On The Semantic Intricacies of Conditioning

Friedrich Gretz, Nils Jansen, Benjamin Lucien Kaminski, Joost-Pieter Katoen and Federico Olmedo
RWTH Aachen University, Aachen, Germany

***Abstract*—We discuss semantic intricacies of conditioning, a main feature in probabilistic programming, and propose how to deal with these issues in an operational way and in a weakest pre-condition semantics. This includes the interplay between conditioning and possible non-termination as well as between conditioning and non-determinism. We propose a program transformation that eliminates conditioning from programs at the expense of introducing loops.**

Introduction: Many probabilistic programming languages exist including *Probabilistic C*, *Rely*, *Figaro*, *ProbLog*, *Tabular*, and *R2* [1]. Probabilistic programs are normal-looking programs describing posterior probability distributions. They are sequential programs having two main features: (1) sampling from probability distributions, and (2) the ability to condition values of variables in a program. Conditioning allows for adding information about observed events into the program that may influence the posterior distribution. The semantics of probabilistic programs without conditioning is rather well–understood. Denotational [2], weakest (liberal) precondition (w(l)p) [3] and operational semantics [4] have, for instance, been provided. However, covering conditioning semantically imposes several challenging problems. We discuss these intricacies in the setting of a simple imperative language, namely a probabilistic variant of Dijkstra's guarded command language. Admittedly, this is not a language used nowadays in probabilistic programming, but due to its simplicity it can be considered as a "core" language in which the semantic intricacies of conditioning can be properly illustrated. Its main restriction is that it does not support continuous distributions. The problems discussed here do, however, also occur when considering such distributions. On the other hand, it contains a non-deterministic choice which is essential for considering probabilistic programs at different abstraction levels and for including multi-threading. We focus on conditioning as expressed by means of so-called observe statements [1], [5].

When to observe?: Consider the program snippet P

$$\{x \coloneqq 0\}\ [1/2]\ \{x \coloneqq 1\};\ \texttt{observe}\ (x{=}1)\ ,$$

which assigns zero to the variable x with probability $1/2$ (modeled by a probabilistic choice) while x is assigned one with the same likelihood, after which we condition to the outcome of x being one. The observe statement blocks all invalid runs violating its condition and renormalizes the probabilities of the remaining valid runs. The interpretation of the program is the expected outcome conditioned on the valid runs. For P, this yields the outcome one. Consider now:

$$\{x \coloneqq 0;\ \texttt{observe}\,(x{=}1)\}\ [1/2]\ \{x \coloneqq 1;\ \texttt{observe}\,(x{=}1)\}$$

The left branch of the probabilistic choice is infeasible. Is this program equivalent to P? We think they are. Setting an infeasible program into context thus can render it feasible.

The interference with non–termination: Consider

$$x \coloneqq 2 \qquad \text{and} \qquad \{x \coloneqq 2\}\ [1/2]\ \{\texttt{abort}\}\ .$$

Both programs assign two to x, but the right one aborts with probability $1/2$. Should these two programs be considered equivalent or not? Most semantics do not distinguish them, as they assume programs to almost-surely terminate. This may make sense for programs in certain application domains. But can we really require a probabilistic programmer to write only terminating programs? Sure, one can prevent a programmer from writing programs containing abort statements, but one cannot avoid divergence—loops may easily not terminate. We take the position that termination is not an issue that is left to the programmer.[1] Semantics thus has to treat divergence. In our proposal, both programs above are thus distinguished.

Observations inside loops: Consider the two programs:

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

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 of the right program is undefined and cannot be measured. Note that programs with (probabilistic) assertions must be loop–free to avoid similar problems [6]; other approaches insist on the absence of diverging loops [7].

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

[1] Almost-sure termination of probabilistic programs is "more undecidable" than termination for ordinary programs; leaving this to the full responsibility of a programmer is rather demanding.

programmer" to write only terminating programs. Therefore, a semantics for conditioning has to take the possibility of conditioning to zero–probability events into account. We propose such a semantics; it distinguishes the two loopy programs above.

The interference with non–determinism: The following example blurs the situation even further. Consider the program:

```
1: repeat
2:    {$x := 1$} [$1/2$] {$x := 0$};
3:    {$x := 1$} $\Box$ {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 not clear: If in line 3, the oracle to resolve the non–determinism always chooses $x := 1$, then this results in certain non–termination. If, on the other hand, the oracle 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 oracle 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 oracle should prefer the latter.

Our proposal: We provide a semantics of a probabilistic variant of Dijkstra's guarded command language. This includes probabilistic and non-deterministic choice, abortion and conditioning by means of observe statements. (Our semantics can be easily adapted to programming languages that support sampling from arbitrary discrete distribution functions.) Given that this language is rather basic our semantics can act as a backbone for full-fledged imperative probabilistic programming languages with conditioning. The operational model is based on Markov decision processes. In absence of non–determinism, this reduces to Markov chains. The crux of our semantics is to distinguish the violation of observe statements and possible divergence. The probability that a given outcome is obtained is normalized with respect to the probability that all observe statements are fulfilled. The latter probability includes the possible diverging runs.

wp-Reasoning: We provide an alternative semantics by generalizing the weakest pre-expectation semantics by McIver and Morgan [3]. The soundness of the semantics is investigated in two directions. The wp-semantics is semantically equivalent to the operational model in the sense that (roughly speaking) weakest pre-expectations correspond to conditional expected rewards. Moreover, this semantics is backward compatible with McIver and Morgan's semantics for programs without conditioning[2]; this *does not* apply to alternative approaches such as R2 [1]. To be more precise, this latter soundness result only holds for programs without non-determinism. In fact, it turns out that combining both non–determinism and conditioning cannot be treated in the wp-semantics. The problem is that the resolution of non-deterministic choices needs to depend on the context of these choices, rendering a definition by structural induction on programs—as is *the* standard approach for defining wp-semantics—impossible.

Conditioning is syntactic sugar: Any program with conditioning can be transformed into an equivalent program without conditioning. This observation is not new, but due to the treatment of possible divergence, in our setting the transformation to eliminate conditioning is different and more involved. Let P be program with observations. We transform this program by repeatedly sampling executions from P until the sampled execution satisfies all its observations. This comes at the expense of introducing a loop. Details will be presented at the workshop.

Discussion: We welcome a thorough discussion about the semantics of conditioning. Even with just discrete probability functions, there are various issues, as made clear in this note. By presenting two consistent semantic views, we hope to provide a good basis for discussion. Issues for discussion are, amongst others:

- Is non-determinism an issue of interest?
- Should conditioning be allowed inside loops?
- How to treat non-terminating programs?
- Can we come up with rules of thumb to avoid some problematic cases?

Acknowledgment

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

References

[1] A. V. Nori, C. Hur, S. K. Rajamani, and S. Samuel, "R2: an efficient MCMC sampler for probabilistic programs," in *Proc. of AAAI*, 2014, pp. 2476–2482.

[2] D. Kozen, "Semantics of probabilistic programs," *J. Comput. Syst. Sci.*, vol. 22, no. 3, pp. 328–350, 1981.

[3] A. McIver and C. Morgan, *Abstraction, Refinement And Proof For Probabilistic Systems*. Springer, 2004.

[4] F. Gretz, J.-P. Katoen, and A. McIver, "Operational versus weakest pre-expectation semantics for the probabilistic guarded command language," *Perform. Eval.*, vol. 73, pp. 110–132, 2014.

[5] C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel, "Slicing probabilistic programs," in *Proc. of PLDI*. ACM Press, 2014, pp. 133–144.

[6] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," in *Proc. of PLDI*. ACM, 2014, p. 14.

[7] A. Chakarov and S. Sankaranarayanan, "Expectation invariants for probabilistic program loops as fixed points," in *Proc. of SAS*, ser. LNCS, vol. 8723. Springer, 2014, pp. 85–100.

[2]Given that their semantics is backward compatible with Dijkstra's guarded command language, we consider this as a desirable property.