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 programs, in which probabilistic choices are involved. For instance, the simple program $\{x := 2\}[\frac{1}{3}]\{x := 5\}$ simulates flipping a biased coin (with one outcome having a probability of $\frac{1}{3}$, and the other one $\frac{2}{3}$). One easily notices that, contrary to classical deterministic programs, executing such a program contaning probabilistic choices with the same input will not always result in the same output. In this whole report, programs containing probabilistic choices will be called *probabilistic programs*. 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 [6], Freivalds' matrix multiplication [5], and certain primality tests, such as the Miller-Rabin test [11] and the Solovay-Strassen test [14] (for positive numbers), and the Berlekamp test (for polynomials) [2]. There also exists probabilistic algorithms to solve the leader election problem (see [12]).
Apart from probabilistic choice, there exists a different kind of choice, frequently used, but also making programs nondeterministic, called nondeterministic choice. One example is the following program: $\{x := 2\}\square\{x := 5\}$. 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. In this whole report, programs containing such choices will be called *nondeterministic programs*. Such programs share with probabilistic programs their ability to return different outputs for one given input. The nondeterministic choice was first introduced by Dijkstra in the GCL language [4] (which did not include the probabilistic choice), and is used, for instance, in the model-checker Spin, to model the unpredictable behavior of a program where several choices are possible. It is also used in programs where the outcomes of a choice are known, but the mecanism making the choices is either unknown, or

uses overly complex probability distributions. **[TO DO: Find a source]** This kind of choice is also a common concept in concurrency theory.

However, one major problem arises when combining the probabilistic and nondeterministic choices in the same program. Programs containing both types are called *fully probabilistic programs*, whereas programs containing none of them are *deterministic programs*. In this report, we will use the following fully probabilistic program as a running example:

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

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

A frequently used semantics, which turned out to be well-suited to program verification, is Dijkstra's predicate transformer semantics [4], which consists in viewing nondeterministic programs (without probabilistic choice) as predicate transformers. More precisely, if P is a program and φ a predicate over the set of states (i.e. an object modeling a property, that each state will or not satisfy), then the predicate transformer semantics describes P as an object $wp[P]$, transforming the predicate φ (called *postcondition* in this context) into another predicate called the *weakest precondition of φ for the program P*, denoted by $wp[P](\varphi)$. As the name explicitely says, this new predicate is the weakest (in the sense of "least restrictive", or "satisfied by the most states") predicate ψ verifying the following property: if P is executed in a state satisfying ψ, then P is guaranteed to terminate in a state satisfying φ.

For instance, consider the following sequential program:

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

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 $wp[P_1]([x \geq 5]) = wp[x := -y]([x \geq 4]) = [y \leq -4]$. (The formal method to get this result is presented in [4] and will be reminded in Section **??**)

Dijkstra's predicate transformer semantics was extended by McIver and Morgan [7] to fully probabilistic programs by changing the semantics, 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, instead of assigning only 0 or 1 values meaning that the execution of P will certainly or certainly not terminate in a state satisfying the postcondition. Note that applying the semantics of McIver and Morgan to nondeterministic programs, without any probabilistic choice, the results are the same as in Dijkstra's semantics.
By applying the semantics of McIver and Morgan 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:

$$P_2\text{: } \{x := 2\}\Box\{x := 5\}$$

Recall that, the semantics of McIver and Morgan, $wp[P](\varphi)$ assigns to each state the probability that the execution of P in this state terminates in a state satisfying φ (it is therefore a function from the set of states to $[0, 1]$). As it is not possible to assign probabilities to the nondeterministic choices, conventions must be defined in order to compute weakest precondition when the program contains nondeterministic choices:

1. One possible choice is to consider that the outcome which is realized is the least desired outcome (for the given postcondition). In this case, the nondeterministic choice is called *demonic choice* [7]. In this example, when computing $wp[P_2]([x = 2])$, the least desired outcome is $x := 5$. It is therefore considered as the chosen outcome, and after its execution, $[x = 2]$ cannot be satisfied. Hence, $wp[P_2]([x = 2])$ assigns to each state the constant 0, and thus, $wp[P_0]([x = 2])$ also assigns to each state the constant 0. We get the same result with $[x = 5]$. However, when computing $wp[P_2]([x = 2 \text{ OR } x = 5])$, we get the constant function 1 (thus, $wp[P_0]([x = 2 \text{ OR } x = 5])$ is the constant function p), because in both possible outcomes of the choice, the postcondition will be satisfied. But this result is counterintuitive, as it violates the modularity law of probabilities (detailed below).
2. An alternative possibility is the *angelic choice* (named in [7]), which is intuitively the opposite of the demonic choice. In this case, the outcome which is realized is the most desired outcome. With this choice, we get that $wp[P_0]([x = 2])$ is the constant function p, and symetrically for $wp[P_0]([x = 5])$. But $wp[P_0]([x = 2 \text{ OR } x = 5])$ is also the constant function p. These results also violates the modularity law of probabilities.

In both cases, counterintuitive results are obtained. Moreover, in the demonic choice case, in order to realise that there exists executions where $[x = 2]$ can be achieved, we have to compute the weakest precondition of another postcondition (namely, $[x = 2 \text{ OR } x = 5]$), because $wp[P_0]([x = 2])$ is the constant function 0. The latter result is therefore missing information. Therefore, in order to obtain more intuitive, or more understandable results, we can either try to define a new convention for the nondeterministic choice, or define new semantics not based on probabilities.

Some authors [3, 15, 16] 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, 13]). It is well-known that Pr is a probability measure over Ω for the σ-algebra $\mathcal{A}$ if it is a mapping from $\mathcal{A}$ to $[0,1]$ satisfying the following axioms:

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

On the contrary, Π is a possibility measure over Ω for the σ-algebra $\mathcal{A}$ if it is a mapping from $\mathcal{A}$ to $[0,1]$ satisfying the following axioms (see [1,13]):

1. $\Pi(\emptyset) = 0$
2. $\Pi(\Omega) = 1$
3. $\forall U, V \in \mathcal{A}$, $\Pi(U \cup V) = \mathrm{Max}(\Pi(U), \Pi(V))$ (max-modularity law)

This type of measure, however, does not give as much 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, but not how often it is realized. This behavior bears a strong resemblance to the nondeterministic choice, whose outcomes are known to be possible, but do not have any probability assigned.

Using possibilities instead of probabilities first seems not to be an interesting choice, since possibilities normally do not bear as much information as probabilities, as we said earlier. But in fact, possibility theory has already been studied [3,15,16] in order to define predicate transformer semantics for fuzzy imperative languages (defined in [8,9], they are languages with more instructions based on possibilities, and are at least as expressive as GCL), and the authors obtained interesting semantics in which each possible outcome is taken in account (even for GCL programs). 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 [10], and then simplified in [13]). It is a way to extend the ideas presented in [3,15,16] using possibility theory to the pGCL language presented in [7].

[TO DO: Insert the organisation of the paper when we are sure of the plan.]

[TO DO: Make the notations uniform in the whole report: use S (or C) in the rules, in the definitions, etc... And use P when refering explicitely to a program defined as an example]

2 Related Work

3 State of the art

In this section, we present the pGCL language and the vision of McIver and Morgan over weakest precondition calculus [7], we quickly explain (see [7]) how this type of calculus can be exploited to compute expected values of random variables, and finally quickly present the current uses of possibility theory as a base for predicate transformer semantics in fuzzy languages.

3.1 The pGCL language

The pGCL language is an extension of Dijkstra's GCL. It was first introduced in [7], and includes a new instruction that GCL did not include: the probabilistic choice. Therefore, the pGCL language allows the generation of fully probabilistic programs. It is defined by the following grammar[1][2]:

$$\begin{aligned} S ::= {} & \texttt{skip} \mid x := A \mid S; S \mid \{S\}\Box\{S\} \mid \{S\}[p]\{S\} \mid \\ & \texttt{if } (b) \texttt{ then } \{S\} \texttt{ else } \{S\} \mid \texttt{ while } (b) \texttt{ do } \{S\} \end{aligned}$$

This language is sufficient to describe any fully probabilistic program. The `if...then...else` and `while` structure are the usual control structures encountered in many languages. The `skip` instruction corresponds to an empty instruction, where nothing is done (it is included in the language so that we do not have to create a specific `if...then...` instruction for the case where no `else` is needed.) $x := A$ is an assignment statement, where the program variable x is set to the value of the expression A in the current state. $S_1; S_2$ is a sequence assignment, modeling the execution of S_1 followed by S_2. We take interest primarly in the two remaining instructions.
The $\{S_1\}[p]\{S_2\}$ (where $p \in]0, 1[$) instruction models the fact that there is a probability p that S_1 is executed, and a probability $1 - p$ that S_2 is executed. Finally, $\{S_1\}\Box\{S_2\}$ is a statement that will nondeterministically execute either S_1 or S_2. The difference with the probabilistic choice is that no probability can be assigned to S_1 or S_2 because it makes no sense. This means that if $\{S_1\}\Box\{S_2\}$ is executed a large number of times, no clear pattern should appear, whereas the execution of $S_1[p]S_2$ a large number of times will tend to have a proportion p of cases where S_1 is executed, and a proportion $1-p$ of cases where S_2 is executed.

[1] The original GCL included an `abort` instruction corresponding to an error case. We chose, not to include it, as it can be represented by `while` (`true`) `do` {`skip`}

[2] The rules for nondeterminism were defined by Dijkstra in guarded commands [4], but McIver and Morgan used an equivalent grammar and added probabilistic choice to get the grammar presented here.

Note that this grammar only has choice instructions with two outcomes; but combination of instructions make possible the realisation of choices with any finite number (and even a countable number when using loops) of outcomes.

3.2 Predicate transformer semantics and weakest precondition calculus

Let Σ be the set of program states. Each state is uniquely characterized by the values of each variable in the state.

Definition 1. (Predicate and binary predicate)
A predicate *φ is a function from Σ to $[0,1]$. One says that each state σ has a probability $\varphi(\sigma)$ of satisfying φ.*
When φ takes only the values 0 and 1, it is called a binary predicate. *In this case, one abusively says that σ satisfy φ if and only if $\varphi(\sigma) = 1$.*

Let P be a pGCL program and let φ be a binary predicate. What is the probability that the execution of P in a state σ is guaranteed to terminate in a state satisfying the postcondition φ ? McIver and Morgan extended Dijkstra's vision of weakest precondition calculus, such that $wp[P](\varphi)(\sigma)$ is the answer, i.e. $wp[P](\varphi)$ is a predicate, and for each $\sigma \in \Sigma$, $wp[P](\varphi)(\sigma)$ is the probability that σ satisfies $wp[P](\varphi)$, i.e. the probability that the execution of P in σ terminates in a state satisfying ψ.
Therefore, P is described by the semantics of McIver and Morgan as an object $wp[P]$ transforming the predicate φ into the predicate $wp[P](\varphi)$. The rules[3] for computing $wp[P](\varphi)$ are given in Table 1.

The state $\sigma[x/A]$ is the state obtained when syntactically replacing the variable x by the value of A in σ; $[\![b : true]\!]$ is a binary predicate, evaluating to 1 in σ if and only if the boolean variable b is true in σ (and symmetrically for $[\![b : false]\!]$). Finally, the form lfp $(\lambda X.f(X))$ denotes the least fixed point of the application f (where the X are predicates). The value of the least fixed-point can be computed with Kleene's fixed-point theorem (under certain hypotheses, verified here; more details in Appendices 6.1 and 6.2).

Note that the rules allow the computation of $wp[P](\varphi)$ for all arbitrary predicates φ, but in the practice, we only want to compute $wp[P](\varphi)$ where φ is a binary predicate, because when checking programs, we want to verify whether the final states satisfy a property or not, and not to verify whether a final states satisfies a property with a certain probability. In the only cases where φ is not a binary predicate, φ will result from the computation of another weakest precondition $wp[P'](\psi)$

[3] These rules correspond to the case where the nondeterministic choice is considered to be the demonic choice, as Dijkstra first intended.

P	$wp[P](\varphi)$
`skip`	φ
$x := A$	$\lambda\sigma.\varphi(\sigma[x/A])$
$P_1; P_2$	$wp[P_1](wp[P_2](\varphi))$
$\{P_1\}\Box\{P_2\}$	$\lambda\sigma.\text{Min}(wp[P_1](\varphi)(\sigma), wp[P_2](\varphi)(\sigma))$
$\{P_1\}[p]\{P_2\}$	$\lambda\sigma.p \cdot wp[P_1](\varphi)(\sigma) + (1-p)\cdot wp[P_2](\varphi)(\sigma)$
`if` (b) `then` $\{P_1\}$ `else` $\{P_2\}$	$\lambda\sigma.[\![b : true]\!](\sigma)\cdot wp[P_1](\varphi)(\sigma) + [\![b : false]\!](\sigma)\cdot wp[P_2](\varphi)(\sigma)$
`while` (b) `do` $\{P\}$	lfp $(\lambda X.(\lambda\sigma.[\![b : true]\!](\sigma)\cdot wp[P_1](X)(\sigma) + [\![b : false]\!](\sigma)\cdot\varphi(\sigma)))$

Table 1: Rules for defining the predicate transformer wp

Applying these rules to the example program P_1 from Section 1 with the postcondition $[x \geq 5]$ (binary predicate evaluating to 1 in state σ if and only if $x \geq 5$ holds in σ), we get (the details with λ-calculus are left to the reader):

$$wp[x := -y; x := x+1]([x \geq 5]) = wp[x := -y](wp[x := x+1]([x \geq 5]))$$

The assignment rules determines that $wp[x := x+1]([x \geq 5]) = \lambda\sigma.[x \geq 5](\sigma[x/x+1])$. It turns out that:

$$wp[x := x+1]([x \geq 5]) = [x \geq 4]$$

Proof. Recall that $[x \geq 5](\sigma) = 1$ if $x \geq 5$ in σ and 0 else. Therefore $[x \geq 5](\sigma[x/x+1]) = 1$ if $x \geq 5$ in $\sigma[x/x+1]$ and 0 else. This is equivalent to: $[x \geq 5](\sigma) = 1$ if $x+1 \geq 5$ in σ and 0 else. Therefore, $[x \geq 5](\sigma[x/x+1]) = [x+1 \geq 5](\sigma)$, and $wp[x := x+1]([x \geq 5]) = [x+1 \geq 5] = [x \geq 4]$.

We resume the computation:

$$wp[x := -y; x := x+1]([x \geq 5]) = wp[x := -y](wp[x := x+1]([x \geq 5]))$$
$$= wp[x := -y]([x+1 \geq 5]) = \; wp[x := -y]([x \geq 4]) = [-y \geq 4] = [y \leq -4]$$

One can also apply these rules to the program P_0 and get the results presented in Section 1. The calculation is done in Appendix 6.3. **[TO DO: Maybe move this calculation here if there is enough place. It should be more logical, since it is our running example.]**

3.3 Computation of expected values

[TO DO: It is the same as computing the expected value of a random variable whose outcomes are known, and are (for each state) values of $\mathbb{R}^{\infty}_{\geq 0}$. Explain that some results, as shown in the intro, seem counterintuitive.]

In the practice, weakest precondition calculus can be used to anticipate the value of certain functions. Recall that with the simple program P_1 given in Section 1, we proved that $wp[P_1]([x \geq 5]) = [y \leq -4]$. As P_1 is deterministic, we get that, if σ satisfies $[y \leq -4]$, then the state obtained after executing P_1 in σ satisfies $[x \geq 5]$; and if σ does not satisfy $[y \leq -4]$, then the state obtained after executing P in σ does not satisfy $[x \geq 5]$. We can say that the function $[y \leq -4]$ anticipates the value of $[x \geq 5]$ after the execution of P_1.

[TO DO: Write references for what follows] It is in fact possible, in the deterministic case, to anticipate values of other functions, such as the function associating to σ the value of x^2, or the value of $|\sin(y)|$ in σ. However, due to certain constraints in order to ensure the existence of least fixed-points and make the computation possible, we must restrict ourselves to functions from σ to $\mathbb{R}^{\infty}_{\geq 0}$. [4] That restriction is not a problem in practice, as one can always decompose an arbitrary function as two functions, namely, its absolute value and its sign.

This computation can still be extended to nondeterministic programs (by considering that, in the nondeterministic choice, the anticipated value is the least one), and to fully probabilistic programs, by taking the ponderated mean between the two possible values. The rules of computation are rigorously the same as in Table 1, but are applied to functions from Σ to $\Rightarrow$.

Example 1. Consider the program P_1 from Section 1, and the function $\sigma \to x^2$ (where x is replaced by its value in σ). Its value after the execution of P_1 can be computed with:

$$wp[x := -y; x := x+1](x^2) = wp[x := -y](wp[x := x+1](x^2))$$
$$= wp[x := -y]((x+1)^2) == (1-y)^2$$

[TO DO: Apply to P_0 and underline the problem of counterintuitive results]

3.4 Chinese work

[TO DO: Obviously change this title when you have a better idea.]

[4] With this restriction, a complete partial order can be defined on $\{f \mid f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\}$ and Kleene's fixed point theorem can be applied. See Appendices 6.1 and 6.2

4 Contribution

In this section, we consider programs over the language presented in Section **??**, and our goal is to define clearer semantics for these programs as the semantics we presented before. Recall that we showed in Section **??** 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 [10], and simplified in [13]) with the computation of expected values.

4.1 Fuzzy random variables

Let Σ be the set of states. We consider that the variables in our programs can only take $\underline{\text{real positive}}$ values.[5] We adapted the following definition from [10].

Definition 2. (Fuzzy Random Variable)
A Fuzzy Random Variable (short: FRV) is a mapping X from the set of states Σ to the powerset $P(\mathbb{R}_{\geq 0}^{\infty})$. This means that, if X is an FRV, then for all $\sigma \in \Sigma$, $X(\sigma)$ is a subset of $\mathbb{R}_{\geq 0}^{\infty}$. In fact, it is a set of possible values for X in the state σ.

Example 2. 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.

4.2 Contribution: the weakest-precondition-style calculus of expected values of fuzzy random variables

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

[5] 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.

Definition 3. *$ev : (Progs) \to (F \to F)$ 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 2 of a predicate-transformer-style calculus for the computation of expected values.

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

Table 2: Rules for defining the FRV transformer ev

Consider the guideline example P: $\{\{x := 2\}\Box\{x := 5\}\}[p]\{x := 7\}$. 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 2, we can see that this requires first to compute the functions $ev[\{x := 2\}\Box\{x := 5\}](\underline{x})$ and $ev[x := 7](\underline{x})$; besides, the former requires to compute $ev[x := 2](\underline{x})$, $ev[x := 5](\underline{x})$.

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:

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

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

Finally,

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

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*. 2015.
2. Elwyn R. Berlekamp. *Factoring Polynomials Over Finite Fields*. 1967.
3. Yixiang Chen and Hengyang Wu. *Domain Semantics of Possibility Computations*. 2008.
4. E.W. Dijkstra. *A Discipline of Programming*. 1976.
5. R. Freivalds. *Probabilistic Machines Can Use Less Running Time*. 1977.
6. C.A.R. Hoare. *Algorithm 64: Quicksort*. 1961.
7. Annabelle McIver and Carroll Morgan. *Abstration, Refinement and Proof for Probabilistic Systems*. 2005.
8. Rafael Morales-Bueno, Ricardo Conejo, José Luis Pérez de la Cruz, and Buenaventura Clares. *An elementary fuzzy programming language*. 1993.
9. Rafael Morales-Bueno, José Luis Pérez de la Cruz, Ricardo Conejo, and Buenaventura Clares. *A family of fuzzy programming languages*. 1997.

10. Madan L. Puri and Dan A. Ralescu. *Fuzzy Random Variables.* 1986.
11. Michael O. Rabin. *Probabilistic Algorithm for Testing Primality.* 1977.
12. Murali Krishna Ramanathan, Ronaldo A. Ferreira, Suresh Jagannathan, and Ananth Y. Grama. *Randomized Leader Election.* 2004.
13. Arnold F. Shapiro. *Fuzzy Random Variables.* 2009.
14. Robert.M Solovay and Volker Strassen. *A fast Monte-Carlo test for primality.* 1977.
15. Hengyang Wu and Yixiang Chen. *The Semantics of wlp and slp of Fuzzy Imperative Programming Languages.* 2011.
16. Hengyang Wu and Yixiang Chen. *Semantics of Non–Deterministic Possibility Computation.* 2012.

6 Appendix

6.1 Kleene's Fixed point theorem

Let P be a set with a partial order $\leq$.

Definition 4. *P is a* complete partial order *if the following conditions are satisfied:*

- *$\leq$ has a minimal element, denoted $\perp$ (such that $\forall x \in P$, $\perp \leq x$)*
- *for each subset $S = \{x_1, x_2, \dots\} \subset P$ where $x_1 \leq f_2 \leq \dots$, S has a least upper bound* Sup(S) *in P.*

Theorem 1. Kleene's fixed-point theorem *If P is a complete partial order and $f : P \to P$ is a monotone function (i.e. $\forall x, y \in P$, $x \leq y \Rightarrow f(x) \leq f(y)$), then f has a least fixed-point, and its value is* $\text{lfp}(f) = \text{Sup}_{n\in\mathbb{N}}(f^n(\perp))$

[TO DO: Find a reference to the proof] [TO DO: Monotone or Scott-Continuous... ? If Scott-Continuous, then I should change all proofs a little, but it should not be a problem)]

6.2 Weakest precondition calculus in pGCL

With the notations of Section 3, let $\mathcal{P}$ be the set of predicates, and consider the relation $\leq_{\mathcal{P}}$ such that $f \leq_{\mathcal{P}} g$ iff $\forall \sigma \in \Sigma$, $f(\sigma) \leq g(\sigma)$. The order $\leq_{\mathcal{P}}$ is called the *pointwise order*, as the value of f must be inferior to the value of g in each point σ.

We immediately get the following theorem:

Theorem 2. *$(\mathcal{P}, \leq_{\mathcal{P}})$ is a partial order.*

We will prove that it is even a complete partial order:

Theorem 3. *$(\mathcal{P}, \leq_{\mathcal{P}})$ is a complete partial order*

Proof.
- One can easily verify that $\perp$ is the constant function $\sigma \to 0$
- Let $S = \{f_1, f_2, \dots\}$ be a subset of P with $f_1 \leq_{\mathcal{P}} f_2 \leq_{\mathcal{P}} \dots$. Then, we define the function $f : \sigma \to \text{Sup}_{n\in\mathbb{N}^*}(f_n(\sigma))$.
  For each σ, $f(\sigma)$ is correctly defined: as the set $\{f_n(\sigma) \mid n \in \mathbb{N}^*\}$ is a subset of $[0, 1]$, it has a least upper bound in $[0, 1]$. Thus, $f \in \mathcal{P}$
  - Let $n \in \mathbb{N}^*$. Then, by construction, for each $\sigma \in \Sigma$, $f_n(\sigma) \leq \text{Sup}_{n\in\mathbb{N}^*}(f_n(\sigma)) = f(\sigma)$. And therefore, $f_n \leq_{\mathcal{P}} f$. As this holds for any n, we get that f is an upper bound of S
  - Let g be another upper bound of S: then, $\forall n \in \mathbb{N}^*, \forall \sigma \in \Sigma$, $f_n(\sigma) \leq g(\sigma)$. Then, $\forall \sigma \in \Sigma$, $f(\sigma) = \text{Sup}_{n\in\mathbb{N}^*}(f_n(\sigma)) \leq g(\sigma)$. Hence, $f \leq g$. Therefore, f is the least upper bound of S

Note that, with an analogue proof, if we denote by $\mathcal{F}$ the set of functions $\{f \mid f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\}$ (where $\mathbb{R}^{\infty}_{\geq 0} = [0; +\infty]$ with $+\infty$ included), and by $\leq_{\mathcal{F}}$ the canonical pointwise order over $\mathcal{F}$, we can prove that $(\mathcal{F}, \leq_{\mathcal{F}})$ is a complete partial order.

[TO DO: Prove that the function constructed at then end of the Table 1 is indeed monotone (or Scott-Continuous ?). To that extent, prove inductively that $wp[P]$ is also monotone (or Scot-contiuous ?)]

6.3 Application of the semantics McIver and Morgan to P_0

Recall that P_0 is the program $\{\{x := 2\}\Box\{x := 5\}\}[p]\{x := 7\}$. Recall that we denoted by P_2 the left member $\{x := 2\}\Box\{x := 5\}$. Let φ be any postcondition.

Looking at the rules in Table 1, one notices that, to compute $wp[P](\varphi)$, one first has to compute $wp[P_2](\varphi)$ and $wp[x := 7](\varphi)$, and to compute the former, we need to compute $wp[x := 2](\varphi)$ and $wp[x := 5](\varphi)$.

Applying the assignment rule, we get that:

$$wp[x := 2](\varphi) = \lambda\sigma.\varphi(\sigma[x/2])$$

And symmetrically:

$$wp[x := 5](\varphi) = \lambda\sigma.\varphi(\sigma[x/5])$$
$$wp[x := 7](\varphi) = \lambda\sigma.\varphi(\sigma[x/7])$$

.

Now, applying the nondeterministic choice rule, we get:

$$\begin{aligned} wp[P_2](\varphi) &= wp[\{x := 2\}\Box\{x := 5\}](\varphi) \\ &= \lambda\sigma.\mathrm{Min}(wp[x := 2](\varphi)(\sigma), wp[x := 5](\varphi)(\sigma)) \\ &= \lambda\sigma.\mathrm{Min}(\varphi(\sigma[x/2]), \varphi(\sigma[x/5])) \end{aligned}$$

And finally, applying the probabilistic choice rule, we get:

$$\begin{aligned} wp[P_0](\varphi) &= wp[\{\{x := 2\}\Box\{x := 5\}\}[p]\{x := 7\}](\varphi) \\ &= \lambda\sigma.p \cdot wp[\{x := 2\}\Box\{x := 5\}](\varphi)(\sigma) + (1-p) \cdot wp[x := 7](\varphi)(\sigma) \\ &= \lambda\sigma.p \cdot \mathrm{Min}(\varphi(\sigma[x/2]), \varphi(\sigma[x/5])) + (1-p) \cdot \varphi(\sigma[x/7]) \end{aligned}$$

Now, we can try replacing φ by interesting postconditions.

- If $\varphi = [x = 7]$, then for each σ, we get $[x = 7](\sigma[x/2]) = 0$, $[x = 7](\sigma[x/5]) = 0$ and $[x = 7](\sigma[x/7]) = 1$, and therefore, $wp[P_O]([x = 7]) = \lambda\sigma.(1-p)$
- If $\varphi = [x = 2]$, then for each σ, we get $[x = 2](\sigma[x/2]) = 1$, $[x = 2](\sigma[x/5]) = 0$ and $[x = 2](\sigma[x/7]) = 0$, and therefore, $wp[P_O]([x = 2]) = \lambda\sigma.0$ (and symmetrically, $wp[P_O]([x = 5]) = \lambda\sigma.0$)
- If $\varphi = [x = 2 \text{ OR } x = 5]$, we get $[x = 2 \text{ OR } x = 5](\sigma[x/2]) = 1$, $[x = 2 \text{ OR } x = 5](\sigma[x/5]) = 1$ and $[x = 2 \text{ OR } x = 5](\sigma[x/7]) = 0$, and therefore, $wp[P_O]([x = 2 \text{ OR } x = 5]) = \lambda\sigma.p$

These are precisely the results stated in Section 1.