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 do not compute simply one output state, but 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 semantic 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: Weakest-precondition ; Possibility theory ; Anticipation of values ; Fuzzy random variables

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 the first 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 on a fixed input will not always result in the same output. In this report, programs containing probabilistic choices are 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. For instance, 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 [8], Freivalds' matrix multiplication [6], and certain primality tests, such as the Miller-Rabin test [17] and the Solovay-Strassen test [20] (for positive numbers), and the Berlekamp test (for polynomials) [3]. There also exist probabilistic algorithms to solve the leader election problem [18].
Apart from probabilistic choice, there exists a different kind of choice, frequently used, which also makes 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 a probability to them. In this report, programs containing such choices are 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 [5] (which did not include the probabilistic choice), and is used, for instance, in the model-checker Spin [9], to model the unpredictable behavior of a program where several choices are possible, for instance when modeling concurrency problems.

It is also used in programs where the outcomes of a choice are known, but the mechanism making the choices is either unknown, or uses overly complex probability distributions, or when there are several possible mechanisms. [12] This instruction, however, is not meant to be executed in a program, it is only used for modeling purposes.

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

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

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

A frequently used semantics, which is well-suited to program verification, is Dijkstra's predicate transformer semantics [5] (in the case of nondeterministic programs without probabilistic choices). More precisely, if S is a program and φ a predicate over the set of states (i.e. an object modeling a property over states), then the predicate transformer semantics describes S as an object $wp[S]$, transforming the predicate φ (called *postcondition* in this context) into another predicate called the *weakest precondition of φ for the program S*, denoted by $wp[S](\varphi)$. As the name indicates, this new predicate is the weakest (in the sense of "least restrictive", or "satisfied by the most states") predicate ψ satisfying the following property: if S is executed in a state satisfying ψ, then S 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 [5] and will be recapped in Section 3, and the calculation is detailed in Appendix 6.3.

Dijkstra's predicate transformer semantics was extended by McIver and Morgan [12] to probabilistic nondeterministic programs by changing the semantics, such that the weakest precondition $wp[S](\varphi)$ assigns to each state σ the probability that the execution of the program S in σ terminates in a state satisfying φ, instead of assigning only 0 or 1 values meaning that the execution of S will certainly or certainly not terminate in a state satisfying the postcondition. Note that, when 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 $1 - p$, which can be written as $wp[P_0]([x = 7]) = \lambda\sigma.1 - p$). Now, consider the subprogram P_2 of P_0:

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

Recall that the semantics of McIver and Morgan, $wp[S](\varphi)$ assigns to each state σ the probability that the execution of S in σ terminates in a state satisfying φ. It is therefore a function from the set of states Σ to $[0, 1]$. As it is not possible to assign probabilities to the outcomes of a nondeterministic choice, 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* [12]. 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 *angelic choice* (named in [12]), 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 symmetrically for $wp[P_0]([x = 5])$. But $wp[P_0]([x = 2 \text{ OR } x = 5])$ is also the constant function p. This result 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 exist 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 [4, 25, 26] tried a different approach to the weakest precondition calculus, based on possibility theory rather than 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, 19]). Recall 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, 19]):

1. $\Pi(\emptyset) = 0$
2. $\Pi(\Omega) = 1$
3. $\forall U, V \in \mathcal{A}$, $\Pi(U \cup V) = \text{Max}(\Pi(U), \Pi(V))$ (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 happens. 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 possible, but we cannot tell that it will almost surely happen. As the name indicates, a possibility measure only indicates how possible an event is, but not how likely it is to be 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 [4, 25, 26] in order to define predicate transformer semantics for fuzzy imperative languages (defined in [14, 15], 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 into 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 a semantics describing the program by anticipating the values of functions after its execution (this kind of semantics can be derived from predicate transformer semantics), where we consider set-valued functions. We show that it is related to the computation of expected values of fuzzy random variables (first defined in [16], and then simplified in [19]). It is a

way to get a possibility-based semantics for probabilistic nondeterministic programs, applying the idea of [4, 25, 26] of using possibility theory to the pGCL language (derived from GCL, including probabilistic choice) presented in [12].

The report is organised as follows: In Section 2, we briefly present the work of other authors using possibility theory to define semantics for languages derived from GCL, or defining another style of semantics for pGCL. In Section 3, we present the pGCL language introduced by McIver and Morgan in [12] 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, called *expected possible value* semantics, based on the semantics of McIver and Morgan. The report is finally concluded in Section 5. Appendix 6 contains several examples and proofs, mainly related to our contribution in Section 4.

2 Related Work

In [4], the authors consider a fuzzy imperative language [14, 15], 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 [26], the same authors explicitly build two semantics based on the previous tools, namely a fuzzy predicate transformer semantics and a state transformer semantics, for programs in the considered fuzzy language, and prove the equivalence between the two semantics. In [23], the authors develop other mathematical tools based on 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; more details about denotational semantics can be found in [22]) for programs including probabilities and nondeterminism.

3 State of the Art: the Language, and McIver and Morgan's Semantics

In this section, we present the pGCL language and the semantics of McIver and Morgan for weakest precondition calculus [12], and explain 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 [12], and includes a new instruction that GCL did not include: the probabilis-

tic choice. Therefore, the pGCL language allows the generation of probabilistic nondeterministic programs. It is defined by the following grammar[1][2]:

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

This language is sufficient to describe any probabilistic nondeterministic program. The `if...then...else...` and `while...do...` 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 primarily 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 to the probabilistic choice is that no probability can be assigned to S_1 or S_2. In this case, we only know that both executions are possible.
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 *satisfies* φ *if and only if* $\varphi(\sigma) = 1$.

[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 [5], but McIver and Morgan used an equivalent grammar and added probabilistic choice to get the grammar presented here.

S	$wp[S](\varphi)$
`skip`	φ
$x := A$	$\lambda\sigma.\varphi(\sigma[x/A])$
$S_1; S_2$	$wp[S_1](wp[S_2](\varphi))$
$\{S_1\}\square\{S_2\}$	$\lambda\sigma.\text{Min}(wp[S_1](\varphi)(\sigma), wp[S_2](\varphi)(\sigma))$
$\{S_1\}[p]\{S_2\}$	$\lambda\sigma.p \cdot wp[S_1](\varphi)(\sigma) + (1-p) \cdot wp[S_2](\varphi)(\sigma)$
`if` (b) `then` $\{S_1\}$ `else` $\{S_2\}$	$\lambda\sigma.[\![b : true]\!](\sigma) \cdot wp[S_1](\varphi)(\sigma) + [\![b : false]\!](\sigma) \cdot wp[S_2](\varphi)(\sigma)$
`while` (b) `do` $\{S\}$	lfp $(\lambda X.(\lambda\sigma.[\![b : true]\!](\sigma) \cdot wp[S](X)(\sigma) + [\![b : false]\!](\sigma) \cdot \varphi(\sigma)))$

Table 1: Rules for defining the predicate transformer wp

Let S be a pGCL program and let φ be a binary predicate. What is the probability that the execution of S in a state σ is guaranteed to terminate in a state satisfying the postcondition φ? McIver and Morgan extended Dijkstra's weakest precondition calculus, such that $wp[S](\varphi)(\sigma)$ is the answer, with the following definition:

Definition 2. *If S is a pGCL program, then the predicate transformer semantics of S is given by $wp[S]$ (defined by the rules[3] in Table 1).*
$wp[S]$ is a predicate transformer[4], i.e. for all postconditions φ, $wp[S](\varphi)$ is a predicate, such that $\forall\sigma \in \Sigma$, $wp[S](\varphi)(\sigma)$ is the probability that the execution of S in σ terminates in a state satisfying φ.

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

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, lfp $(\lambda X.f(X))$ denotes the least fixed-point of the function f (in this case, 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.

[4] Note that the rules allow the computation of $wp[S](\varphi)$ for all arbitrary predicates φ, but in the practice, we only want to compute $wp[S](\varphi)$ 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 $wp[S'](\psi)$

Remark 1. We will often simplify the expressions in the table by writing for instance $\mathrm{Min}(wp[S_1](\varphi), wp[S_2](\varphi))$ instead of $\lambda\sigma.\mathrm{Min}(wp[S_1](\varphi)(\sigma), wp[S_2](\varphi)(\sigma))$ or $p \cdot wp[S_1](\varphi) + (1-p) \cdot wp[S_2](\varphi)$ instead of $\lambda\sigma.p \cdot wp[S_1](\varphi)(\sigma) + (1-p) \cdot wp[S_2](\varphi)(\sigma)$. But recall that these objects are predicates.

Example 1. We can now apply these rules with the program P_0 given in Section 1. Recall that P_0 is the program $\{\{x := 2\}\square\{x := 5\}\}[p]\{x := 7\}$. Recall that we denoted by P_2 the left member $\{x := 2\}\square\{x := 5\}$. Let φ be any postcondition.

Looking at the rules in Table 1, one notices that, to compute $wp[P_0](\varphi)$, 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:

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

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

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

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

These are precisely the results stated in Section 1.

One can also apply these rules to the simpler program P_1 and get the results presented in Section 1. The calculation is done in Appendix 6.3.

3.3 Anticipating values, link with random variables

In practice, the 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 $[x \geq 5]$. One can say that the function $[y \leq -4]$ anticipates the value of $[x \geq 5]$ after the execution of P_1.

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. [24] However, due to certain constraints in order to ensure the existence of least fixed-points and make the computation possible, one must restrict ourselves to functions from Σ to $\mathbb{R}^{\infty}_{\geq 0}$ (where $\mathbb{R}^{\infty}_{\geq 0} = [0, +\infty]$). Let $\mathcal{F} = \{f \,|\, f : \Sigma \to \mathbb{R}^{\infty}_{\geq 0}\}$. [5] 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 probabilistic nondeterministic programs, by taking the weighted average between the two possible values. The rules of computation are exactly the same as in Table 1. But in this case, they are applied to a function $f \in \mathcal{F}$, and the result is another function $wp[S](f) \in \mathcal{F}$. (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 introduce 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, $\underline{x} \in \mathcal{F}$ is a function giving the value of x in each state.*

Example 2. 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:

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

[5] With this restriction, a chain-complete partial order can be defined on $\mathcal{F}$ (the inclusion of ∞ in the set of possible values is necessary for this definition) and Kleene's fixed point theorem can be applied. See Appendices 6.1 and 6.2. Note that ∞ is not only included to get a chain-complete partial order, it can be used as a possible value. One interesting function of $\mathcal{F}$, for example, is the running time of a program, possibly infinite, as in [10]

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 $x^2_{\sigma'} = 4$. Therefore, $(1-\underline{y})^2$ anticipates the value that $\underline{x}^2$ takes after the execution of the program.

Example 3. Consider the program P_0 from Section 1, we can anticipate the value of the variable x by computing the anticipated value of the function $\underline{x}$ (we should normally anticipate the value of $\underline{|x|}$, but we easily see that they are the same in this example):

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

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 probability p, and the value 7 with probability $1-p$.

Remark 3. The previous remark can be generalized this way: Let S be a probabilistic program (without nondeterminism), and let $f \in \mathcal{F}$ be a function. For each initial state σ, the execution of S in σ 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, S terminated, or is looping infinitely. Let Ω_σ be the set of execution schemes obtained when executing S 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 S in σ as a positive random variable $X_\sigma : \Omega_\sigma \to \mathbb{R}^\infty_{\geq 0}$.
Thus, $\boxed{wp[S](f) = \lambda\sigma.E(X_\sigma).}$ This result can be more clearly observed in Example 4 in Appendix 6.4. A proof can be deduced from the rules in Table 1, notably from the probabilistic choice rule.

Remark 4. The previous remark can be extended to probabilistic nondeterministic programs by applying rules in Table 1 and by considering the nondeterministic choice as the demonic choice (which is how we were able to compute $wp[P_0](\underline{x})$). Note however that the Ω_σ that we obtain for each σ depends on the postcondition we are computing (as shown in Example 5 in Appendix 6.4), because the demonic nature in nondeterministic choice will restrict each of these choices to one outcome, depending on the postcondition.

Remark 5. The result of the computation of $wp[P_0](\underline{x})$ still does not take into account the possibility that x can take the value 5. Analogously, if the nondeterministic choice was to be considered as angelic choice, we would get that $wp[P_0](\underline{x}) = 5p + 7(1-p)$ and the value 2 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 unsatisfying results, both when computing weakest preconditions of predicates in probabilistic nondeterministic programs and when anticipating values of functions in probabilistic nondeterministic programs. Our goal is to define a new semantics, which should be based on possibility theory, and represents each possible outcome more clearly. Recall from Section 3 that the weakest-precondition calculus can 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 a *Fuzzy random variable* (introduced in [16], and simplified in [19]).

Remark 6. In the following, we will consider ∞ as a positive value, and will write "positive values" for "finite positive values or ∞".

4.1 Fuzzy random variables

We adapted the following definition from [16].

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

In [16], 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 are taking as values only sets of positive values (as we previously defined them). Then, by using the definition of expected value in [16] 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 $P(\mathbb{R}^{\infty}_{\geq 0})$), then we can define the expected value $E(X)$ as follows:*

$$E(X) = \left\{ \sum_{n=1}^{+\infty} t_n Pr(\omega_n) \,|\, \forall n \geq 1, t_n \in X(\omega_n) \right\} \subset \mathbb{R}^{\infty}_{\geq 0}$$

One can interpret 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 $X(\omega_n)$).

4.2 Contribution: anticipating values of set-valued functions, link with fuzzy random variables

Let $\mathcal{F}' = \{f \,|\, f : \sigma \to P(\mathbb{R}^{\infty}_{\geq 0}) \backslash \{\emptyset\}\}$ be a new set of functions. We want to anticipate the values of these functions after the execution of a program. What it intuitively means is that f maps each σ to a non-empty set of "possible values".

Notation 2 *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).*

Our suggestion is to replace the *wp* operator by the following operator:

Definition 5. *If S is a pGCL program, then our* expected possible value *semantics of S is given by $ev[S]$ (defined by the rules in Table 2).*
$ev[S]$ is a function transformer, i.e. if $f \in \mathcal{F}'$, then $ev[S](f) \in \mathcal{F}'$.

However, we have not proven yet that ev is well defined in the presence of loops (it is the only problematic point, since it requires the computation of least-fixed points). This requires the existence of a chain-complete partial order over $\mathcal{F}'$, and that for all programs S, $ev[S]$ is monotonic (more details in Appendix 6.1). However, in programs without loops, we do not need such a structure, and we can directly use ev, as we do in this section.

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

Consider the running example P_0: $\{\{x := 2\} \square \{x := 5\}\}[p]\{x := 7\}$. The value of x after the execution of P_0 in a given initial state is given by a fuzzy random variable. We want to compute its expected value. Thus, we have to

S	$ev[S](f)$
`skip`	f
$x := A$	$\lambda\sigma.f(\sigma[x/A])$
$S_1; S_2$	$ev[S_1](ev[S_2](f))$
$S_1[p]S_2$	$\lambda\sigma.\{t_1p + t_2(1-p) \mid t_1 \in ev[S_1](f)(\sigma), t_2 \in ev[S_2](f)(\sigma)\}$
$\{S_1\}\Box\{S_2\}$	$\lambda\sigma.ev[S_1](f)(\sigma) \cup ev[S_2](f)(\sigma)$
`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` $\{S\}$	$\text{lfp}\ (\lambda X.(\lambda\sigma.\{[\![b : true]\!](\sigma) \cdot t_1 + [\![b : false]\!](\sigma) \cdot t_2 \mid$ $t_1 \in ev[S](X)(\sigma), t_2 \in f(\sigma)\}))$

Table 2: Rules for defining the transformer ev

determine $ev[P_0](\underline{x})$. It is a function of σ, but the form of P_0 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,

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

Contrary to what was observed with the semantics of McIver and Morgan, no information is lost here. The result 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 from 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 probability p (corresponding to the first execution scheme, where the left branch of the probabilistic choice is executed. In this case, x has 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 probability $1 - p$.

Finally, we can explain how our semantics is related to possibility theory. Recall that McIver and Morgan extended wp to probabilistic nondeterministic 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 predicate transformer semantics of pGCL programs, detailed by McIver and Morgan in [12]. We explained how the weakest-precondition calculus could be used in order to anticipate the values of particular functions, and explained the link with random variables, i.e. the value of a function after the execution of a program in a given initial state is a random variable, and the weakest precondition calculus gives access to its expected value. We underlined that nevertheless, the results obtained with the semantics of McIver and Morgan are 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 sets of positive real numbers. We explained the link between our semantics and the expected value of fuzzy random variables.
However, our semantics is defined by an operator which is not yet proven to be well-defined for programs containing loops. This requires the definition of a chain-complete partial order over $\mathcal{F}'$ which makes the function transformer $ev[S]$ monotonic for each program S (see Appendix 6.1 for more details). Up to now, our attempts (detailed in Appendix 6.5) to build such an order failed. There still exists one solution, consisting in restricting the possible values of the functions to downward closed sets, and build a pointwise order based on inclusion, but this solution builds in practice a new set of functions isomorphic to $\mathcal{F}$, and the

resulting semantics suffers from the same loss of information as in the semantics of McIver and Morgan.
Therefore, the most important part of the future work will consist in finding a chain-complete partial order over $\mathcal{F}'$, verifying the above-mentioned properties, in order for our semantics to be well-defined.

References

1. Parul Agarwal and H.S. Nayal. *Possibility Theory versus Probability Theory in Fuzzy Measure Theory.* 2015.
2. Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction.* 1997.
3. Elwyn R. Berlekamp. *Factoring Polynomials Over Finite Fields.* 1967.
4. Yixiang Chen and Hengyang Wu. *Domain Semantics of Possibility Computations.* 2008.
5. E.W. Dijkstra. *A Discipline of Programming.* 1976.
6. R. Freivalds. *Probabilistic Machines Can Use Less Running Time.* 1977.
7. G. Gierz, K.H. Hofmann, K. Keimel, J.D. Lawson, M.W. Mislove, and D.S. Scott. *A Compendium of Continuous Lattices.* 1980.
8. C.A.R. Hoare. *Algorithm 64: Quicksort.* 1961.
9. Gerard J. Holzmann. *The Model Checker Spin.* 1997.
10. Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Federico Olmedo. *Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs.* 2016.
11. Jacques Loeckx and Kurt Sieber. *The Foundations of Program Verification, 2nd Edition.* 1987.
12. Annabelle McIver and Carroll Morgan. *Abstraction, Refinement and Proof for Probabilistic Systems.* 2005.
13. Filip Moons. *The Scott Topology (Bachelor Thesis II).* 2013.
14. Rafael Morales-Bueno, Ricardo Conejo, José Luis Pérez de la Cruz, and Buenaventura Clares. *An elementary fuzzy programming language.* 1993.
15. Rafael Morales-Bueno, José Luis Pérez de la Cruz, Ricardo Conejo, and Buenaventura Clares. *A family of fuzzy programming languages.* 1997.
16. Madan L. Puri and Dan A. Ralescu. *Fuzzy Random Variables.* 1986.
17. Michael O. Rabin. *Probabilistic Algorithm for Testing Primality.* 1977.
18. Murali Krishna Ramanathan, Ronaldo A. Ferreira, Suresh Jagannathan, and Ananth Y. Grama. *Randomized Leader Election.* 2004.
19. Arnold F. Shapiro. *Fuzzy Random Variables.* 2009.
20. Robert.M Solovay and Volker Strassen. *A fast Monte-Carlo test for primality.* 1977.
21. V. Stoltenberg-Hansen, I. Lindstrom, and E. R. Griffor. *Mathematical Theory of Domains.* 1994.
22. Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* 1977.
23. Regina Tix, Klaus Keimel, and Gordon Plotkin. *Semantics Domain for Combining Probability and Non-Determinism.* 2009.
24. Glynn Winskel. *The Formal Semantics of Programming Languages.* 1993.
25. Hengyang Wu and Yixiang Chen. *The Semantics of wlp and slp of Fuzzy Imperative Programming Languages.* 2011.
26. Hengyang Wu and Yixiang Chen. *Semantics of Non–Deterministic Possibility Computation.* 2012.

6 Appendix

6.1 Kleene's Fixed point theorem

The following definitions can be found in [7].

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

Definition 6. (Chain-complete partial order) $(P, \leq)$ *is a* chain-complete partial order *if the following conditions are satisfied:*

- $\leq$ *has a minimal element, denoted* $\bot$ *(such that* $\forall x \in P,\ \bot \leq x$*)*
- *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 1. (Kleene's fixed-point theorem) *If* P *is a chain-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* $\mathrm{lfp}(f) = \mathrm{Sup}_{n\in\mathbb{N}}(f^n(\bot))$

In fact, Kleene's Theorem has more general hypotheses (such as f being a mapping from a chain-complete partial order P to a chain-complete partial order Q and being Scott-Continuous). A proof of this more general theorem can be found in [21], and clearer explanations can be found in [13]. However, in our case, when P and Q are the same chain-complete partial order, the monotonicity of f is sufficient. A proof of this latter point can be found in [11]

6.2 Chain-complete partial order over the set of predicates and set $\mathcal{F}$ of functions 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 lifted from* $\leq$, 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 chain-complete partial order:

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

Proof. – One can easily verify that $\bot$ 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 $\{f_n(\sigma) \mid n \in \mathbb{N}^*\}$ 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 $\sigma \in \Sigma$, $f_n(\sigma) \leq \mathrm{Sup}_{n\in\mathbb{N}^*}(f_n(\sigma)) = f(\sigma)$. 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, $\forall \sigma \in \Sigma$, $f(\sigma) = \text{Sup}_{n \in \mathbb{N}^*}(f_n(\sigma)) \leq g(\sigma)$. Hence, $f \leq g$. Therefore, f is the least upper bound of S

Remark 7. 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 chain-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.

In order for the weakest-precondition semantics to make sense, it is sufficient that the operator wp is monotone and preserves Sups. A proof is available in [2] for wp over GCL programs, and can be extended to pGCL.

Knowing that $\mathcal{P}$ is a chain-complete partial order and that, for all S, the operator $wp[S]$ is monotone, then we can prove that the function whose least fixed point we want to compute (when S includes a loop) is monotone, and as it is a function over $\mathcal{P}$, then the least fixed-point exists. Therefore, the computation of $wp[S]$ with the rules presented in Table 1 is always possible.
With the previous remark, this result can be extended to $\mathcal{F}$.

6.3 Application of the semantics McIver and Morgan to P_1

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

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

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:

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

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:

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

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 Attempts of defining a complete partial order over $\mathcal{F}'$ in Section 4

Recall that we defined $\mathcal{F}' = \{f : \mid f : \Sigma \to \mathcal{P}(\mathbb{R}^\infty_{\geq 0})\backslash\{\emptyset\}\}$, and we defined the semantics of the pGCL program P as a transformer $ev[P]$, 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 chain-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)

The monotonicity in the second point and the existence of a chain-complete partial order both require the existence of a partial order over $\mathcal{F}'$. Here are our attempts, and explanations of their failure in satisfying the two properties mentioned above.

First attempt: The pointwise order based on inclusion. In this attempt, instead of considering functions mapping elements of Σ to non-empty subsets of $\mathbb{R}^\infty_{\geq 0}$, we considered functions mapping elements of Σ to arbitrary (and then possibly empty) subsets of $\mathbb{R}^\infty_{\geq 0}$. We therefore had another set of functions $\mathcal{F}'' = \{f \mid f : \Sigma \to P(\mathbb{R}^\infty_{\geq 0})\}$.

The easiest way to build a partial order over $\mathcal{F}''$ is to build a partial order over $P(\mathbb{R}^\infty_{\geq 0})$ and to lift this order into a pointwise order over $P(\mathbb{R}^\infty_{\geq 0})$. The inclusion $\subset$ is obviously a partial order over $P(\mathbb{R}^\infty_{\geq 0})$. Therefore, we can consider the following relation:

$$f \leq_1 g \text{ iff } \forall \sigma \in \Sigma, f(\sigma) \subset g(\sigma)$$

One can easily see that this defines a partial order over $\mathcal{F}''$, and one can prove that it defines a complete partial order over $\mathcal{F}''$; we will however not detail this proof, because it is long and because the problem does not reside in this first property.

In fact, one can easily see that, for this partial order, the minimal element is $\bot = \lambda\sigma.\emptyset$, since $\emptyset$ is included in each subset of $\mathbb{R}^\infty_{\geq 0}$. However, the rules in Table 2 are not well defined when considering $\bot$ as a postcondition. Nevertheless, they have to be well defined, since the computation of the least fixed-point in the `while`. The main problem resides in the fact that, when constructing a set containing elements of the form $[\![b : true]\!](\sigma)t_1 + [\![b : false]\!](\sigma)t_2$, with t_1 and t_2 belonging in certain sets, then what happens when the set where we should pick t_1 (or symmetrically t_2) is empty ?

We can decide that, in this case, the set containing all elements of the form $[\![b : true]\!](\sigma)t_1 + [\![b : false]\!](\sigma)t_2$ is also empty. But in this case, when computing a least fixed-point by applying Kleene's Theorem, we start with $\bot$, and each time that we apply the function whose least-fixed point we seek to $\bot$, the result is $\bot$. Therefore, each least fixed-point that we compute is $\bot$, which is not what we expect.

Second attempt: no empty set. We tried to solve the problem by considering $\mathcal{F}'$ as we did in Section 4, i.e. by considering functions mapping states to non-empty subsets of $\mathcal{F}'$. In this case, the previous relation $\leq_1$ remains a partial order.

However, it does not define a chain-complete partial order anymore, because there is no minimal element. This can be proven by showing that, for each $f\mathcal{F}'$, there exists $g \in \mathcal{F}'$ such that $f \leq_1 g$ does not hold.

Proof. Let $f \in \mathcal{F}'$. We will build a function g as follows: Let $\sigma_0 \in \Sigma$.

- $\forall \sigma' \neq \sigma_0$, $g(\sigma') = f(\sigma')$
- If $f(\sigma_0)$ is a singleton $\{x\}$, then $g(\sigma_0) = \{x + 1\}$
- If $f(\sigma_0)$ is not a singleton, then we can tell that it contains at least two elements x and y, because $f(\sigma_0)$ is non-empty. Then, we can decide for instance that $g(\sigma_0) = f(\sigma_0) \backslash \{x\}$.

In all cases, we get that $\forall \sigma \in \Sigma$, $g(\sigma)$ is not empty, and all its elements are positive real numbers (or the infinity). Therefore, $g \in \mathcal{F}'$. However, by construction, $f \leq_1 g$ does not hold, because there exists one state (namely, σ_0) for which $f(\sigma_0) \not\subset g(\sigma_0)$.
Therefore, f cannot be the minimal element for $\leq_1$. As this holds for all $f \in \mathcal{F}'$, there exists no minimal element for $\leq_1$ in $\mathcal{F}'$.

Third attempt: downward closures. In this case, we consider another relation inspired from the pointwise order, but where the inclusion must hold on downward closures.

Definition 7. (Downward closure)
Let $U \subset \mathbb{R}^{\infty}_{\geq 0}$. Then, the downward closure *of U, denoted by $\downarrow U$, is a subset of $\mathbb{R}^{\infty}_{\geq 0}$ such that:*

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

In other terms, $\downarrow U$ contains exactly all elements of U and all elements of $\mathbb{R}^{\infty}_{\geq 0}$ inferior to one element of U.

Consider the following relation:

$$f \leq_2 g \text{ iff } \forall \sigma \in \Sigma, \downarrow (f(\sigma)) \subset \downarrow (g(\sigma))$$

This relation is supposed to counter the default of the previous relation of having no minimal element. In this case, we easily see that the minimal element is $\bot = \lambda\sigma.\{0\}$. However, this relation does not even define a partial order over $\mathcal{F}'$, because it is not antisymmetric.

This can easily be seen by the fact that $\downarrow \{2,5\} = \downarrow \{5\}$, but $\{2,5\} \neq \{5\}$. However, it is interesting to note that we can build a partial order with this relation over another set of functions.

Let $\downarrow P(\mathbb{R}^{\infty}_{\geq 0})$ denote $\{\downarrow U \,|\, U \in P(\mathbb{R}^{\infty}_{\geq 0})\}$, the set of downward closures of all subsets of $\overline{\mathbb{R}}^{\infty}_{\geq 0}$. It is possible to use the previous relation to build a chain-complete partial order over the set of functions $\mathcal{F}_{\downarrow} = \{f : \,|\, f : \Sigma \to \downarrow P(\mathbb{R}^{\infty}_{\geq 0})\}$. However, this leads to a lack of information, as some possibilities cannot be represented. For example, it would be impossible in this case to obtain the result that we got for P_0 in Section 4. This is linked to the fact that $\mathcal{F}_{\downarrow}$ is isomorphic to the set $\mathcal{F}$ presented in Section 3, and therefore suffers from the same loss of information.

Fourth attempt: surjective mapping. Another possibility (which seems complex, but is in fact often used to define chain-complete partial orders on other sets) is to consider the following relation over $\mathcal{F}'$ (we keep the property that our functions must not map any state to the empty set):

$$f \leq_3 g \text{ iff } \forall \sigma \in \Sigma, \exists \alpha : g(\sigma) \to f(\sigma) \text{ surjective, such that } \forall r \in g(\sigma), \alpha(r) \leq r$$

However, this does not even define a partial order. One can prove that this relation is reflexive and transitive. However, it is not antisymmetric:

Proof. Let $f, g \in \mathcal{F}'$ such that, for one $\sigma \in \Sigma$, we have:

- $f(\sigma) = \{\dots \frac{1}{6}, \frac{1}{4}, \frac{1}{2}, 1, 2, 3, 4 \dots\}$
- $g(\sigma) = \{\dots \frac{1}{7}, \frac{1}{5}, \frac{1}{3}, 1, 2, 3, 4 \dots\}$

In practice, this is the same as considering two sequences $(f_n)_{n\in\mathbb{Z}}$ and $(g_n)_{n\in\mathbb{Z}}$ such that:

$$\forall n \in \mathbb{Z}, f_n = \begin{cases} \frac{1}{2|n|} & \text{if } n \leq -1 \\ n+1 \text{ if } & n \geq 0 \end{cases} \tag{1}$$

$$\forall n \in \mathbb{Z}, g_n = \begin{cases} \frac{1}{2|n|+1} & \text{if } n \leq -1 \\ n+1 \text{ if } & n \geq 0 \end{cases} \tag{2}$$

With these notations, we have $f(\sigma) = \{f_n \,|\, n \in \mathbb{Z}\}$ and $g(\sigma) = \{g_n \,|\, n \in \mathbb{Z}\}$. Now, we shall consider the following mappings:

$$\alpha : \begin{cases} g(\sigma) \to f(\sigma) \\ g_n \quad \to \quad f_{n-1} \end{cases} \tag{3}$$

$$\beta : \begin{cases} f(\sigma) \to g(\sigma) \\ f_n \quad \to \quad g_{n-1} \end{cases} \tag{4}$$

We easily see that, by construction, α and β are both surjective, and one can verify with the expressions of f_n and g_n that $\forall n \in \mathbb{Z}$, $f_{n-1} = \alpha(g_n) \leq g_n$ and $g_{n-1} = \beta(f_n) \leq f_n$. Therefore, $\forall r \in f(\sigma)$, $\beta(r) \leq r$ and $\forall r \in g(\sigma)$, $\alpha(r) \leq r$.
Hence, $f \leq_3 g$ and $g \leq_3 f$. However, for the σ previously defined, we have $f(\sigma) \neq g(\sigma)$, and therefore, $f \neq g$.
Thus, $\leq_3$ is not a partial order (and as a consequence, it is not possible to use it to build a chain-complete partial order).