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: $\{x := 2\}\Box\{x := 5\}$. 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 3)

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, 16, 17] 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 [3,16,17] 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 get a possibility-based semantics for pGCL programs, applying the idea of [3,16,17] of using possibility theory to the pGCL language presented in [7].

The report is organised as follows: In Section 2, we quickly present the work of other authors using possibility theory to define semantics for language derived from GCL, or defining other semantics for pGCL. In Section 3, we present the pGCL language introduced by McIver and Morgan in [7], as well as their predicate transformer semantics (based on Dijkstra's semantics), and how it can be used to anticipate values of positive functions, and relate it to random variables. In Section 4, we present our own semantics, based on the semantics of McIver and Morgan, but used to anticipate values of functions taking sets as values (to model the possibility of different values without more information, resulting of the nondeterministic choice), and explain how it counters the loss of information previously mentioned. The report is finally conclued in Section 5.

[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

In [3], the authors consider a fuzzy imperative language [8, 9], which is an extended version of GCL with more instructions based on possibility theory (notably fuzzy assignments), but which does not include the probabilistic choice. They build mathematical tools (based on domain theory, more precisely on powerdomains), and present the semantics they intend to define with the tools. In [17], the same authors explicitely build two semantics based on the previous tools, namely a fuzzy predicate transformer semantics and a state transformer semantics, for program in the concerned fuzzy language, and prove the equivalence between the two semantic. In [15], the authors develop other mathematical tools based an domain theory, and present as further work the idea of using their tools to develop a denotational semantics (otherwise known as state transformer semantics, an alternative semantics type to the predicate transformer semantics; both types are dual) for programs including probabilities and nondeterminism. (It is, to our knownledge, the only other attempt to define clearer semantics for languages including probabilities AND nondeterminism, but it focuses on another style of semantics.)

3 State of the art: the language, and McIver and Morgan's semantics

In this section, we present the pGCL language and the vision of McIver and Morgan over weakest precondition calculus [7], and explain (**[TO DO: insert reference here]**) how this type of calculus can be exploited to compute expected values of random variables.

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. 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*.*

[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.

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, with the following definition:

Definition 2. *If P is a pGCL program, then the predicate transformer semantics of P is given by $wp[P]$ (defined by the rules[3] in Table 1).*
$wp[P]$ is a predicate transformer, i.e. for all postconditions φ, $wp[P](\varphi)$ is a predicate, such that $\forall \sigma \in \Sigma$, $wp[P](\varphi)(\sigma)$ is the probability that the execution of P in σ terminates in a state satisfying φ

Therefore, P is described by the semantics of McIver and Morgan as an object $wp[P]$ transforming the predicate φ into the predicate $wp[P](\varphi)$.

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` $\{P\}$	$\text{lfp}\ (\lambda X.(\lambda\sigma.[\![b : true]\!](\sigma) \cdot wp[P_1](X)(\sigma) + [\![b : false]\!](\sigma) \cdot \varphi(\sigma)))$

Table 1: Rules for defining the predicate transformer wp

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 expression 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).

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

Remark 1. We will often simplify the expressions in the table by writing for instance $\mathrm{Min}(wp[P_1](\varphi), wp[P_2](\varphi))$ instead of $\lambda\sigma.\mathrm{Min}(wp[P_1](\varphi)(\sigma), wp[P_2](\varphi)(\sigma))$, or $p \cdot wp[P_1](\varphi) + (1-p) \cdot wp[P_2](\varphi)$ instead of $\lambda\sigma.p \cdot wp[P_1](\varphi)(\sigma) + (1-p) \cdot wp[P_2](\varphi)(\sigma)$. But recall that these objects are predicates.

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 $wp[P'](\psi)$

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:

$$\begin{aligned} wp[x := -y; x := x+1]([x \geq 5]) &= wp[x := -y](wp[x := x+1]([x \geq 5])) \\ = wp[x := -y]([x+1 \geq 5]) = \ & wp[x := -y]([x \geq 4]) = [-y \geq 4] = [y \leq -4] \end{aligned}$$

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 Anticipating values, link with random variables

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 in σ, 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 $\mathbb{R}^{\infty}_{\geq 0}$ (where $\mathbb{R}^{\infty}_{\geq 0} = [0; +\infty]$). Let $\mathcal{F} = \{f \mid f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\}$. [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 (and carefully redefine all operations between functions, such as $+$, $\times$, etc.).

Therefore, it is possible to extend the previous predicate transformer semantics in an "anticipated-value" semantics for deterministic programs, by viewing programs as function transformers $\mathcal{F} \to \mathcal{F}$. The anticipation of values can 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 in this case, we apply them to a function $f \in \mathcal{F}$, and obtain another function $wp[P](f) \in \mathcal{F}$. (Note that predicates are also elements of $\mathcal{F}$. Hence, the previous computations still work in this context.)

Notation 1 *In the following, we will denote by x_σ the value of the program variable x in σ, and we will assimilate the program variable x to the function $\underline{x} : \sigma \to x_\sigma$. (This is a simplified notation. In the practice, we should consider one function giving the absolute value of x_σ and another giving its sign.) Hence, $\underline{x} \in \mathcal{F}$, it is a function giving the value of x in each state.*

Example 1. Consider the program P_1 from Section 1, and the function $\underline{x}^2 \in \mathcal{F}$. Its value after the execution of P_1 can be computed with:

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

For example, if $(1 - \underline{y})^2(\sigma) = 4$, then σ is a state where $(1 - y_\sigma)^2 = 4$, for instance where $y_\sigma = 3$. Then by executing P_1 in σ, x is set to -3, and then to -2. In the final state σ', we have $x^2_{\sigma'} = 2$. Therefore, $(1 - \underline{y})^2$ anticipates the value that $\underline{x}^2$ takes after the execution of the program.

Example 2. Consider the program P_0 from Section 1, we can anticipate the value of the variable x by computing the anticipate value of the function $\underline{x}$ (we should

[4] With this restriction, a complete partial order can be defined on $\mathcal{F}$ and Kleene's fixed point theorem can be applied. See Appendices 6.1 and 6.2

normally anticipate the value of $\underline{|x|}$, but we easily see that they are the same in this context):

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

We leave the result in this form to underline interesting remarks.

Remark 2. For each σ, $wp[P_0](\underline{x})(\sigma)$, which is the anticipated value of x after the execution of P_0 in σ, has the form of the expected value $E(X)$ of a positive random variable X taking the value 2 with the probability p, and the value 7 with the probability $1-p$.

Theorem 1. *Let P be a probabilistic program (without nondeterminism), let σ be an initial state, and let $f \in \mathcal{F}$ be a function. For each ω, the execution of P in ω can be informally divided as several "branches", called execution schemes ω (at each choice, the current branch is divided into two branches). We consider an execution scheme to be a full execution, i.e. at the end of an ω, P terminated, or is looping infinitely. Let Ω_σ be the set of execution schemes obtained when executing P in σ. A probability can be assigned to each $\omega \in \Omega_\sigma$. We can therefore see the value of f after the execution of P in σ as a positive random variable $X_\sigma : \Omega_\sigma \to \mathbb{R}^{\infty}_{\geq 0}$.*
Thus, $wp[P](\bar{f}) = \lambda\sigma.E(X_\sigma)$.

This result can be more clearly observed in Example 4 in Subsection 6.4. **[TO DO: Maybe move the example here.]**

Remark 3. The previous theorem can be extended to fully probabilistic programs by applying rules in Table 1 and by considering the nondeterministic choice as the demonic choice. Note however that the Ω_σ that we obtain depend on the postcondition we are computing (as shown in Example 5 in Subsection 6.4), because the the demonism in nondeterministic choice will restrict each of these choices to one outcome, depending on the postcondition.

Remark 4. The result of the computation of $wp[P_0](\underline{x})$ still does not take in account the possibility that x can take the value 5. Symmetrically, if the nondeterministic choice was to be considered as the angelic choice, we would get that $wp[P_0](\underline{x}) = 5p + 7(1-p)$ and one result would be missing.

These results are therefore counterintuitive. At this point, we defined our own semantics by considering fuzzy random variables, instead of random variables.

4 A new semantics based on possibilities

In this section, we consider programs over the pGCL language presented in Section 3. As we stated before, previous existing semantics give nonsatisfying results, both when computing weakest preconditions of predicates in fully probabilistic programs and when anticipating values of functions in fully probabilistic programs. Our goal is to define a new semantics, which should be based on possibility theory, and would more clearly represent each possible outcome. Recall that we mentioned in Section 3 that weakest-precondition calculus could be used in order to anticipate values of functions of $\mathcal{F}$ (i.e. functions from Σ to $\mathbb{R}^{\infty}_{\geq 0}$), and that the corresponding computation in fact gave, for each state σ, the expected value of a random variable (i.e. the value of the function we are anticipating, after the execution of the program in σ). Our main idea here is to replace functions from Σ to $\mathbb{R}^{\infty}_{\geq 0}$ by functions from Σ to $\mathcal{P}(\mathbb{R}^{\infty}_{\geq 0})$, and to identify the anticipation of their values as the expected value of *Fuzzy random variable* (introduced in [10], and simplified in [13]).

4.1 Fuzzy random variables

We adapted the following definition from [10].

Definition 3. (Fuzzy Random Variable)
A Fuzzy Random Variable (short: FRV) is a mapping X from a probabilistic space $(\Omega, \mathcal{A}, Pr)$ to the powerset $P(\mathbb{R}^{\infty}_{\geq 0})$. This means that, if X is an FRV, then for all $\omega \in \Omega$, $X(\omega)$ is a subset of $\mathbb{R}^{\infty}_{\geq 0}$. In fact, it is a set of possible values for X in the event ω.

In [10], there exists a definition of the expected value $E(X)$ of an arbitrary fuzzy random variable X over an arbitrary probabilistic space. However, in our context, it suffices to consider Ω as a countable space $\{\omega_1, \omega_2, \dots\}$ (recall that our events are execution schemes) and fuzzy random variables taking as values only sets of positives values (as we previously defined them). Then, by using the definition of expected value in [10] and applying our simplifications, we come to the following definition of the expected value, which is sufficient in our context:

Definition 4. *If X is a fuzzy random variable from Ω to $\mathbb{R}^{\infty}_{\geq 0}$, then we can define the expected value $E(X)$ as follows:*

$$E(X) = \{\sum_{n=1}^{+\infty} t_i Pr(\omega_i) \,|\, \forall i \in \mathbb{N}^*, t_i \in X(\omega_i)\} \subset \mathbb{R}^{\infty}_{\geq 0}$$

One can interprete this definition as follows: if X is a positive fuzzy random variable (i.e. a random variable associating to each event a set of positive values), then $E(X)$ is the set obtained by computing all possible (even infinite) averages (by picking one value in each $X(\omega_i)$).

4.2 Contribution: the weakest-precondition-style calculus of anticipation of values of set-valued functions, link with fuzzy random variables

Let $\mathcal{F}' = \{f \mid f : \sigma \to P(\mathbb{R}^{\infty}_{\geq 0})\}$ be the new set of functions whose values we want to anticipate. What it intuitively means is that f maps each σ to a set of "possible values". (As we previously explained, restriction to positive values only is not a problem in the practice.)

Example 3. In the following, we redefine the application $\underline{x}$ by : $\underline{x} = \lambda\sigma.\{x_\sigma\}$, meaning that $\underline{x}$ associates to σ a set containing the only possible value for x in this state (which is directly accessible, since each state is characterized uniquely by the values of each variable).

We replace the *wp* operator by the following operator:

Definition 5. *If P is a pGCL program, then our* ***[TO DO: find a proper name]*** *semantics of P is given by $ev[P]$ (defined by the rules in Table 2).*
$ep[P]$ is a function transformer, i.e. for all functions $f \in \mathcal{F}'$, then $ev[P](f) \in \mathcal{F}'$.

Recall that we mentioned in Subsection 3.3 that, if $f \in \mathcal{F}$ and if P is a pGCL program, then the value of f after the execution of P in a state σ was given by a random variable X_σ, and $wp[P](f)$ in fact gave $E(X_\sigma)$ for each σ.
Symmetrically, here, if $f \in \mathcal{F}'$, then the value of f (which is a set) after the execution of P in a state σ is given by a fuzzy random variable X_σ, and $ev[P](f)$ in fact gives $E(X_\sigma)$ (which is a set) for each σ. This confirms that $ev[P]$ transforms a function of $\mathcal{F}'$ in another function of $\mathcal{F}'$.

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

Table 2: Rules for defining the FRV transformer *ev*

Consider the running example P: $\{\{x := 2\}\square\{x := 5\}\}[p]\{x := 7\}$. The value of x after the execution of P in a given initial state is given by a fuzzy random

variable. We want to compute its expected value. Thus, we have to determine $ev[P](\underline{x})$. 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,

$$\begin{aligned} ev[P_0](\underline{x}) &= \lambda\sigma.\{t_1p + t_2(1-p) \mid t_1 \in ev[\{x := 2\}\Box\{x := 5\}](\underline{x})(\sigma), t_2 \in ev[x := 7](\underline{x})(\sigma)\} \\ &= \lambda\sigma.\{t_1p + t_2(1-p) \mid t_1 \in \{2, 5\}, t_2 \in \{7\}\} \\ &= \lambda\sigma.\{2p + 7(1-p), 5p + 7(1-p)\} \end{aligned}$$

Contrary to what was observed with the semantics of McIver and Morgan, no information is lost here. The results we just showed means that, when we execute P_0 in a state σ, there are two possible values for the program variable x after the execution. The first one results of the case where $x := 2$ is executed in the nondeterministic choice. In this case, anticipating the value of x is the same as computing the expected value of a classical random variable as we did in Subsection 3.3. The second one is obtained symmetrically in the case where $x := 5$ is executed.
The link with fuzzy random variables is clearer, as the previous results shows that, for each σ, $ev[P_0](\underline{x})(\sigma)$ is the expected value of a fuzzy random variable taking the value $\{2, 5\}$ with a probability p (this corresponds to the first execution scheme, where the left branch of the probabilistic choice is executed. And this case, x has indeed a probability p of being set to 2 or 5. Both choices are possible, and that is why both values belong to the set), and the value $\{7\}$ with a probability $1 - p$.

Finally, we can explain how our semantics is related to possibility theory. Recall that McIver and Morgan extended wp to fully probabilistic programs

such that, if φ is a predicate, then $wp[P](\varphi)(\sigma)$ is the probability that P, when executed in σ, terminates in a state where φ is satisfied.
If we denote by $\mathbb{1}_U$ the indicator function of a set U, and if we take $f \in \mathcal{F}'$, then $\mathbb{1}_{ev[P](f)(\sigma)}(a)$ is the possibility that a is an expected value of f after the execution of P in the state σ. And we can extend this result: if $V \subset \mathbb{R}^{\infty}_{\geq 0}$, then $\mathbb{1}_{ev[P](f)(\sigma)}(V) = \mathrm{Sup}_{a\in V}(\mathbb{1}_{ev[P](f)(\sigma)}(a))$ is the possibility that at least one value in V is an expected value of f after the execution of P in σ.

5 Conclusion

In this report, we presented the vision of weakest-precondition calculus over pGCL program, detailed by McIver and Morgan in [7]. We explained how the weakest-precondition calculus could be used in order to anticipate the values of particular functions (the restriction to positive functions is due to constraints of having a complete partial order over the set of functions in order to ensure the existence of least fixed-points and make the computation possible), and explained the link with random variables, i.e. the value of a function after the execution of program in a given initial state is a random variable, and the weakest precondition calculus gives access to its expected values. We underlined that nevertheless, the results obtained with the semantics of McIver and Morgan were not satisfying with the demonic choice, and neither with the angelic choice, as both conventions omitted essential information. Finally, we presented the notion of fuzzy random variables, and our own semantics, where we anticipate values of functions taking not positive real numbers as values, but set of positive real numbers. We explained the link between our semantics and the expected value of fuzzy random variables.

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. Regina Tix, Klaus Keimel, and Gordon Plotkin. *Semantics Domain for Combining Probability and Non-Determinism.* 2009.
16. Hengyang Wu and Yixiang Chen. *The Semantics of wlp and slp of Fuzzy Imperative Programming Languages.* 2011.
17. 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 6. *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 $x_1 \leq fx2 \leq \dots$, S has a least upper bound* $\mathrm{Sup}(S)$ *in P (Such a set S is called a* chain*)*

Theorem 2. 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* $\mathrm{lfp}(f) = \mathrm{Sup}_{n\in\mathbb{N}}(f^n(\perp))$

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

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

Theorem 4. *$(\mathcal{P}, \leq_{\mathcal{P}})$ 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 $f : \sigma \to \mathrm{Sup}_{n\in\mathbb{N}^*}(f_n(\sigma))$.
For each σ, $f(\sigma)$ is correctly defined: as the set $\{f_n(\sigma) \,|\, n \in \mathbb{N}^*\}$ 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 $\sigma \in \Sigma$, $f_n(\sigma) \leq \mathrm{Sup}_{n\in\mathbb{N}^*}(f_n(\sigma)) = f(\sigma)$. 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, $\forall \sigma \in \Sigma$, $f(\sigma) = \mathrm{Sup}_{n\in\mathbb{N}^*}(f_n(\sigma)) \leq g(\sigma)$. 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. The inclusion of $+\infty$ in the set of values taken by the function is necessary, otherwise the proof does not work, we can find chains for which the Sup does not exist.

[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, $wp[P_0]([x = 7]) = \lambda\sigma.(1-p)$
- 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, $wp[P_0]([x = 2]) = \lambda\sigma.0$ (and symmetrically, $wp[P_0]([x = 5]) = \lambda\sigma.0$)

– 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, $wp[P_0]([x = 2 \text{ OR } x = 5]) = \lambda\sigma.p$

These are precisely the results stated in Section 1.

6.4 Example of anticipation of values in Section 3.3

Example 4. Here is an example of program where the random variables associated to the anticipated value differ, depending on the initial state.

$$P_3 : \{x := y + 1\}[p]\{x := y - 1\}$$

The value of y is unknown and depends on the initial state. Let $\sigma \in \Sigma$ be an initial state. Consider that we want to anticipate the value of the program variable x (and therefore, of the function $\underline{x}$). Two execution schemes are possible:

– ω_1, with probability p, executes $x := y + 1$
– ω_2, with probability $1 - p$, executes $x := y - 1$

When computing the anticipated value of $\underline{x}$, we get:

$$\begin{aligned} wp[P_3](\underline{x})(\sigma) &= p \cdot wp[x := y + 1](\underline{x})(\sigma) + (1 - p) \cdot wp[x := y - 1](\underline{x})(\sigma) \\ &= p \cdot \underline{x}(\sigma[x/y + 1]) + (1 - p) \cdot \underline{x}(\sigma[x/y - 1]) \\ &= p(y_\sigma + 1) + (1 - p)(y_\sigma - 1) \end{aligned}$$

Therefore:

$$wp[P_3](\underline{x}) = p(\underline{y} + 1) + (1 - p)(\underline{y} - 1)$$

In this example, it is clear that for each σ, $wp[P_3](\underline{x})$ appears as the expected value of a random variable, taking the value $(y_\sigma + 1)$ with probability p and the value $(y_\sigma - 1)$ with probability $1 - p$. However, this random variable depends on σ. Therefore, when you change σ, you change the random variable.

Example 5. Here is an example of program where the execution scheme differ, depending on the postcondition. Consider the program P_0 defined in Section 1.

– When considering as postcondition the predicate $[x = 2]$, we get two execution schemes:
  - ω_1 (executed with a probability p), executing the left part of the probabilistic choice, and then automatically picking the least desired outcome in the nondeterministic choice, i.e. $x := 5$
  - ω_2 (executed with a probability $1 - p$), executing the right part of the probabilistic choice
– When considering as postcondition the predicate $[x = 5]$, we get two other execution schemes:
  - ω_1 (executed with a probability p), executing the left part of the probabilistic choice, and then automatically picking the least desired outcome in the nondeterministic choice, i.e. $x := 2$
  - ω_2 (executed with a probability $1 - p$), executing the right part of the probabilistic choice

6.5 Anticipation of values of set-valued functions in Section 4

Recall that we defined $\mathcal{F}' = \{f : | f : \Sigma \to \mathcal{P}(\mathbb{R}^{\infty}_{\geq 0})\}$, and we defined the semantics of the pGCL program P as a transformer $\overline{ev}[P]$, transforming $f \in \mathcal{F}'$ in $ev[P](f) \in \mathcal{F}'$. But for the definition of ev to make sense (i.e. for the least fixed-point in the loop rule to exist, by applying Kleene's Theorem), we want to prove two points:

1. $\mathcal{F}'$ is a complete partial order.
2. For all programs P, the function transformer $ev[P] : \mathcal{F}' \to \mathcal{F}'$ is monotone (this is sufficient to make the function whose least fixed-point is calculated monotone)

We define the lower closure of a subset of $\mathbb{R}^{\infty}_{\geq 0}$.

Definition 7. *If U is a subset of $\mathbb{R}^{\infty}_{\geq 0}$, then the lower closure $\downarrow U$ of U is defined as:*

$$x \in \downarrow U \text{ iff } \exists y \in U, x \leq y$$

We immediately get the following lemma, which will be used a lot in our proofs:

Lemma 1. *1. If U is a subset of $\mathbb{R}^{\infty}_{\geq 0}$, then $U \subset \downarrow U$.*
2. $\downarrow U = \emptyset$ iff $U = \emptyset$

Theorem 5. *The following order is a partial order over $\mathcal{F}'$:*

$$f \leq_{\mathcal{F}'} g \text{ iff } \forall \sigma \in \Sigma, \downarrow (f(\sigma)) \subset \downarrow (g(\sigma))$$

Proof. Reflexivity, antisymmetry and transitivity are respectively obtained from the reflexivity, antisymmetry and transitivity of $\subset$ as an order relation over $\mathbb{R}^{\infty}_{\geq 0}$.

Theorem 6. *$(\mathcal{F}', \leq_{\mathcal{F}'})$ is a complete partial order.*

Proof.
- One easily sees that $\bot : \sigma \to \emptyset$ is the minimal element
- Let $S = \{f_1, f_2, \dots\}$ be a chain of $\mathcal{F}'$. Therefore, we have $f_1 \leq_{\mathcal{F}'} f_2 \leq_{\mathcal{F}'} \dots$. We shall consider $f \in \mathcal{F}'$ defined by: $f : \sigma \to \cup_{n \in \mathbb{N}^*} f_n(\sigma)$. We will prove that f is the least upper bound of S.
  - Let $n \in \mathbb{N}^*$ and $\sigma \in \Sigma$. We have to prove that $\downarrow (f_n(\sigma)) \subset \downarrow (f(\sigma))$. If $f_n(\sigma) = \emptyset$, then we immediately get the result.
    Otherwise, let $x \in \downarrow (f_n(\sigma))$. Therefore, there exists $y \in f_n(\sigma)$ such that $x \leq y$. We immediately deduce that $y \in f_n(\sigma) \in \cup_{n \in \mathbb{N}^*} f_n(\sigma) = f(\sigma)$, and as $x \leq y$, then $x \in \downarrow (f(\sigma))$. As this holds for all $n \in \mathbb{N}^*$ and for all $\sigma \in \Sigma$, we get that:

    $$\forall n \in \mathbb{N}^*, \forall \sigma \in \Sigma, \downarrow (f_n(\sigma)) \subset \downarrow (f(\sigma))$$

    And therefore, $\forall n \in \mathbb{N}^*$, $f_n \leq_{\mathcal{F}'} f$. Thus, f is an upper bound for S.

- Let g be another upper bound of S. Therefore, we have:
$$\forall n \in \mathbb{N}^*, \forall \sigma \in \Sigma, \downarrow (f_n(\sigma)) \subset \downarrow (g(\sigma))$$
Let $\sigma \in \Sigma$. We want to prove that $\downarrow (f(\sigma)) \subset \downarrow (g(\sigma))$. If $f(\sigma) = \emptyset$, the result is immediate.
Otherwise, let $x \in \downarrow (f(\sigma))$. Therefore, there exists $y \in f(\sigma)$ such that $x \leq y$. As $y \in f(\sigma)$, there exists $n \in \mathbb{N}^*$ such that $y \in f_n(\sigma)$. With the Lemma 1, we get that $y \in \downarrow (f_n(\sigma))$, and then, with the previous hypothesis, $y \in \downarrow (g(\sigma))$. Hence, there exists $z \in g(\sigma)$ with $y \leq z$. Thus, $x \leq z$, and therefore, $x \in \downarrow (g(\sigma))$.
Therefore, $\forall \sigma \in \Sigma$, $\downarrow (f(\sigma)) \subset \downarrow (g(\sigma))$, and thus, $f \leq_{\mathcal{F}'} g$. Hence, f is the least upper bound of S.

We proved the first point: $\mathcal{F}'$ is a complete partial order.
Now, we want to prove the following theorem.

Theorem 7. *For all pGCL programs P, $ev[P] : \mathcal{F}' \to \mathcal{F}'$ is a monotone function*

Proof. We will prove it by induction, using the rules in Table 2.

– Obviously, when $P = \texttt{skip}$, if $f \leq_{\mathcal{F}'} g$, then $ev[skip](f) = f \leq_{\mathcal{F}'} g = ev[skip](g)$

– $P = \{x := A\}$, if $f \leq_{\mathcal{F}'} g$, then we get that $\forall \sigma \in \Sigma$, $\downarrow (f(\sigma)) \subset \downarrow (g(\sigma))$. Particularly, for all $\sigma' \in \Sigma$ such that there exists $\sigma \in \Sigma$ where $\sigma' = \sigma[x/A]$, we get $\downarrow (f(\sigma')) \subset \downarrow (g(\sigma'))$. Therefore, $ev[x := A](f) = \lambda\sigma.f(\sigma[x/A]) \leq_{\mathcal{F}'} \lambda\sigma.g(\sigma[x/A]) = ev[x := A](g)$

– When $P = P_1; P_2$, assume that $ev[P_1]$ and $ev[P_2]$ are monotone. Then, if $f \leq_{\mathcal{F}'} g$, we get that $ev[P_2](f) \leq_{\mathcal{F}'} ev[P_2](g)$, and then, $ev[P_1](ev[P_2](f)) \leq_{\mathcal{F}'} ev[P_1](ev[P_2](g))$. Finally, we proved that $ev[P_1; P_2](f) \leq_{\mathcal{F}'} ev[P_1; P_2](g)$

– When $P = P_1 \Box P_2$, assume that $ev[P_1]$ and $ev[P_2]$ are monotone. Then, if $f \leq_{\mathcal{F}'} g$, we shall prove that $ev[P](f) = \lambda\sigma.ev[P_1](f)(\sigma) \cup ev[P_2](f)(\sigma) \leq_{\mathcal{F}'} \lambda\sigma.ev[P_1](g)(\sigma) \cup ev[P_2](g)(\sigma) = ev[P](g)$.
To that extent, we have to prove that,
$$\forall \sigma \in \Sigma, \downarrow (ev[P_1](f)(\sigma) \cup ev[P_2](f)(\sigma)) \subset \downarrow (ev[P_1](g)(\sigma) \cup ev[P_2](g)(\sigma))$$
Let $\sigma \in \Sigma$. If $ev[P_1](f)(\sigma) \cup ev[P_2](f)(\sigma) = \emptyset$, then there is nothing to prove.

Otherwise, let $x \in \downarrow (ev[P_1](f)(\sigma) \cup ev[P_2](f)(\sigma))$. Then, there exists $y \in ev[P_1](f)(\sigma) \cup ev[P_2](f)(\sigma)$ such that $x \leq y$. We assume that $y \in ev[P_1](f)(\sigma)$. Then, with the Lemma 1, $y \in \downarrow (ev[P_1](f)(\sigma)) \subset \downarrow (ev[P_1](g)(\sigma))$. Therefore, there exists $z \in ev[P_1](g)(\sigma)$ such that $y \leq z$. One the one hand, we can notice that $z \in ev[P_1](g)(\sigma) \subset ev[P_1](g)(\sigma) \cup ev[P_2](g)(\sigma)$, and on the other hand, that $x \leq y \leq z$. Therefore, $x \in \downarrow (ev[P_1](g)(\sigma) \cup ev[P_2](g)(\sigma))$. The proof is symmetrical if $y \in ev[P_2](f)(\sigma)$. Therefore, we got our result.

– When $P = P_1[p]P_2$, assume that $ev[P_1]$ and $ev[P_2]$ are monotone. Then, if $f \leq_{\mathcal{F}'} g$, we shall prove that $ev[P](f) = \lambda\sigma.\{t_1p+t_2(1-p) \mid t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\} \leq_{\mathcal{F}'} \lambda\sigma.\{t_1p+t_2(1-p) \mid t_1 \in ev[P_1](g)(\sigma), t_2 \in ev[P_2](g)(\sigma)\} = ev[P](g)$.
To that extent, we have to prove that:

$$\begin{aligned}\forall\sigma \in \Sigma, \downarrow(\{t_1p + t_2(1-p) \mid t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\})\\ \subset \downarrow(\{t_1p + t_2(1-p) \mid t_1 \in ev[P_1](g)(\sigma), t_2 \in ev[P_2](g)(\sigma)\})\end{aligned}$$

Let $\sigma \in \Sigma$. As usual, if $\{t_1p+t_2(1-p) \mid t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\} = \emptyset$, there is nothing to prove.

Otherwise, let $x \in\downarrow(\{t_1p + t_2(1-p) \mid t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\})$.
Then, there exists $y \in \{t_1p + t_2(1-p) \mid t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\}$ such that $x \leq y$. We can tell that there exists $t_1 \in ev[P_1](f)(\sigma)$ and $t_2 \in ev[P_1](2)(\sigma)$ such that $y = t_1p + t_2(1-p)$.
With the Lemma 1, $t_1 \in ev[P_1](f)(\sigma) \subset\downarrow(ev[P_1](f)(\sigma)) \subset\downarrow(ev[P_1](g)(\sigma))$, and therefore, there exists $z_1 \in ev[P_1](g)(\sigma)$ such that $t_1 \leq z_1$. Symmetrically, there exists $z_2 \in ev[P_2](g)(\sigma)$ such that $t_2 \leq z_2$.

Now, we have $x \leq y = t_1p+t_2(1-p) \leq z_1p+z_2(1-p)$, with $z_1 \in ev[P_2](g)(\sigma)$ and $z_2 \in ev[P_2](g)(\sigma)$.
This proves that $x \in\downarrow(\{t_1p+t_2(1-p) \mid t_1 \in ev[P_1](g)(\sigma), t_2 \in ev[P_2](g)(\sigma)\})$.

– When $P =$ `if` b `then` P_1 `else` P_2, assume that $ev[P_1]$ and $ev[P_2]$ are monotone. Then, if $f \leq_{\mathcal{F}'} g$, we shall prove that $ev[P](f) = \lambda\sigma.\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \mid t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\} \leq_{\mathcal{F}'} \lambda\sigma.\{[\![b : true]\!](\sigma)\cdot t_1+[\![b : false]\!](\sigma)\cdot t_2 \mid t_1 \in ev[P_1](g)(\sigma), t_2 \in ev[P_2](g)(\sigma)\} = ev[P](g)$.
To that extent, we have to prove that:

$$\begin{aligned}\forall\sigma \in \Sigma, \downarrow\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \mid\\ t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\}\\ \subset \downarrow\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \mid\\ t_1 \in ev[P_1](g)(\sigma), t_2 \in ev[P_2](g)(\sigma)\}\end{aligned}$$

Let $\sigma \in \Sigma$. If $\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \mid t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\} = \emptyset$, then the result is immediate.

Otherwise, let $x \in\downarrow\{[\![b : true]\!](\sigma)\cdot t_1+[\![b : false]\!](\sigma)\cdot t_2 \mid t_1 \in ev[P_1](f)(\sigma), t_2 \in ev[P_2](f)(\sigma)\}$.
Then, there exists $t_1 \in ev[P_1](f)(\sigma)$ and $t_2 \in ev[P_2](f)(\sigma)$ such that $x \leq [\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2$. Let $y = [\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2$

Using the Lemma 1, we get that $t_1 \in ev[P_1](f)(\sigma) \subset\downarrow(ev[P_1](f)(\sigma)) \subset\downarrow(ev[P_1](g)(\sigma))$. Therefore, there exists $z_1 \in ev[P_1](g)(\sigma)$ such that $t_1 \leq z_1$.

And symmetrically, there exists $z_2 \in ev[P_2](g)(\sigma)$ such that $t_2 \leq z_2$.

Let $z = [\![b : true]\!](\sigma) \cdot z_1 + [\![b : false]\!](\sigma) \cdot z_2$.
If b holds in σ, then $y = t_1$ and $z = z_1$. We had $x \leq y$, and $t_1 \leq z_1$. Thus, $x \leq z$. Symmetrically, if b does not hold in σ, then, $y = t_2$ and $z = z_2$ and we symmetrically prove that $x \leq z$. And we easily see that $z \in \{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \,|\, t_1 \in ev[P_1](g)(\sigma), t_2 \in ev[P_2](g)(\sigma)\}$. This concludes this point of the proof.

– When $P =$ `while` (b) `do` $\{P_1\}$, assume that $ev[P_1]$ is monotone. Then, if $f \leq_{\mathcal{F}'} g$, we shall prove that $ev[P](f) = \text{lfp } (\lambda X.(\lambda\sigma.\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \,|\, t_1 \in ev[P_1](X)(\sigma), t_2 \in f(\sigma)\}) \leq_{\mathcal{F}'} \text{lfp } (\lambda X.(\lambda\sigma.\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \,|\, t_1 \in ev[P_1](X)(\sigma), t_2 \in g(\sigma)\}) = ev[P](g)$.

  We proved previously that $\mathcal{F}'$ was a complete partial order. Therefore, Kleene's fixed-point theorem can be applied here, and we can compute the two least fixed-point previously mentioned.

  Let $F_f(X)$ denote $\lambda X.(\lambda\sigma.\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \,|\, t_1 \in ev[P_1](X)(\sigma), t_2 \in f(\sigma)\}$, and symmetrically, let $F_g(X)$ denote $\lambda X.(\lambda\sigma.\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \,|\, t_1 \in ev[P_1](X)(\sigma), t_2 \in g(\sigma)\}$.

  We seek to prove that lfp $F_f \leq_{\mathcal{F}'}$ lfp F_g. Applying Kleene's Theorem (recall that $\perp : \sigma \rightarrow \emptyset$), we get that

$$\text{lfp } F_f = \text{Sup}_{n \in \mathbb{N}^*}(F_f^n(\perp))$$
$$\text{lfp } F_g = \text{Sup}_{n \in \mathbb{N}^*}(F_g^n(\perp))$$

  Therefore, we have to prove that:

$$\forall \sigma \in \Sigma, \downarrow((\text{Sup}_{n \in \mathbb{N}^*}(F_f^n(\perp)))(\sigma)) \subset \downarrow((\text{Sup}_{n \in \mathbb{N}^*}(F_g^n(\perp)))(\sigma))$$

  One easily verifies that, to prove the previous proposition, it is sufficient to prove that:

$$\forall n \in \mathbb{N}, F_f^n(\perp) \leq_{\mathcal{F}'} F_g^n(\perp)$$

  Or, equivalently:

$$\forall n \in \mathbb{N}, \forall \sigma \in \Sigma, \downarrow(F_f^n(\perp)(\sigma)) \subset \downarrow(F_g^n(\perp)(\sigma))$$

  This is proven by induction:
  - For $n = 0$, $F_f^0 = F_g^0 = Id_{\mathcal{F}'}$, and therefore, the result is immediate.
  - If, for one $n \in \mathbb{N}$, we get that $\forall \sigma \in \Sigma, \downarrow(F_f^n(\perp)(\sigma)) \subset \downarrow(F_g^n(\perp)(\sigma))$, then, let $\sigma \in \Sigma$ and let $x \in \downarrow(F_f^{n+1}(\perp)(\sigma))$. We shall prove that $x \in \downarrow(F_g^{n+1}(\perp)(\sigma))$.

We know that there exists $y \in F_f^{n+1}(\bot)(\sigma)$ such that $x \leq y$. And as $y \in F_f^{n+1}(\bot)(\sigma) = F_f(F_f^n(\bot))(\sigma) = \{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \,|\, t_1 \in ev[P_1](F_f^n(\bot))(\sigma), t_2 \in f(\sigma)\}$.

Hence, there exists $t_1 \in ev[P_1](F_f^n(\bot))(\sigma)$ and $t_2 \in f(\sigma)$ such that $y = [\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2$.
With the Lemma 1 we get that $t_1 \in ev[P_1](F_f^n(\bot))(\sigma) \subset\downarrow (ev[P_1](F_f^n(\bot))(\sigma))$.
With the induction hypothesis, that $F_f^n(\bot) \leq_{\mathcal{F}'} F_g^n(\bot)$, and with the monotonicity of $ev[P_1]$, we get that $ev[P_1](F_f^n(\bot)) \leq_{\mathcal{F}'} ev[P_1](F_g^n(\bot))$, and particularly, that $\downarrow (ev[P_1](F_f^n(\bot))(\sigma)) \subset\downarrow (ev[P_1](F_g^n(\bot))(\sigma))$. Hence, $t_1 \in\downarrow (ev[P_1](F_g^n(\bot))(\sigma))$.
Symetrically, recall that $f \leq_{\mathcal{F}'} g$, and therefore, $\downarrow (f(\sigma)) \subset\downarrow (g(\sigma))$. And as $t_2 \in f(\sigma)$, then $t_2 \in\downarrow (f(\sigma))$, and hence, $t_2 \in\downarrow (g(\sigma))$.

Hence, e get that there exists $z_1 \in ev[P_1](F_g^n(\bot))(\sigma)$ and $z_2 \in g(\sigma)$ such that $t_1 \leq z_1$ and $t_2 \leq z_2$.
Therefore, $y = [\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \leq [\![b : true]\!](\sigma) \cdot z_1 + [\![b : false]\!](\sigma) \cdot z_2 \in F_g(F_g^n(\bot))(\sigma) = F_g^{n+1}(\sigma)$. Therefore, $y \in\downarrow (F_g^{n+1}(\sigma))$. And as $x \leq y$, then $x \in\downarrow (F_g^{n+1}(\sigma))$.

This concludes the proof.