, 10].

5 Program Transformations

In this section, we use the semantics to show the correctness of a program transformation aimed at removing observations from programs. The program transformation basically allows removing observations from programs through the introduction of a global loop. It is motivated by a well–known technique to simulate a uniform distribution in some interval $[a, b]$ using fair coins [26, Th. 9.2]. The technique is illustrated by a program simulating a six–sided die:

```
1: repeat
2:     {a_0 := 0} [1/2] {a_0 := 1};
3:     {a_1 := 0} [1/2] {a_1 := 1};
4:     {a_2 := 0} [1/2] {a_2 := 1};
5:     i := 4a_0 + 2a_1 + a_0 + 1
6: until (1 ≤ i ≤ 6)
```

The body of the loop simulates a uniform distribution over the interval $[1, 8]$, which is repeatedly sampled (in variable i) until its outcome lies in $[1, 6]$. The effect of the repeated sampling is precisely to condition the distribution of i to $1 \leq i \leq 6$. As a result, $\Pr[i = N] = \frac{1}{6}$ for all $N = 1, \ldots, 6$.

Our program transformation follows the same idea. Given a program P with observations, we repeatedly sample executions from P until the sampled execution satisfies all observations in P. To implement this, we have to take into account three issues. First, we introduce a flag that signals whether all observations along a program execution were satisfied or not. Let variable *flag* be initially `true` and replace every observation `observe` (G) in the original program by the assignment $\mathit{flag} := \mathit{flag} \wedge G$. In this way, the variable *flag* is true until an observation is violated. Secondly, since a program execution is no longer blocked on violating an observation, we need to modify the program to avoid any possible divergence after an observation has been violated. This is achieved

by adapting the loop guards. For instance loop `while` (G) $\{P\}$ is transformed into `while` $(G \wedge \mathit{flag})$ $\{P\}$, whereas loop `repeat` $\{P\}$ `until` (G) is changed into `repeat` $\{P\}$ `until` $(G \vee \neg \mathit{flag})$. Finally, observe that we need to keep a permanent copy of the initial program state since every time we sample an execution, the program must start from its original initial state. In general, the transformed program will have the following shape:

```
1: s_1, ..., s_n := x_1, ..., x_n;
2: repeat
3:    flag := true;
4:    x_1, ..., x_n := s_1, ..., s_n;
5:    modified version of original program;
6: until (flag)
```

Here $x_1, \ldots, x_n$ denote the set of variables that occur in the original program and $s_1, \ldots, s_n$ are auxiliary variables used to store the initial program state; note that if the original program is closed (i.e. independent of its input), Lines 1 and 4 can be omitted. Line 5 includes the modified version of the original program which accounts for the replacement of observations by flag updates and, possibly, the adaptation of loop guards.

We illustrate the program transformation on the left program below:

```
1: {x := 0} [1/2] {x := 1};
2: {y := 0} [1/2] {y := −1};
3: observe (x + y = 0)
```

```
1: repeat
2:    flag := true;
3:    {x := 0} [1/2] {x := 1};
4:    {y := 0} [1/2] {y := 1};
5:    flag := flag ∧ (x + y = 0)
6: until (flag)
```

The transformed—`observe`–free—program is given on the right. Using the operational semantics from Section 4 we establish that the transformation is semantic-preserving:

Theorem 3 (Correctness of the program transformation). *Let P be a probabilistic program and let P' be the result of applying the above transformation to program P. Then for initial state σ_I and reward function f,*

$$\mathsf{cer}(P,\, f)(\sigma_I) \;=\; \mathsf{er}(P',\, f)(\sigma_I) \ .$$

In some circumstances it is possible to apply a dual program transformation that replaces program loops with observations. This is applicable when there is no data flow between loop iterations and the samplings across iterations are thus independent and identically distributed. This is the case, e.g. for the earlier program that simulates a six-sided dice. One can show that this program is

semantically equivalent to the program

$$
\begin{aligned}
&1\text{: } \{a_0 \coloneqq 0\}\ [1/2]\ \{a_0 \coloneqq 1\};\\
&2\text{: } \{a_1 \coloneqq 0\}\ [1/2]\ \{a_1 \coloneqq 1\};\\
&3\text{: } \{a_2 \coloneqq 0\}\ [1/2]\ \{a_2 \coloneqq 1\};\\
&4\text{: } i \coloneqq 4a_0 + 2a_1 + a_0 + 1;\\
&5\text{: } \texttt{observe}\ (1 \leq i \leq 6)
\end{aligned}
$$

6 Conclusion

We have presented two views on the semantics of probabilistic programs and showed their relationship for purely probabilistic programs. Whereas the operational semantics can cope with all features—loops, conditioning, non–termination, and non–determinism—the weakest pre–expectation approach cannot be directly applied to handle non–determinism in this setting. We believe that formal semantics, verification, and program analysis has much to offer to improve modern probabilistic programming, and consider this as an interesting and challenging avenue for further research.

Acknowledgement. We thank the reviewers for their valuable feedback.

References

1. Barthe, G., Köpf, B., Olmedo, F., Béguelin, S.Z.: Probabilistic relational reasoning for differential privacy. ACM Trans. Program. Lang. Syst. 35(3), 9 (2013)
2. Chakarov, A., Sankaranarayanan, S.: Expectation invariants for probabilistic program loops as fixed points. In: Proc. of SAS. LNCS, vol. 8723, pp. 85–100. Springer (2014)
3. Cock, D.: Verifying probabilistic correctness in Isabelle with pGCL. El. Proc. in Th. Comp. Sc. 102, 167–178 (2012)
4. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall (1976)
5. Feldman, Y.A., Harel, D.: A probabilistic dynamic logic. In: Proc. of STOC. pp. 181–195. ACM (1982)
6. Gordon, A.D.: An agenda for probabilistic programming: Usable, portable, and ubiquitous (2013), http://research.microsoft.com/en-us/projects/fun
7. Gordon, A.D., Graepel, T., Rolland, N., Russo, C.V., Borgström, J., Guiver, J.: Tabular: a schema-driven probabilistic programming language. In: Proc. of POPL. pp. 321–334. ACM Press (2014)
8. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Proc. of FOSE. pp. 167–181. ACM Press (2014)
9. Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J.P., McIver, A., Olmedo, F.: Conditioning in probabilistic programming. In: Proc. of MFPS. pp. 1–12. ENTCS (2015)
10. Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J.P., McIver, A., Olmedo, F.: Conditioning in probabilistic programming. CoRR (2015)

11. Gretz, F., Katoen, J.P., McIver, A.: Operational versus weakest pre-expectation semantics for the probabilistic guarded command language. Perform. Eval. 73, 110–132 (2014)
12. Hur, C.K., Nori, A.V., Rajamani, S.K., Samuel, S.: Slicing probabilistic programs. In: Proc. of PLDI. pp. 133–144. ACM Press (2014)
13. Hurd, J., McIver, A., Morgan, C.: Probabilistic guarded commands mechanized in *HOL*. Theor. Comput. Sci. 346(1), 96–112 (2005)
14. Kaminski, B.L., Katoen, J.P.: On the hardness of almost-sure termination. In: Proc. of MFCS. LNCS, vol. 9234. Springer (2015)
15. Katoen, J.P., McIver, A., Meinicke, L., Morgan, C.C.: Linear-invariant generation for probabilistic programs. In: Proc. of SAS. LNCS, vol. 6337, pp. 390–406. Springer (2010)
16. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. 22(3), 328–350 (1981)
17. McIver, A., Morgan, C.: Abstraction, Refinement And Proof For Probabilistic Systems. Springer (2004)
18. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995)
19. Nori, A.V., Hur, C.K., Rajamani, S.K., Samuel, S.: R2: An efficient MCMC sampler for probabilistic programs. In: Proc. of AAAI. AAAI Press (2014)
20. Olderog, E.R.: Nets, Terms and Formulas: Three Views of Concurrent Processes and their Relationship. Cambridge Tracts in Theoretical Computer Science, Cambridge Universoty Press (1990)
21. Paige, B., Wood, F.: A compilation target for probabilistic programming languages. In: Proc. of ICML. JMLR Proceedings, vol. 32, pp. 1935–1943. JMLR.org (2014)
22. Pfeffer, A.: Figaro: An object-oriented probabilistic programming language. Technical report, Charles River Analytics (2000)
23. Puterman, M.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley and Sons (1994)
24. Sampson, A., Panchekha, P., Mytkowicz, T., McKinley, K.S., Grossman, D., Ceze, L.: Expressing and verifying probabilistic assertions. In: Proc. of PLDI. p. 14. ACM (2014)
25. Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM Journal on Computing 13(2), 292–314 (1984)
26. Shoup, V.: A Computational Introduction to Number Theory and Algebra. Cambridge University Press (2009)