ENS Rennes - University of Rennes 1

Internship report: Master Degree, first year

Possibility Distribution Semantics for Probabilistic Programs with Nondeterminism

Intern: Joshua Peignier
Supervisors: Benjamin Kaminski, Christoph Matheja
Team: Software Modeling and Verification
Institute: RWTH Aachen University (Aachen, Germany)
Dates: 15/05/2017 - 11/08/2017

Abstract. In contrast to ordinary programs, probabilistic programs compute a probability distribution of output states for each given input state. One benefit of adding randomness into programs is that computationally hard problems, such as matrix multiplication or leader election protocols, can be solved (on average) more efficiently. A frequently used design concept to model unknown or overly complex probability distributions is nondeterministic choice. However, having probabilistic and nondeterministic choice within the same program (or even within loops) leads to subtle semantical intricacies. The goal of this report is to capture the semantics of both concepts uniformly using possibility theory. This would allow to simplify existing weakest-precondition style calculi for reasoning about probabilistic and nondeterministic programs.

Keywords: Fuzzy imperative languages ; Fuzzy relations ; Possibility theory ; Weakest-precondition

1 Introduction

Probabilistic programs are a particular class of programs, in which probabilistic choices are involved. For instance, the simple program $\{x := 2\}[\frac{1}{3}]\{x := 5\}$ is a program that has a probability $\frac{1}{3}$ of setting the program variable x to 2, and a probability $\frac{2}{3}$ of setting x to 5; it is therefore a probabilistic program. What makes this kind of program different from classical deterministic programs is that executing such a program with always the same input will not always result in the same output. Though they seem more complex than deterministic programs, probabilistic programs are useful in many ways. For example, it is simpler to solve some problems with probabilistic programs than with deterministic ones, as we can build probabilistic programs which have a better average complexity than deterministic programs (sometimes at the cost of a small probability of error). Classical examples of such programs are the quicksort algorithm, Freivalds' matrix multiplication, and some primality tests, such as the Miller-Rabin test and the Solovay-Strassen test (for positive numbers), and the Berlekamp test (for polynomials). There also exists probabilistic algorithms to solve the leader election problem. In this whole report, programs containing such choices will be called *probabilistic programs*.
Besides probabilistic choice, there exists a frequently used concept in nondeterministic programs, called nondeterministic choice. One example is the following program: $\{x := 2\}[\!]\{x := 5\}$. In this example, the program will set x nondeterministically to 2, or to 5, but we cannot assign a probability to any of these choices. More precisely, it makes no sense to assign them a probability. The nondeterministic choice is used in some programs where the outcomes of a choice are known, but making the mecanism behind the choices is either unknown, or uses overly complex probability distributions. **[TO DO: Find a source]** This instruction was first introduced by Dijkstra [3] in the GCL language, and is used, for instance, in the model-checker Spin, to model the unpredictable behavior of a program where several choices are possible. In this whole report, programs containing such choices will be called *nondeterministic programs*. Programs con-

taining both types will be called *probabilistic nondeterministic programs*, whereas programs containing none of them are *deterministic programs.*

However, major problems arise with the use of nondeterministic choice, including: defining proper semantics for this instruction, and the complication arising from combination of probabilistic and nondeterministic choices in the same program. In this report, we will use the following probabilistic nondeterministic program as a guildeline example:

$$P_0\colon \{x := 2 [\!] x := 5\}[p]\{x := 7\} (\text{with } p \in]0, 1[)$$

This program intuitively seems to set the program variable x to 7 with a probability $1 - p$, and with a probability p, to set nondeterministically x either to 2 or to 5. How can we semantically describe this program ?

When introducing the GCL language [3] (whose semantics allow it to generate nondeterministic programs, but not probabilistic programs), Dijkstra also introduced *predicate transformer semantics*, and particularly the calculus of weakest precondition. This concept relies on Hoare triples: A *Hoare triple* is a triple of the form $\{\psi\}P\{\varphi\}$, where P is a program, and where ψ and φ are predicates respectively called *precondition* and *postcondition.* A Hoare triple is said to be *valid* if and only if, each time that an initial state satisfies ψ, the final state obtained after executing P satisfies φ. Thus, Dijkstra introduced the weakest preconditions: if P is a program and φ is a postcondition, then $wp[P](\varphi)$ is the weakest (in the sense of "satisfied by the most states") precondition ψ making the Hoare triple $\{\psi\}P\{\varphi\}$ valid. The predicate transformer computation is a set of rules which, knowing a program P and a postcondition ψ, allow the computation of the associated weakest precondition $wp[P](\psi)$. One can therefore see P as a predicate transformer, transforming the predicate φ into the predicate $wp[P](\varphi)$.
For instance, consider the following program:

$$P_1\colon x := -y + 1$$

If we denote by $[x \geq 5]$ the predicate satisfied by all states where x is greater or equal to 5, then we get that $wp[P_1]([x \geq 5]) = [y \leq -4]$. (The formal method to get this result is presented in [3] and will be reminded in Section 4)

This method was later further extended in [4] to probabilistic nondeterministic programs (generated by the pGCL language, which is an extended version of GCL including probabilistic choice), by changing the semantics a little, such that the weakest precondition $wp[P](\varphi)$ assigns to each state σ the probability that the execution of the program P in σ terminates in a state satisfying φ. By applying semantics defined in [4] to the program P_0 (note that, the way the program is written, the final state should not depend on the initial state in this example), we get that the probability, for each initial state, that the the final state satisfies $[x = 7]$ is $1 - p$, which is intuitively expected (formally, this means that $wp[P_0]([x = 7])$ assigns to each state the constant $1-p$). Now, consider the subprogram P_2 of P_0:

$$P_2\colon x := 2 [\!] x := 5$$

We also get that the probability, for each state, that the final state satisfies $[x = 2]$ is 0 (and symmetrically for $[x = 5]$), but the probability for each state that the final state satisfies $[x = 2 \text{ OR } x = 5]$ is p. Theses results are not unexpected, as the execution of P_0 has a probability p of executing the P_2, in which we know that x is set either to 2 or to 5. However, as it is not possible to assign probabilities to the nondeterministic choices, conventions must be defined in order to compute weakest preconditions. In [4], the authors chose a convention such that, in this case, the probability of getting to a state satisfying $[x = 2]$ is 0, as there is no guarantee that the corresponding instruction is executed and no way to assign a probability.
However, this result seems to lack information, as there is a possibility that the final state satisfies $[x = 2]$, but the corresponding probability is 0, and in order to know that this program can achieve a state where $[x = 2]$, we have to compute the weakest precondition of another postcondition as $[x = 2]$ (in this case, $[x := 2 \text{ OR } x = 5]$).
Moreover, the convention chosen in [4] (and in [10]) is not universal, as other authors [2,9] chose conventions with which we get other results (notably the probability of achieving a state where $[x = 2]$ holds becomes p (and symmetrically for $[x = 5]$), and the probability of achieving a state where $[x = 2 \text{ OR } x = 5]$ holds remains p).

Some authors [2, 9, 10] tried a different approach to weakest precondition calculus, based on possibility theory rather on probabilities, in order to define different semantics and possibly give a better comprehension of how nondeterministic programs behave. Possibility theory is an alternative approach to probabilities to model uncertainty (detailed in [1,8]). It is well-known that if Pr is a probability measure over Ω for the σ-algebra $\mathcal{A}$, then it satisfies the following axioms:

1. $\forall U \in \mathcal{A}$, $Pr(U) \in [0, 1]$
2. $Pr(\emptyset) = 0$
3. $Pr(\Omega) = 1$
4. $\forall U, V \in \mathcal{A}$, $Pr(U \cup V) = Pr(U) + Pr(V) - Pr(U \cap V)$

On the contrary, a possibility measure over Ω for the σ-algebra $\mathcal{A}$ is a mapping Π satisfying the following axioms (adapted from [1,8]):

1. $\forall U \in \mathcal{A}$, $\Pi(U) \in [0, 1]$
2. $\Pi(\emptyset) = 0$
3. $\Pi(\Omega) = 1$
4. $\forall U, V \in \mathcal{A}$, $\Pi(U \cup V) = \mathrm{M}ax(\Pi(U), \Pi(V))$

This type of measure, however, does not give as many information as probabilities. Indeed, when $Pr(U) = 1$, we know that an event of U happens almost surely, and when $Pr(U) = 0$, we know that any event of U almost never happen. However, when $\Pi(U) = 0$, we know that all events of U are impossible (thus cannot happen), but when $\Pi(U) = 1$, it just means that at least one event of U is totally possible, but we cannot tell that it will almost surely happen. As the name says, a possibility measure only indicates how possible an event is.

Jumping from probabilities to possibilities also feels like a loss of information. But in fact, possibility theory has already been studied [2,9,10] in order to define predicate transformer semantics for fuzzy imperative languages (defined in [5,6], they are languages with more instructions based on possibilities, and are at least as expressive as GCL). However, these languages do not include the probabilistic choice.

In this report, our goal is to give our own semantics of probabilistic nondeterministic programs, and more precisely an expected value semantics (which can be derived from predicate transformer semantics), where we consider expected values of fuzzy random variables (first defined in [7], and then simplified in [8]). It is a way to extend the ideas presented in [2, 9, 10] using possibility theory to the pGCL language presented in [4].
[TO DO: Insert a quick organisation of the report here, reminding what to find in each section]

2 Related Work

3 Preliminaries

4 Contribution

In this section, we consider programs over the language presented in Section 4, and our goal is to define clearer semantics for these programs as the semantics we presented before. Recall that we showed in Section 4 that weakest-precondition calculus could be used in order to compute expected values of random variables. Our main idea here is to combine the concept of *Fuzzy random variable* (introduced in [7], and simplified in [8]) with the computation of expected values.
Let Σ be the set of states. We consider that the variables in our programs can only take real positive values.[1] We adapted the following definition from [7].

Definition 1. (Fuzzy Random Variable)
A Fuzzy Random Variable (short: FRV) is a mapping X from the set of states Σ to the powerset $P(\mathbb{R}^{\infty}_{\geq 0})$. This means that, if X is an FRV, then for all $\sigma \in \Sigma$,

[1] This is due to constraints explained further. But note that we can implement an arbitrary real variable by two variables, one for its absolute value, and one for its sign.

$X(\sigma)$ is a subset of $\mathbb{R}^{\infty}_{\geq 0}$. In fact, it is a set of possible values for X in the state σ.

Example 1. In the following, for each program variable x, we denote by $\underline{x}$ the FRV such that, for all $\sigma \in \Sigma$, $\underline{x}(\sigma) = \{x_\sigma\}$, meaning that $\underline{x}$ associates to each state σ the singleton containing the value x_σ, which is the value of the program variable x in the set σ. (Recall that each state is characterized uniquely by a tuple containing the values of each variable in a given order.). Therefore, we can say that the FRV $\underline{x}$ models the program variable x.

A fuzzy random variable X over a program can model several concepts. For instance, it can be used to model the runtime of a program executed in a state σ (which is why we chose to include ∞); in this example, $X(\sigma)$ would give all possible runtimes for the execution of one program starting from σ. It can also model the value of a program variable (or any function over the program variables) after the execution of a program.

Let $F = \{f \mid f : \sigma \to P(\mathbb{R}^{\infty}_{\geq 0})\}$ be the set of fuzzy random variables, and $Progs$ be the set of pGCL programs. We define the following application:

Definition 2. *$ev : (Progs) \to (F \to F)$ is an application mapping each program S to its FRV transformer semantics. This means that, for all $S \in Progs$, $ev[S]$ transforms a FRV $X \in F$ in another FRV $ev[S](X)$.*

Concretely, if X is a FRV (i.e. an application mapping each state σ to a subset $X(\sigma)$ of $\mathbb{R}^{\infty}_{\geq 0}$), then $ev[S](X)(\sigma)$ is the set of possible expected values of the FRV X after executing S in the state σ.
For instance, if we consider a program variable x, then the FRV $\underline{x}$ maps each state σ to the set $\{x_\sigma\}$. Our goal is that the FRV $\underline{x}$ is transformed by $ev[S]$ into the FRV $ev[S](\underline{x})$, which maps each σ to the set $ev[S](\underline{x})(\sigma)$ of possible values of the variable x after executing S in the state σ. Therefore, we can say that $ev[S](\underline{x})(\sigma)$ is the *expected value* (or rather *set of possible expected values*) of the FRV $\underline{x}$ after executing S in the state σ; or, with other words, the set of possible expected values of the program variable x after executing S in σ.

We give rules in Table 5 of a predicate-transformer-style calculus for the computation of expected values.

Consider the guideline example P: $\{x := 2 \| x := 5\}[p]\{x := 7\}$. We want to compute the expected value of x after the execution of P. Thus, we have to determine $ev[P](\underline{x})$,which will map each state σ to the expected value of x after executing P in σ. It is a function of σ, but the form of P gives the intuition that it will be a constant function, not depending on σ.
With the rules presented in Table 5, we can see that this requires first to compute the functions $ev[x := 2 \| x := 5](\underline{x})$ and $ev[x := 7](\underline{x})$; besides, the former requires to compute $ev[x := 2](\underline{x})$, $ev[x := 5](\underline{x})$.

S	$ev[S](X)$
`skip`	X
$y := A$	$\lambda\sigma.X(\sigma[y/A])$
$S_1; S_2$	$ev[S_1](ev[S_2](X))$
$S_1[p]S_2$	$\lambda\sigma.\{t_1 p + t_2(1-p) \,\vert\, t_1 \in ev[S_1](X)(\sigma), t_2 \in ev[S_2](X)(\sigma)\}$
$S_1 \| S_2$	$\lambda\sigma.ev[S_1](X)(\sigma) \cup ev[S_2](X)(\sigma)$
`if` (b) `then` $\{S_1\}$ `else` $\{S_2\}$	$\lambda\sigma.\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \,\vert\, t_1 \in ev[S_1](X)(\sigma), t_2 \in ev[S_2](X)(\sigma)\}$
`while` (b) `do` $\{S\}$	$\mathrm{lfp}\ (\lambda Y.(\lambda\sigma.\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \,\vert\, t_1 \in ev[S](Y)(\sigma), t_2 \in X(\sigma)\}))$

Table 1. Rules for defining the FRV transformer ev

We get that:

$$ev[x := 2](\underline{x}) = \lambda\sigma.\underline{x}(\sigma[x/2]) = \lambda\sigma.\{x_{\sigma[x/2]}\} = \lambda\sigma.\{2\}$$

Indeed, the value of x in $\sigma[x/2]$ is necessarily 2. Recall that $\sigma[x/2]$ is the state obtained after setting x to 2 in the state σ. This means that the expected value of x after executing $x := 2$ in any state σ can only be 2. By the same computation, we get $ev[x := 5](\underline{x}) = \lambda\sigma.\{5\}$ and $ev[x := 7](\underline{x}) = \lambda\sigma.\{7\}$.

Now, we can compute that:

$$\begin{aligned} ev[x := 2 \| x := 5](\underline{x}) &= \lambda\sigma.ev[x := 2](\underline{x})(\sigma) \cup ev[x := 5](\underline{x})(\sigma) \\ &= \lambda\sigma.\{2\} \cup \{5\} = \lambda\sigma.\{2, 5\} \end{aligned}$$

This means that the possible expected values of x after executing $x := 2 \| x := 5$ in any state σ are 2 and 5.

Finally,

$$\begin{aligned} ev[P](\underline{x}) &= \lambda\sigma.\{t_1 p + t_2(1-p) \,\vert\, t_1 \in ev[x := 2 \| x := 5](\underline{x})(\sigma), t_2 \in ev[x := 7](\underline{x})(\sigma)\} \\ &= \lambda\sigma.\{t_1 p + t_2(1-p) \,\vert\, t_1 \in \{2, 5\}, t_2 \in \{7\}\} \\ &= \lambda\sigma.\{2p + 7(1-p), 5p + 7(1-p)\} \end{aligned}$$

5 Conclusion

[TO DO: Explain that, for a program S, by considering each of its variables x and computing $ev[S](\underline{x})$, we are able to fully characterize the program S and get an understanding of what it does. Explain that it is clearer than what was done before.]

References

1. Parul Agarwal and H.S. Nayal. *Possibility Theory versus Probability Theory in Fuzzy Measure Theory.*
2. Yixiang Chen and Hengyang Wu. *Domain Semantics of Possibility Computations.*
3. E.W. Dijkstra. *A Discipline of Programming.*
4. Annabelle McIver and Carroll Morgan. *Abstration, Refinement and Proof for Probabilistic Systems.*
5. Rafael Morales-Bueno, Ricardo Conejo, José Luis Pérez de la Cruz, and Buenaventura Clares. *An elementary fuzzy programming language.*
6. Rafael Morales-Bueno, José Luis Pérez de la Cruz, Ricardo Conejo, and Buenaventura Clares. *A family of fuzzy programming languages.*
7. Madan L. Puri and Dan A. Ralescu. *Fuzzy Random Variables.*
8. Arnold F. Shapiro. *Fuzzy Random Variables.*
9. Hengyang Wu and Yixiang Chen. *Semantics of Non–Deterministic Possibility Computation.*
10. Hengyang Wu and Yixiang Chen. *The Semantics of wlp and slp of Fuzzy Imperative Programming Languages.*