The Semantics of Probabilistic Programs *

Ashwin Nayak †

May 1997

Abstract

The use of randomness has gained much popularity in the algorithms community, and not without reason: randomised algorithms are often not only more time efficient, but are also much simpler than their deterministic counterparts. Moreover, many problems which are provably either intractable or unsolvable deterministically admit randomized solutions. If formal verification of realistic programs were to be achieved, it would be imperative to include constructs for randomization in programming languages, and to develop appropriate semantics for such languages. A more compelling reason—from the point of view of algorithm design—for exploring the semantics of probabilistic programs is that it might provide another approach to understanding how the use of randomness leads to simplicity and efficiency in algorithms.

Much work has been done on the semantics of probabilistic programs since the seminal paper by Kozen [6] in 1981. This report concentrates on the semantics of a simple imperative language equipped with a probabilistic construct. A weakest pre-condition semantics for this language (in the style of Dijkstra [1]) is presented, and studied with an eye on its ultimate use as a vehicle for proving properties of realistic probabilistic programs, and how closely it reflects reasoning about such programs. Much of the work reported can be attributed to the Probabilistic Systems Group at Oxford University [12].

1 Introduction

1.1 Motivation

Recent times have seen an increased use of randomized algorithms in a host of areas [11]. The reasons for their immense popularity are not hard to understand: they are much simpler and

*Project Report, CS 263: Design of Programming Languages, Spring 1997.
†Computer Science Division, UC Berkeley. Email: `ashwin@cs.berkeley.edu`.

more often than not, more time efficient than their deterministic counterparts. Moreover, the only known efficient solutions to some inherently intractable problems (sometimes provably so) employ randomization. It is then not very surprising that these algorithms have found their way into the practical domain as well. ([11] has quite a comprehensive coverage of such algorithms.) It is then very natural to demand that traditional methods for program verification be extended to handle reasoning about probabilistic programs. This report looks at one such extension.

Another compelling reason to study probabilistic programs from a semantics point of view is that it might provide an alternative approach to understanding the exact role of randomness in their efficiency and simplicity. Very little is known to date about the formal relationship between deterministic and randomised algorithms. For instance, it is not known whether for every problem and for every deterministic algorithm for it, there is a randomized algorithm that is either more efficient, or is at least as efficient but is in some well-defined sense much simpler. While the notion of efficiency is quite well understood, there have been no attempts to formalise the notion of simplicity. It is hoped that a reasonable definition of simplicity will emerge from a study of the semantics of probabilistic programs. It would also be interesting to see if this increased understanding can lead to the formal derivation of new and better probabilistic solutions to problems.

Yet another motivating factor for the present study is the element of unreliability in certain systems (such as faulty hardware) which implement programs. Such unreliability is often accurately modelled as being probabilistic. A formal 'probabilistic semantics' may be useful in arguing about the degree of fault-tolerance of a program when run on unreliable hardware.

Not surprisingly, the theories developed for dealing with standard deterministic programs are grossly inadequate for handling probabilistic programs. For one, the demands on a standard program are very high: we require that they terminate with the correct result on every input. Probabilistic programs, on the other hand, are allowed a less stringent correctness requirement: they can deliver incorrect results, albeit only with a small probability (independent of the input). Moreover, they are not required to deliver results in a fixed amount of time (determined by the input). A good average running time is satisfactory in many cases. Thus, there is a need to develop a new language in which one can express and manipulate the (probabilistic) concepts encountered while analysing properties of randomized programs.

1.2 Scope of the report and related work

This report is confined to the study of a simple *imperative* language equipped with a probabilistic primitive. There are two reasons for concentrating on an imperative language. Firstly, most currently popular languages happen to be imperative (hence most existing software is written in such languages), and so the first attempt at formalizing reasoning about probabilistic programs would naturally be to extend semantics of imperative languages to include randomness. The second reason, which is related to the one of the motivations for this study, is that algorithm design has traditionally involved an imperative model of computation (the RAM), and theorists understand such models better. So it would presumably be simpler to analyse the effect of introducing a variation in such a model rather than in another, less familiar model. The report also tries to follow the axiomatic approach brought to fore by Hoare [4] (following Floyd) since it most closely reflects the reasoning about programs in practice. We also suspect that this is more suitable for analysing the role of randomization in algorithms.

The exact object of study is the semantics of a modification (mostly to include a probabilistic primitive) of Dijkstra's language of guarded commands [1]. Encouraged by the success of the predicate transformer semantics approach in the domain of non-probabilistic programs, we look at such a semantics for the language mentioned.

The predicate transformer view was first proposed by Kozen [6, 7] as one of two possible ways of looking at probabilistic programs: either as a function on the 'actual' inputs adjoined with an infinite stream of random ones, or as a transformation on distributions on the inputs. He shows that these semantics are, in fact, equivalent. Following the trend set in [7], there has been much work related to this approach by He, who takes a 'relational' view of programs, and by Jones [5], who shows how one can derive Kozen's semantics via a more general method of injecting probability into a semantic domain. These are finally unified in the work of Morgan *et al.* in [10] where non-determinism is introduced into the model of Kozen and Jones. We study this end product while deliberately ignoring explicit non-determinism.

It should be noted that by spelling out the scope of the report as above, we are disregarding a big body of work on related topics which are also relevant to the goals of the project. Rao develops semantics for a probabilistic parallel language in [13] with which once can reason about properties of programs that hold either deterministically, or with probabilistic certainty. Randomness is introduced and studied in the 'reactive' model of computation in works such as [3]. Logics of probability have also been proposed (see, for example [2]). A study of these topics has been left out in favour of a deeper treatment of the other topics mentioned before.

1.3 Organisation of the report

The rest of the report is organized as follows. In Section 2, we present a well-studied imperative language and its weakest precondition semantics. We show how this semantics can be extended to probabilistic programs by generalizing the notion of pre- and post conditions in Section 3. Next, in Section 4, we describe the probabilistic analogue of the standard 'healthiness conditions' and their relation to general properties of programs. We conclude by evaluating the 'predicate transformer' view of probabilistic programs in Section 5.

2 Semantics of an imperative language

The syntax for the language we will be looking at is based on Dijkstra's language of guarded commands [1]. It is defined formally below:

$$\begin{array}{rcl} \mathit{prog} & := & \textbf{abort} \mid \textbf{skip} \mid x := E \mid \mathit{prog}; \mathit{prog} \mid \\ & & \mathit{prog}\ \textbf{if}\ B\ \textbf{else}\ \mathit{prog} \mid (\mu X \cdot \mathcal{C}) \end{array}$$

In the above, *prog* ranges over programs, x over the set of variables, E over expressions, B over Boolean expressions, and $\mathcal{C}$ over program *schema* in which the program *name* X can appear. Thus the language consists of the conventional constructs for non-terminating computation, a 'no-op', assignments, sequential composition, conditional statements, and recursive invocation. Note that loops can be expressed in terms of the recursion primitive:

$$\textbf{do}\ B \rightarrow \mathit{prog}\ \textbf{od} \quad := \quad (\mu X \cdot (\mathit{prog}; X)\ \textbf{if}\ B\ \textbf{else}\ \textbf{skip})$$

and that recursion can be simulated using loops, so that they are really equally expressive.

In the *predicate transformer* view of programs, a program is considered as an entity which effects some transformation on the variables that occur in it. The mapping between the variables and the values they can assume characterise the *state* of a program. The transformation effected by the program can then be described in terms of the *precondition* and the *postcondition* satisfied by the initial and the final state, respectively, of the program. The pre- and postconditions are predicates over the set of states S of the program. The *weakest precondition* corresponding to a postcondition Q is that set of states starting from which the program *prog* is guaranteed to terminate in a state satisfying Q. It is denoted by $\mathrm{wp}.\mathit{prog}.Q$. Predicate transformer semantics identifies the meaning of a program with the mapping it induces between a postcondition and the weakest precondition corresponding to it, thus embedding it in the function space $\mathcal{D}S := 2^S \rightarrow 2^S$. $\mathcal{D}S$ inherits a *refinement* ordering $\sqsubseteq$ from the underlying space 2^S in the natural way.

The standard wp-semantics for the language defined above is can be written as:

$$\begin{array}{rcl} \mathrm{wp}.\textbf{abort}.Q & := & \mathit{false} \\ \mathrm{wp}.\textbf{skip}.Q & := & Q \\ \mathrm{wp}.(x := E).Q & := & Q[x := E] \\ \mathrm{wp}.(\mathit{prog}; \mathit{prog}').Q & := & \mathrm{wp}.\mathit{prog}.(\mathrm{wp}.\mathit{prog}'.Q) \\ \mathrm{wp}.(\mathit{prog}\ \textbf{if}\ B\ \textbf{else}\ \mathit{prog}').Q & := & (B \wedge \mathrm{wp}.\mathit{prog}.Q) \vee (\neg B \wedge \mathrm{wp}.\mathit{prog}'.Q) \\ \mathrm{wp}.(\mu X \cdot \mathcal{C}).Q & := & \mathrm{wp}.\mathrm{LFP}(\mu X \cdot \mathcal{C}).Q \end{array}$$

The abbreviation $\mathrm{LFP}(\mu X \cdot \mathcal{C})$ stands for the least fixed point of the function $f : \mathcal{D}S \rightarrow \mathcal{D}S$ defined as $f(\mathrm{wp}.X) = \mathrm{wp}.\mathcal{C}$. The semantics given above is quite intuitive, and we omit further explanation. We only pause to point out that these correspond closely to the Floyd-Hoare approach (this becomes obvious once further 'healthiness conditions' of predicate transformers are proved).

3 Extension to probabilistic programs

We now add a probabilistic primitive to the language defined in Section 2:

$$prog \quad := \quad \cdots \mid prog \; {}_P\!\oplus \; prog$$

where P ranges over real number expressions (possibly involving the state variables) in the range $[0, 1]$. The intended meaning of the construct is that when executed, the left choice is selected with probability P, and the right one with probability $1 - P$. (Note that this is expressive enough for translating 'real' randomized primitives into this language.)

In order to extend predicate transformer semantics for the earlier constructs to this variant of the language, we first note that a standard predicate Q can be viewed alternatively as a $\{0, 1\}$-valued function: a state s satisfies Q if $Q.s = 1$.[1] Similarly, a state s satisfies $\mathrm{wp}.prog.Q$ if $\mathrm{wp}.prog.Q.s = 1$. As a step towards defining probabilistic predicates, we can interpret the $\{0, 1\}$ values as being the *probabilities* of the events $s \in Q$ and $s \in \mathrm{wp}.prog.Q$. One might be tempted to define a probabilistic predicate Q as a function on S taking values in the interval $[0, 1]$, and thus, for a (standard) predicate Q, $\mathrm{wp}.prog.Q.s$ would give the probability that starting in the state s, the program terminates in a state satisfying Q. This interpretation breaks down when we consider intermediate preconditions (in the rule for sequencing), where the postcondition for the first part of the program may not be a standard predicate taking $\{0, 1\}$ values. So we generalise the earlier interpretation: we consider $\mathrm{wp}.prog.Q.s$ as the *expectation* of the function Q over the set of states s' that the program may terminate in when started in the state s. When Q is standard, this reduces to the probability interpretation. There is no reason we should restrict Q to take values in $[0, 1]$, and thus we define a probabilistic predicate as a function from the states to all non-negative reals $\mathcal{R}_{\geq}$. Probabilistic programs are then functions from probabilistic postconditions to weakest probabilistic preconditions. The space of probabilistic predicates is denoted by $\mathcal{P}S := S \to \mathcal{R}_{\geq}$, and it inherits the natural ordering $\Rrightarrow$ from the usual ordering on $\mathcal{R}_{\geq}$. Probabilistic programs are seen to be embedded in $\mathcal{J}S := \mathcal{P}S \to \mathcal{P}S$, the refinement ordering $\sqsubseteq$ on which is derived pointwise from $\Rrightarrow$. Under various conditions (including the boundedness of the probabilistic predicates), both $\mathcal{P}S$ and $\mathcal{J}S$ can be shown to be complete partial orders.

We are now ready to define the new semantics in terms of probabilistic predicates. It is spelt out below.

$$\begin{array}{rcl}
\mathrm{wp}.\mathbf{abort}.Q & := & 0 \\
\mathrm{wp}.\mathbf{skip}.Q & := & Q \\
\mathrm{wp}.(x := E).Q & := & Q[x := E] \\
\mathrm{wp}.(prog; prog').Q & := & \mathrm{wp}.prog.(\mathrm{wp}.prog'.Q) \\
\mathrm{wp}.(prog \;\mathbf{if}\; B \;\mathbf{else}\; prog').Q & := & [B] \times \mathrm{wp}.prog.Q + [\neg B] \times \mathrm{wp}.prog'.Q
\end{array}$$

[1] We borrow the notation $f.x$ of Morgan *et al.* [12] for function application $f(x)$.

$$\begin{aligned}
\text{wp}.(\mu X \cdot \mathcal{C}).Q &\;:=\; \text{wp}.\text{LFP}(\mu X \cdot \mathcal{C}).Q \\
\text{wp}.(prog\ {}_P\!\oplus\ prog').Q &\;:=\; P \times \text{wp}.prog.Q + (1-P) \times \text{wp}.prog'.Q
\end{aligned}$$

Here, 0 stands for the constant function taking the value 0 everywhere, $[\cdot]$ maps the Boolean values *true* and *false* to 1 and 0 respectively, and LFP now stands for the least fixed point in the space $\mathcal{J}S$.

There seem to be no great surprises in the semantics given above, and the ease with which it generalizes the standard semantics should be considered a merit of this approach.

We now take a detour to illustrate the above semantics using a tiny program. All that the program does is flip two fair coins. Let $p := p_1; p_2$, where $p_1 := (n := 0\ {}_{\frac{1}{2}}\!\oplus\ n := 1)$, and $p_2 := (m := 0\ {}_{\frac{1}{2}}\!\oplus\ m := 1)$. Let $[n = m]$ stand for the standard predicate which is satisfied by exactly those states in which $n = m$. wp.p.$[n = m]$ should give us the probability that we end up getting the same result on both coins (starting from *any* state). We verify this using the formal rules:

$$\begin{aligned}
\text{wp}.p_2.[n = m] &= (1/2)\ \text{wp}.(m := 0).[n = m] + (1/2)\ \text{wp}.(m := 1).[n = m] \\
&= (1/2)\ ([n = 0] + [n = 1])
\end{aligned}$$

Denote wp.p_2.$[n = m]$ by Q. Then, wp.p.$[n = m]$ = wp.p_1.Q, which is given by:

$$\begin{aligned}
\text{wp}.p_1.Q &= (1/2)\ \text{wp}.(n := 0).Q + (1/2)\ \text{wp}.(n := 1).Q \\
&= (1/4)\ ([0 = 0] + [0 = 1] + [1 = 0] + [1 = 1]) \\
&= (1/2)
\end{aligned}$$

This was a toy program, and it turned out to be very easy to prove a property about it. Another reason why the example was a triviality was that it did not contain any loop (or recursive call) in it. As in the non-probabilistic case, one has to prove some general properties about the wp-semantics given above in order to arrive at Hoare-like rules that help us analyse looping programs. We turn to this next.

4 Healthiness conditions

In [1], Dijkstra imposes 'healthiness' (well-formedness) conditions on standard predicate transformers, namely:

1. feasibility : for all *prog*, wp.*prog*.0 $=$ 0.
2. conjunctivity : for all *prog*, wp.*prog*$(Q \wedge Q')$ $=$ wp.*prog*.$Q \wedge$ wp.*prog*.Q'.
3. monotonicity : for all *prog*, if $Q \Rightarrow Q'$, then wp.*prog*.$Q \Rightarrow$ wp.*prog*.Q'.

4. continuity : for all *prog*, wp.*prog*.$(\sqcup \mathbf{Q}) = \sqcup$ wp.*prog*.Q_i, where $\mathbf{Q} = (Q_i)$ is an non-decreasing chain in 2^S.

5. disjuntivity : for all *prog*, wp.*prog*$(Q \vee Q')$ = wp.*prog*.$Q \vee$ wp.*prog*.Q'.

These conditions exactly characterize programs that can be given an alternate semantics as relations between initial and final states, and they can be used to derive general laws for reasoning about programs (in particular, the loop-invariant rule of Hoare). The question that arises then is whether we can come up with a similar set of rules that make reasoning about programs easier.

As a first attempt, we can try following the paradigm of interpreting standard concepts in terms of their probabilistic counterparts ($\vee$ is interpreted as the maximum $\sqcup$, and $\wedge$ is interpreted as minimum $\sqcap$). Not all the conditions then still hold. For one, probabilistic programs don't distribute over $\sqcap$ in general: $\mathbf{Pr}[X \cap Y]$ may be much lesser than than $\min\{\mathbf{Pr}[X], \mathbf{Pr}[Y]\}$. It so happens that in the probabilistic case, all well-formedness conditions can be unified into one: *linearity.* A predicate transformer $j \in \mathcal{J}S$ is said to be linear if for all predicates $Q, R \in \mathcal{P}S$, and scalars $\alpha, \beta \in \mathcal{R}_{\geq}$, $j.(\alpha Q + \beta R) = \alpha(j.Q) + \beta(j.R)$. Note that for all *prog*, wp.*prog* is linear. (This can be proved by induction.) The feasibility condition can be generalized so that j is said to be feasible if $j.Q \Rrightarrow \sqcup Q$. (All this says is that a convex combination of a set of values cannot be greater than the maximum of those values.) By taking $Q = 0$, we get the standard feasibility condition. One can similarly derive the analogues of the other conditions from the 'master' condition of linearity, but they are not very interesting. An interesting consequence of linearity is the following theorem about loops.

Theorem 1 *Let* $h.R := [B] \times$ wp.*prog*.R, *where* B *is a Boolean expression. Then,*

$$\text{wp}.(\mathbf{do}\ B \rightarrow prog\ \mathbf{od}).Q \;=\; \sum_{i=0}^{\infty} h^i.([\neg B] \times Q)$$

The importance of this theorem can be judged by the fact that it enables the computation of the expected number of steps the loop takes to terminate (say, by setting $Q.s$ to the value of a loop counter in the state s). The most important consequence of linearity is a generalization of the Floyd-Hoare loop-invariance rule to the probabilistic case.

Theorem 2 *Let* $loop := \mathbf{do}\ B \rightarrow prog\ \mathbf{od}$. *If* I *is a* wp-*invariant of loop, i.e. if* $[B] \sqcap I \Rrightarrow$ wp.*prog*.I, *and* I *ensures termination of the loop* ($I \Rrightarrow$ wp.*loop*.1), *then,* $I \Rrightarrow$ wp.*loop*.$([\neg B] \sqcap I)$.

We can also prove another result (very well-known in probability circles in a different garb), which is helpful in proving termination of loops. It basically states that if starting in any state in say I, the probability of a loop eventually terminating is at least $p > 0$, then, the loop terminates with *certainty* when started in a state in I.

Lemma 3 *Let I be a* wp*-invariant of loop. If for some fixed $p > 0$, we have $p(I) \Rightarrow$* wp.*loop*.1*, then in fact $I \Rightarrow$* wp.*loop*.1.

In the case of standard programs, termination of loops is proved using 'variants' based on the state of the program. Variants are integer valued expressions over the state that are bounded below, but decrease strictly after each iteration of the loop. When dealing with probabilistic programs, it often happens that no such variant exists, eventhough the loop is guaranteed to terminate with certainty. If, however, we are assured of a variant that is bounded above, we can prove the termination of the loop from the properties discussed above.

5 Conclusion

Having looked at some of the literature on probabilistic program semantics, in particular predicate transformer semantics, one can definitely say that we are a long way off from achieving complete formalization of the kind of reasoning about randomized algorithms one sees in practice. The following inadequacies come to mind:

1. The absence of an interface with a logic of probability. Often very little can be inferred about the correctness of a randomized algorithm from text of the algorithm alone. One needs to establish properties of the particular mathematical structures over which the computation is being done (e.g. graphs, or finite groups) before the correctness of the algorithm can be deduced. Sometimes, we need to establish results about the state of a program at an intermediate state in order to conclude some property of the entire program, and proving such a result could call for more sophisticated tools from a formal logic of probability. Such a logic may also help bring formal reasoning closer (by abstraction) to everyday reasoning about randomized algorithms.

2. A priori knowledge of properties (such as loop invariants or variants) of a program is required for any non-trivial program. This is a problem also with axiomatic semantics for deterministic programs, and inherently intractable. But certainly, it would be interesting to see if we could get rid of it in restricted non-trivial cases.

3. The lack of interesting general laws about termination in the probabilistic case is striking. How would, for example, one argue that the simple random walk on the line with an absorbing barrier at 0 terminates with certainty?

References

[1] E. W. Dijkstra. *A discipline of programming.* Prentice Hall International, Englewood Cliffs, 1976.

[2] R. Fagin, J. Y. Halpern and N. Megiddo. A logic for reasoning about probabilities. *Information and Computation* **87** (1990), pp. 78–128.

[3] R. van Glabbeek, S. A. Smolka, B. Steffen and C. M. N. Tofts. Reactive, generative, and stratified models of probabilistic processes. *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, 1990, pp. 130–141.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM* **12**(10) (1969), pp. 576–580.

[5] C. Jones. Probabilistic nondeterminism. PhD Thesis, Monograph ECS-LFCS-90-105, Edinburgh University, 1990.

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

[7] D. Kozen. A probabilistic PDL. *Proceedings of the 15th ACM Symposium on Theory of Computing*, 1983, pp. 291–297.

[8] C. Morgan, A. McIver, K. Seidel and J. W. Sanders. Probabilistic predicate transformers. Technical Report PRG-TR-4-95, Programming Research Group, Oxford University, 1995.

[9] C. Morgan. Proof rules for probabilistic loops. Technical Report PRG-TR-25-95, Programming Research Group, Oxford University, 1995.

[10] C. Morgan, A. McIver and K. Seidel. Probabilistic predicate transformers. *ACM Transactions on Programming Languages and Systems* **18**(3) (1996), pp. 325-353.

[11] R. Motwani and P. Raghavan. *Randomized algorithms.* Cambridge University Press, Cambridge, 1995.

[12] Probabilistic Systems Group. Oxford University.
`http://www.comlab.ox.ac.uk/oucl/groups/probs/project.html`

[13] J. R. Rao. Reasoning about probabilistic parallel programs. *ACM Transactions on Programming Languages and Systems* **16**(3) (1994), pp. 798–842.

[14] K. Seidel, C. Morgan and A. McIver. An introduction to probabilistic predicate transformers. Technical Report PRG-TR-6-96, Programming Research Group, Oxford University, 1996.