Probabilistic unifying relations for modelling epistemic and aleatoric uncertainty: semantics and automated reasoning with theorem proving*

Kangfeng Ye^{a,*}, Jim Woodcock^a, Simon Foster^a

^aDepartment of Computer Science, University of York, Deramore Lane, Heslington, YO10 5GH, York, United Kingdom

Abstract

Probabilistic programming combines general computer programming, statistical inference, and formal semantics to help systems make decisions when facing uncertainty. Probabilistic programs are ubiquitous, including having a significant impact on machine intelligence. While many probabilistic algorithms have been used in practice in different domains, their automated verification based on formal semantics is still a relatively new research area. In the last two decades, it has attracted much interest. Many challenges, however, remain. The work presented in this paper, probabilistic unifying relations (ProbURel), takes a step towards our vision to tackle these challenges.

Our work is based on Hehner's predicative probabilistic programming, but there are several obstacles to the broader adoption of his work. Our contributions here include (1) the formalisation of its syntax and semantics by introducing an Iverson bracket notation to separate relations from arithmetic; (2) the formalisation of relations using Unifying Theories of Programming (UTP) and probabilities outside the brackets using summation over the topological space of the real numbers; (3) the constructive semantics for probabilistic loops using Kleene's fixed-point theorem; (4) the enrichment of its semantics from distributions to subdistributions and superdistributions to deal with the constructive semantics; (5) the unique fixed-point theorem to simplify the reasoning about probabilistic loops; and (6) the mechanisation of our theory in Isabelle/UTP, an implementation of UTP in Isabelle/HOL, for automated reasoning using theorem proving.

We demonstrate our work with six examples, including problems in robot localisation, classification in machine learning, and the termination of probabilistic loops.

Keywords: formal semantics, fixed-point theorems, predicative programming, UTP, probabilistic models, probabilistic programs, probabilistic distributions, formal verification, quantitative verification, automated reasoning, theorem proving, Isabelle/HOL, robotics, machine learning, classification

1. Introduction

Motivations. Probabilistic programming combines general computer programming, statistical inference, and formal semantics to help systems make decisions when facing uncertainty. Probabilistic programs are ubiquitous but are particularly important in machine intelligence applications. Probabilistic algorithms have been used in practice for a long time in autonomous robots, self-driving cars, and artificial intelligence. There are tools for automated formal verification, particularly model checkers. However, many challenges remain, such as the following. What is the mathematical meaning of a probabilistic program? Are probabilistic programming languages expressive enough to capture rich features in real-world applications, such as epistemic and aleatoric uncertainty, discrete and continuous distributions, and real-time? How can we

^{*}This document is the results of the research project RoboTest (https://robostar.cs.york.ac.uk/) funded by EPSRC.

^{*}Corresponding author

Email addresses: kangfeng.ye@york.ac.uk (Kangfeng Ye), jim.woodcock@york.ac.uk (Jim Woodcock), simon.foster@york.ac.uk (Simon Foster)

compare two programs? How can we implement a probabilistic specification as a probabilistic program? Can formal verification be largely automated and scaled to large systems without sacrificing accuracy? Does a probabilistic program almost surely terminate? What is the expected runtime of this program? The work presented in this paper, probabilistic unifying relations (ProbURel), takes a step towards our vision to tackle these challenges.

Uncertainty is essential to cyber-physical systems, particularly in autonomous robotics where we work. Such systems are subjected to various uncertainties, including real-world environments and physical robotic platforms, which present significant challenges for robots. Robots are usually equipped with probabilistic control algorithms to deal with these uncertainties. For example, a modern robot may use SLAM for localisation and mapping, the value iteration algorithm for probabilistic planning [1].

Probabilistic algorithms are intrinsically more difficult to program and analyse than non-probabilistic algorithms. For a specific input, the output of a probabilistic algorithm might be a distribution of possible outputs, not just a single output. Outputs with low probabilities are rare, which makes testing difficult. Probabilistic behaviour may be challenging to capture and model correctly because assumptions may not be obvious and may be left implicit. We need more precision in understanding an autonomous robot that may impose safety-critical issues on humans and their environments. One approach to addressing these challenges is providing probabilistic programs with formal syntax, semantics, and verification techniques to ensure they behave as expected in a real-world environment.

According to Gordon et al. [2], probabilistic programming includes two basic constructs to draw values from probabilistic distributions such as uniform distributions and condition values of variables. Probabilistic inference is the problem in probabilistic programming to compute explicit probability distributions or to compute relevant probabilities for particular events from probabilistic programs.

Examples. We illustrate a few examples and informally discuss their modelling and inference.

Example 1.1 (The (forgetful) Monty Hall problem). The problem [3] is a puzzle based on an American television game show *Let's Make a Deal*. It is named after its original host, Monty Hall. Suppose you are the contestant and are given a choice of opening one of three doors. Behind one door is a car, and behind the others are goats. You pick a door, say No. 1, and the host, who knows what is behind each door, opens another door, say No. 3 and reveals a goat. He then asks, "Do you want to pick door No. 2 instead of your original choice?" The problem is simple: should you change your choice to maximise your chance to win a car?

To model this problem, we define three variables p, c, and m for the door number having the prize (car), the contestant chooses, and the Monty chooses. We model the problem below.

```
p := rand({0..2}); c := rand({0..2});//Prize and contestant's choice are random
if(p=c) { // If the contestant's choice is the prize,
    m := (c+1)%3 pc{1/2} m := (c+2)%3; // Monty randomly chooses other two doors
} else {
    m := 3-c-p; // Monty chooses another door which has no prize
}
c := 3-c-m; // If the contestant changes the choice
```

We use two constructs: rand(S) to draw a random value from set S and P pc{r} Q, a binary probabilistic choice, to choose P with probability r and Q with probability 1 - r.

The last line corresponds to the strategy for the contestant to change the initial choice. The question becomes "which strategy will have the higher winning (c=p) probability?"

Suppose now that Monty forgets which door has the prize behind it. He opens either of the doors not chosen by the contestant. The contestant switches their choice to that door if the prize is revealed (m = p). So the contestant will surely win. However, should the contestant switch if the prize is not revealed $(m \neq p)$? In this forgetful Monty, the new knowledge $(m \neq p)$ is learned.

Accordingly, we model the forgetful Monty problem below.

We introduce another construct $P \mid \mid Q$ to model the new knowledge (encoded in Q, e.g. $m \mid = p$ denoting the prize is not revealed) learned after P is executed. So the question becomes how the distribution is updated after learning the new evidence? What is the winning probability?

Example 1.2 (Classification - COVID-19 diagnosis). We consider people using a COVID-19 test to diagnose if they may or may not have contracted COVID-19. The test result is binary and could be positive or negative. The test, however, is imperfect. It doesn't always give a correct result.

We model the prior, the test, and the first test result positive below.

```
1 {
2    c := True pc{p1} c := False; // Prior probability of a person having COVID
3    // A test
4    if c { ct := Pos pc{p2} ct := Neg; } // True positive and True negative
5    else { ct := Pos pc{p3} ct := Neg; } // False positive and False negative
6 } | | (ct = Pos) // Learn the result is positive
```

In the program, c and ct denotes if a person has COVID or not, and the test result; and p1, p2, and p3 are parameters of this program. We are interested in several questions. How likely is a randomly selected person to have COVID-19 if the first test result is positive? Is it necessary to have the second test to reassure the result?

Taken the second test into account, the new program is as follows.

But how much can the second test contribute to the diagnosis? How the result changes if the parameters are changed?

Example 1.3 (Robot localisation (RL)). A circular room has two doors and a wall. A robot with a noisy door sensor maps position to *door* or *wall*. Doors are at positions 0 and 2; position 1 is a blank wall. We introduce a program variable $bel \in \{0...2\}$ to denote the position of the robot that we believe. When the reading of the door sensor is door, it is four times more likely to be right than wrong and likewise when the reading is wall.

The following program models two sensor readings and one movement in between the readings.

The door(bel) (or wall(bel)) is a function returning 1 if the bel is 0 or 2 (or 1) and returning 0 otherwise. We are interested in questions like how many measurements and moves are necessary to estimate the robot's location accurately.

Example 1.4 (Flip a coin till heads). We consider the simplest probabilistic program with a loop: flip a coin until the outcome is heads, defined as follows.

```
c := heads;
while (c=tail) {
   c := heads pc{p} c:= tail;
}
```

The p above is a parameter denoting the probability of getting a heads for a coin flip. It is 1/2 for a fair coin. Does this program terminate? What is the probability distribution on termination? How is the distribution related to p? What is the semantics of this program? What is its expected runtime?

Example 1.5 (Throw a pair of dice). This example [4] is about throwing a pair of dice till they have the same outcome. We model it below.

```
while (d1 != d2) {
    d1 := rand({1..6});
    d2 := rand({1..6});
}
```

This is slightly complex than the coin program in Example 1.4 because two variables are declared. Does this program terminate? What is the probability distribution on termination? What is the semantics of this program? Is it still as simple as the coin example? What is its expected runtime?

Example 1.6 (One-dimensional simple random walk (SRW)). Grimmett and Welsh [5] defined various random walks. A random walk is *simple* if at each time step it can move only to its next (or neighbouring) positions randomly in one of the lattice directions. A *symmetric* simple random walk has the equal probability for each direction. It is also the Gambler's Ruin Problem with an absorbing barrier at 0. We model it as a probabilistic program below.

In the program, m and p are parameters. The program with p=1/2 (that is, symmetric) is widely studied, for example, in [6, 7, 8, 9]. Unlike the coin and dice examples where each experiment (flip a coin or throw a pair of dice) is independent, each experiment in this example is not independent because the value of x is updated.

Does this program terminate? How does the termination relate to the parameter p? What is the probability distribution on termination? What is the semantics of this program? What is its expected runtime?

There are several challenges to model and answer the questions of these programs: (1) the capability to model the learning process using conditional probability and joint probability as used in the Bayesian approach, (2) the reasoning about probabilistic loops to give them a precise semantics (probability invariant) and their termination, (3) the inference to get exact probability distributions and exact expected runtime, especially for programs with loops, and (4) the guarantee of the correctness of the analysis. A considerable amount of literature has been published on addressing these problems, but none of them can address all these challenges.

McIver and Morgan's weakest pre-expectation [10] is based on the pGCL [11, 10], an extension of Dijkstra's Guarded Command Language (GCL) [12] with a probabilistic choice construct. It is mechanised in High-Order Logic (HOL) [13] by Hurd et al. [14], enabling verification of partial correctness of probabilistic programs. However, pGCL does not support conditional probability, so it cannot model the examples: the forgetful Monty, COVID, and the robot localisation, presented in Examples 1.1 to 1.3.

Based on the weakest pre-expectation semantics, Kaminski [15] developed an advanced weakest precondition calculus which supports conditioning in cpGCL [16] using an **observe** statement and expected
runtimes in the calculus [17]. The observe statement, however, conditions only boolean expressions. It
cannot support the general likelihood functions as discussed in Example 1.3 where the two functions are
real-valued expressions. The expected runtime analysis [18] is based on upper-bounds. It cannot reason
about the exact expected runtimes which requires the reasoning of exact probability distributions.

Barthe et al. [19] presented ELLORA, an assertion-based logic for probabilistic programs, implemented in the EasyCrypt theorem prover [20]. However, pGCL does not support conditional probability, so it cannot model the examples: the forgetful Monty, COVID, and the robot localisation.

Schröer et al. [21] developed expectation-based reasoning using a deductive verification infrastructure, based on the weakest pre-expectation semantics. Similarly, it cannot reason about the exact probability distributions and the expected runtimes. For example, the random walk is verified to be almost-surely terminated, but without its semantics or exact distributions. It is also not able to model the general likelihood functions.

Hehner's probabilistic predicate programming (PPP) [22, 4] can model and reason about all these examples. But PPP is not formalised and implemented in any tool for automated verification.

The work presented in this paper aims to support modelling and analysis of these probabilistic programs and answer the questions which we are interested in. Additionally, as discussed later in Sect. 8.1, we aim to pursue a probabilistic semantic framework (1) having an expressive language with rich semantics to model systems not only from abstract specification level but also concrete implementation level; (2) able to unify different probabilistic models and programmings, so their tools can be integrated; (3) extendible to support more features like more discrete distributions, nondeterminism, continuous distributions, time, communication and concurrency, because these features are essential in modelling robotic applications; (4) providing a practical and decidable method to approximate the semantics for probabilistic loops because it is non-trivial to construct an invariant and prove it for a loop; and (5) most importantly, supporting theorem proving because these programs usually have unbounded variables and infinite state space.

Our approach. Our previous work probabilistic RoboChart [23] and probabilistic designs [24] model aleatoric uncertainty describing the natural randomness of physical processes. Another category is epistemic uncertainty due to the lack of knowledge of information, which is reducible by gaining more knowledge. Usual probabilistic choice can model aleatoric uncertainty but not epistemic uncertainty because it requires the capability to update distributions or beliefs after learning new knowledge. To model this process, for example, in the Bayesian approach, conditional and joint probabilities should be supported. In this paper, we present a probabilistic programming language, called probabilistic unifying relations (ProbURel), based on Hehner's probabilistic predicative programming [22, 4], to model both aleatoric and epistemic uncertainty. This programming uses the subjective Bayesian approach to reason about epistemic uncertainty.

In Hehner's original work [4], a probabilistic program is given relational semantics, and its syntax is a mixture of relations and arithmetic. The presentation of syntax and semantics in the paper is not formal. For example, the semantics of a probabilistic ok (skip) is given as $ok = (x' = x) \times (y' = y)$. There is a benefit to introducing semantics using examples, but it lacks formalisation. The operators like = and \times are not formally defined, and the types for variables and expressions are not given. The lack of this information makes the paper not easily accessible to readers, particularly for researchers aiming to use the work for automated reasoning of probabilistic programs. Therefore, our first contribution to this paper is formalising its syntax and semantics. We introduce a notation called Iverson brackets, such as [r], to establish a correspondence between relations r and arithmetic (0 or 1). For ok, we could formalise it as [v' = v] where v denotes the state space (composed of all variables) of a program. This notation separates relations (v' = v) with arithmetic, so expressions and operators in a program all have clear meanings or definitions depending on their contexts (relations or arithmetic) where the contexts can be easily derived because of the separation.

In addition to syntax and semantics, the semantics for probabilistic loops are not formally presented and argued. Hehner proposed a more straightforward but more potent (than total correctness) approach [25]: partial correctness + time to deal with the termination of loops (for conventional programs) with extra

information about run time. His approach introduces a time variable t with a healthiness condition, strict incremental for each iteration. The variable t can be discrete or continuous and is an extended natural or real number to have ∞ for nontermination. The same approach is also applied to probabilistic programs [4], but the partial correctness of probabilistic loops is not formally reasoned about. For this reason, our second contribution in this paper is to bridge the semantic gap for probabilistic loops by establishing its semantics using fixed-point theorems: specifically Kleene's fixed-point theorem, to construct the fixed points using iterations. The advantage of having an iterative (or constructive) fixed point includes both theoretical semantics and practical computation or approximation. A hint of this would be possible to verify probabilistic loops using both theorem proving and model checking (based on approximation).

To give the semantics using the fixed-point theorem, we define a complete lattice ($[0,1], \leq$) over the real unit interval (the real numbers between 0 and 1 inclusive). We restrict to the unit interval simply because probability values are between 0 and 1. To apply Kleene's theorem, we prove the loop function is Scott-continuous for the state space in which only finite states have positive probabilities. Then we define the semantics of loops as the least fixed point (lfp) of the function where lfp can be calculated iteratively as the supremum of the ascending Kleene chain (from the bottom of the complete lattice). This bridges the semantics gap, but it is still challenging to calculate lfp because the chain is infinite. We, therefore, also present the strongest fixed point (gfp) of the function, calculated iteratively as the infimum of the descending Kleene chain (from the top of the complete lattice) of the function. We prove a unique fixed point theorem where lfp and gfp are the same based on particular assumptions. The unique fixed point theorem makes reasoning about loops much more accessible because it is unnecessary to calculate lfp. Instead, a fixed point must be constructed and proved with the unique fixed point theorem. This, eventually, is consistent with the loop semantics using Hehner's more straightforward approach. In particular, our semantics can be mechanised and automated.

In Kleene's theorem, the ascending and descending chains start from a pointwise constant function 0 and 1 (the bottom and the top of the complete lattice). The two pointwise functions are not distributions (where probabilities of the state space sum to 1). Indeed, the pointwise function 0 is a subdistribution (the probabilities sum to less than or equal to 1), and the pointwise function 1 is a superdistribution (the probabilities sum to larger than 1). For this reason, our third contribution is to extend the semantic domain of the probabilistic programming language from distributions to subdistributions and superdistributions. Eventually, constructs like conditional, probabilistic choice, and sequential composition will not be restricted to programs that are distributions. This brings us to the required semantics to use Kleene iterations for the semantics of loops.

The introduction of Iverson brackets also has another benefit: the relations inside the brackets can be easily characterised using alphabetised relations in UTP because relations in both Hehner's work and UTP are of the predicative style. Relations in our probabilistic programs are indeed UTP alphabetised relations, which allows us to reason about probabilistic programs using the existing theorem prover Isabelle/UTP for UTP. Our final contribution, therefore, is to mechanise the semantics of the probabilistic programming language in Isabelle/UTP. Our reasoning is primarily automated thanks to the various relation tactics in Isabelle/UTP. Six examples presented in this paper are all verified. All definitions and theorems in this paper are mechanised, and accompanying icons (**) encoding a hyperlink (available only in the electronic version of this paper) to corresponding repository artefacts.

Paper structure. The remainder of this paper is organised as follows. We review related work in Sect. 2. Section 3 provides the necessary background for further presentation of our work in the subsequent sections. In Sect. 4, we define the complete lattice over the unit interval and then lift it to a complete lattice over pointwise functions. Section 5 formalises our probabilistic programming with Iverson brackets defined. We also present proven algebraic laws for each construct. In Sect. 6, we present the semantics of probabilistic loops and the fixed point theorem. Afterwards, we illustrate our reasoning approach using six examples. Two are classification problems in machine learning, and two contain probabilistic loops (see Sect. 7). Finally, we discuss future work in Sect. 8.

2. Related work

Imperative probabilistic sequential programming languages and their semantics. Imperative probabilistic programs are the extension of conventional imperative programs with the capability to model randomness (typically from random number generators), usually using a binary probabilistic choice construct [10, 4] or a random number generator (rand) [22, 26] sampling from the uniform distribution over a set (either finite or infinite). McIver and Morgan [27] show any discrete distribution, including uniform distributions, can be achieved through a binary probabilistic choice using a fair coin. Our work presented in this paper, ProbURel, uses both a probabilistic choice and a construct to draw a discrete uniform distribution from a finite set (similar to rand).

The "predicate style" semantics (for example, weakest precondition [12], Hoare logic [28], and predicative programming [29]) for conventional imperative programs are boolean functions over state space. The semantics is sufficient to reason about these programs in the qualitative aspect, including termination or total correctness. Still, reasoning about probabilistic programs with natural quantitative measurements is insufficient. For this reason, boolean functions are generalised to real-valued functions over state space [30, 31, 10, 22]. One exception is the relational semantics [32] that embeds standard programs into the probabilistic world. In this semantics, probability distributions over states are captured in a special program variable (prob, a total function) in a standard program, and, therefore, the semantics of such probabilistic programs are still boolean functions over state space. Programs in our ProbURel are real-valued functions over state space.

Kozen's extension [30, 31] replaced nondeterministic choice in conventional imperative programs with probabilistic choice, while McIver and Morgan [11] added a probabilistic choice construct (and so it has both nondeterministic and probabilistic choice). McIver and Morgan's weakest pre-expectation or expectation transformer semantics [10], the real-valued expressions over state space are called expectations (indeed random variables). The weakest pre-expectation expressed as pre = wp (P, [post]) (where the square bracket [_] converts a boolean-valued predicate to an arithmetic value, especially [true] = 1 and [false] = 0), is the least pre-expectations (evaluated in the initial state of P) to ensure that the probabilistic program P terminates with post-expectation [post] in its final state. For example,

$$wp(x := x + 1_{(1/3)} \oplus x := x - 1, [x \ge 0]) = (1/3) * [x = -1 \lor x = 0] + [x \ge 1]$$

means that in order for the program to establish $x \ge 0$, the probability of x being -1 or 0 (or $x \ge 1$, or x < -1) in its initial state is at least 1/3 (or 1, or 0). An extensive set of algebraic laws has been presented for reasoning about probabilistic programs, including loops and termination in [7, Appendix B]. Using this semantics, Kaminski [15] developed an advanced weakest precondition calculus. Based on the work, Schröer et al. [21] developed a deductive verification infrastructure for verifying discrete probabilistic programs in terms of bounded expectations and expected runtimes, and termination probabilities using an intermediate verification language from which verification conditions are generated and verified in SMT solvers. Our ProbURel uses a similar notation to the square bracket, called Iverson brackets. The predicates in ProbURel are UTP's alphabetised relations which have been used to establish program correctness using the weakest precondition calculus [33]. For this reason, our ProbURel can also describe the weakest pre-expectation semantics.

The pGCL [11, 10], an extension of Dijkstra's Guarded Command Language (GCL) [12] with a probabilistic choice construct, is a widely studied imperative probabilistic programming language. The weakest pre-expectation semantics is based on pGCL. It is formalised in High-Order Logic (HOL) [13] by Hurd et al. [14] (based on the quantitative logic [34]), enabling verification of partial correctness of probabilistic programs, and also formalised in Isabelle/HOL by Cock [35] using shallow embedding (where probabilities are just primitive real numbers) to achieve improved proof automation. The pGCL has simple operational semantics [36] using (parametric) Markov Decision Processes (MDPs) to establish a semantic connection with the weakest pre-expectation semantics and has relational semantics [37, 32, 38], which is based on the theory of designs in UTP and mechanised in Isabelle/UTP [24]. The pGCL contains a nondeterministic choice construct, but ProbURel in this paper does not include it. It is part of our future work to introduce

nondeterminism. We also use Isabelle/UTP for automated verification, but the theory we use here is the theory of relations in UTP, which is more general than the theory of designs.

Probabilistic programs can be modelled as functions using a monadic interpretation. Hurd [6] developed a formal HOL framework for modelling and verifying probabilistic algorithms using theorem proving. The work uses mathematical measure theory to represent probability space to model a random bit generator (an infinite stream of independent coin-flips). It uses a monadic state transformer to model probabilistic programs with higher-order logic functions. A probabilistic program consumes some bits from the front of the stream for randomisation and returns the remains. Audebaud et al. [39] use the monadic interpretation of randomised programs for probabilistic distributions (instead of measure theory) and mechanise their work in the Coq theorem prover [40]. They consider probabilistic choice (without nondeterminism) in a functional language with recursion (instead of an imperative language). Programs in ProbURel are interpreted in imperative instead of monadic, and probabilistic loops are reasoned using fixed-point theories.

Dahlqvist et al.'s simple imperative probabilistic language [26] uses two constructs, coin() and rand(), to introduce discrete and continuous uniform distributions, and both operational and denotational semantics are presented. In its operational semantics, a probabilistic program is assumed to start in two fixed infinite streams (one for coin and one for rand), and the execution of each random sampling reads and removes the head from its corresponding stream. Eventually, the program is deterministic, and randomness is present in the infinite streams. This is similar to Hehner's probabilistic predicative programming [22] where each call to rand is stored in a mathematical variable (not a program variable). The denotational semantics of the simple language is given in terms of probability distributions. In ProbURel, we use a similar notation to rand to draw a discrete uniform distribution from a finite set. The semantics of ProbURel are denotational.

Hehner [22, 4] also generalises boolean functions for predicative programming to real-valued functions for probabilistic predicative programming. In his language, conditional and joint probability are modelled through sequential and parallel composition. One unique feature of the language is its capability to model epistemic uncertainty, due to the lack of knowledge of information and reducible after gaining more knowledge, and aleatoric uncertainty, due to the natural randomness of physical processes. Epistemic uncertainty is modelled through parallel composition using the subjective Bayesian approach. Our work, presented here, is based on Hehner's work. We formalise the syntax and semantics of the work, introduce UTP's alphabetised relations, bridge the semantics gap in dealing with probabilistic loops and mechanise it in Isabelle/UTP for automated reasoning.

Researchers also use Hoare logic to reason about probabilistic programs, such as the work presented in [41, 42, 43], and VPHL [44] uses a weighted tree structure to represent probability distributions in its semantics and can reason about the partial correctness of probabilistic programs. The probabilistic relational Hoare logic (pRHL) [45] is a Hoare quadruple that establishes the equivalence of two programs and the usual Hoare logic to relate programs as pre- and post-conditions. ELLORA [19] is an assertion-based program logic for probabilistic programs and mechanised in the EasyCrypt theorem prover [20]. The logic is presented in both abstract and concrete. The abstract logic is used for reasoning about loops and adversaries while the concrete logic facilitates formal verification. The logic features the reasoning of the broad class of loops for absolute termination, AST, and general termination using different assertions. ProbURel uses UTP's alphabetised relations, which have been used to establish program correctness using Hoare logic [33]. For this reason, our ProbURel can also describe probabilistic Hoare logic semantics. The semantics of ProbURel are denotational, and two programs are equivalent if they are equal functions over the same state space.

Recursion and Almost-sure termination. Reasoning about recursion is usually hard and is especially harder [46] for probabilistic programs because semantically probabilistic programs associate states with probability distributions other than merely boolean information for conventional programs. From this aspect, conventional programs can be regarded as a particular case of probabilistic programs where probability is always 1 or 0, so probabilistic programs are a more general paradigm.

Morgan and McIver's early work [47, 7] uses general techniques invariants and variants for reasoning about loops. Invariants for probabilistic loops are now expectations. Variants (V) are still integer-valued expressions but a) they are bounded below and above ($L \le V < H$) by fixed integer constants (L and H) if the states that satisfy the conjunction ($G \land Inv$) of the loop guard condition G and the invariant Inv,

are infinite; b) for every iteration, there is a fixed non-zero probability ε that the invariants are strictly decreased. A variant is not required always to be strictly decreased now; it could also be increased. The variant rule is strengthened later by McIver et al. to remove the need to bound above, allow (quasi-) variants to be real-valued expressions, and ε to vary [8, Theorem 4.1]. Their new variant rule relies on a supermartingale, a sequence of random variables (RVs) for which the expected value of the current random variable is larger than or equal to that of the subsequent random variable. Two parametric antitone functions characterise the quasi-variant called p and d for a lower bound d on how much a program must decrease the variant with at least probability p. The new rule enables them to reason about the two-dimensional random walk, which is believed to be hard. Chakarov et al.'s expectation invariants [48] and Kaminski's sub- and superinvariants [15] are similar to Morgan and McIver's probabilistic invariants to use expectations for invariants. Our approach to reasoning about probabilistic loops is based on fixed-point theories, and the semantics of a loop is its unique fixed point when additional conditions are satisfied. We also use an iterative way to construct the fixed point, which is the supremum of an ascending chain.

As a consequence of the generality of probabilistic programs from conventional programs, it is much harder [49] to analyse the termination of probabilistic programs because now the program terminates with probability (instead of absolute termination [7] for conventional programs). The usual knowledge for conventional programs, such as termination or nontermination, finite run-time, and compositionality, is not valid for probabilistic programs. This problem has attracted a lot of interest in recent decades, such as [7, 50, 51, 52, 53, 8, 54, 55, 56]. The research area of interest for probabilistic programs is a weakened termination, called almost-sure termination (AST), or termination with probability one. In other words, a probabilistic program may not always terminate, but the probability of divergence is 0. For example, flipping a fair coin until the outcome is heads is such a program. As Esparza et al. [51] pointed out, (conventional) termination is a purely topological property, namely the absence of cycles, while AST requires arithmetic reasoning. Some recent studies have investigated the positive almost-sure termination [50] where probabilistic programs terminate in the finite expected time, and several studies have also assessed the null almost-sure termination where probabilistic programs terminate almost-surely but not in the finite expected time. Hehner inspires our approach to reason about termination.

In Hehner's semantics [4], a time variable t is introduced and is strictly increased in each iteration of a loop. For example, it can be an extended integer number (including ∞ for nontermination) and is used to count iterations. His approach [25] for termination of (probabilistic) loops is stronger than total correctness (equal to partial correctness plus termination) because the time variable allows reasoning about not only whether a loop terminates but also when it terminates (run-time analysis). One example from his paper is a probabilistic loop about throwing a pair of dice till they have the same outcome. The invariant (or hypothesis) of this loop, shown below and proved in Sect. 7.7, gives the distribution of final states (primed variables).

$$H \widehat{=} [d'_1 = d'_2] * [t' \ge t + 1] * (\frac{5}{6})^{(t'-t-1)} * (\frac{1}{36})$$

We are interested in the probability distribution in terms of iterations or t, and so we substitute $[d'_1 = d'_2]$ with 6 because there are 6 possible combinations of d'_1 and d'_2 in each experiment to have them equal.

$$Ht \stackrel{\frown}{=} 6 * [t' \ge t + 1] * \left(\frac{5}{6}\right)^{(t'-t-1)} * \left(\frac{1}{36}\right) = [t' \ge t + 1] * \left(\frac{5}{6}\right)^{(t'-t-1)} * \left(\frac{1}{6}\right)$$

Provided the initial value of t is 0, we plot the program's termination probability using this invariant in Fig. 1. Because t counts iterations, the diagram also shows the probability of the termination in the exact iteration t. The probability for t' = 1 is 1/6 (≈ 0.167), so the program terminates in the first iteration with probability 1/6 as expected (6 among total 6*6=36 combinations).

Reasoning about almost-sure termination becomes an arithmetic summation of this distribution, as

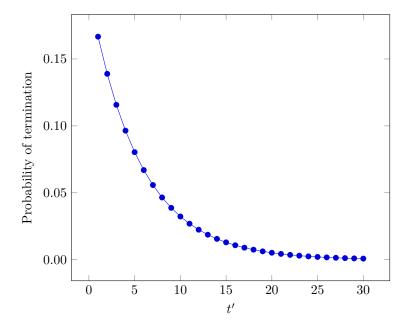


Figure 1: Termination probability over t' for dice.

shown below, which sums to 1.

$$\sum_{t'=0}^{\infty} \llbracket t' \geq 1 \rrbracket * \left(\frac{5}{6}\right)^{\binom{t'-1}{2}} * \left(\frac{1}{6}\right) = \sum_{t'=1}^{\infty} \left(\frac{5}{6}\right)^{\binom{t'-1}{2}} * \left(\frac{1}{6}\right) = \left(\frac{1}{6}\right) * \sum_{t'=1}^{\infty} \left(\frac{5}{6}\right)^{\binom{t'-1}{2}} = 1.0$$

The expected run-time is the expectation of t', simply the sequential composition of (Ht; t') = t + 6, denoting, on average, it takes six throws to have their outcomes equal.

Summary. In Table 1, we summarise the comparison of different probabilistic semantics in terms of four perspectives: modelling features, advanced features, reasoning and verification, and automation. In modelling features, we consider the support of constructs for usual probabilistic choice, uniform distributions, conditioning, joint probability, nondeterministic choice, and expression and type systems. Our work supports all but nondeterminism. Notably, our language has a rich expression and type system which is based on the Z notation [58, 59] and the mechanised Z mathematical toolkit¹ in Isabelle/HOL, which entitles us to model abstract probabilistic programs and capture rich semantics. Our language and semantic framework can support parametric models, is able to unify other semantics and support refinement thanks to our UTP relations. Probabilistic designs are also based on UTP and so the theory supports similar features as ours. In terms of verification, our work can be used to unify the WPE semantics (but this needs the new development), and support parametric verification.

3. Preliminaries

3.1. Unifying Theories of Programming

In UTP [60, 33], the meaning (denotational semantics) of programs is given as predicates, called "programs-as-predicates" [61]. In this approach, the alphabetised relational calculus [33], a combination of standard

¹https://github.com/isabelle-utp/Z_Toolkit.

Table 1: Comparison of different probabilistic semantics

Approach	Modelling features						$\frac{\text{Advanced}}{\text{Para Unf Refn}}$			Reasoning and Verification WP ExaPD ExpRT Inv Para					Auto
	$\overline{\text{PrCh UniD Cond Jnt NonD Expr}}$														
PPDL[31]	✓	✓				Bsc					√	✓	√		
WPE[57]	\checkmark				\checkmark	Bsc			\checkmark	\checkmark			\checkmark		$\sqrt{\mathrm{TP}}$
AWPE[15]	\checkmark	\checkmark	\checkmark		\checkmark	Bsc			\checkmark	\checkmark		\checkmark	\checkmark		\checkmark
DVWPE[21]	\checkmark	\checkmark	\checkmark		\checkmark	Bsc				\checkmark		\checkmark	\checkmark		\checkmark
ELLORA[19]	\checkmark	\checkmark				Bsc				ASS	\checkmark	\checkmark	\checkmark		$\sqrt{\mathrm{TP}}$
PDs[32]	\checkmark				\checkmark	Rich	\checkmark	\checkmark	$\sqrt{0}$	CTr	\checkmark		\checkmark	\checkmark	$\sqrt{\mathrm{TP}}$
PPP[22]	\checkmark	\checkmark	\checkmark	\checkmark		Bsc			$\sqrt{0}$		\checkmark	\checkmark	\checkmark		
Our	\checkmark	\checkmark	\checkmark	\checkmark		Rich	\checkmark	\checkmark	$\sqrt{0}$	$\sqrt{0}$	\checkmark	\checkmark	\checkmark	\checkmark	$\sqrt{\mathrm{TP}}$

Acronym: ASS: assertion-based; Auto: automation; AWPE: advanced WPE; Bsc: basic; CTr: contract-based; Cond: conditional probability; DVWPE: WPE-based deductive verification infrastructure; ExaPD: exact probability distributions; Exp: expression and type system; ExpRT: expected runtime; Inv: loop invariant; Jnt: joint probability, especially supporting general likelihood functions; NonD: nondeterministic choice; Para: parametric models or reasoning; PDs: probabilistic designs; PrCh: probabilistic choice; Refn: refinement; TP: theorem proving; Unf: unification; UniD: construct for uniform distributions; ✓o: supported but not yet developed; ✓TP: automation through theorem proving;

predicate calculus operators and Tarski's relation algebra [62], is used as the basis for its semantic model to denote programs as binary relations between initial observations and their subsequent observations. An alphabetised relation is an alphabet-predicate pair $(\alpha P, P)$ where the accompanying alphabet of P, αP , is composed of undashed variables (x) and dashed variables (x'), representing observations made initially and subsequently. For example, a program (x' := x + 1) with two observable variables x and y can be modelled as a relational predicate $(x' = x + 1 \land y' = y)$ with its alphabet $\{x, x', y, y'\}$.

Alphabetised predicates are presented in this representation of UTP through alphabetised expressions [V,S]expr, parametric over the value type V and the observation space S and defined as total functions $S \to V$. Predicates² are boolean expressions: $[S]pred \cong [bool,S]expr$, an expression whose value type is boolean. Relations are predicates over a product space: $[S_1,S_2]urel \cong [S_1 \times S_2]pred$, where S_1 and S_2 are the initial and final observation space, corresponding to undashed variables (input alphabet) and dashed variables (output alphabet). Here urel means the UTP relations. Homogeneous relations have the same initial and final observation space: $[S]hrel \cong [S,S]urel$.

The denotational semantics of a sequential program is given as relations by the composition of constructors, including conditional, assignment, skip, sequential composition, and nondeterministic choice. These constructors are defined below.

Definition 3.1 (Constructs of sequential programs).

$$(P \lhd b \rhd Q) \stackrel{\widehat{=}}{=} (b \land P) \lor (\neg b \land Q)$$
 (conditional)

$$(x :=_A e) \stackrel{\widehat{=}}{=} (x' = e \land w' = w)$$
 (assignment)

$$\mathcal{I} \stackrel{\widehat{=}}{=} (v' = v)$$
 (skip)

$$P; \ Q \stackrel{\widehat{=}}{=} (\exists s_0 \bullet P[s_0/s'] \land Q[s_0/s])$$
 (sequential composition)

$$P \sqcap Q \stackrel{\widehat{=}}{=} (P \lor Q)$$
 if $\alpha P = \alpha Q$ (nondeterminism)

Particularly, we need to emphasise the type of programs and their alphabets. In (conditional), b is type $[S_1]$ pred, P and Q are of type $[S_1, S_2]$ urel, and $\alpha b \subseteq \alpha P = \alpha Q$. In (assignment), A is the observation

 $^{^{2}}$ In the rest of the paper, we use expressions, predicates, and relations to refer to alphabetised counterparts for simplicity.

space of a program, including variable x and a set w of other variables. We also use a simple syntax w' = w here to denote conjunctions of equations over each variable in w. For example, if $w = \{y, z\}$, then $(w' = w) \stackrel{\frown}{=} (y' = y \land z' = z)$. The subscript A in $:=_A$ is usually omitted because it can automatically be derived from its context. Skip \mathbb{I} (skip) is a special assignment where no variable changes (here v denotes the set of all variables), so the observation space stays the same. In sequential composition P; Q (sequential composition), two relations P of type $[S_1, S_2]$ urel and Q of type $[S_2, S_3]$ urel are composed because the output alphabet S_2 of P is the same as the input alphabet S_2 of Q. The relational composition gives a program of type $[S_1, S_3]$ urel with $s_0 : S_2$ denotes the entire state, $P[s_0/s']$ for the substitution of the final observation s' of P by s_0 , and $Q[s_0/s]$ for the substitution of the initial observation s of Q by s_0 . Nondeterministic choice $P \sqcap Q$ (nondeterminism) is simply a disjunction of relations if they have the same alphabets. The introduction of the new notation \sqcap emphasises this condition.

UTP uses refinement to deal with program development or correctness. A specification S is refined by a program P, denoted as $S \sqsubseteq P$, if and only if that P implies S is universally closed. For example, (x := x + 1) is a refinement of (x' > x) because for any x and x', $(x' = x + 1) \Rightarrow (x' > x)$. Relations of type $[S_1, S_2]$ urel, for any given S_1 and S_2 , are partially ordered by \sqsubseteq where **false** and **true**, special relations whose predicates are **false** and **true**, are at its extremes: **true** $\sqsubseteq P \sqsubseteq \textbf{false}$.

3.2. Isabelle/UTP

$$\lambda s. get_x (get_{snd_L} s) = get_x (get_{fst_L} s) + 1 \wedge get_y (get_{snd_L} s) = get_y (get_{fst_L} s)$$
 [Lens representation]

where fst_L and snd_L are the lenses to project the first and the second element of a product space. By substituting the state s with a pair (s, s'), this expression can be simplified to

$$\lambda(s, s')$$
. $get_x s' = get_x s + 1 \land get_y s' = get_y s$ [Simplified lens representation]

However, writing UTP expressions this way is tedious and not very useful and intuitive for good programming practice because too many implementation details are presented. For this reason, Isabelle/UTP implemented a lifted parser to provide a transparent conversion between the lens's representation and the programming syntax like (x' := x + 1). We denote this representation of UTP expressions as $(expr)_e$, such as $(x' := x + 1)_e$, which is converted to [Lens representation].

In Isabelle/UTP, we use \mathbf{v} to denote the universe alphabet of a program. In other words, it is the set of all observable variables. We also use \mathbf{v}' to denote the set of all dashed observable variables. For the previous example, \mathbf{v} denotes $\{x, y\}$ and \mathbf{v}' denotes $\{x', y'\}$.

3.3. Probabilistic predicative programming

Predicative programming [29, 66], or programs-as-predicates [67], describes programs using first-order semantics or relational semantics as boolean expressions (predicates). A program has its input denoted by undashed variables and output denoted by dashed variables. Predicative programming also uses refinement for program correctness.

Probabilistic predicative programming [22, 4] generalises predicative programming from boolean to probabilistic. Notations are introduced for probabilistic programming, such as skip, assignment, conditional choice, probabilistic choice, sequential composition (conditional probability), parallel composition (joint probability), and recursion. Except for parallel composition, these constructors deal with probabilistic programs whose outputs are distributions or distribution programs. Parallel composition can deal with

probabilistic programs whose outputs might not be distributions (non-distribution programs), and uses normalisation to give a distribution program.

This programming supports the subjective Bayesian approach through parallel composition. From a given distribution program, we can learn a new fact by placing the fact in parallel with the distribution program to allow beliefs to be updated.

To reason about the termination of loops, a time variable is introduced to count iterations. This gives more information (time) than just termination [25]. In this programming, the expected value of a number expression e according to a distribution program P is just the sequential composition of P and e. If e is a boolean expression, the sequential composition gives the probability that e is valid after the execution of P. With the time variable, the programming allows reasoning about the average termination time. For example, on average, it takes two flips of a fair coin to see heads or tails. The termination probability of a loop (**while** e **do** e can be specified using the sequential composition of the solution (of the loop) and the negation of the loop condition (e e e). If the result is 1, it means the loop almost surely terminates. If the result is not 1, the loop may diverge.

3.4. Complete lattices and fixed-point theorems

A partially ordered set (poset) (X, \leq) is a complete lattice if every subset of X has a supremum and an infimum.

$$\forall A \subseteq X \bullet \left(\bigcap A \right) \in X$$
 (Inf exists)
$$\forall A \subseteq X \bullet \left(\bigcup A \right) \in X$$
 (Sup exists)

We use a tuple $(X, \leq, <, \perp, \top, \sqcap, \sqcup, \sqcap, \sqcup)$ to represent a complete lattice (X, \leq) with a strict binary relation <, the bottom element \perp , the top element \top , the infimum \sqcap of two elements, the supremum \sqcup of two elements, the infimum \sqcap of a (finite or infinite) set of elements, and the supremum \sqcup of a (finite or infinite) set of elements.

A complete lattice satisfies more laws below.

$$x \leq x \qquad \qquad \text{(reflexive)} \\ x \leq y \land y \leq z \Rightarrow x \leq z \qquad \qquad \text{(transitive)} \\ x \leq y \land y \leq x \Rightarrow x = y \qquad \qquad \text{(antisym)} \\ x \sqcap y \leq x \qquad \qquad \text{(inf_le1)} \\ x \subseteq y \land x \leq z \Rightarrow x \leq y \sqcap z \qquad \qquad \text{(inf_le2)} \\ (x \leq y) \equiv (x \sqcap y = x) \qquad \qquad \text{(inf_greatest)} \\ (x \leq y) \equiv (x \sqcap y = x) \qquad \qquad \text{(inf_iff)} \\ x \leq x \sqcup y \qquad \qquad \text{(sup_ge1)} \\ y \leq x \sqcup y \qquad \qquad \text{(sup_ge2)} \\ y \leq x \land z \leq x \Rightarrow y \sqcup z \leq x \qquad \qquad \text{(sup_least)} \\ (x \leq y) \equiv (x \sqcup y = y) \qquad \qquad \text{(sup_least)} \\ (x \leq y) \equiv (x \sqcup y = y) \qquad \qquad \text{(Inf_lower)} \\ (\forall x . x \in A \Rightarrow x \leq x) \Rightarrow z \leq \prod A \qquad \qquad \text{(Inf_greatest)} \\ x \in A \Rightarrow x \leq x \Rightarrow x \leq$$

Monotonic and antimonotonic functions in order theory are characterised using *mono* and *antimono* defined below.

Definition 3.2 (Monotone and anti-monotone). Provided (X, \leq) and (X', \leq') are posets and f is a function of type $X \to X'$, then

$$mono(f) \cong \forall x \bullet \forall y \bullet x \leq y \Rightarrow f(x) \leq' f(y)$$
 (monotone) $antimono(f) \cong \forall x \bullet \forall y \bullet x \leq y \Rightarrow f(y) \leq' f(x)$ (anti-monotone)

Ascending and descending chains are monotonic and antimonotonic functions whose domain is natural numbers.

Definition 3.3 (Chains). Provided (X, \leq) is a complete lattice and f is a function of type $\mathbb{N} \to X$, then

$$incseq(f) \stackrel{\frown}{=} mono(f)$$
 (ascending chain)
 $decseq(f) \stackrel{\frown}{=} antimono(f)$ (descending chain)

We particularly define *incseq* and *decseq* to be over complete lattices which we are interested in this paper. This is to simplify the specification of premises in lemmas and theorems because *incseq* and *decseq* impose a type restriction to complete lattices directly. Otherwise, we need additional premises if we use the more general *mono* and *antimono*.

We show the application of a monotonic function f to an ascending chain c is also an ascending chain.

Theorem 3.1. We fix
$$c: \mathbb{N} \to X$$
 and $f: X \to Y$, then $\mathsf{incseq}(c) \land \mathsf{mono}(f) \Rightarrow \mathsf{incseq}(\lambda \ n \bullet f(c(n)))$

We show the application of a monotonic function f to a descending chain c is also a descending chain.

Theorem 3.2. We fix
$$c: \mathbb{N} \to X$$
 and $f: X \to Y$, then $\mathsf{decseq}(c) \land \mathsf{mono}(f) \Rightarrow \mathsf{decseq}(\lambda \ n \bullet f(c(n)))$

If f is an ascending or descending chain, its limit is the supremum or infimum of the chain.

Theorem 3.3 (Limit as supremum and infimum). Provided (X, \leq) is a complete lattice and also totally ordered, and we fix $f : \mathbb{N} \to X$, then

$$incseq(f) \Rightarrow f \xrightarrow{n \to \infty} \left(\bigsqcup n \bullet f(n) \right)$$
 (limit as supremum)
 $decseq(f) \Rightarrow f \xrightarrow{n \to \infty} \left(\bigcap n \bullet f(n) \right)$

Here we use $f \xrightarrow{n \to \infty} v$ to denote the limit of f is v: $\lim_{n \to \infty} f(n) = v$. The definition of the limit of a sequence is given below.

Definition 3.4 (Limit of a sequence). A sequence f converges to v if and only if

$$\forall \varepsilon : \mathbb{R} > 0 \bullet \exists N : \mathbb{N} \bullet \forall n > N \bullet \mid f(n) - v \mid < \varepsilon$$

Theorem 3.4 (Knaster-Tarski fixed-point theorem). Provided (X, \leq) is a complete lattice and $F: X \to X$ is monotonic, the set of fixed points of F also forms a complete lattice. The least fixed point is the infimum of the pre-fixed points.

$$\mu F \stackrel{\frown}{=} \prod \{u : X \mid F(u) \le u\}$$
 (least fixed point)

The greatest fixed point is the supremum of the post-fixed points.

$$\nu F \,\widehat{=}\, \big|\, \big|\, \big\{u: X \mid u \leq F(u)\big\} \hspace{1cm} \text{(great fixed point)}$$

Definition 3.5 (Scott continuity [68]). Suppose (X, \leq) and (X', \leq') are complete lattices. A function $F: X \to X'$ is **Scott-continuous** or **continuous** if, for every non-empty chain $S \subseteq X$,

$$F\left(\left|\begin{array}{c} |_{x} S\right) = \left|\begin{array}{c} |_{x} F(S) \end{array}\right.$$
 (continuous)

Here we use F(S) to denote the set $\{d \in S \bullet F(d)\}$, the relational image of S under F or the range of F domain restricted to S.

In the original definition of Scott continuity, both X and X' are directed-complete partial orders (dcpo). We fix them to be complete lattices because every complete lattice is a dcpo [68], and we only consider complete lattices in this paper. If X and X' are identical lattices, the subscript of \square in Definition (continuous) can be omitted.

Theorem 3.5 (Monotonicity). A continuous function is also monotonic.

Theorem 3.6 (Kleene fixed-point theorem). Provided (X, \leq) is a complete lattice with a least element \perp and a top element \top , and $F: X \to X$ is continuous, then F has a least fixed point μ F and a greatest fixed point ν F.

$$\mu F = \bigsqcup_{n \ge 0} F^n(\bot)$$
 (least fixed point)
$$\nu F = \bigsqcup_{n \ge 0} F^n(\top)$$
 (greatest fixed point)

Here we use $\bigsqcup_{n>0} F^n(\bot)$ to denote $\bigsqcup \{n: \mathbb{N} \bullet F^n(\bot)\}$

PROOF. We prove μF is a fixed point first and then prove μF is the least one.

```
F(\mu F)
= { Definition (least fixed point) }
  F\left(\left| \begin{array}{c} |_{n\geq 0} F^n(\bot) \end{array}\right)
= { Continuity Definition (continuous) }
  | |_{n \geq 0} F(F^n(\perp))
= { Defintion of F^n: F(F^m(x)) = F^{m+1}(x) }
  | |_{n>0} (F^{n+1}(\perp))
= { Rewrite index }
  | |_{m \geq 1} (F^m(\perp))
= \{ \text{Law (sup\_iff) and } \perp \text{ is the least element } \}
  \perp \sqcup \left( \bigsqcup_{m \geq 1} \left( F^m(\perp) \right) \right)
= { Definition of F^0: F^0(\bot) = \bot }
  F^0(\bot) \sqcup \left( \bigsqcup\nolimits_{{}^{m} \, \geq \, {}^{1}} \, \left( F^m(\bot) \right) \right)
= \{ \text{ Definition of } \big| \big|_{m \ge 0} \}
  | |_{m \geq 0} (F^m(\perp))
= { Rewite index }
   | |_{n \geq 0} (F^n(\perp))
= { Definition (least fixed point) }
   \mu F
```

So μF is a fixed point of F.

Suppose fb is also a fixed point of F that is, F(fb) = fb.

```
\{ \perp \text{ is the least element } \}
   \perp \leq fb
\Rightarrow { F is continuous and so is monotonic by Theorem 3.5 }
   F(\perp) \leq F(fb)
\Rightarrow \{ F(fb) = fb \}
   F(\perp) \leq fb
\Rightarrow { F^2(\bot) = F(F(\bot)) and F is monotonic }
   F^2(\perp) < F(fb)
\Rightarrow \{ F(fb) = fb \}
  F^2(\perp) < fb
\Rightarrow { Induction }
   F^n(\perp) < fb
\Rightarrow \{ \text{Law (Sup\_least)} \}
  | |_{n>0} (F^n(\perp)) \leq fb
= { Definition (least fixed point) }
   \mu F \leq fb
```

So μF is the least fixed point.

Similarly, we prove νF is a fixed point of F and is also the greatest.

3.5. Summation over topological space

Summation considered in this paper could range over an infinite set, called infinite sums. We use corresponding theories in Isabelle/HOL to deal with convergence and infinite sums.

We say a function f is summable on a (potentially infinite) set A, denoted as summable(f, A) if the sum of f on A converges to a particular value. The convergence is expressed as the existence of a limit of f over finite subsets B of A when B is approaching A. In Isabelle/HOL, the limit is generalised to arbitrary topological space using filters [69]. Its definition is parametrised over two filters. To the infinite sums, they are the open neighbourhood filter, interpreted as "for all points in some open neighbourhood of a point" and the subset inclusion ordered at-top filter, interpreted as "for sufficiently large finite subsets when it approaches its top A". The infinite sums of f over A, denoted as $\Sigma_{\infty}x \in A \bullet f(x)$, is the limit if summable(f, A) and 0 otherwise. The definitions of summable and finite sums can be found in Isabelle/HOL.

We list some laws of summation below.

Theorem 3.7.

```
c \neq 0 \land summable(f,A) \Rightarrow summable(\lambda x \bullet f(x)/c,A) \qquad \text{(division by constant summable)} c \neq 0 \land summable(f,A) \Rightarrow \Sigma_{\infty} x \in A \bullet f(x)/c = (\Sigma_{\infty} x \in A \bullet f(x))/c \qquad \text{(division by constant)} summable(f,A) \Rightarrow summable(\lambda x \bullet f(x) * c,A) \qquad \text{(multiplication of constant summable)} summable(f,A) \Rightarrow \Sigma_{\infty} x \in A \bullet f(x) * c = (\Sigma_{\infty} x \in A \bullet f(x)) * c \qquad \text{(multiplication of constant)} summable(f,A) \Rightarrow summable(\lambda x \bullet c * f(x),A) \qquad \text{(multiplication of constant summable)} summable(f,A) \Rightarrow \Sigma_{\infty} x \in A \bullet c * f(x) = c * (\Sigma_{\infty} x \in A \bullet f(x)) \qquad \text{(multiplication of constant)} summable(f,A) \land summable(g,A) \Rightarrow summable(\lambda x \bullet f(x) + g(x),A) \qquad \text{(addition summable)}
```

$$\begin{aligned} \textit{summable}(f,A) \wedge \textit{summable}(g,A) \Rightarrow \Sigma_{\infty} x \in A \bullet f(x) + g(x) = \Sigma_{\infty} x \in A \bullet f(x) + \Sigma_{\infty} x \in A \bullet g(x) \\ & \text{(addition)} \end{aligned}$$

$$\begin{aligned} \textit{summable}(f,A) \wedge \textit{summable}(g,A) \Rightarrow \Sigma_{\infty} x \in A \bullet f(x) - g(x) = \Sigma_{\infty} x \in A \bullet f(x) - \Sigma_{\infty} x \in A \bullet g(x) \\ & \text{(subtraction)} \end{aligned}$$

4. Unit real interval (complete lattice)

Our probabilistic programs are real-valued functions over state space, and specifically, they are the functions from state space to real numbers between 0 and 1 inclusive, or the *unit real interval* (*ureal*). We call them *ureal*-valued functions, denoted as $S \to ureal$. To deal with the semantics of probabilistic loops in Section 6 using the Knaster–Tarski and Kleene fixed-point theorems [70], we define a complete lattice containing a set of these functions together with a pointwise comparison relation \leq .

Section 4.1 defines *ureal* and constructs a complete lattice containing the set *ureal* with relation \leq . Then we define *ureal*-valued functions and the pointwise comparison relation in Section 4.2. With these definitions, we construct the required complete lattice for characterising probabilistic loops.

4.1. Definition of ureal

The *ureal* is defined below as a set of real numbers between 0 and 1.

Definition 4.1 (Unit real interval). $ureal = \{0...1\}$

We also define two functions to get the smaller and larger value of two comparable numbers, such as real numbers and *ureal* numbers.

Definition 4.2 (Maximum and minimum of real numbers).

$$max(x, y) = (if x \le y then y else x)$$
 $min(x, y) = (if x \le y then x else y)$

The two functions *min* and *max* entitle us to define conversions between real numbers and *ureal* numbers.

Definition 4.3 (Conversion between ureal and \mathbb{R}). We define functions $u2r(\mathfrak{P})$ and $r2u(\mathfrak{P})$ (notations \overline{x} and y) to convert x of ureal to \mathbb{R} , and y of \mathbb{R} to ureal.

$$\overline{x} \mathrel{\widehat{=}} (x :: \mathbb{R}) \qquad y \mathrel{\widehat{=}} \min(\max(0, y), 1)$$

The conversion of a *ureal* number x to a real number, using the function u2r, is simply a type cast from *ureal* to \mathbb{R} . However, the conversion of a real number y to *ureal* by the function r2u needs to deal with the cases when y is out of the unit interval. We use *min* and *max* to bound it to 0 or 1 in these cases and keep its value if y is between 0 and 1. Based on the conversions, we define the comparison functions over *ureal*.

Definition 4.4 (Comparison functions of ureal). Provided both x and y are of type ureal.

$$x = y = \overline{x} = \overline{y}$$
 $x < y = \overline{x} < \overline{y}$ $x \le y = \overline{x} \le \overline{y}$
 $\max(x, y) = (\text{if } x \le y \text{ then } y \text{ else } x)$ $\min(x, y) = (\text{if } x \le y \text{ then } x \text{ else } y)$

From these comparisons, we show both conversion functions are monotonic.

Lemma 4.1. The function u2r is strictly monotonic (**). That is, if x < y, then $\overline{x} < \overline{y}$. The function r2u is monotonic (**), but not strictly. For example, $\underline{2} = \underline{3}$ (both equal to 1) though 2 < 3.

The function r2u is the inverse of u2r.

Lemma 4.2.
$$(\overline{x}) = x$$

The function u2r is the inverse of r2u only if the real number to be converted is between 0 and 1.

Lemma 4.3.
$$(x \ge 0 \land x \le 1) \Rightarrow \overline{(\underline{x})} = x$$



Infimum and supremum of *ureal* are defined using Hilbert's ε operator, an indefinite description, written ε $x \bullet P(x)$ denoting some x such that P(x) is true. We note that ε used below denotes the Hilbert's operator and a real number elsewhere in the paper.

Definition 4.5 (Infimum and supremum of *ureal*).

The infimum satisfies Laws (Inf_lower) and (Inf_greatest), and the supremum satisfies Laws (Sup_upper) and (Sup_least).

A complete lattice is now formed using these definitions.

Theorem 4.4. The poset (ureal, \leq) with the least element 0 ($\underline{0}$), the greatest element 1 ($\underline{1}$), \sqcap (min), \sqcup (max), \square , and \sqcup forms a complete lattice (ureal, \leq , <, 0, 1, \sqcap , \sqcup , \square , \sqcup).

This complete lattice is illustrated in the left diagram of Fig. 2. Indeed, it is a totally ordered set.

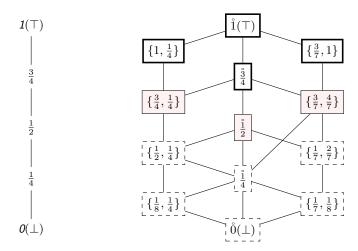


Figure 2: Complete lattices: left (ureal, \leq) and right ($S \to ureal$, \leq). We use $\left\{\frac{3}{4}, \frac{1}{4}\right\}$ denotes a function $\left\{s_1 \mapsto \frac{3}{4}, s_2 \mapsto \frac{1}{4}\right\}$ whose domain contains two elements s_1 and s_2 and their corresponding probabilities are $\frac{3}{4}$ and $\frac{1}{4}$ respectively. $\frac{1}{4}$ denotes a constant function which maps every element in its domain to $\frac{1}{4}$. Dashed box: subdistributions; normal: distributions; thick: superdistributions.

The addition, real numbers' subtraction, and multiplication operators are lifted for ureal.

Definition 4.6 (Bounded plus and minus). Provided both x and y are of type *ureal*.



$$x+y \mathbin{\widehat{=}} (\min(1,\overline{x}+\overline{y})) \qquad x-y \mathbin{\widehat{=}} (\max(0,\overline{x}-\overline{y})) \qquad x*y \mathbin{\widehat{=}} (\overline{x}*\overline{y})$$

The addition + and subtraction - are bounded to *ureal* by using *min* and *max*. For example, 0.5 + 0.7 = 1 and 0.5 - 0.7 = 0.

4.2. The ureal-valued functions

We now consider *ureal*-valued functions and define several constant functions for real-valued and *ureal*-valued.

Definition 4.7 (Real- and *ureal*-valued constant functions).



$$\dot{0} = \lambda s \bullet (0::\mathbb{R})$$
 $\dot{1} = \lambda s \bullet (1::\mathbb{R})$ $\dot{0} = \lambda s \bullet (0::ureal)$ $\dot{1} = \lambda s \bullet (1::ureal)$

The $\dot{0}$ and $\dot{1}$ are real-valued constant functions, and $\dot{0}$ and $\dot{1}$ are *ureal*-valued constant functions.

The addition and subtraction operators are also lifted to functions in a pointwise manner, and relations \leq and \leq are also lifted to functions.

Definition 4.8 (Operators and relations on functions).

$$\begin{split} f - g & \mathrel{\widehat{=}} (\lambda \, x \bullet f(x) - g(x)) \\ f & \leq g \mathrel{\widehat{=}} (\forall \, x \bullet f(x) \leq g(x)) \end{split} \qquad f + g \mathrel{\widehat{=}} (\lambda \, x \bullet f(x) + g(x)) \\ f & < g \mathrel{\widehat{=}} (\forall \, x \bullet f(x) < g(x)) \end{split}$$

The complete lattice for *ureal*-valued functions is formed.

Theorem 4.5. The poset $(S \to \mathsf{ureal}, \leq)$ with the least element $\mathring{0}$, the greatest element $\mathring{1}$, the infimum and supremum in a pointwise manner forms a complete lattice $(S \to \mathsf{ureal}, \leq, <, \mathring{0}, \mathring{1}, \sqcap, \sqcup, \sqcap, \sqcup)$.

We illustrate an example of the complete lattice $(S \to \textit{ureal}, \leq)$ in the right diagram of Fig. 2. Here we consider S containing two elements and use $\{\frac{1}{2}, \frac{1}{4}\}$ to denote probabilities over S: $\frac{1}{2}$ and $\frac{1}{4}$ respectively. Constant functions such as $\frac{1}{2}$ are on the central column. In the diagram, we only show a few functions where a dashed box, a normal box, or a thick box denotes a subdistribution, a distribution, or a superdistribution whose probabilities sum to less than or equal to 1, equal to 1, or larger than 1.

5. Probabilistic programming

This section concerns our probabilistic programming language's syntax and denotational semantics. Probabilistic recursion is not considered here, and its syntax and semantics will be introduced in Sect. 6.

Before presenting the semantics, we define a notation of Iverson brackets in Sect. 5.1 and introduce various expression types used in our language to constrain programs in Sect. 5.2. Our probabilistic programs are functions characterised as probabilistic distributions or subdistributions in Sect. 5.3. Our definition of Iverson brackets is real-valued functions, but probabilistic programs are *ureal*-valued functions. We must convert between these functions to use Iverson brackets in our semantics. The conversion is defined in Sect. 5.2.

After the presentation of syntax and semantics, we show a collection of proved algebraic laws for each construct in Sects. 5.5 to 5.12. These laws are used in compositional reasoning to simplify probabilistic programs.

5.1. Iverson brackets

Iverson brackets establish a correspondence between the predicate calculus and arithmetic, generalising the Kronecker delta.³

Definition 5.1 (Iverson bracket). The Iverson bracket of a predicate P of type [S] pred defines a function $S \to \mathbb{R}$, which gives a real number 0 or 1 if P is false or true for a particular state s (of type S).

$$[P] = (\mathbf{if} \ P \ \mathbf{then} \ 1 \ \mathbf{else} \ 0)_e$$

³Iverson brackets are a notation for the characteristic function on predicates. The convention was invented by Kenneth Eugene Iverson in 1962. Donald Knuth advocated using square brackets to avoid ambiguity in parenthesised logical expressions.

Several laws follow immediately from this definition.

$$\llbracket \text{false} \rrbracket = \dot{0} \tag{1}$$

$$\llbracket true \rrbracket = \dot{1}$$
 (2)

$$Q \sqsubseteq P \Rightarrow \llbracket P \rrbracket \le \llbracket Q \rrbracket \tag{3}$$

$$\llbracket \neg P \rrbracket = (1 - \llbracket P \rrbracket)_{e} \tag{4}$$

$$\llbracket P \wedge Q \rrbracket = (\llbracket P \rrbracket * \llbracket Q \rrbracket)_{e} \tag{5}$$

$$[\![P \lor Q]\!] = ([\![P]\!] + [\![Q]\!] - [\![P]\!] * [\![Q]\!])_{e}$$
(6)

$$[\![\lambda s \bullet s \in A \cap B]\!] = ([\![\lambda s \bullet s \in A]\!] * [\![\lambda s \bullet s \in B]\!])_e \tag{7}$$

$$(\llbracket \lambda \, s \bullet s \in A \rrbracket + \llbracket \lambda \, s \bullet s \in B \rrbracket)_e = (\llbracket \lambda \, s \bullet s \in A \cap B \rrbracket + \llbracket \lambda \, s \bullet s \in A \cup B \rrbracket)_e \tag{8}$$

$$(\max(x,y))_e = (x * [x > y] + y * [x \le y])_e \tag{9}$$

$$(\min(x,y))_e = (x * [x \le y] + y * [x > y])_e \tag{10}$$

$$\sum_{P(k)} f(k) = \sum_{k} \left(f * \llbracket P \rrbracket \right)_{e} (k) \tag{11}$$

Laws (1) and (2) show the arithmetic representations of UTP predicates false and true of type [S] pred are simply constant functions $\dot{0}$ and $\dot{1}$. Iverson brackets are monotone, as shown in Law (3). Laws (4) to (8) establish direct correspondence between arithmetic, logic, and set operations. Laws (9) and (10) show the maximum and minimum operations that can be implemented using the Iverson bracket. Law (11) shows summation over a subset of indices characterised by P(k) can be expressed as a summation over whole indices with the summation function f(x) multiplied by the Iverson bracket of P. According to Donald E. Knuth [71], it is not easy to make a mistake when dealing with summation indices by using the notation of the right side of the law. We omit other properties of Iverson brackets here for simplicity.

5.2. Type Abbreviations

We define several type abbreviations for real-valued and *ureal*-valued functions used to type constructs in our language.

$$[S] \textit{rexpr} = [\mathbb{R}, S] \textit{expr} \qquad \qquad (\text{Real-valued expression})$$

$$[S_1, S_2] \textit{rvfun} = [\mathbb{R}, S_1 \times S_2] \textit{expr} \qquad (\text{Relational real-valued expression})$$

$$[S] \textit{rvhfun} = [S, S] \textit{rvfun} \qquad (\text{Homogeneous relational real-valued expression})$$

$$[S] \textit{ureal-valued expression}$$

$$[S_1, S_2] \textit{prfun} = [\textit{ureal}, S_1 \times S_2] \textit{urexpr} \qquad (\text{Relational } \textit{ureal-valued expression})$$

$$[S] \textit{prhfun} = [S, S] \textit{prfun} \qquad (\text{Homogeneous relational } \textit{ureal-valued expression})$$

We define two functions $rvfun_of_prfun$ (and $prfun_of_rvfun$ (to convert P of type $[S_1, S_2]prfun$ to an expression of type $[S_1, S_2]rvfun$, and f of type $[S_1, S_2]rvfun$ to an expression of type $[S_1, S_2]prfun$.

Definition 5.2 (Conversion of relational real-valued and *ureal*-valued functions).

$$\textit{rvfun_of_prfun}(P) \mathrel{\widehat{=}} \left(\overline{P}\right)_e$$
 (Probabilistic programs to real-valued functions)
 $\textit{prfun_of_rvfun}(f) \mathrel{\widehat{=}} \left(f\right)_e$ (Real-valued functions to probabilistic programs)

The notations \overline{p} and \underline{r} (Definition 4.3) used ablove convert a *ureal* number to a real number and a real number to a *ureal* number, respectively. In this paper, we also use symbols \overline{P} and \underline{f} for $rvfun_of_prfun(P)$ and $prfun_of_rvfun(f)$, the conversions of functions.

Remark 5.1. For two reasons, we define two types *prfun* and *rvfun*. First, infinite summation and limits are defined over topological space, and so over real numbers, which form a Banach space, but not over *ureal*. We, therefore, need to convert probabilistic programs into real-valued functions to calculate summation and limits. After calculation, the results are converted back to probabilistic programs. Second, two functions in parallel composition or joint probability, introduced later in Sect. 5.4, are not necessary to be probabilistic, and they can be more general real-valued functions.

5.3. Distribution functions

A real-valued expression p is nonnegative if its range is real numbers larger than or equal to 0.

Definition 5.3 (Nonnegative).

 $\mathit{is_nonneg}(p) \mathrel{\widehat{=}} p \geq 0$

where p is a tautology on predicate p and is expanded to $\forall s \bullet (p)_e(s)$.

A real-valued expression p is probabilistic if its range is real numbers between 0 and 1 inclusive. This expression is characterised by a function is_prob of type $[S]rexpr \rightarrow bool$.

Definition 5.4 (Probability expression).

 $\mathit{is_prob}(p) \mathrel{\widehat{=}} p \geq 0 \land p \leq 1$

Theorem 5.2 (Iverson bracket is probabilistic). $is_prob(\llbracket p \rrbracket)$

A probabilistic function is called a *distribution* function if the probabilities of all states sum to 1, which is characterised by a function is_dist of type $[S]rexpr \rightarrow bool$.

Definition 5.5 (Probabilistic distributions).

 $is_dist(p) \stackrel{\frown}{=} is_prob(p) \land \Sigma_{\infty} s \bullet p(s) = 1$

where Σ_{∞} denotes a summation over possible infinite states.

A probabilistic function is called a *subdistribution* function if the probabilities of all states sum to less than or equal to 1, which is characterised by a function *is_subdist*.

Definition 5.6 (Probabilistic subdistributions).

 $\textit{is_subdist}(p) \mathrel{\widehat{=}} \textit{is_prob}(p) \land \Sigma_{\infty} s \bullet p(s) > 0 \land \Sigma_{\infty} s \bullet p(s) \leq 1$

We note that a probabilistic distribution is also a subdistribution but $\dot{0}$ (probabilities are zero everywhere) is not. We exclude $\dot{0}$ in subdistributions because (1) Σ_{∞} in Isabelle/HOL is defined to be 0 when p is not summable or divergent, and so we cannot differentiate this case from $\dot{0}$ from the summation result; and (2) the probability summation is the denominator in the definitions of normalisation in Definitions 5.10 and 5.11, and so subdistributions allow us to characterise the non-zero result to deal with the division-by-zero error.

$\mathbf{Lemma~5.3.}~\textit{is_dist}(p) \Rightarrow \textit{is_subdist}(p)$

For relational real-valued expressions p of type $[S_1, S_2]$ rvfun, we define three functions to specify if the final state of a program is characterised by the expression p is probabilistic, distributions, or subdistributions. To specify these functions, we define a curried operator \tilde{p} ($\hat{=} \lambda s s' \bullet p(s, s')$) to turn p into lambda terms, and so $\tilde{p}(s)$ is a function from the final state s' to real numbers.

Definition 5.7 (Final states are probabilistic, distributions, and subdistributions).



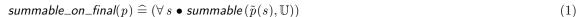
$$is_final_dist(p) \stackrel{.}{=} \underbrace{is_dist(\tilde{p})}$$
 (final distributions)
$$is_final_subdist(p) \stackrel{.}{=} is_subdist(\tilde{p})$$
 (final subdistributions)

For all initial states s, if such a curried expression \tilde{p} is probabilistic, a distribution, or a subdistribution, then we say p is probabilistic, a distribution, or a subdistribution over the final states, characterised by functions is_final_prob, is_final_dist, and is_final_subdist.

For an expression P of type $[S_1, S_2]$ prfun, if is_final_prob(\overline{P}), we say P is probabilistic. Similarly, we say P is a distribution or a subdistribution (over its final states), if $is_final_dist(\overline{P})$ or $is_final_subdist(\overline{P})$.

Using the function summable, we introduce convergence for relational expressions and for the product of relational expressions over final states.

Definition 5.8 (Summable on final states).



$$summable_on_final2(p,q) \stackrel{\frown}{=} (\forall s \bullet summable(\lambda s' \bullet p(s,s') * q(s,s'), \mathbb{U})) \tag{2}$$

The function summable_on_final characterises the relational expression p(s) over final states are summable on the universe \mathbb{U} of state space for any initial state s. The function summable_on_final2 characterises the product of the expressions p(s) and q(s) over final states that are summable on \mathbb{U} .

We also define functions below to characterise if the final states of a relational expression are reachable and the final states of two relational expressions are reachable at the same states.

Definition 5.9 (Reachable final states).

$$final_reachable(p) \stackrel{\frown}{=} (\forall s \bullet \exists s' \bullet p(s, s') > 0) \tag{1}$$

$$final_reachable2(p,q) \stackrel{\frown}{=} (\forall s \bullet \exists s' \bullet p(s,s') > 0 \land q(s,s') > 0)$$

$$\tag{2}$$

The final states of p are reachable, $final_reachable(p)$, if from any initial state s there exists at least one final state s' such that p(s, s') is larger than 0, or reaching s' from s is possible. The final_reachable2(p, q)characterises the possibility for p(s) and q(s) to reach the same state s' from any initial state s.

Convergence and reachability of a relational expression p can be derived from whether p is a distribution or subdistribution over its final states, as shown below.

$$is_final_dist(p) \Rightarrow \begin{pmatrix} is_prob(p) \land (\forall s \bullet \Sigma_{\infty} s' \bullet p(s, s') = 1) \land \\ summable_on_final(p) \land final_reachable(p) \end{pmatrix}$$
 (1)

$$is_final_dist(p) \Rightarrow \left(\begin{array}{c} is_prob(p) \wedge (\forall \, s \, \bullet \, \Sigma_{\infty} s' \, \bullet \, p(s,s') = 1) \wedge \\ summable_on_final(p) \wedge final_reachable(p) \end{array} \right)$$

$$is_final_subdist(p) \Rightarrow \left(\begin{array}{c} is_prob(p) \wedge (\forall \, s \, \bullet \, \Sigma_{\infty} s' \, \bullet \, p(s,s') > 0) \wedge (\forall \, s \, \bullet \, \Sigma_{\infty} s' \, \bullet \, p(s,s') \leq 1) \\ summable_on_final(p) \wedge final_reachable(p) \end{array} \right)$$

$$(2)$$

Law 1 shows if p is a distribution over its final states, then p is probabilistic, summable over its final states, and reachable. The second conjunct restates p as a distribution over its final states. Law 2 shows if p is a subdistribution over the final states, then p is probabilistic, the summation of p over its final states is larger than 0, and less than or equal to 1, and p is summable over its final states and reachable. The second and third conjuncts restate p as a subdistribution over its final states.

Normalisation $\mathcal{N}(p)$ is the distribution whose values are in the same proportion as the values of p. Here, p is not required to be a distribution, but the result of normalisation is a distribution.

Definition 5.10 (Normalisation). We fix p of type [S] rexpr.

 $\mathcal{N}(p) \mathrel{\widehat{=}} (p/(\Sigma_{\infty}s : S \bullet p(s)))_{e}$

In the definition, the division operator / in Isabelle/HOL is implemented as $inverse_divide^4$ in the division ring. The result a/b is 0 if either a or b is 0. The $\mathcal{N}(p)$ gives the distribution of the state space (both the initial and final states for a relational expression) in p. For example, suppose that x is in the 1..n range.

Example 5.1.

```
\mathcal{N}(\llbracket x' = x + 1 \lor x' = x + 2 \rrbracket)
= { Definition 5.10 }
    ([x' = x + 1 \lor x' = x + 2] / (\Sigma_{\infty}(x, x') : (1..n) \times (1..n) \bullet [x' = x + 1 \lor x' = x + 2]))_{\alpha}
= { Theorem 5.1 Law 11 }
    \left(\left(\llbracket x'=x+1\vee x'=x+2\rrbracket\right)/\left(\Sigma_{\infty}(x,x')\bullet\left(\llbracket x'=x+1\vee x'=x+2\rrbracket\right)*\llbracket(x,x')\in(1..n)\times(1..n)\rrbracket\right)\right)_{e}
= \{ \text{ Theorem } 5.1 \text{ Law } 6 \}
     \left( \begin{array}{l} (x'=x+1 \vee x'=x+2) \, / \\ \left( \Sigma_{\infty}(x,x') \bullet \left( \begin{array}{l} \llbracket x'=x+1 \rrbracket + \llbracket x'=x+2 \rrbracket - \\ \llbracket x'=x+1 \rrbracket * \llbracket x'=x+2 \rrbracket \end{array} \right) * \llbracket (x,x') \in (1..n) \times (1..n) \rrbracket \right) \right)_e 
= { Theorem 5.1 Law 5, \llbracket x' = x + 1 \rrbracket * \llbracket x' = x + 2 \rrbracket = \dot{0}, and summation distributes through sum }
     \left( \begin{array}{c} ([\![x'=x+1 \lor x'=x+2]\!]) / \\ \left( \begin{array}{c} \Sigma_{\infty}(x,x') \bullet [\![x'=x+1]\!] * [\![(x,x') \in (1..n) \times (1..n)]\!] + \\ \Sigma_{\infty}(x,x') \bullet [\![x'=x+2]\!] * [\![(x,x') \in (1..n) \times (1..n)]\!] \end{array} \right) \right) 
= { Theorem 5.1 Law 5 and Iverson bracket summation one-point rule }
   \left(\left(\llbracket x'=x+1\vee x'=x+2\rrbracket\right)/\left(\begin{array}{c} \Sigma_{\infty}x\bullet\llbracket(x,x+1)\in(1..n)\times(1..n)\rrbracket+\\ \Sigma_{\infty}x\bullet\llbracket(x,x+2)\in(1..n)\times(1..n)\rrbracket\end{array}\right)\right)_{e}
    (([x' = x + 1 \lor x' = x + 2]) / (\Sigma_{\infty} x \bullet [x \in 1..n - 1] + \Sigma_{\infty} x \bullet [x \in 1..n - 2]))_{e}
= { Iverson summation }
   (([x' = x + 1 \lor x' = x + 2]) / (n - 1 + n - 2))_e
= \{ arithmetic \}
    (([x' = x + 1 \lor x' = x + 2]) / (2 * n - 3))_{a}
```

Often we want the distribution of just the final state if p is a relational expression.

Definition 5.11 (Normalisation of the final state). We fix p of type $[S_1, S_2]$ rvfun,

$$\mathcal{N}_f(p) = (p/(\Sigma_{\infty} v_0 : S_2 \bullet p[v_0/\mathbf{v}']))_e$$

The value p(s, s') of p for a pair (s, s') of the initial state s and the final state s' is divided by the summation of $p(s, v_0)$ over the all final states v_0 for the initial state s. The normalisation of p is a distribution over its final states, given p is nonnegative and reachable, and $\tilde{p}(s)$ is convergent for any state s.

Theorem 5.5. is_nonneg(p)
$$\land$$
 final_reachable(p) \land summable_on_final(p) \Rightarrow is_final_dist($\mathcal{N}_f(p)$)

A probabilistic program P is a relational *ureal*-valued expression of type $[S_1, S_2]$ prfun. We show the conversion of P to a real-valued function \underline{P} is probabilistic and pointwise subtraction of \underline{P} from the constant 1 function is also probabilistic.

Theorem 5.6. Provided P is an expression of type $[S_1, S_2]$ prfun, then is_prob (\overline{P}) and is_prob $(1 - \overline{P})$.

⁴https://isabelle.in.tum.de/library/HOL/HOL/Fields.html.

The conversion of P to a real-valued function and then back to the ureal-valued function is still P.

Theorem 5.7. prfun_of_rvfun is the inverse of rvfun_of_prfun, that is,
$$(\overline{P}) = P$$
.

The conversion of p of type $[S_1, S_2]$ prfun to a ureal-valued function and then back to a real-valued function is still p if p is probabilistic.

Theorem 5.8. rvfun_of_prfun is the inverse of prfun_of_rvfun if
$$p$$
 is a probabilistic, that is, is_prob $(p) \Rightarrow \overline{(p)} = p$.

A corollary of this theorem, given below, states that the conversion of an Iverson bracket expression to *prfun* and then back to *rvfun* gives the expression itself because Iverson bracket expressions are probabilistic (Theorem 5.2).

Theorem 5.9.
$$\overline{\left(\llbracket p\rrbracket\right)}=\llbracket p\rrbracket.$$

5.4. Syntax and semantics

Our probabilistic programming language includes six constructs (except probabilistic recursions), and their semantics is given as follows.

Definition 5.12 (Probabilistic programs). We define probabilistic programs, interpreted as $[S_1, S_2]$ prfun (that is, ureal-valued functions), constructed from the syntax below where we fix P and Q as probabilistic programs of type $[S_1, S_2]$ prfun, r as an expression of type $[S_1, S_2]$ prfun, r as a relation of type $[S_1, S_2]$ urel, and r and r as relational real-valued expressions of type $[S_1, S_2]$ rvfun.

$$\begin{array}{c} \mathbb{Z}_p \triangleq \underline{\llbracket \mathbb{Z} \rrbracket} & \text{(skip)} \\ (x :=_p e) \triangleq \underline{\llbracket x := e \rrbracket} & \text{(assignment)} \\ (P \oplus_r Q) \triangleq \underline{\left(\overline{r} * \overline{P} + \left(\overline{1} - \overline{r}\right) * \overline{Q}\right)_e} & \text{(probabilistic choice)} \\ (\textbf{if}_c \, b \, \textbf{then} \, P \, \textbf{else} \, Q) \triangleq \underline{\left(\textbf{if} \, b \, \textbf{then} \, \overline{P} \, \textbf{else} \, \overline{Q}\right)_e} & \text{(conditional choice)} \\ P :_p Q \triangleq \underline{\overline{P} :_f \overline{Q}} \text{ where } R :_f T \triangleq (\Sigma_\infty v_0 \bullet R[v_0/\mathbf{v}'] * T[v_0/\mathbf{v}])_e & \text{(sequential composition)} \\ R \parallel T \triangleq \mathcal{N}_f \, (R * T)_e & \text{(parallel composition)} \end{array}$$

We note that the semantics of all these constructs are converting the corresponding real-valued expressions to the ureal-valued expressions by $prfun_of_rvfun$.

The probability skip \mathbb{Z}_p is the *ureal* version of the Iverson bracket of the relational skip \mathbb{Z} . It changes no variables and terminates immediately. On termination, the final state equals the initial state with probability 1; all other assignments to the final state have probability 0. Similarly, the probability assignment $(x :=_p e)$ is the *ureal* version of the Iverson bracket of the relational assignment (x := e). An assignment is a one-point distribution of the final state.

The probabilistic choice $(P \oplus_r Q)$, also denoted **if**_p r **then** P **else** Q, is the weighted sum of \overline{P} (the conversion of P to the real-valued expression) and \overline{Q} based on their weights \overline{r} and $(\dot{1} - \overline{r})$. Because of the type $[S_1, S_2]$ prfun of r, both \overline{r} and $\dot{1} - \overline{r}$ are probabilistic (or between 0 and 1) by Theorem 5.6.

In the conditional choice (if $_c$ b then P else Q), b is a relation. If b is evaluated to true, the choice is \overline{P} . Otherwise, it is \overline{Q} .

Sequential composition $(P;_p Q)$ is the serial composition of \overline{P} with \overline{Q} by $;_f$ where both P and Q have the same type of [S] prhfun. The $R;_f T$, where both R and T are type of [S] prfun, is the conditional probability of T given R. Indeed, it is the summation of the product of $R[v_0/\mathbf{v}']$, the substitution of v_0 for \mathbf{v}' in R, and $T[v_0/\mathbf{v}]$, the substitution of v_0 for \mathbf{v} in T, over their intermediate states v_0 . Its semantics can be interpreted as starting from an initial state s, the probability of $(P;_p Q)$ reaching a final state s' is equal

to the summation of the probability of \overline{Q} reaching state s' from v_0 , given the probability of \overline{P} reaching v_0 from s, over all intermediate states v_0 .

The $R \parallel T$ is the parallel composition of R with T, semantically as normalisation of the product of R and T. It is the joint probability of R and T. In its most general form, neither R nor T need to be proper probabilistic programs, but the result will be a probabilistic program.

In Bayesian inference, the posterior probability of A given B is computed based on a prior probability, which is estimated before B is observed, a likelihood function over B given fixed A, and model evidence B according to Bayes' theorem given below.

$$posterior = \frac{prior * likelihood}{evidence} \qquad or \qquad P(A \mid B) = \frac{P(A)P(B \mid A)}{P(B)}$$

where B is a new observed data or evidence. In our programming language, the update of the posterior probability is modelled using parallel composition for learning new facts and sequential composition for making actions, such as the movement of robots. This is illustrated in the forgetful Monty, the robot localisation, and the COVID diagnosis examples in Sects. 7.3, 7.4, and 7.5.

In other work [2, 15], this learning or conditioning is encoded using an **observe** statement such as **observe**(ϕ) where ϕ is a boolean expression or a predicate defined over program variables. This statement normalises all valid executions that satisfy ϕ with respect to the probability of total valid executions and blocks invalid executions (with probability 0). Comparatively, our parallel composition or Hehner's is not restricted to predicates. Indeed, R and T in $R \parallel T$ can be any real-valued or *ureal*-valued expressions, which are more general likelihood functions. To encode predicates similar to ϕ in the **observe** statement, we need to use $\llbracket \phi \rrbracket$ to convert it. As illustrated in the examples in Sects. 7.3, 7.4, and 7.5, three likelihood functions are $\llbracket m' \neq p' \rrbracket$, $(3 * \llbracket door(bel') \rrbracket + 1)_e$, and $\llbracket ct' = Pos \rrbracket$.

5.5. Top and bottom

According to Theorem 4.5, the set of probabilistic programs is a complete lattice under \leq . The top and bottom elements of the lattice satisfy the properties below.

Theorem 5.10. Provided that P is a probabilistic program and p is a real-valued function.

The top element \top is $\mathring{1}$ and the bottom \bot is $\mathring{0}$. Any probabilistic program P is between \bot and \top . The constant $\mathring{1}$ and $\mathring{1}$ mutually correspond in *ureal*-valued and real-valued functions, and they can be converted to each other. This is similar for $\mathring{0}$ and $\mathring{0}$. Real-valued functions and *ureal*-valued functions satisfy the right-zero and right-one laws.

5.6. Skip

The \mathbb{I}_p is a special case of assignment $x :=_p x$ and is also a distribution as shown below.

$$\mathbf{I}_p = (x :=_p x) \tag{1}$$

$$is_final_dist(\overline{\mathbb{I}_p})$$
 (2)

$$\overline{\left(\llbracket \mathbf{I} \rrbracket \right)} = \llbracket \mathbf{I} \rrbracket \tag{3}$$

Law 3 shows that the conversion of $\llbracket \mathbb{I} \rrbracket$ to the *ureal*-valued function, and then back to the real-valued function is still $\llbracket \mathbb{I} \rrbracket$.

5.7. Assignments

A probabilistic assignment is a distribution.

Theorem 5.12. is_final_dist
$$(\overline{x} :=_p e)$$

5.8. Probabilistic choice

Probabilistic choice preserves various properties below.

$$\textit{is_final_dist}\left(\overline{P}\right) \land \textit{is_final_dist}\left(\overline{Q}\right) \Rightarrow \textit{is_final_dist}\left(\overline{P} \oplus_r \overline{Q}\right) \tag{1}$$

$$(P \oplus_r Q) = (Q \oplus_{\mathring{1}-r} P) \tag{2}$$

$$(P \oplus_{\mathring{0}} Q) = Q \tag{3}$$

$$(P \oplus_{\hat{1}} Q) = P \tag{4}$$

$$(P \oplus_r Q) = \overline{r} * \overline{P} + (\dot{1} - \overline{r}) * \overline{Q}$$
 (5)

Law 1 shows if P and Q are distributions, then the probabilistic choice is also a distribution. Laws 2 to 4 state the probabilistic choice is quasi-commutative, a zero, and a unit.

The probabilistic choice is also quasi-associative.

Theorem 5.14 (Quasi-associativity). We fix
$$w_1, w_2, r_1, r_2 : [S]$$
 urexpr,

 $(\mathring{1} - w_1) * (\mathring{1} - w_2) = (\mathring{1} - r_2) \land \underline{w_1 = r_1 * r_2} \Rightarrow (P \oplus_{w_1} (Q \oplus_{w_2} R)) = ((P \oplus_{r_1} Q) \oplus_{r_2} R)$

The probabilistic choice is quasi-associative under the two assumptions involving $w_1, w_2, r_1, and r_2$.

5.9. Conditional choice

Conditional choice satisfies various properties below.

$$\textit{is_final_dist}\left(\overline{P}\right) \land \textit{is_final_dist}\left(\overline{\mathbf{Q}}\right) \Rightarrow \textit{is_final_dist}\left(\overline{\mathbf{if}_c \ b \ \mathbf{then} \ P \ \mathbf{else} \ Q}\right) \tag{1}$$

$$(\mathbf{if}_c \ b \ \mathbf{then} \ P \ \mathbf{else} \ P) = P \tag{2}$$

$$(\mathbf{if}_c \ b \ \mathbf{then} \ P \ \mathbf{else} \ Q) = \left(P \oplus_{\llbracket b \rrbracket} \ Q\right) \tag{3}$$

$$(P_1 \le P_2 \land Q_1 \le Q_2) \Rightarrow (\mathbf{if}_c \ b \ \mathbf{then} \ P_1 \ \mathbf{else} \ Q_1) \le (\mathbf{if}_c \ b \ \mathbf{then} \ P_2 \ \mathbf{else} \ Q_2)$$
 (4)

Law 1 shows if P and Q are distributions, then the conditional choice is also a distribution. Law 2 shows the conditional choice between P and P is P itself. A conditional choice is a special form of probabilistic choice, as given in Law 3, with its weight $\llbracket b \rrbracket$ being the Iverson bracket of b: either 0 (if b is evaluated to false) or 1 (if b is evaluated to true). The conditional choice is also monotonic, shown in Law 4.

5.10. Sequential Composition

A variety of properties are held for sequential composition. We note that r and t below are predicates of type [S] pred.

Theorem 5.16.

$$is_final_dist\left(\overline{P}\right) \land is_final_dist\left(\overline{Q}\right) \Rightarrow is_final_dist\left(\overline{P};_{p}\overline{Q}\right)$$
 (1)

$$\mathring{0};_{p} P = \mathring{0}$$
(2)

$$P;_{p} 0 = 0$$

$$\mathbb{I}_p :_{n} P = P \tag{4}$$

$$P;_{n} \mathbb{I}_{p} = P \tag{5}$$

$$is_final_dist(\overline{P}) \Rightarrow P;_p \mathring{1} = \mathring{1}$$
 (6)

$$(P_1 \le P_2 \land Q_1 \le Q_2) \Rightarrow (P_1;_p Q_1) \le (P_2;_p Q_2) \tag{7}$$

$$is_final_dist\left(\overline{P}\right) \land is_final_dist\left(\overline{Q}\right) \land is_final_dist\left(\overline{R}\right) \Rightarrow \left(P;_{p}\left(Q;_{p}R\right) = \left(P;_{p}Q\right);_{p}R\right) \tag{8}$$

$$\textit{is_final_subdist}\left(\overline{P}\right) \land \textit{is_final_subdist}\left(\overline{Q}\right) \land \textit{is_final_subdist}\left(\overline{R}\right) \Rightarrow \left(P :_{p} \left(Q :_{p} R\right) = \left(P :_{p} Q\right) :_{p} R\right) \tag{9}$$

$$\textit{is_final_subdist}\left(\overline{P}\right) \Rightarrow \left(P ;_p \left(\mathbf{if}_c \ b \ \mathbf{then} \ Q \ \mathbf{else} \ R\right) = \underline{\left(\overline{\left(P ;_p \left(\llbracket b \rrbracket \ast Q\right)\right)} + \overline{\left(P ;_p \left(\llbracket \neg \ b \rrbracket \ast R\right)\right)}\right)_e}\right) \tag{10}$$

$$\llbracket r \rrbracket :_p \llbracket t \rrbracket = (\Sigma_{\infty} v_0 \bullet \llbracket r[v_0/\mathbf{v}'] \wedge t[v_0/\mathbf{v}] \rrbracket)_e \tag{11}$$

If P and Q are distributions, then sequential composition of P and Q is also a distribution (Law 1). Sequential composition is left zero (Law 2), right zero (Law 3), left unit (Law 4), right unit (Law 5), monotonic (Law 7), and associative (Laws 8 and 9 if P, Q, and R are distributions or subdistributions). If P is a subdistribution, then Law 10 shows P is distributive through conditional choice.

An interesting Law 6 ($^{\diamond}$) states if P is a distribution over the final state, then sequence composition of P with $\mathring{1}$ is $\mathring{1}$. Its proof is given below.

Proof.

```
\begin{split} &P :_{p} \mathring{1} \\ &= \{ \text{ Definition (sequential composition)} \} \\ &\underbrace{\left( \Sigma_{\infty} v_{0} \bullet \overline{P}[v_{0}/\mathbf{v}'] * \mathring{1}[v_{0}/\mathbf{v}] \right)_{e}}_{e} \\ &= \{ \text{ Theorem 5.10 and } \mathbb{1}[v_{0}/\mathbf{v}] = \mathbb{1} \} \\ &\underbrace{\left( \Sigma_{\infty} v_{0} \bullet \overline{P}[v_{0}/\mathbf{v}'] * \mathbb{1} \right)_{e}}_{e} \\ &= \{ \text{ Pointwise multiplication and multiplication unit law: } x * 1 = x \} \\ &\underbrace{\left( \Sigma_{\infty} v_{0} \bullet \overline{P}[v_{0}/\mathbf{v}'] \right)_{e}}_{e} \\ &= \{ \text{ Assumption: } is\_final\_dist\left( \overline{P} \right) \text{ and Theorem 5.4 Law 1: } \left( \forall s \bullet \Sigma_{\infty} s' \bullet \overline{P}(s,s') = 1 \right) \} \\ &\underline{\mathbb{1}} \\ &= \{ \text{ Theorem 5.10} \} \\ &\mathring{\mathbb{1}} \end{split}
```

The sequential composition of two Iverson bracket expressions can be simplified to the summation of the Iverson bracket of conjunction, as shown in Law 11 (3). This law is proved below.

Proof.

$$= \{ \text{ Theorem 5.9} \}$$

$$\underline{(\Sigma_{\infty}v_0 \bullet \llbracket r \rrbracket \llbracket (v_0/\mathbf{v}') * \llbracket t \rrbracket \llbracket (v_0/\mathbf{v}) \rrbracket_e}$$

$$= \{ \text{ Substitution distributive through Iverson bracket} \}$$

$$\underline{(\Sigma_{\infty}v_0 \bullet \llbracket r \llbracket (v_0/\mathbf{v}') \rrbracket * \llbracket t \llbracket (v_0/\mathbf{v}) \rrbracket)_e}$$

$$= \{ \text{ Theorem 5.1 Law 5} \}$$

$$\underline{(\Sigma_{\infty}v_0 \bullet \llbracket r \llbracket (v_0/\mathbf{v}') \wedge t \llbracket (v_0/\mathbf{v}) \rrbracket)_e}$$

A corollary of this law, given below, states that if the two expressions cannot agree on an intermediate state v_0 , the sequence is just $\mathring{0}$.

Theorem 5.17.
$$c_1 \neq c_2 \Rightarrow [x' = c_1];_p [x = c_2] = 0$$

The intermediate state can be ignored if the two expressions agree on one intermediate state c_1 .

Theorem 5.18.
$$[\![x=c_0 \land x:=c_1]\!] :_p [\![x=c_1]\!] = [\![x=c_0]\!]$$

The intermediate state can be ignored if the second expression is also a point distribution, but its final state is still specified.

Theorem 5.19.
$$[\![x=c_0 \land x:=c_1]\!] :_p [\![x=c_1 \land x:=c_2]\!] = [\![x=c_0 \land x'=c_2]\!]$$

5.11. Normalisation

The \mathcal{N}_f in Definition 5.11 gives a distribution of the final state, that is, over all variables in the state space. We also want the distribution of just one particular variable instead of all, for example, to define a uniform distribution. For this purpose, we define the alphabetised normalisation below.

Definition 5.13 (Alphabetised normalisation). We fix p of type $[S_1, S_2]$ rvfun and a program variable x of type T_x ,

$$\mathcal{N}_{\alpha}(x,p) \stackrel{\frown}{=} (p/(\Sigma_{\infty}x_0: T_x \bullet p[x_0/x']))_{\alpha}$$

Uniform distributions are defined using \mathcal{N}_{α} .

Definition 5.14 (Uniform distributions). We fix a program variable x of type T_x and a finite set A of type \mathbb{P} T_x ,

$$\mathcal{U}\left(x,A\right) \widehat{=} \mathcal{N}_{\alpha}\left(x,\left[\!\left[\bigsqcup v \in A \bullet x := v\right]\!\right]\right)$$

A uniform distribution of x from a finite set A is an alphabetised normalisation of a program $[\![\, \,] \, v \in A \bullet x := v]\!]$, nondeterministic choice $[\![\, \,] \,]$ of the value of x from A, over x'. Here, $[\![\, \,] \,]$ is inside the Iverson bracket and is a UTP relation operator infimum, simply disjunction $[\![\, \,] \,]$.

The uniform distribution operator satisfies the properties below.

Theorem 5.20. We fix P of type $[S_1, S_2]$ prfun,

$$\mathcal{U}\left(x,\varnothing\right) = \dot{0}\tag{1}$$

$$finite(A) \Rightarrow is_prob(\mathcal{U}(x, A))$$
 (2)

$$finite(A) \land A \neq \varnothing \Rightarrow is_final_dist(\mathcal{U}(x, A))$$

$$\tag{3}$$

$$finite(A) \land A \neq \varnothing \Rightarrow (\forall v \in A \bullet \mathcal{U}(x, A);_{n} \llbracket x = v \rrbracket = (1/\operatorname{card}(A))_{e}) \tag{4}$$

$$\mathit{finite}(A) \land A \neq \varnothing \Rightarrow \Big(\mathcal{U}\left(x,A\right) = \left[\left[\bigcup v \in A \bullet x := v \right] \right] / \mathit{card}(A) \Big) \tag{5}$$

$$\mathit{finite}(A) \land A \neq \varnothing \Rightarrow \left(\underline{\mathcal{U}(x,A)};_{p} P = \underline{\left(\Sigma_{\infty} v \in A \bullet \overline{P}[v/x] \right) / \mathit{card}(A)} \right) \tag{6}$$

Law 1 shows the distribution over an empty set \varnothing is just the constant function $\dot{0}$. Provided A is finite, then $\mathcal{U}(x,A)$ is probabilistic (Law 2). If A is also not empty $(A \neq \varnothing)$, $\mathcal{U}(x,A)$ is also a distribution (Law 3). Under the same assumptions about A, $\mathcal{U}(x,A)$ is truly a uniform distribution, that is, x being any value from A is equally likely $(1/\operatorname{card}(A))$ where $\operatorname{card}(A)$ is the cardinality of A), as shown in Law 4, where we use sequential composition $\mathcal{U}(x,A)$; $\mathbb{I}[x=v]$ to express the probability of x being a particular value x. The distribution $\mathcal{U}(x,A)$ can be simplified to another form, shown in Law 5. The sequence of a uniform distribution and x is a left one-point, shown in Law 6.

5.12. Parallel composition

The (parallel composition) is defined over real-valued functions and specifies a *ureal*-valued function. It satisfies the properties below.

Theorem 5.21. Fix
$$p$$
, q , and r of type $[S_1, S_2]$ rvfun, and P , Q , and R of type $[S_1, S_2]$ prfun.

$$is_nonneg(p*q) \Rightarrow is_prob(\mathcal{N}_f(p*q)_g)$$
 (1)

$$\left(\begin{array}{c} \textit{is_final_prob}(p) \land \textit{is_final_prob}(q) \land \\ (\textit{summable_on_final}(p) \lor \textit{summable_on_final}(q)) \land \textit{final_reachable2}(p,q) \end{array} \right) \Rightarrow \textit{is_final_dist}(p \parallel q)$$
 (2)

$$(is_nonneg(p) \land is_nonneg(q) \land \neg final_reachable2(p,q)) \Rightarrow p \parallel q = \mathring{0}$$

$$(3)$$

$$\dot{0} \parallel p = \mathring{0} \tag{4}$$

$$p \parallel \dot{0} = \mathring{0} \tag{5}$$

$$c \neq 0 \land is_final_dist(p) \Rightarrow (\lambda s \bullet c) \parallel p = p \tag{6}$$

$$c \neq 0 \land is_final_dist(p) \Rightarrow p \parallel (\lambda s \bullet c) = p$$
 (7)

$$p \parallel q = q \parallel p \tag{8}$$

$$\left(\begin{array}{l} \textit{is_nonneg}(p) \land \textit{is_nonneg}(q) \land \textit{is_nonneg}(r) \land \\ \textit{summable_on_final2}(p,q) \land \textit{summable_on_final2}(q,r) \land \\ \textit{final_reachable2}(p,q) \land \textit{final_reachable2}(q,r) \end{array} \right) \Rightarrow (p \parallel q) \parallel r = p \parallel (q \parallel r)$$
 (9)

$$summable_on_final(\overline{Q}) \Rightarrow (\overline{P} \parallel \overline{Q}) \parallel \overline{R} = \overline{P} \parallel (\overline{Q} \parallel \overline{R})$$
 (10)

$$\mathit{finite}(A) \land A \neq \varnothing \Rightarrow \Big(\mathcal{U}\left(x,A\right) \parallel p = \underline{\left(\left(\Sigma_{\infty}v \in A \bullet \llbracket x := v \rrbracket * p[v/x']\right) / \left(\Sigma_{\infty}v \in A \bullet p[v/x']\right)\right)_e} \Big) \tag{11}$$

Law 1 shows the normalisation of the product p*q of p, and q is probabilistic if p*q is nonnegative. If both p and q are probabilistic and summable on their final states, reachable on at least one same final state at the same time, then $p \parallel q$ is also a distribution of the final state (Law 2). If both p and q are nonnegative and not reachable on at least one same final state at the same time (or a contradiction between p and q), then $p \parallel q$ is a zero (Law 3).

Parallel composition is a left zero (Law 4) and a right zero (Law 5), and a left unit (Law 6) and a right unit (Law 7) if a constant c is not 0 and p is a distribution. It is also commutative (Law 8).

Law 9 shows if p, q, and r are nonnegative, both p and q are summable and reachable on their product, and both q and r are summable and reachable on their product, then the parallel composition is associative. If, however, p, q, and r are converted from probabilistic programs P, Q, and R, and also Q is summable on its final state, then the parallel composition is also associative (Law 10).

Law 11 shows if A is finite and not empty, then the parallel composition of a uniform distribution over x from A and p can be simplified to a division whose numerator denotes the value of p reaching a final state with x being a particular value v and whose denominator represents the summation of the values of p reaching final states with x being any value v from A.

6. Recursion

This section presents the syntax and semantics of probabilistic recursions. We use the Kleene fixed-point theorem to construct fixed points iteratively.

6.1. Probabilistic loops

We introduce the syntax for the least and greatest fixed points based on the Knaster–Tarski fixed-point theorem 3.4.

Definition 6.1 (Least and greatest fixed points).



$$\mu_{p} X \bullet P \stackrel{\frown}{=} \mu \ (\lambda X \bullet P)$$
 (least fixed point)
$$\nu_{\theta} X \bullet P \stackrel{\frown}{=} \nu \ (\lambda X \bullet P)$$
 (great fixed point)

We define a loop function \mathcal{F} below and use it to define probabilistic while loops later.

Definition 6.2 (Loop function). We fix a homogeneous relation b : [S] hrel, and homogeneous probabilistic programs P and X of type [S] prhfun, then

$$\mathcal{F}(b, P, X) \stackrel{\triangle}{=} \mathbf{if}_c b \mathbf{then} (P;_p X) \mathbf{else} \, \mathbb{I}_p$$
 (loop function)

The function \mathcal{F} is a conditional choice between the sequence $(P;_p X)$ and the skip \mathbb{I}_p , depending on the relation b. We use \mathcal{F}_P^b as a shorthand for $\lambda X \bullet \mathcal{F}(b, P, X)$. Then $\mathcal{F}_P^b(X)$ can be expressed below.

If a probabilistic program P is a distribution, then \mathcal{F}_{P}^{b} is monotonic.

Theorem 6.1.
$$is_final_dist(\overline{P}) \Rightarrow mono(\mathcal{F}_P^b)$$



With \mathcal{F} , we define two probabilistic loops using the least and the greatest fixed points. The reason we define two probabilistic loops is to establish the uniqueness theorem of fixed points. Based on Knaster–Tarski fixed-point Theorem 3.4, the uniqueness is equivalent to the equality of the least and greatest fixed points.

Definition 6.3 (Probabilistic loops).



while_p b do P od
$$\hat{=} \mu_p X \bullet \mathcal{F}_P^b(X)$$
 (while loop by least fixed point)
while_p b do P od $\hat{=} \mu_p X \bullet \mathcal{F}_P^b(X)$ (while loop by greatest fixed point)

The denotational semantics of recursions in programming is usually defined on the lfp [72] or the weakest fixed point in UTP [60], as we do here for the probabilistic loop (**while**_p b **do** P **od**). But why the lfp is commonly used to give the denotation semantics for recursive, instead of the gfp? Gunter and Scott [72] stated that it is intuitively reasonable and lfp yields a canonical solution. Hoare and He [60] argued that the

weakest fixed point (wfp) is more implementable (but might be non-terminating) and the strongest fixed point (sfp) might be not implementable such as the miracle. The sfp is useful to prove the correctness of recursion programs, but subject to two problems: invalidate the implication in UTP to model correctness of designs, and difficult to implement nondeterminism. However, sfp can be used to establish the uniqueness of fixed points, and so wfp and sfp are the same and will not be subject to the problems for each of them. Our definition of $(\mathbf{while}_p^{\mathsf{T}} \ b \ \mathbf{do} \ P \ \mathbf{od})$ is for the same reason, merely for the proof of the unique fixed point theorem.

The **while**_p b **do** P **od** satisfy several laws below.

$$is_final_dist(\overline{P}) \Rightarrow \textit{while}_p \ b \ \textit{do} \ P \ \textit{od} = \mathcal{F}_P^b (\textit{while}_p \ b \ \textit{do} \ P \ \textit{od})$$
 (unfold)

$$while_p \ false \ do \ P \ od = II_p$$
 (false)

while_p true do
$$P$$
 od = $\mathring{0}$ (true)

If P is a distribution, **while**_p b **do** P **od** can be unfolded without its semantics changed. If b is **false**, the loop is just \mathbb{I}_p . It is $\mathring{0}$ if b is **true**.

The **while**_p^{\top} b **do** P **od** satisfy similar laws below.

$$is_final_dist\left(\overline{P}\right) \Rightarrow \textit{while}_p^{\top} \ b \ \textit{do} \ P \ \textit{od} = \mathcal{F}_P^b \left(\textit{while}_p^{\top} \ b \ \textit{do} \ P \ \textit{od}\right)$$
 (unfold)

$$\mathbf{\textit{while}}_{p}^{\top} \mathbf{\textit{false do}} P \mathbf{\textit{od}} = \mathbf{I}_{p}$$
 (false)

is_final_dist
$$(\overline{P}) \Rightarrow \mathbf{while}_n^{\mathsf{T}} \mathbf{true} \, \mathbf{do} \, P \, \mathbf{od} = \mathring{1}$$
 (true)

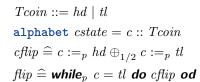
We note that if P is a distribution and b is **true**, the loop is just 1, instead of 0 for **while**_p. In Theorems 6.2 and 6.3, both loops are well defined if P is a distribution. To reason about nested loops, we need to give the semantics (that is, the fixed point p) to the innermost loop first, prove p is a distribution, and then move to the next loop closed to that loop. If, however, the inner loop does not almost surely terminate, currently we cannot give the semantics to the outer loop because this loop is not well defined. One line of our future work is to investigate the weakened condition of P from distributions to subdistributions to allow us to give the semantics to nested loops if the inner loop does not terminate almost surely.

As the semantics for probabilistic loops in our programming language is the (least fixed point) and the (great fixed point) in Knaster-Tarski fixed-point Theorem 3.4, this does not give information about how iterations can compute fixed points. We resort to Kleene fixed-point Theorem 3.6 for iterations. Our use of iterations to compute fixed points is motivated by a simple probabilistic program: flip a coin until the outcome is heads, defined as follows.

6.2. Motivation example

This example [31, 34, 26] is about flipping a coin till the outcome is heads.

Definition 6.4 (Flip a coin).



Tcoin is a free type in the Z notation or an enumerable data type in Isabelle, and it contains two constants hd and tl of type Tcoin. The keyword **alphabet** is used to declare the state space of a program, and it is cstate in this case. This state space is composed of only one variable c of type Tcoin, denoting the outcome

of the coin flip experiment. The cflip is a probabilistic choice denoting both hd and tl are equally likely, or have a uniform distribution over two outcomes. Finally, this program flip is a while loop whose condition is c = tl, specifying that the outcome is tl and whose body is cflip. The cflip is simplified as follows.

```
cflip = \{ \text{ Definition of } cflip \} 
c :=_{p} hd \oplus_{1/2} c :=_{p} tl 
= \{ \text{ Law 5 in Theorem 5.13 and Definition (assignment)} \} 
\overline{1/2} * \overline{\left( \llbracket c :=_{p} hd \rrbracket \right)} + \left( \overline{1} - \overline{1/2} \right) * \overline{\left( \llbracket c :=_{p} tl \rrbracket \right)} 
= \{ \text{ Conversion Definitions 5.2 and 4.3, and Theorem 5.8} \} 
1/2 * \llbracket c :=_{p} hd \rrbracket + 1/2 * \llbracket c :=_{p} tl \rrbracket 
= \{ \text{ Definition (assignment)} \} 
1/2 * \llbracket c' = hd \rrbracket + 1/2 * \llbracket c' = tl \rrbracket  (cflip altdef)
```

The cflip is also a distribution.

Lemma 6.4.
$$is_final_dist(\overline{cflip})$$

Proof.

```
{ Theorem 5.12 } is\_final\_dist(\overline{c} :=_p \overline{hd}) \land is\_final\_dist(\overline{c} :=_p \overline{tl}) \Rightarrow \{ \text{ Theorem 5.13 Law 1} \} is\_final\_dist(\overline{cflip})
```

Then the loop function $\mathcal{F}_{cflip}^{c=tl}(X)$, denoted as \mathcal{F}_c , is further simplified.

Now consider $F^n(\perp)$ in the Kleene fixed-point theorem, and here F is \mathcal{F}_c and \perp is $\mathring{0}$.

$$\begin{split} \left(\mathcal{F}_{c}\right)^{0}(\mathring{0}) = &\mathring{0} \\ \left(\mathcal{F}_{c}\right)^{1}(\mathring{0}) = &\{\text{Law (loop body of } \textit{flip)}\} \\ & \underline{ \left[c = tl \right] * \left(\left(\underline{1/2 * \left[c' = hd \right] + 1/2 * \left[c' = tl \right] \right) ;_{p} \mathring{0} \right) + \left[c = hd \right] * \left[c' = c \right] } \\ = &\{\text{Right Zero Theorem 5.16 Law 3}\} \\ & \underline{ \left[c = hd \right] * \left[c' = c \right] } \\ \left(\mathcal{F}_{c}\right)^{2}(\mathring{0}) = &\{F^{2}(\mathring{0}) = F(F^{1}(\mathring{0})) \text{ and Law } (\mathcal{F}_{P}^{b} \text{ altdef})} \} \end{split}$$

The $(\mathcal{F}_c)^n$ ($\mathring{0}$) corresponds to the termination probability after up to n-1 iterations of the loop body of flip in Definition 6.4. For example, $(\mathcal{F}_c)^1$ ($\mathring{0}$) corresponds to zero iterations or immediate termination. Its semantics is $\underline{\llbracket c = hd \rrbracket} * \underline{\llbracket c' = c \rrbracket}$ which means if the initial value of c is hd, then the final value is also hd with probability 1. For example, $(\mathcal{F}_c)^3$ ($\mathring{0}$) corresponds to the termination after up to two iterations, including the immediate termination, the termination after exact one iteration ([c = tl] * [c' = hd]/2, meaning the initial value of c is tl, and the outcome of the flip is hd with probability 1/2), and exact two iterations ($[c = tl] * [c' = hd]/2^2$).

Now consider $F^n(\top)$ in the Kleene fixed-point theorem, and here \top is $\mathring{1}$.

```
\begin{split} (\mathcal{F}_c)^0 \,(\mathring{\mathbf{1}}) = & \{ \text{Law (loop body of } \mathit{flip}) \} \\ & \mathbb{I}_c = \mathit{tt} \mathbb{I} * \overline{\left( \left( \frac{1}{2} * \mathbb{I}_c' = \mathit{hd} \mathbb{I} + 1/2 * \mathbb{I}_c' = \mathit{tt} \mathbb{I} \right) ;_p \mathring{\mathbf{1}} \right)} + \mathbb{I}_c = \mathit{hd} \mathbb{I} * \mathbb{I}_c' = \mathit{c} \mathbb{I} \\ & = \{ \text{Lemma } 6.4 \text{ and Theorem } 5.16 \text{ Law } 6 \} \\ & \mathbb{I}_c = \mathit{tt} \mathbb{I} + \mathbb{I}_c = \mathit{hd} \mathbb{I} * \mathbb{I}_c' = \mathit{c} \mathbb{I} \\ & (\mathcal{F}_c)^2 \,(\mathring{\mathbf{1}}) = \{ F^2 \,(\mathring{\mathbf{1}}) = F(F^1 \,(\mathring{\mathbf{1}})) \text{ and Law } (\mathcal{F}_P^b \text{ altdef}) \} \\ & \mathbb{I}_c = \mathit{tt} \mathbb{I} * \overline{\left( \mathit{cflip} ;_p \left( F_{\mathit{cflip}}^{c=\mathit{tt}} \right)^1 \,(\mathring{\mathbf{1}}) \right)} + \mathbb{I}_c = \mathit{tt} \mathbb{I} * \mathbb{I} \mathbb{I} \\ & = \{ \text{Law } \, (\mathit{cflip} \text{ altdef}), \, (F_{\mathit{cflip}}^{c=\mathit{tt}})^1 \,(\mathring{\mathbf{1}}), \, \text{Theorem } 3.1 \text{ Law } \, (\text{skip}), \, \text{and Theorem } 5.1 \text{ Law } 5 \} \\ & \mathbb{I}_c = \mathit{tt} \mathbb{I} * \overline{\left( \frac{1}{2} * \mathbb{I}_c' = \mathit{hd} \mathbb{I} + \frac{1}{2} * \mathbb{I}_c' = \mathit{tt} \mathbb{I} \right)} :_p \left( \mathbb{I}_c = \mathit{tt} \mathbb{I} + \frac{1}{2} * \mathbb{I}_c' = \mathit{c} \mathbb{I} \right) \right)} + \mathbb{I}_c = \mathit{hd} \mathbb{I} * \mathbb{I}_c' = \mathit{c} \mathbb{I} \\ & = \{ \text{Definition } \, (\text{sequential composition}), \, \text{Theorem } 5.17, \, \cdots \} \end{split}
```

We notice that $(\mathcal{F}_c)^n$ ($\mathring{1}$) above is an addition of three operands of which the last two are the same as $(\mathcal{F}_c)^n$ ($\mathring{0}$) in (iteration from bot). The extra operand $\llbracket c = t t \rrbracket / 2^{n-1}$ converges to 0 when n approaches ∞ , and so eventually $(\mathcal{F}_c)^n$ ($\mathring{0}$) and $(\mathcal{F}_c)^n$ ($\mathring{1}$) coincide at ∞ . This is illustrated in Fig. 3 where the common part $\llbracket c = h d \rrbracket * \llbracket c' = c \rrbracket$ in $(\mathcal{F}_c)^n$ ($\mathring{0}$) and $(\mathcal{F}_c)^n$ ($\mathring{1}$) is omitted. As shown in the diagram, along with the

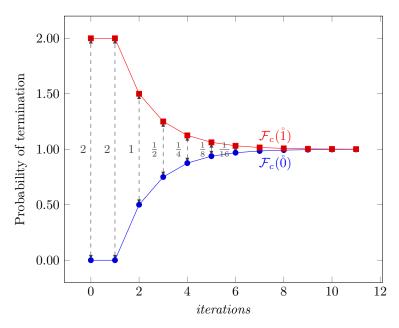


Figure 3: Termination probability over iterations from bottom and top for coin flip.

increasing iterations, $(\mathcal{F}_c)^n$ ($\mathring{0}$) increases towards 1 from the initial 0, while $(\mathcal{F}_c)^n$ ($\mathring{1}$) decreases towards 1 from the initial 2. Their differences, marked with dashed lines, are becoming smaller and smaller. From this example, we observe that there is one unique fixed point where the least fixed point and the greatest fixed point are the same. The precondition for this uniqueness is their iteration differences converging to 0. If this is the case, we must find a fixed point and prove it to reason about a probabilistic loop. Then the fixed point is the semantics of the loop. We do not need to compute it by iterations.

This example motivates us to give semantics to probabilistic loops as follows. First, we need to prove the loop function is continuous. Then if the differences of iterations from top and bottom converge to 0, there is a unique fixed point. Otherwise, we use the Kleene fixed-point Theorem 3.6 to compute the least and greatest fixed points by iterations.

6.3. Fixed point theorems

We define the function \mathcal{I} below recursively for iterations $(\mathcal{F}_P^b)^n(X)$, and the function $\mathcal{I}\mathcal{D}$ for the differences between iterations from top and bottom.

Definition 6.5 (Iteration and iteration difference).

$$\mathcal{I}(n,b,P,X) \cong \left(\mathbf{if} \ n = 0 \ \mathbf{then} \ X \ \mathbf{else} \ \mathcal{F}_P^b \left(\mathcal{I}(n-1,b,P,X)\right)\right) \tag{iteration}$$

$$\mathcal{F}_0(b,P,X) \cong \left(\mathbf{if} \ c \ b \ \mathbf{then} \ \left(P \ _{:p} \ X\right) \ \mathbf{else} \ \mathring{\mathbf{0}} \right)$$

$$\mathcal{ID}(n,b,P) \cong \left(\mathbf{if} \ n = 0 \ \mathbf{then} \ \mathring{\mathbf{1}} \ \mathbf{else} \ \mathcal{F}_0 \left(b,P,\mathcal{ID} \left(n-1,b,P\right)\right)\right) \tag{iteration difference}$$

The \mathcal{F}_0 is similar to \mathcal{F} in Definition (loop function) except that if the condition b does not hold, it is $\mathring{0}$ instead of \mathbb{I}_p in \mathcal{F} .

We show that $\mathcal{ID}(n, b, P)$ indeed captures the difference between iterations from top and bottom for any n.

Theorem 6.5. Provided P is a distribution, that is, is_final_dist(P).

$$\forall n : \mathbb{N} \bullet \mathcal{F}_{D}^{b^{n}}(\mathring{1}) - \mathcal{F}_{D}^{b^{n}}(\mathring{0}) = \mathcal{ID}(n, b, P)$$

The proof of this theorem is shown in Appendix A.1.

We show that the iteration from the bottom is an ascending chain and the iteration from the top is a descending chain if P is a distribution.

Theorem 6.6. is_final_dist
$$(\overline{P}) \Rightarrow incseq(\lambda n \bullet \mathcal{I}(n, b, P, \mathring{0}))$$

Theorem 6.7. is_final_dist
$$(\overline{P}) \Rightarrow decseq(\lambda n \bullet \mathcal{I}(n, b, P, \mathring{1}))$$

For an ascending or descending chain f, we define \mathcal{FS}_{\uparrow} and $\mathcal{FS}_{\Downarrow}$ to denote there are only finite states to have their suprema or infima different from their initial values f(0).

Definition 6.6 (Finite states). We fix $f: \mathbb{N} \to [S]$ prhfun, then define



$$\begin{split} \mathcal{F}\mathcal{S}_{\Uparrow}(f) & \; \widehat{=} \; \mathit{finite}\left(\left\{s: S \mid \left(\left(\bigsqcup n \bullet f(n,s)\right) > f(0,s)\right)\right\}\right) \\ \mathcal{F}\mathcal{S}_{\Downarrow}(f) & \; \widehat{=} \; \mathit{finite}\left(\left\{s: S \mid \left(\left(\bigcap n \bullet f(n,s)\right) < f(0,s)\right)\right\}\right) \end{split}$$

The intuition behind the definitions of \mathcal{FS}_{\uparrow} and $\mathcal{FS}_{\Downarrow}$ is that if an ascending or descending chain f has its supremum or infimum not equal to f(0) for a particular state s, then for any $\varepsilon : \mathbb{R} > 0$, there exists a $m : \mathbb{N}$ such that $(\bigsqcup n \bullet f(n,s) - f(m,s) < \varepsilon)$ or $(f(m,s) - \bigcap n \bullet f(n,s) < \varepsilon)$.

We show that if f is an ascending chain and $\mathcal{FS}_{\uparrow}(f)$ also holds, then there exists a $N : \mathbb{N}$ such that for any $n \geq N$, f(n, s) is close to its supremum in a given distance $\varepsilon : \mathbb{R} > 0$ for any s.

Theorem 6.8. We fix
$$f : \mathbb{N} \to [S_1, S_2]$$
 prfun, then



$$\mathit{incseq}(f) \, \wedge \, \mathcal{FS}_{\Uparrow}(f) \Rightarrow \forall \, \varepsilon : \mathbb{R} > 0 \bullet \exists \, N : \mathbb{N} \bullet \forall \, n \geq N \bullet \forall \, s \bullet \left(\bigsqcup m \, \bullet \, f(m,s) \right) - f(n,s) < \varepsilon$$

This is explained and illustrated in Fig. 4 where f is a monotonic function, such as $(\lambda n \bullet \mathcal{I}(n, b, P, 0))$, whose domain is a complete lattice, and so its limit exists and is the supremum $(\sqsubseteq n \bullet f(n, s_i))$ of the increasing chain of this function for a particular state s_i . In this diagram, we show there are four states (four combinations of the product states (s, s'), indeed) in the observation space, denoted as s_1, s_2, s_3 , and s_4 . We draw the function f of the four states for n up to 11, as shown in the diagram as $f(n, s_1)$, $f(n, s_2)$, $f(n, s_3)$, and $f(n, s_4)$. The function for each state has a corresponding supremum, such as $\sqsubseteq n \bullet f(n, s_1)$ for s_1 , and a densely dashed line denotes the supremum. We also consider a real number $\varepsilon > 0$, and so ε

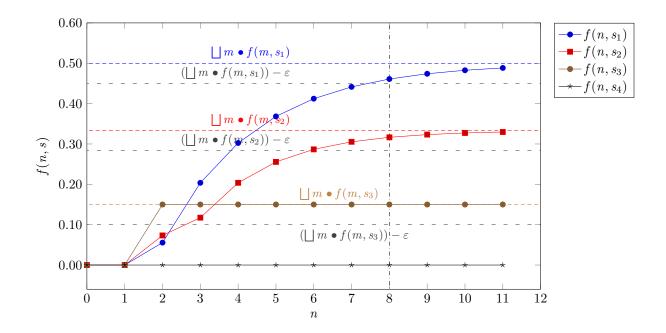


Figure 4: Illustration of limits of an increasing chain for various states.

regions (that is, areas between two parallel lines whose distance is ε) are formed between the densely dashed lines (the supremum) and the loosely dashed line $((\bigsqcup n \bullet f(n,s_i)) - \varepsilon)$. The theorem above shows there is always a N for all $m \geq N$ such that the closeness of $f(m,s_i)$ to its supremum $(\bigsqcup n \bullet f(n,s_i))$ is less than ε for any s_i . Because the limit of $f(n,s_i)$ is $(\bigsqcup n \bullet f(n,s_i))$, there always exists a N_i to satisfy this closeness. For constant zero functions, such as $f(n,s_4)$, N_i is just 0. According to the assumption of finiteness, there are finite states to have their N_i larger than 0, the states $\{s_1,s_2,s_3\}$, in this example, whose N_i are 8, 6, and 2 respectively. We can choose N as the maximum number from this N_i set, and it is 8 (illustrated as a dotted dash line at x=8) here. Now for all $n \geq N$ and any state s, the function f(n,s) is close to its supremum within the given ε . This theorem is necessary to prove Theorem 6.10 below and eventually the continuity theorem 6.12.

A descending chain f satisfies the similar theorem below.

Theorem 6.9. We fix
$$f : \mathbb{N} \to [S_1, S_2]$$
 prfun, then

$$\textit{decseq}(f) \land \mathcal{FS}_{\Downarrow}(f) \Rightarrow \forall \, \varepsilon : \mathbb{R} > 0 \bullet \exists \, N : \mathbb{N} \bullet \forall \, n \geq N \bullet \forall \, s \bullet f(n,s) - \left(\left| \begin{array}{c} m \bullet f(m,s) \\ \end{array} \right| < \varepsilon \right) < \varepsilon = 0$$

From Theorems 6.6 and 6.8, we prove the following theorem stating that for any state s the limit of the application of \mathcal{F} to \mathcal{I} is the application of \mathcal{F} to the supremum of iterations.

$$\begin{array}{l} \left(\textit{is_final_dist} \left(\overline{P} \right) \land \mathcal{FS}_{\uparrow} \left(\lambda \: n \: \bullet \: \mathcal{I} \left(n, b, P, \mathring{0} \right) \right) \right) \Rightarrow \\ \forall \: s \: \bullet \: \left(\lambda \: n : \mathbb{N} \: \bullet \: \mathcal{F}_{P}^{b} \left(\mathcal{I} \left(n, b, P, \mathring{0} \right) \right) (s) \right) \xrightarrow{n \to \infty} \left(\mathcal{F}_{P}^{b} \left(\bigsqcup n \: \bullet \: \mathcal{I} \left(n, b, P, \mathring{0} \right) \right) (s) \right) \end{array}$$

From Theorems 6.7 and 6.9, we prove the following similar theorem stating that for any state s the limit of the application of \mathcal{F} to \mathcal{I} is the application of \mathcal{F} to the infimum of iterations.

Theorem 6.11 (Limit as infimum).

$$\begin{array}{l} \left(\text{is_final_dist} \left(\overline{P} \right) \land \mathcal{FS}_{\Downarrow} \left(\lambda \, n \, \bullet \, \mathcal{I} \left(n, b, P, \mathring{1} \right) \right) \right) \Rightarrow \\ \forall \, s \, \bullet \, \left(\lambda \, n : \mathbb{N} \, \bullet \, \mathcal{F}_{P}^{b} \left(\mathcal{I} \left(n, b, P, \mathring{1} \right) \right) (s) \right) \xrightarrow{n \to \infty} \left(\mathcal{F}_{P}^{b} \left(\bigcap n \, \bullet \, \mathcal{I} \left(n, b, P, \mathring{1} \right) \right) (s) \right) \end{array}$$

Continuity of \mathcal{F} for iterations from bottom and top is derived from Theorems 6.10 and 6.11 and presented below.

Theorem 6.12 (Continuity - iteration from bottom).

$$\left(\text{is_final_dist}\left(\overline{P}\right) \land \mathcal{FS}_{\Uparrow}\left(\lambda \: n \: \bullet \: \mathcal{I}\left(n, b, P, \mathring{0}\right)\right)\right) \Rightarrow \mathcal{F}_{P}^{b}\left(\bigsqcup n \: \bullet \: \mathcal{I}\left(n, b, P, \mathring{0}\right)\right) = \left(\bigsqcup n \: \bullet \: \mathcal{I}\left(n, b, P, \mathring{0}\right)\right)$$

Theorem 6.13 (Continuity - iteration from top).



$$\left(\text{is_final_dist}\left(\overline{P}\right) \land \mathcal{FS}_{\Downarrow}\left(\lambda \ n \bullet \mathcal{I}\left(n,b,P,\mathring{1}\right)\right)\right) \Rightarrow \mathcal{F}_{P}^{b}\left(\prod n \bullet \mathcal{I}\left(n,b,P,\mathring{1}\right)\right) = \left(\prod n \bullet \mathcal{I}\left(n,b,P,\mathring{1}\right)\right)$$

We show that in Appendix A.3 to establish the continuity above, \mathcal{FS}_{\uparrow} ($\lambda n \bullet \mathcal{I}(n, b, P, \mathring{0})$), indeed, is required. The standard non-probabilistic continuity theorem [73, Section 5.3] does not have the similar requirement because the semantics of sequential composition in the language is the functional composition over one (deterministic) intermediate state. The semantics of sequential composition in our language, however, is the infinite summation over all possible intermediate states. We discuss this required premise in detail in Appendix A.3.

The Kleene fixed-point theorem 3.6 states the least (or greatest) fixed point of a continuous function \mathcal{F} is the supremum (or infimum) of the ascending (or descending) chain of the function. This is just the semantics of while loops.

Theorem 6.14 (Least fixed point by construction).



$$(is_final_dist(\overline{P}) \land \mathcal{FS}_{\uparrow}(\lambda n \bullet \mathcal{I}(n, b, P, \mathring{0}))) \Rightarrow \textit{while}_p \ \textit{b} \ \textit{do} \ P \ \textit{od} = (\bigsqcup n \bullet \mathcal{I}(n, b, P, \mathring{0}))$$

Theorem 6.15 (Greatest fixed point by construction).



$$\left(\textit{is_final_dist}\left(\overline{P}\right) \land \mathcal{FS}_{\Downarrow}\left(\lambda \: n \: \bullet \: \mathcal{I}\left(n, b, P, \mathring{1}\right)\right)\right) \Rightarrow \textit{while}_{p}^{\intercal} \: b \: \textit{do} \: P \: \textit{od} = \left(\prod n \: \bullet \: \mathcal{I}\left(n, b, P, \mathring{1}\right)\right)$$

There are several benefits in having the semantics of probabilistic loops constructed by iterations, as shown in Theorems 6.14 and 6.15. Essentially, the theorems give the semantics to probabilistic loops theoretically. In practice, they also enable us to compute the semantics by approximation or iterations, for example, in model checking. Another benefit is facilitating the proof of the unique fixed point theorem.

Theorem 6.16 (Unique fixed point - finite final states).



$$\left(\begin{array}{l} \textit{is_final_dist}\left(\overline{P}\right) & \land \mathcal{FS}_{\uparrow}\left(\lambda \, n \, \bullet \, \mathcal{I}\left(n, b, P, \mathring{0}\right)\right) \, \land \\ \left(\forall \, s \, \bullet \, \left(\lambda \, n \, \bullet \, \overline{\mathcal{ID}\left(n, b, P\right)}(s)\right) \xrightarrow{n \to \infty} 0 \right) & \land \, \mathcal{F}_{P}^{b}\left(\textit{fp}\right) = \textit{fp} \\ \Rightarrow \left(\textit{while}_{p} \, \, b \, \, \textit{do} \, P \, \, \textit{od} = \textit{fp}\right) \land \left(\textit{while}_{p}^{\top} \, b \, \, \textit{do} \, P \, \, \textit{od} = \textit{fp}\right) \end{array} \right)$$

There are four assumptions in the theorem. The third one corresponds to the differences between iterations from top and bottom, illustrated as dashed lines in Fig. 3. If for any state s, the difference tends to 0, then the least fixed point by Theorem 6.14 and the greatest fixed point by Theorem 6.15 coincide. The fourth assumption states fp is a fixed point of \mathcal{F} . The conclusion states that both the least fixed point and the greatest fixed point are just fp. This theorem largely simplifies the proof obligation for reasoning about loops to establish these assumptions.

The second assumption $\mathcal{FS}_{\uparrow}(\lambda n \bullet \mathcal{I}(n, b, P, 0))$ restricts the application of the theorem above and previous theorems to a limited subset of probabilistic programs. For example, the program with a time variable t to model the dice example, described in Sect. 2, does not satisfy the assumption because the set of final states with positive probabilities is infinite. We have proved more general theorems to support wider probabilistic programs, including those with a time variable. First, we define *finite_final* to characterise a program that always produces finitely many final states.

finite_final(P)
$$\widehat{=} \forall s \bullet finite\{s' : S \mid P(s, s') > 0\}$$

P is finite_final if for any initial state s, P has finitely many reachable states. The assumption \mathcal{FS}_{\uparrow} or $\mathcal{FS}_{\downarrow}$ in Theorems 6.10 to 6.16 is now replaced by finite_final(P). The conclusions of these theorems are still valid. We present the updated unique fixed theorem below and omit others here for brevity.

Theorem 6.17 (Unique fixed point - finite final states for each iteration).



$$\left(\begin{array}{c} \textit{is_final_dist}\left(\overline{P}\right) \land \textit{finite_final}(P) \land \left(\forall \, s \bullet \left(\lambda \, n \bullet \overline{\mathcal{ID}\left(n, \, b, \, P\right)}(s) \right) \xrightarrow{n \to \infty} 0 \right) \land \mathcal{F}_P^b\left(\mathit{fp} \right) = \mathit{fp} \end{array} \right) \\ \Rightarrow \left(\textit{while}_p \ b \ \textit{do} \ P \ \textit{od} = \mathit{fp} \right) \land \left(\textit{while}_p^\top \ b \ \textit{do} \ P \ \textit{od} = \mathit{fp} \right)$$

7. Examples and case studies

7.1. Doctor Who's Tardis Attack

Two robots, the Cyberman C and the Dalek D, attack Doctor Who's Tardis once a day between them. C has a probability of 1/2 of a successful attack, while D has a probability of 3/10 of a successful attack. C attacks more often than D, with a probability of 3/5 on a particular day (and so D attacks with a probability of 2/5 on that day). What is the probability that there will be a successful attack today?

We model the problem in our probabilistic programming in the definition below.

Definition 7.1 (Doctor Who's Tardis Attack).



```
\begin{array}{l} \textit{Attacker} ::= C \mid D \\ \textit{Status} ::= S \mid F \\ \textbf{alphabet} \ \textit{dwtastate} = r :: \textit{Attacker} \qquad a :: \textit{Status} \\ \textit{dwta} \ \widehat{=} \ \Big( (r :=_p C) :_p \Big( a :=_p S \oplus_{1/2} a :=_p F \Big) \Big) \oplus_{3/5} \Big( (r :=_p D) :_p \Big( a :=_p S \oplus_{3/10} a :=_p F \Big) \Big) \end{array}
```

We define the attackers C and D of type Attacker, and S and F of type Status for a successful or failed attacker. The observation space of this program is dwtastate containing two variables r and a to record the attacker and the attack status. The problem is modelled as a program dwta, composed of probabilistic choice, assignment, and sequential composition.

Using the reasoning framework and the algebraic laws mechanised in Isabelle, dwta is simplified and proved semantically equal to a probabilistic program shown below.

Theorem 7.1 (Simplified program).



$$dwta = \left(\begin{array}{c} 3/10 * \llbracket r' = C \wedge a' = S \rrbracket + 3/10 * \llbracket r' = C \wedge a' = F \rrbracket + \\ 6/50 * \llbracket r' = D \wedge a' = S \rrbracket + 14/50 * \llbracket r' = D \wedge a' = F \rrbracket \end{array}\right)_e$$

This law shows C has a probability of 3/10 of a successful or failed attack, while D has a probability of 6/50 of a successful attack and 14/50 of a failed attack. We note that this simplified program is a distribution of the final state because the sum of the probabilities of these combinations equals 1: 3/10+3/10+6/50+14/50=1.

With this simplified program, we can use it to answer interesting quantitative queries using sequential composition. The answer to the question "What is the probability of a successful attack?", for example, is 21/50.

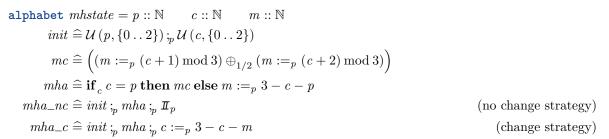
Theorem 7.2.
$$\overline{dwta}$$
; $[a = S] = (21/50)_e$



7.2. The Monty Hall problem

We model the problem in the program below, where the doors are numbered as natural numbers: 0, 1, and 2.

Definition 7.2 (Monty hall).



The observation space mhstate contains three variables of type \mathbb{N} : p for the number of the prize door, c for the contestant's choice, and m for the door Monty opens. The init is the initial configuration of the problem where the values of p and c follow a uniform distribution from an interval between 0 and 2 inclusive, so the prize and the contestant's choice are random. The mha models if the contestant's choice is the prize (c = p), the Monty randomly (with probability 1/2) chooses m from the other two doors, denoted as $(c+1) \mod 3$ and $(c+2) \mod 3$. Otherwise, the prize is not revealed, and then Monty chooses the one that is not p (he knows the value of p, the prize door) where 3-c-p guarantees that m is different from both c and p. The mha_nc models a no-change strategy, and the mha_nc models a change strategy after Monty reveals one.

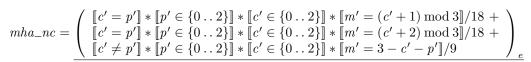
We simplify *init* according to the law below.

Theorem 7.3 (Initial).
$$init = (\llbracket p' \in \{0..2\} \rrbracket * \llbracket c' \in \{0..2\} \rrbracket * \llbracket m' = m \rrbracket / 9)_e$$

In the initial configuration, the combinations of p and c have an equal probability 1/9 with m unchanged. The *init* is also a distribution: the summation over its final states equals 1.

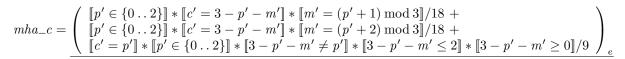
The mha_nc is proved to be equal to the program below.

Theorem 7.4 (No change strategy).



The mha_c is also proved to be equal to the program below.

Theorem 7.5 (Change strategy).



With these laws, we can answer questions like the probability of winning for each strategy and whether you change the choice.

Theorem 7.6 (Winning probability).

$$\overline{mha_nc} :_p \llbracket c = p \rrbracket = (1/3)_e$$
 (winning probability of no-change strategy)
$$\overline{mha_c} :_p \llbracket c = p \rrbracket = (2/3)_e$$
 (winning probability of change strategy)

🎒 and 🗳

The above law shows that the winning probabilities are 1/3 for the no-change strategy and 2/3 for the change strategy, so you should change the choice because you have a higher probability of winning.

7.3. The forgetful Monty

The new problem is modelled below.

Definition 7.3 (Forgetful Monty).



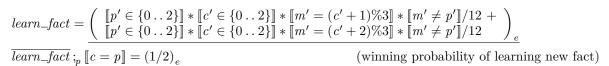
and

$$forgetful_monty \stackrel{\frown}{=} init;_p mc \qquad \qquad \text{(the forgetful Monty)}$$

$$learn_fact \stackrel{\frown}{=} forgetful_monty \parallel \llbracket m' \neq p' \rrbracket \qquad \qquad \text{(learn new fact that the prize is not revealed)}$$

After initialisation, the forgetful Monty randomly chooses one from the other two doors. The learned new fact that the prize is not revealed $(m' \neq p')$ is fed into the program by parallel composition as shown in the definition of $learn_fact$. The program equals the one below, and the winning probability is queried.

Theorem 7.7.



The probability of winning is now 1/2 and so if Monty is forgetful, and the contestant happens to choose a door with no prize, it does not matter whether the contestant sticks or switches because they have the equal probability of 1/2.

7.4. Robot localisation

The likelihood functions are defined below.

Definition 7.4 (Likelihood functions).



$$scale_door = (3 * \llbracket door(bel') \rrbracket + 1)_e$$

 $scale_wall = (3 * \llbracket \neg door(bel') \rrbracket + 1)_e$

We are interested in questions like how many measurements and moves are necessary to estimate the robot's location accurately.

7.4.1. Initialisation

Initially, the robot is randomly placed, and so a uniform distribution. This is defined below by the program init.

Definition 7.5 (Initialisation).
$$init = \mathcal{U}(bel, \{0...2\})$$



7.4.2. First sensor reading

The sensor detects a door. We learn new knowledge and update our beliefs accordingly using parallel composition. The prior probability distribution (prior) is *init*, and the likelihood function is *scale_door*. The posterior probability distribution is given in the theorem below.

Theorem 7.8 (First posterior).
$$init \parallel scale_door = 4/9 * [bel' = 0] + 1/9 * [bel' = 1] + 4/9 * [bel' = 2]$$

We have a high probability of 4/9 to believe the robot is in front of a door at position 0 or 2.

7.4.3. Move one space to the right

Now the robot takes an action to move one space to the right, and the belief position is shifted one to the right. This is defined as $move_right$ below.

Definition 7.6 (Move to the right).
$$move_right = (bel :=_p (bel + 1) \mod 3)$$

An action updates the belief using sequential composition. The posterior probability distribution after the move is given as follows.

Theorem 7.9 (Posterior after the first move).

$$(init \parallel scale_door) :_{p} move_right = 4/9 * \llbracket bel' = 0 \rrbracket + 4/9 * \llbracket bel' = 1 \rrbracket + 1/9 * \llbracket bel' = 2 \rrbracket$$

We observe that the probability values are not changing, but the positions are shifted in the distribution.

7.4.4. Second sensor reading

The sensor detects a door again. The posterior probability distribution is updated accordingly.

Theorem 7.10 (Second posterior).

$$\left((init \parallel scale_door) :_{p} move_right \right) \parallel scale_door = 2/3 * \llbracket bel' = 0 \rrbracket + 1/6 * \llbracket bel' = 1 \rrbracket + 1/6 * \llbracket bel' = 2 \rrbracket$$

We have a high probability of 2/3 to believe the robot is in front of a door at position 0 and a low probability of 1/6 in the other two positions.

7.4.5. Move one space to the right

Another action is to move the robot to its right, and the posterior probability distribution is shifted accordingly.

Theorem 7.11 (Posterior after the second move).

$$\begin{split} & \left(\left((init \parallel scale_door) ;_p move_right \right) \parallel scale_door \right) ;_p move_right \\ &= 1/6 * \llbracket bel' = 0 \rrbracket + 2/3 * \llbracket bel' = 1 \rrbracket + 1/6 * \llbracket bel' = 2 \rrbracket \end{split}$$

7.4.6. Third sensor reading

The learn sensor detects a wall. The posterior probability distribution is updated accordingly.

Theorem 7.12 (Third posterior).

$$\left(\left(\left((init \parallel scale_door) ;_p \; move_right \right) \parallel scale_door \right) ;_p \; move_right \right) \parallel scale_wall \\ = 1/18 * \llbracket bel' = 0 \rrbracket + 8/9 * \llbracket bel' = 1 \rrbracket + 1/18 * \llbracket bel' = 2 \rrbracket$$

After three sensor readings and two moves, our beliefs about the robot's position are 8/9 at position 1 and 1/18 at position 0 or 2. We plot the beliefs in Fig. 5 where each position has six updates corresponding to the initial prior (I1), the three sensor readings (door - D2 and D4, and wall - W6), and the two moves to the right (M3 and M5). The diagram shows that the difference between the highest and lowest probability for each update becomes big or stays the same: from 0 for I1 to 15/18 (= 8/9 - 1/18) for W6. So the robot gains more knowledge in each update. From the diagram, we are confident of (probability 8/9) the robot's localisation after three measurements and two moves.

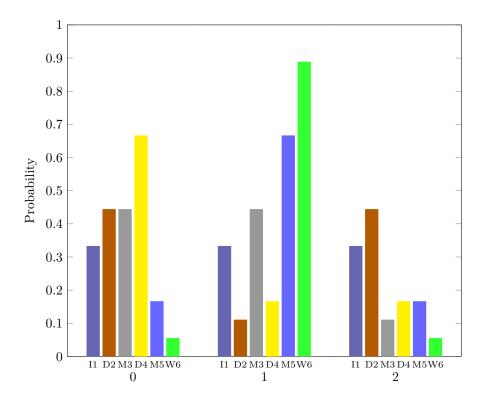


Figure 5: The update of the robot's belief at different positions after three measurements and two moves with a prior.

7.5. Classification - COVID-19 diagnosis

We define the state space cdstate of this example as follows.

Definition 7.7 (State space).

 $CovidTest ::= Pos \mid Neg$ $alphabet \ cdstate = c :: bool \ ct :: CovidTest$

Whether a person has COVID or not is recorded in a boolean variable c: true for COVID and false for no COVID. The test result is recorded in a variable ct of type CovidTest whose value could be Positive or Negative.

The prior probability of a randomly selected person having COVID is p_1 of type *ureal*. The prior probability distribution, therefore, is a probabilistic choice, defined below.

Definition 7.8 (Prior). $Init \stackrel{\frown}{=} \mathbf{if}_p \ p_1 \ \mathbf{then} \ c :=_p \mathit{True} \ \mathbf{else} \ c :=_p \mathit{False}$

So the probability of a person having COVID is p_1 and having no COVID is $(1 - p_1)$.

The test is imperfect. Its sensitivity (true positive) is p_2 , and specificity (true negative) is $1 - p_3$. It means if a person with COVID is tested, the probability of a positive result is p_2 , and if a person without COVID is tested, the probability of a negative result is $1 - p_3$. We, therefore, define the action of a test as below.

Definition 7.9 (Test). TestAction $\widehat{=}$ if $_c$ c then $(ct:=_p Pos \oplus_{p_2} ct:=_p Neg)$ else $(ct:=_p Pos \oplus_{p_3} ct:=_p Neg)$

It is a conditional choice between two probabilistic choices, defining the probabilities of true positive, false negative, false positive, and true negative. A positive test result is a new learned evidence, defined as follows.

Definition 7.10.
$$TestResPos \cong \llbracket ct' = Pos \rrbracket$$

It, essentially, is an Iverson bracket expression of a relation ct' = Pos stating that the test result ct is positive.

We conduct the first test and learn its positive result, modelled as the program below.

Definition 7.11 (First test).
$$FirstTestPos = (Init;_p TestAction) \parallel TestResPos$$



As usual, the action TestAction is sequentially composed, and the evidence is learned in parallel. We show the posterior in the theorem below.

Theorem 7.13 (Posterior after the first test)



$$\mathit{FirstTestPos} = \left(\left(\begin{array}{c} \llbracket c' \rrbracket * \llbracket ct' = \mathit{Pos} \rrbracket * p_1 * p_2 + \\ \llbracket \neg \ c' \rrbracket * \llbracket ct' = \mathit{Pos} \rrbracket * (1-p_1) * p_3 \end{array} \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) \right)_{e} = \left(\left(\begin{array}{c} \llbracket c' \rrbracket * \llbracket ct' = \mathit{Pos} \rrbracket * (1-p_1) * p_3 \end{array} \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) \right)_{e} = \left(\left(\begin{array}{c} \llbracket c' \rrbracket * \llbracket ct' = \mathit{Pos} \rrbracket * (1-p_1) * p_3 \end{array} \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) \right)_{e} = \left(\left(\begin{array}{c} \llbracket c' \rrbracket * \llbracket ct' = \mathit{Pos} \rrbracket * p_1 * p_2 + (1-p_1) * p_3 \end{array} \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) \right)_{e} = \left(\left(\begin{array}{c} \llbracket c' \rrbracket * \llbracket ct' = \mathit{Pos} \rrbracket * p_1 * p_2 + (1-p_1) * p_3 \end{array} \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) \right)_{e} = \left(\left(\begin{array}{c} \llbracket c' \rrbracket * \llbracket ct' = \mathit{Pos} \rrbracket * p_1 * p_2 + (1-p_1) * p_3 \end{array} \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) \right)_{e} = \left(\begin{array}{c} \llbracket c' \rrbracket * \llbracket ct' = \mathit{Pos} \rrbracket * p_1 * p_2 + (1-p_1) * p_3 \end{array} \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_1) * p_3 \right) / \left(p_1 * p_2 + (1-p_2) * p_3 \right) / \left(p_1 * p_2 + (1-p_2) * p_3 \right) / \left(p_1 * p_2 + (1-p_2) * p_3 \right) / \left(p_1 * p_2 + (1-p_2) * p_3 \right) / \left(p_1 * p_2 + (1-p_2) * p_3 \right) / \left(p_1 * p_2 + (1-p_2) * p_3 \right) / \left(p_1 * p_2 + (1-p_2) * p_3 \right) / \left(p_1 * p_2 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_3 + (1-p_2) * p_3 \right) / \left(p_1 * p_$$

From the theorem, we know that the probability that the person has COVID, given a positive test, is

$$(p_1 * p_2) / (p_1 * p_2 + (1 - p_1) * p_3)$$

Provided $p_1 = 0.002$, $p_2 = 0.89$, and $p_3 = 0.05$, the probability of the person having COVID is 0.0344 (much less than we may think?), and so the probability without COVID is 0.9656.

If a second test is conducted, the result is still positive, and so new evidence is learned again.

Definition 7.12 (Second test). SecondTestPos
$$\hat{=}$$
 (FirstTestPos $;_p$ TestAction) \parallel TestResPos



We show the posterior after the second test with the learned fact in the theorem below.

Theorem 7.14 (Posterior after the second test).



$$SecondTestPos = \left(\left(\begin{array}{c} \llbracket c' \rrbracket * \llbracket ct' = Pos \rrbracket * p_1 * p_2^2 + \\ \llbracket \neg \ c' \rrbracket * \llbracket ct' = Pos \rrbracket * (1-p_1) * p_3^2 \end{array} \right) / \left(p_1 * p_2^2 + (1-p_1) * p_3^2 \right) \right)_e$$

From the theorem, we know that the probability that the person has COVID, given a positive test, is

$$(p_1 * p_2^2) / (p_1 * p_2^2 + (1 - p_1) * p_3^2)$$

Provided $p_1 = 0.002$, $p_2 = 0.89$, and $p_3 = 0.05$, the probability of the person having COVID is 0.3884, so the probability without COVID is 0.6116. With the second test, it is more likely (38.84% vs. 3.44%) that the person may have COVID. In this case, a second test should be conducted, given the first test is positive.

7.6. (Parametrised) coin flip

The program flip in Definition 6.4 is for an unbiased coin (a Bernoulli distribution with p=1/2), and pflip below defines a parametrised program where the parameter p denotes the probability to have its outcome as heads (a Bernoulli distribution with probability p). So it could be a biased coin.

Definition 7.13 (Parametrised coin).
$$pflip(p) = while_p \ c = tl \ do \ c :=_p hd \oplus_p c :=_p tl \ od$$



Both flip and pflip contain probabilistic loops. We use the unique fixed point theorem 6.16 to give semantics to them where P in the theorem is cflip here for flip. Previously, we have shown that cflip is a distribution. And obviously, the observation space cstate is finite (two elements hd and tl), so the product $cstate \times cstate$ is also finite. We also show that the differences in iterations from top and bottom tend to 0, which is illustrated in Fig. 3.

Theorem 7.15.
$$\forall s : cstate \times cstate \bullet \left(\lambda \ n \bullet \overline{\mathcal{ID}(n, c = tl, cflip)}(s)\right) \xrightarrow{n \to \infty} 0$$

Additionally, [c' = hd] is a fixed point of the loop function.

Theorem 7.16.
$$\mathcal{F}_{cflip}^{c=tl}\left(\llbracket c' = hd \rrbracket \right) = \llbracket c' = hd \rrbracket$$

All four assumptions of Theorem 6.16 are now established. The *flip*, therefore, is semantically (surprisingly) just the fixed point [c' = hd].

Theorem 7.17.
$$flip = [c' = hd]$$

The *flip* terminates almost surely and is almost impossible for non-termination.

This theorem shows the probability of the final state c' being hd is 1 and is not hd is 0. This is equivalent to the termination probability.

We also show pflip(p) is semantically equal to flip if p is not 0.

Theorem 7.19.
$$p \neq 0 \Rightarrow pflip(p) = [c' = hd]$$

If p is 0, we know this program pflip is non-terminating because the probabilistic choice in the loop body always chooses tl, and so not possible to terminate.

Though flip and pflip(p) are semantically equal, we expect there are differences between the two programs in other aspects, such as average termination time. Consider a biased coin with probability p = 0.75 to see heads. Then we know, on average, it needs fewer flips than an unbiased coin to see heads. In other words, pflip(p) has a smaller average termination time than flip. This is modelled in Hehner's work [4] by a time variable t of type natural numbers to count iterations in a loop. In our language, it is defined below.

$$\begin{split} & \texttt{alphabet} \ cstate_t = t :: \mathbb{N} \qquad c :: Tcoin \\ & flip_t \ \widehat{=} \ \textit{while}_p \ c = tl \ \textit{do} \ \left(c :=_p \ hd \oplus_{1/2} c :=_p \ tl \right);_p t :=_p t + 1 \ \textit{od} \end{split}$$

The new state space is $cstate_t$ with an additional variable t of \mathbb{N} , and the loop $flip_t$ will increase t by 1 in each iteration. After the introduction of t, we use Theorem 6.17 to prove the semantics of $flip_t$ is

Theorem 7.20.
$$flip_{-}t = \begin{pmatrix} [\![c = hd]\!] * [\![c' = hd]\!] * [\![t' = t]\!] + [\![c = tl]\!] * [\![c' = hd]\!] * [\![t' \ge t + 1]\!] * (1/2)^{t'-t} \end{pmatrix}_e$$

If the initial value of c is hd (that is, c = hd), $flip_{-}t$ terminates immediately (t' = t) and its final value of c is hd (c' = hd). If the initial value of c is tl (c = tl), $flip_{-}t$ terminates (c' = hd) only when t' is larger than or equal to t+1, that is, at least one flip of the coin. The probability that $flip_{-}t$ terminates at time t' is given by $(1/2)^{t'-t}$ which can be regarded as t'-t-1 tails followed by heads: The termination probability of $flip_{-}t$ is the sum of $(1/2)^{t'-t}$ over natural numbers starting from 1: $\sum_{t'=t+1}^{\infty} (1/2)^{t'-t} = \sum_{n=1}^{\infty} (1/2)^n$. It is a geometric series with a common ratio 1/2, and so its sum is equal to (1/(1-(1/2)))-1=1. This is shown in the theorem below.

Theorem 7.21.
$$\overline{flip_t}$$
; $[c = hd] = (1)_e$

With t, we can quantify the expected value of t (the number of flips on average to get the loop terminated) by sequential composition.

Theorem 7.22.
$$\overline{flip_{-}t}:_{p} t = ([\![c = hd]\!] * t + [\![c = tl]\!] * (t+2))_{e}$$

The expectation of t given the distribution by $flip_{-}t$ is t itself (terminate immediately) if the initial value of c is hd, and t+2 (2 flips on average) otherwise.

We consider the parametrised version with t where p is a ureal number.

Definition 7.15 (Parametrised coin flip with time).

$$pflip_t(p) \mathrel{\widehat{=}} extbf{while}_p \ c = tl \ extbf{do} \ \left(c :=_p hd \oplus_p c :=_p tl
ight);_p t :=_p t+1 \ extbf{od}$$

We show its semantics below.

$$\textbf{Theorem 7.23.} \ \ p \neq 0 \Rightarrow pflip_t(p) = \underbrace{\left(\begin{array}{c} \llbracket c = hd \rrbracket * \llbracket c' = hd \rrbracket * \llbracket t' = t \rrbracket + \llbracket c = tl \rrbracket * \llbracket c' = hd \rrbracket * \llbracket t' \geq t + 1 \rrbracket * (1 - \overline{p})^{t' - t - 1} * \overline{p} \end{array} \right)_e}_{e}$$

If p is not 0, then the probability that $pflip_{-}t(p)$ terminates at t' now is (t'-t-1) tails (probability $(1-\overline{p})^{t'-t-1}$, \overline{p} is the conversion of p to \mathbb{R}) and followed by heads (probability \overline{p}) when the initial c is tl. The following theorem shows the program $pflip_{-}t(p)$ terminates almost surely.

Theorem 7.24.
$$p \neq 0 \Rightarrow \overline{pflip_t(p)};_p \llbracket c = hd \rrbracket = (1)_e$$

Its expected termination time is $1/\overline{p}$ flips, shown below.

Theorem 7.25.
$$p \neq 0 \Rightarrow \overline{pflip_{-}t(p)};_{p} t = ([\![c = hd]\!] * t + [\![c = tl]\!] * (t + 1/\overline{p}))_{e}$$

In essence, the proof of this theorem is the calculation of the following summation.

$$\left(\Sigma_{\infty}v_0 \mid c_v(v_0) = hd \wedge Suc(t) \leq t_v(v_0) \bullet (1 - \overline{p})^{(t_v(v_0) - Suc(t))} * \overline{p} * t_v(v_0)\right)$$

where $c_v(v_0)$ and $t_v(v_0)$ extract the values of the variables c and t from the state v_0 . The calculation involves several important steps:

- (1). find an injective function to reindex the summation over v_0 into the summation over n of natural numbers: $(\Sigma_{\infty} n : \mathbb{N} \bullet f(n))$ where $f(n) = ((1 \overline{p})^n * \overline{p} * (Suc(t) + n);$
- (2). prove f(n) is summable using the ratio test for convergence by supplying a constant ratio c that is less than 1 and a natural number N so that for all numbers larger than N, the ratio f(n+1)/f(n) is less than c;
- (3). because f(n) is summable, we can assume f(n+1) sums to l, then f(n) must sum to $l+f(0)=l+\overline{p}*Suc(t)$;
- (4). alternatively, $f(n+1) = (1-\overline{p})^{n+1} * \overline{p} * (Suc(t) + n + 1) = f(n) * (1-\overline{p}) + (1-\overline{p})^n * \overline{p} * (1-\overline{p}),$ and so $(\Sigma_{\infty}n : \mathbb{N} \bullet f(n+1)) = (\Sigma_{\infty}n : \mathbb{N} \bullet f(n) * (1-\overline{p})) + (\Sigma_{\infty}n : \mathbb{N} \bullet (1-\overline{p})^n * \overline{p} * (1-\overline{p}));$
 - $(\Sigma_{\infty} n : \mathbb{N} \bullet (1 \overline{p})^n * \overline{p} * (1 \overline{p}))$ is a geometric series and equal to $\overline{p} * (1 \overline{p}) * (1/(1 (1 \overline{p}))) = 1 \overline{p}$
- (5). get an equation $l = (l + \overline{p} * Suc(t)) * (1 \overline{p}) + (1 \overline{p})$, solve this equation and we get the result of l, then we know f(n) sums to $(t + 1/\overline{p})$.

We also note that the parameter p is not present in the semantics (see Theorem 7.19) of pflip as long as p is larger than 0. We have seen that the average termination time $1/\overline{p}$ is a function of the parameter p, which entitles us to reason about parametric probabilistic models intrinsically, not like approximation and limitations in probabilistic model checking [74, 75].⁵

 $^{^{5}}$ https://www.prismmodelchecker.org/manual/RunningPRISM/ParametricModelChecking

7.7. Dice

This example [4] is about throwing a pair of dice till they have the same outcome. The *dice* program is defined below.

Definition 7.16.

$$Tdice ::= \{1..6\}$$

alphabet $fdstate = d_1 :: Tdice \qquad d_2 :: Tdice$
 $throw = \underbrace{\mathcal{U}(d_1, Tdice)}_{dice} :_p \underbrace{\mathcal{U}(d_2, Tdice)}_{do throw od}$
 $dice = \underbrace{\textbf{while}_p}_{d_1} d_1 \neq d_2 \underbrace{\textbf{do} throw od}$

The outcome of a die is from 1 to 6 as given in Tdice. The observation space of this program is fdstate, containing two variables d_1 and d_2 of type Tdice, denoting the outcome of each dice in an experiment. The program throw is the sequential composition of two uniform distributions to choose d_1 and d_2 independently, and dice models the example: continue throwing till the outcomes of two dice are equal $(d_1 = d_2)$.

We use the unique fixed point theorem 6.16 to give semantics. First, throw is a distribution.

Theorem 7.26.
$$is_final_dist(\overline{throw})$$

Second, the observation space Tdice is finite, and set $fdstate \times fdstate$ is also finite. Third, the differences of iterations from top and bottom tend to 0.

Theorem 7.27.
$$\forall s : fdstate \times fdstate \bullet \left(\lambda \ n \bullet \overline{\mathcal{ID}(n, d_1 \neq d_2, throw)}(s)\right) \xrightarrow{n \to \infty} 0$$

Finally, we define H,

$$H \stackrel{\frown}{=} ([\![d_1 = d_2]\!] * [\![d'_1 = d_1 \land d'_2 = d_2]\!] + [\![d_1 \neq d_2]\!] * [\![d'_1 = d'_2]\!]/6)_e$$

and prove it is a fixed point.

Theorem 7.28.
$$\mathcal{F}_{throw}^{d_1 \neq d_2}(\underline{H}) = \underline{H}$$

The H gives the distribution on the final states. If initially, d_1 is equal to d_2 ; it has probability 1 to establish that both d'_1 and d'_2 are equal to their initial states, so they are identical too. This is the semantics of \mathbb{Z}_p . However, if d_1 is not equal to d_2 initially, it has a probability 1/6 to establish $d'_1 = d'_2$. Because there are six combinations of the equal values of d'_1 and d'_2 , the total probability is still 1 (6*1/6), so H is a distribution.

All four assumptions of Theorem 6.16 are now established. The *dice*, therefore, is semantically just the fixed point \underline{H} .

Theorem 7.29.
$$dice = \underline{H}$$

The dice terminates almost surely and is almost impossible for non-termination.

Theorem 7.30.

$$\overline{dice} :_{p} [d_{1} = d_{2}] = (1)_{e}$$
 (termination probability)
$$\overline{dice} :_{p} [d_{1} \neq d_{2}] = (0)_{e}$$
 (non-termination probability)

We now consider the dice program with a time variable t.

Definition 7.17 (Dice with time).

$$\begin{array}{ll} \textbf{alphabet} \ dstate_t = t :: \mathbb{N} \qquad d_1 :: Tdice \qquad d_2 :: Tdice \\ throw_t \ \widehat{=} \ \underline{\mathcal{U} \left(d_1, Tdice \right)};_p \underline{\mathcal{U} \left(d_2, Tdice \right)};_p t :=_p t + 1 \\ dice_t \ \widehat{=} \ \textit{while}_p \ d_1 \neq d_2 \ \textit{do} \ throw_t \ \textit{od} \\ \end{array}$$

We show that

Theorem 7.31.
$$throw_t = [t' = t + 1]/36$$

The [t' = t + 1]/36 is a distribution.

Theorem 7.32.
$$is_final_dist([[t'=t+1]]/36)$$

So the conversion of throw_t to real-valued functions is just the distribution.

Theorem 7.33.
$$\overline{throw_{-}t} = [t' = t + 1]/36$$

PROOF. This can be proved using Theorems 5.8, 7.32, and 5.4.

We define Ht.

Definition 7.18 (Ht).

$$Ht \stackrel{\triangle}{=} \left(\begin{array}{c} [\![d_1 = d_2]\!] * [\![t' = t \land d_1' = d_1 \land d_2' = d_2]\!] + \\ [\![d_1 \neq d_2]\!] * [\![d_1' = d_2'\!] * [\![t' \geq t + 1]\!] * (5/6)^{t' - t - 1} * (1/36) \end{array} \right)_e$$

The Ht is a distribution.

Theorem 7.34.
$$is_final_dist(Ht)$$

Theorem 7.35.
$$\overline{(Ht)} = Ht$$

PROOF. This can be proved using Theorems 5.8, 7.34, and 5.4.

The Ht is proved to be a fixed point of $dice_{-}t$.

Theorem 7.36.
$$\mathcal{F}_{throw_t}^{d_1 \neq d_2}(\underline{Ht}) = \underline{Ht}$$

Proof.

$$\begin{split} &\mathcal{F}_{throw_t}^{d_1 \neq d_2} \left(\underline{Ht} \right) \\ &= \{ \text{Law } \left(\mathcal{F}_P^b \text{ altdef} \right) \} \\ &= \{ \text{Law } \left(\mathcal{F}_P^b \text{ altdef} \right) \} \\ &= \{ \text{Definition 5.12 for } ;_p \underline{Ht} \right) + \llbracket \neg \ d_1 \neq d_2 \rrbracket * \llbracket \underline{\pi} \rrbracket \\ &= \{ \text{Definition 5.12 for } ;_p \} \\ &= \{ \text{Definition 5.12 for } ;_f \overline{\left(\underline{Ht} \right)} \right) + \llbracket \neg \ d_1 \neq d_2 \rrbracket * \llbracket \underline{\pi} \rrbracket \\ &= \{ \text{Theorems 7.33 and 7.35 } \} \\ &= \{ \text{Definition 5.12 for } ;_f \} \\ &= \{ \text{Definition 5.12 for } ;_f \} \\ &= \{ \text{Definition 5.12 for } ;_f \} \\ &= \{ \text{Definitions 7.18, substitution, and omit } \llbracket \neg \ d_1 \neq d_2 \rrbracket * \llbracket \underline{\pi} \rrbracket \} \\ \end{split}$$

⁶We note that our Ht is different from that of [4] where the probability of having $d'_1 = d'_2$ is 1/6 (instead of 1/36 in ours). After a careful comparison of our mechanised proof and the pencil-and-paper proof in [4], we figured out the mistake is introduced in a step of the proofs in [4].

$$\boxed{ \begin{bmatrix} d_1 \neq d_2 \end{bmatrix} * \overline{ \begin{pmatrix} \begin{bmatrix} t'' = t + 1 \end{bmatrix}/36 \end{pmatrix} * \\ \sum_{\infty} \mathbf{v}'' \bullet \begin{pmatrix} \begin{bmatrix} d_1'' = d_2'' \end{bmatrix} * \begin{bmatrix} t' = t'' \land d_1' = d_1'' \land d_2' = d_2'' \end{bmatrix} + \\ \begin{bmatrix} d_1'' \neq d_2'' \end{bmatrix} * \begin{bmatrix} d_1' = d_2' \end{bmatrix} * \begin{bmatrix} t' \geq t'' + 1 \end{bmatrix} * (5/6)^{t'-t''-1} * (1/36) \end{pmatrix} } + \cdots}$$

= { Multiplication distributive over addition }

$$\underbrace{ \begin{bmatrix} (\llbracket t'' = t + 1 \rrbracket/36) * \llbracket d_1'' = d_2'' \rrbracket * \llbracket t' = t'' \wedge d_1' = d_1'' \wedge d_2' = d_2'' \rrbracket + }_{ \Sigma_{\infty} \mathbf{v}'' \bullet (\llbracket t'' = t + 1 \rrbracket/36) *} + \cdots$$

$$\underbrace{ \begin{bmatrix} d_1 \neq d_2 \rrbracket * \underbrace{ \sum_{\infty} \mathbf{v}'' \bullet (\llbracket t'' = t + 1 \rrbracket/36) * }_{ \llbracket d_1'' \neq d_2'' \rrbracket * \llbracket d_1' = d_2' \rrbracket * \llbracket t' \geq t'' + 1 \rrbracket * (5/6)^{t'-t''-1} * (1/36) \end{bmatrix}}_{ \bullet \bullet \bullet \bullet} + \cdots$$

 $= \{ Law (addition) and proofs of summable omitted \}$

$$= \{ \text{ In the first summation, only one state } \mathbf{v}''[t'' = t+1, d_1'' = d_1', d_2'' = d_2'] \text{ satisfies the prediction} \\ & \boxed{ \begin{bmatrix} d_1' = d_2' \end{bmatrix} * \begin{bmatrix} t' = t+1 \end{bmatrix} / 36 + \\ \sum_{\infty} \mathbf{v}'' \bullet (\llbracket t'' = t+1 \rrbracket / 36) * \\ & \llbracket d_1'' \neq d_2'' \rrbracket * \llbracket d_1' = d_2' \rrbracket * \llbracket t' \geq t''+1 \rrbracket * (5/6)^{t'-t''-1} * (1/36) \end{bmatrix} + \cdots } \\ = \{ \text{There are 30 states } \mathbf{v}''[t+1/t'', x/d_1'', y/d_2''] \text{ where } x \neq y \text{ satisfies the predicates } \} \\ & \boxed{ \begin{bmatrix} d_1' = d_2' \end{bmatrix} * \begin{bmatrix} t' = t+1 \end{bmatrix} / 36 + \\ & \boxed{ \begin{bmatrix} d_1' = d_2' \end{bmatrix} * \begin{bmatrix} t' = t+1 \end{bmatrix} / 36 + \\ & \boxed{ \begin{bmatrix} d_1' = d_2' \end{bmatrix} * \begin{bmatrix} t' \geq t+1+1 \end{bmatrix} * (5/6)^{t'-(t+1)-1} * 30 * (1/36) * (1/36) \end{bmatrix} + \cdots } \\ = \{ 30/36 = 5/6 \} \\ & \boxed{ \begin{bmatrix} d_1 \neq d_2 \end{bmatrix} * \underbrace{ \begin{bmatrix} \begin{bmatrix} d_1' = d_2' \end{bmatrix} * \begin{bmatrix} t' \geq t+1 \end{bmatrix} * (5/6)^{t'-t-1} * (1/36) \end{bmatrix} + \cdots } \\ = \{ \text{Merged } \} \\ & \boxed{ \begin{bmatrix} d_1 \neq d_2 \end{bmatrix} * \underbrace{ \begin{bmatrix} \begin{bmatrix} d_1' = d_2' \end{bmatrix} * \begin{bmatrix} t' \geq t+1 \end{bmatrix} * (5/6)^{t'-t-1} * (1/36) \end{bmatrix} + \cdots } \\ = \{ \text{Theorems 5.8, 7.34, and 5.4, and the omitted } \} \\ & \boxed{ \begin{bmatrix} d_1 \neq d_2 \end{bmatrix} * (\boxed{ \begin{bmatrix} d_1' = d_2' \end{bmatrix} * \begin{bmatrix} t' \geq t+1 \end{bmatrix} * (5/6)^{t'-t-1} * (1/36) \end{bmatrix} + \boxed{ \begin{bmatrix} 1 \neq d_2 \end{bmatrix} * \begin{bmatrix} 1 \neq d$$

$$[\![d_1 \neq d_2]\!] * \overline{\left(\begin{array}{c} [\![d_1' = d_2']\!] * [\![t' = t+1]\!]/36 + \\ [\![d_1' = d_2']\!] * [\![t' \geq t+1+1]\!] * (5/6)^{t'-(t+1)-1} * 30 * (1/36) * (1/36) \end{array} \right)} + \cdots$$

$$[\![d_1 \neq d_2]\!] * \overline{\left([\![d'_1 = d'_2]\!] * [\![t' = t + 1]\!] / 36 + [\![d'_1 = d'_2]\!] * [\![t' \geq t + 2]\!] * (5/6)^{t'-t-1} * (1/36) \right)} + \cdots$$

$$[d_1 \neq d_2] * \overline{([d'_1 = d'_2] * [t' \geq t + 1] * (5/6)^{t'-t-1} * (1/36))} + \cdots$$

= { Definition (skip) }

 \underline{Ht}

Using Theorem 6.17, we prove the semantics of $dice_t$ is just Ht.

Theorem 7.37. $dice_t = \underline{Ht}$

We note that the semantics of $pflip_t$ in Theorem 7.23 has a pattern

$$[c = tl] * [c' = hd] * [t' \ge t + 1] * (5/6)^{t'-t-1} * (1/6)$$

if p = 1/6, and the semantics of dice_t here has a pattern

$$[d_1 \neq d_2] * [d'_1 = d'_2] * [t' \geq t + 1] * (5/6)^{t'-t-1} * (1/36)$$

The (1/6) or (1/36) above denotes the success probability of each experiment in terms of a particular valuation of the variables in the observation space. For example, (1/6) denotes the probability of [c' = hd] for a particular t' and c' (where c' = hd is the only value to establish [c' = hd]), and (1/36) denotes the probability of $[d'_1 = d'_2]$ for a particular t', d'_1 , and d'_2 (where for each t', there are overall 6 values of d'_1 and d'_2 to establish $[d'_1 = d'_2]$, that is, both take the same value from 1 to 6).

The $dice_t$ terminates almost surely.

Theorem 7.38.
$$\overline{dice_{-}t}$$
; $[\![d_1 = d_2]\!] = (1)_e$

On average, the *dice_t* takes six dice throws to get an equal outcome.

Theorem 7.39.
$$\overline{dice_{-}t}$$
; $t = ([d1 = d2] * t + [d1 \neq d2] * (t+6))_e$

8. Conclusion

Previous work [23, 24] has shown the modelling of aleatoric uncertainty in RoboChart based on the semantics of MDP and in pGCL based on the theory of probabilistic designs and the automated verification of probabilistic behaviours using probabilistic model checking and theorem proving. This work presents a new probabilistic semantic framework, ProbURel, and probabilistic programming to cover modelling both aleatoric and epistemic uncertainties and the automated verification of probabilistic systems exhibiting both uncertainties using theorem proving. We discuss our probabilistic vision in Sect. 1, and the new semantic framework is our first step in the big picture. With ProbURel, we can give semantics to deterministic, probabilistic sequential programs with the support of discrete distributions and time.

We have based our work on Hehner's predicative probabilistic programming and addressed obstacles to applying his work by formalising and mechanising its semantics in Isabelle/UTP. We have introduced an Iverson bracket notation to separate arithmetic semantics from relational semantics so that reasoning about a probabilistic program can reuse existing reasoning techniques for both arithmetic and relational semantics. We have used the UTP's alphabetised relational calculus to formalise its relational semantics, and so probabilistic programs benefit from automated reasoning in Isabelle/UTP. We have used the summations over the topological space of real numbers for arithmetic semantics, and so probabilistic programs also benefit from mechanised theories in Isabelle/HOL for reasoning. We have enriched the semantics domains from probabilistic distributions to subdistributions and superdistributions to use the constructive Kleene fixed point theorem to give semantics to probabilistic loops based on the least fixed point and derive a unique fixed point theorem to vastly simplify the reasoning of probabilistic loops. With formalisation and mechanisation, we have reasoned about six examples of probabilistic programs.

8.1. Our probabilistic vision

Recently, we presented a probabilistic extension [23] to RoboChart [76], a state machine-based DSL for robotics, to allow the modelling of probabilistic behaviour in robot control software. RoboChart [76, 23] is a core notation in the RoboStar⁷ framework [77] that brings modern modelling and verification technologies into software engineering for robotics. In this framework, three key elements are models, formal mathematical semantics for models, and automated modelling and verification tool support. RoboChart is a UML-like architectural and state machine modelling notation featuring discrete time and probabilistic modelling. It has formal semantics: state machines and architectural semantics [76] based on the CSP process algebra [78, 79] and time semantics [76] based on tock-CSP [80, 79]. CSP is a formal notation to describe concurrent systems where processes interact using communication. In the framework, robot hardware and control software are captured in robotic platforms, and controllers of RoboChart. The environment is captured in RoboWorld [81] whose semantics is based on CyPhyCircus [82], a hybrid process algebra, because of the continuous nature of the environment.

While RoboChart and RoboWorld are high-level specification languages, RoboSim [83] is a cycle-based simulation-level notation in the RoboStar framework. RoboSim also has semantics in CSP and *tock*-CSP.

⁷robostar.cs.york.ac.uk.

A RoboSim model can be automatically transformed from a RoboChart model directly, and its correctness is established by refinements [79] in CSP.

Probability is used to capture uncertainties from physical robots and the environment and randomisation in controllers. The probabilistic extension in the RoboStar framework requires its semantic extension to either base on process algebras and hybrid process algebras or have the richness to deal with probabilistic concurrent and reactive systems. The features of its probabilistic semantics that we consider in this big vision include discrete and continuous distributions, discrete time, nondeterminism, concurrency, and refinement. The type system and the comprehensive expression language of RoboChart [76], additionally, are based on those of the Z notation [58, 59] and include mathematical data types such as relations and functions, quantifications, and lambda expressions. Because of such richness in semantics and language features of RoboChart, the formal verification support of RoboChart requires theorem proving and model checking.

Our immediate thought is to consider existing probabilistic extensions to process algebras, including CSP-based [84, 85, 86, 87, 88, 89], CCS-based [90, 91, 92, 93], and ACP-based [94]. The main difference between these extensions is how existing constructs or operators, particularly nondeterministic and external choice, interact with probabilistic choice. We also looked at extensions based on probabilistic transition systems [95, 96, 97] and automata [98, 99]. To preserve the distributivity of existing operators over probabilistic choice, some algebraic properties are lost, such as the congruence for hiding and asynchronous parallel composition [88], idempotence for nondeterministic choice [34], or even no standard nondeterministic choice [87, 100]. The critical problem, however, is the lack of tool support for these extensions. For example, FDR [101], a refinement model checker for CSP and tock-CSP, cannot verify the probabilistic extensions in CSP. For this reason, we explored other solutions.

In [38, 23], we give probabilistic semantics of RoboChart on probabilistic designs [24] in Hoare and He's Unifying Theories of Programming (UTP) [60] and then use the theorem prover Isabelle/UTP [63], an implementation of UTP in Isabelle/HOL, to verify probabilistic models. Probabilistic designs are an embedding of standard non-probabilistic designs into the probabilistic world. The theory of probabilistic designs gives probabilistic semantics to the imperative nondeterministic probabilistic sequential programming language pGCL [57], but not reactive aspects of RoboChart. We have thought about lifting probabilistic designs into probabilistic reactive designs. Still, the main obstacle is the complexity of reasoning about probabilistic distributions in probabilistic designs because distributions are captured in a dedicated variable prob, representing a probability mass function. In particular, the definition [24] of sequential composition includes an existential quantification over intermediate distributions. The proof of sequential composition needs to supply a witness for the intermediate distributions, which is helpful but non-trivial.

We also gave RoboChart probabilistic semantics [23, 102] in the PRISM language [103]. We developed plugins for RoboTool,⁸ an accompanying tool for RoboChart, to support automated verification through probabilistic model checking using PRISM. PRISM, however, employs a closed-world assumption: systems are not subjected to environmental inputs. To verify a RoboChart model, such as a high voltage controller⁹ for a painting robot [104] and an agricultural robot ¹⁰ for UV-light treatment using PRISM, we need to constrain the environmental input and verify its expected outputs through an additional PRISM module being in parallel with the corresponding PRISM model that is automatically transformed from the RoboChart model. Finally, the safety and reachability properties of the RoboChart model (checked by the trace refinement in FDR) become deadlock freedom problems in PRISM. However, this cannot verify other properties like liveness, which requires failures-divergences refinement in CSP and FDR.

The research question that we aim to answer is a probabilistic semantic framework that (1) has rich semantics to capture our probabilistic vision, (2) is simple and flexible to allow further extensions, and (3) supports theorem proving. This question is comprehensive and needs a research programme, instead of a project, to address it. The work we present in this paper is our first step to answering this question.

⁸www.cs.york.ac.uk/robostar/robotool/

⁹github.com/UoY-RoboStar/hvc-case-study/tree/prism_verification/sbmf

¹⁰github.com/UoY-RoboStar/uvc-case-study

8.2. Future work

We have not proved and mechanised the SRW example 1.6. Our immediate future work is to verify SRW: its semantics, termination, and expected runtime in terms of the parameters m and p. We are also interested in the mathematical (that is, the probability theory) way to calculate the termination distribution and comparing it with our programming way (that is, lfp) to establish the equivalence between them.

Our fixed point theorems, such as Theorems 6.14, 6.15, 6.16, and 6.17 for probabilistic loops, cannot deal with the programs (the loop body) whose final observation space contains infinite states with positive probabilities. The restriction is introduced in Theorems 6.8 and 6.9, which are used to prove continuity theorems 6.12 and 6.13, and eventually for the least and greatest fixed point theorems 6.14 and 6.15. Our immediate future work is to extend our fixed point theorem to support such countably infinite state space, enabling us to give semantics to loops containing such programs. Our approach is to use Cousot's constructive version of the Knaster-Tarski fixed point theorem [105] to weaken continuity to monotonicity and treat the least fixed point as the stationary limit of transfinite iteration sequences. With this extension, our semantics can tackle more general probabilistic programs with countably infinite state space. Hehner [4] presented a simpler semantics for loops. His approach is to include a time variable of type extended integer or real numbers to count iterations, similar to the t (but its type is natural numbers) in our examples. He argued that if a fixed point is proved for a loop, then it is the only fixed point, and so the semantics for the loop. This is very interesting to us. We could formalise his proof and mechanise it in Isabelle/UTP, which may benefit our approach to simplify reasoning of loops or address the limitation of finite states with positive probabilities.

The probabilistic programming we present in this paper only considers discrete probabilistic distributions. One of our future works is to support continuous distributions such as normal or Gaussian distributions, uniform distributions, and exponential distributions, which are naturally presented in many physical systems in our semantics. Each point has zero probability in (absolute) continuous distributions, so the probability mass functions for discrete distributions could not describe them. Instead, they are characterised by probability density functions, which require measure theory to deal with probabilities and integration [26] over intervals. We, therefore, will introduce measure theory to our semantics and mechanise it in Isabelle/UTP based on the measure theory in Isabelle. After these lines of future work are complete, our probabilistic programming can automatically model a wide range of probabilistic systems and reasoning about them.

With UTP and ProbURel, we could bring different approaches to handling uncertainty, such as epistemic mu-calculus and probabilistic synthesis, together and unify these approaches. Our semantics for ProbURel are denotational, which could underpin the operational semantics for other approaches. By unifying these theories, we could link different tools. For example, one model could be analysed using our theorem prover, and it could also be transformed into another probabilistic programming language and analysed by the supported tools for it, such as PRISM. This will be beneficial for analysis by leveraging the advantages of different tools.

Acknowledgements

This work is funded by the EPSRC projects RoboCalc (Grant EP/M025756/1) and RoboTest (Grant EP/R025479/1).

Appendix A. Proofs

Appendix A.1. Proof of Theorem 6.5

We present and prove three theorems first and then use them to prove Theorem 6.5.

Theorem Appendix A.1. Provided P is a distribution, that is, is_final_dist(P).

$$\begin{split} \left(\mathcal{F}_{P}^{b}\right)^{0}(\mathring{0}) &= \mathring{0} \\ \left(\mathcal{F}_{P}^{b}\right)^{1}(\mathring{0}) &= \lambda(s,s'). \ \llbracket \neg \ b(s) \rrbracket * \llbracket s' = s \rrbracket \end{split}$$

If
$$n > 1$$
, then

$$\begin{split} \left(\mathcal{F}_{P}^{b}\right)^{n}(\mathring{0}) &= \\ \lambda(s,s'). \left(\sum_{i=1}^{n-1} \left(\sum_{\infty} s_{i-1}. \llbracket b(s) \rrbracket * \overline{P}(s,s_{i-1}) * \atop \left(\sum_{\infty} s_{i-2}. \llbracket b(s_{i-1}) \rrbracket * \overline{P}(s_{i-1},s_{i-2}) * \atop \left(\vdots * \atop \left(\sum_{\infty} s_{0}. \llbracket b(s_{1}) \rrbracket * \overline{P}(s_{1},s_{0}) * \llbracket \neg b(s_{0}) \rrbracket * \llbracket s' = s_{0} \rrbracket \right) \right) \right) \right) \right) \\ + (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket \end{aligned}$$

PROOF. We show below that $(\mathcal{F}_P^b)^n$ ($\mathring{0}$) for n=0 to 3 satisfies the theorem.

$$(\mathcal{F}_P^b)^0(\hat{0}) = \lambda(s,s').0 = \hat{0}$$

$$(\mathcal{F}_P^b)^1(\hat{0}) = \{ \operatorname{Defintion} (\operatorname{loop function}) \}$$

$$\text{if }_c b \text{ then } (P;_p \hat{0}) \text{ else } \mathbb{I}_p$$

$$= \{ \operatorname{Law} (\mathcal{F}_P^b \text{ altdef}) \}$$

$$\underline{ [b]} * \overline{(P;_p \hat{0})} + [\neg b] * \underline{ [ll]}$$

$$= \{ \operatorname{Theorem 5.16 \text{ Law } 3} \}$$

$$\underline{ [\neg b]} * \underline{ [ll]} = \{ \operatorname{Expand as a function form, use } s \text{ and } s' \text{ for initial and final observation states} \}$$

$$\{ \operatorname{Substitution and Definition (skip)} \}$$

$$\lambda(s,s'). \underline{ [\neg b(s)]} * \underline{ [s'=s]}$$

$$(\mathcal{F}_P^b)^2(\hat{0}) = \{ \mathcal{F}^2(\hat{0}) = \mathcal{F}(\mathcal{F}^1(\hat{0})) \text{ and Law } (\mathcal{F}_P^b \text{ altdef}) \}$$

$$\text{if }_c b \text{ then } (P;_p \underline{ [\neg b]} * \underline{ [ll]}) \text{ else } \underline{ [ll]}) \text{ else } \underline{ [ll]}$$

$$= \{ \operatorname{Theorem 5.15 \text{ Law } 3 \text{ and Theorem 5.13 Law 5} \}$$

$$\underline{ [[b])} * \overline{ (P;_p \underline{ [\neg b]} * \underline{ [ll]}) } + (\underline{ [i-[b])}) * \underline{ [ll]}$$

$$= \{ \operatorname{Theorem 5.9}, \operatorname{Definition (skip)} \}$$

$$\underline{ [b]} * \overline{ (P;_p \underline{ [\neg b]} * \underline{ [ll]}) } + (\underline{ [i-[b])} * \underline{ [ll]}] [v_0/v]) + (\underline{ [\neg b]}) * \underline{ [ll]}$$

$$= \{ \operatorname{Theorem 5.9} \}$$

$$\underline{ [b]} * \overline{ (\sum_{\infty} v_0 \bullet \overline{ P}[v_0/v'] * (\underline{ [\neg b]} * \underline{ [ll]}) [v_0/v]) } + (\underline{ [\neg b]}) * \underline{ [ll]}$$

$$= \{ \operatorname{Expand as a function form, use } s \text{ and } s' \text{ for initial and final observation states} \}$$

$$\{ \operatorname{Substitution and Definition (skip)} \}$$

$$\lambda(s,s').[b(s)] * \overline{ (\sum_{\infty} v_0 \bullet \overline{ P}[v_0/v'] * (\underline{ [\neg b]} * \underline{ [ll]}) [v_0/v]) } + (\underline{ [\neg b]}) * \underline{ [ll]}$$

$$= \{ \operatorname{Expand as a function form, use } s \text{ and } s' \text{ for initial and final observation states} \}$$

$$\{ \operatorname{Substitution and Definition (skip)} \}$$

 $= \{ \text{ Theorem 5.8 where the proof of } \textit{is_prob} \text{ is omitted} \}$

$$\lambda(s, s'). \llbracket b(s) \rrbracket * \left(\Sigma_{\infty} s_0 \bullet \overline{P}(s, s_0) * (\llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket) \right) + (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket$$

= { Law (multiplication of constant) and proof of summable is omitted }

$$\lambda(s,s').\underline{(\Sigma_{\infty}s_0 \bullet \llbracket b(s) \rrbracket * \overline{P}(s,s_0) * (\llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket)) + (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket} (\mathcal{F}_P^b)^3 (\mathring{0})$$

 $= \{\, \mathcal{F}^3(\mathring{0}) = \mathcal{F}(\mathcal{F}^2(\mathring{0})) \text{ and same as previous proof} \,\}$

$$\lambda(s,s'). \left(\begin{bmatrix} b(s) \end{bmatrix} * \overline{\left(\left(\frac{\sum_{\infty} s_1 \bullet \overline{P}(s,s_1) *}{\left(\left(\sum_{\infty} s_0 \bullet \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) * (\llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket) \right) \right)} \right) \right) + (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket \right)} + (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket$$

= { Theorem 5.8 where the proof of is_prob is omitted }

$$\lambda(s,s'). \left(\begin{bmatrix} b(s) \end{bmatrix} * \overline{\left(\begin{bmatrix} \sum_{\infty} s_1 \bullet \overline{P}(s,s_1) * \\ \left(\sum_{\infty} s_0 \bullet \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) * (\llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket) \right) \right) \right)} \\ + (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket \right) \right)$$

= { Law (addition) and proofs of summable omitted }

$$\lambda(s,s'). \left(\begin{bmatrix} b(s) \end{bmatrix} * \overline{\left(\begin{pmatrix} \Sigma_{\infty} s_1 \bullet \overline{P}(s,s_1) * \\ (\Sigma_{\infty} s_0 \bullet \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) * (\llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket)) \\ + \Sigma_{\infty} s_1 \bullet \overline{P}(s,s_1) * \llbracket \neg b(s_1) \rrbracket * \llbracket s' = s_1 \rrbracket \right) \right)$$

= { Theorem 5.8 where the proof of is_prob is omitted }

$$\lambda(s,s'). \begin{pmatrix} \llbracket b(s) \rrbracket * \Sigma_{\infty} s_1 \bullet \overline{P}(s,s_1) * \\ (\Sigma_{\infty} s_0 \bullet \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) * (\llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket)) \\ \llbracket b(s) \rrbracket * \Sigma_{\infty} s_1 \bullet \overline{P}(s,s_1) * \llbracket \neg b(s_1) \rrbracket * \llbracket s' = s_1 \rrbracket \\ + (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket \end{pmatrix}$$

= { Law (multiplication of constant) and proof of summable is omitted }

$$\lambda(s,s'). \underbrace{\begin{pmatrix} \Sigma_{\infty}s_1 \bullet \llbracket b(s) \rrbracket * \overline{P}(s,s_1) * \\ (\Sigma_{\infty}s_0 \bullet \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) * (\llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket)) \\ \Sigma_{\infty}s_1 \bullet \llbracket b(s) \rrbracket * \overline{P}(s,s_1) * \llbracket \neg b(s_1) \rrbracket * \llbracket s' = s_1 \rrbracket \end{pmatrix}}_{+(\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket}$$

 $= \{ Rewrite \}$

$$\lambda(s,s'). \left(\sum_{i=1}^{2} \begin{pmatrix} \sum_{\infty} s_{i-1}. \llbracket b(s) \rrbracket * \overline{P}(s,s_{i-1}) * \\ \sum_{i=1}^{2} \begin{pmatrix} \sum_{\infty} s_{i-2}. \llbracket b(s_{i-1}) \rrbracket * \overline{P}(s_{i-1},s_{i-2}) * \\ \vdots * \\ (\sum_{\infty} s_{0}. \llbracket b(s_{1}) \rrbracket * \overline{P}(s_{1},s_{0}) * \llbracket \neg b(s_{0}) \rrbracket * \llbracket s' = s_{0} \rrbracket) \end{pmatrix} \right) \right) \right)$$

Assume $(\mathcal{F}_P^b)^n(\mathring{0})$ satisfies the theorem, we show below that $(\mathcal{F}_P^b)^{n+1}(\mathring{0})$ also satisfies the theorem.

$$\left(\mathcal{F}_{P}^{b}\right)^{n+1}(\mathring{0})$$

= { Assumption, $\mathcal{F}^{n+1}(\mathring{0}) = \mathcal{F}(\mathcal{F}^n(\mathring{0}))$, and same as previous proof }

$$\lambda(s,s').\underbrace{\begin{pmatrix} \begin{bmatrix} b(s) \end{bmatrix} * \\ \\ \left(\begin{bmatrix} \sum_{\infty} s_{n-1} \bullet \overline{P}(s,s_{n-1}) * \\ \sum_{i=1}^{n-1} \begin{bmatrix} \sum_{i=1}^{n-1} \begin{bmatrix} b(s_{n-1}) \end{bmatrix} * \overline{P}(s_{n-1},s_{i-1}) * \\ \sum_{i=1}^{n-1} \begin{bmatrix} \sum_{i=1}^{n-1} \begin{bmatrix} \sum_{i=1}^{n-1} \begin{bmatrix} b(s_{n-1}) \end{bmatrix} * \overline{P}(s_{n-1},s_{i-2}) * \\ \vdots * \\ \sum_{i=1}^{n-1} \begin{bmatrix} \sum_{i=1}^{n-1} \begin{bmatrix} \sum_{i=1}^{n-1} b(s_{n-1}) \end{bmatrix} * \overline{P}(s_{1},s_{0}) * [\neg b(s_{0})] * [s'=s_{0}] \end{pmatrix} \right) \end{pmatrix} \right)} \\ + ([\neg b(s_{n-1})]) * [s'=s_{n-1}]$$

= { Expand finite summation }

= { Law (addition) and proofs of summable omitted }

$$\lambda(s,s'). \left(\frac{\left(\begin{bmatrix} \sum_{\infty} s_{n-1} \bullet \overline{P}(s,s_{n-1}) * \\ \sum_{\infty} s_{n-1-1} . \llbracket b(s_{n-1}) \rrbracket * \overline{p}(s_{n-1},s_{n-1-1}) * \\ \sum_{\infty} s_{n-1-2} . \llbracket b(s_{n-1}) \rrbracket * \overline{p}(s_{n-1},s_{n-1-2}) * \\ \vdots * \\ \left(\sum_{\infty} s_{0} . \llbracket b(s_{1}) \rrbracket * \overline{p}(s_{1},s_{0}) * \llbracket \neg b(s_{0}) \rrbracket * \llbracket s' = s_{0} \rrbracket \right) \right) \right) \right) \\ + \cdots + \\ \sum_{\infty} s_{n-1} \bullet \overline{P}(s,s_{n-1}) * \\ \sum_{\infty} s_{0} . \llbracket b(s_{n-1}) \rrbracket * \overline{p}(s_{n-1},s_{0}) * \llbracket \neg b(s_{0}) \rrbracket * \llbracket s' = s_{0} \rrbracket \\ + \sum_{\infty} s_{n-1} \bullet \overline{P}(s,s_{n-1}) * (\llbracket \neg b(s_{n-1}) \rrbracket) * \llbracket s' = s_{n-1} \rrbracket \right) \right) \right)$$

= { Law (multiplication of constant) and combine summation }

$$\lambda(s,s'). \underbrace{ \begin{bmatrix} \sum_{i=1}^n \left[\sum_{\infty} s_{i-1}. \llbracket b(s) \rrbracket * \overline{P}(s,s_{i-1}) * \\ \sum_{i=1}^n \left(\sum_{\infty} s_{i-2}. \llbracket b(s_{i-1}) \rrbracket * \overline{P}(s_{i-1},s_{i-2}) * \\ \vdots * \left(\sum_{\infty} s_0. \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) * \llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket \right) \right) }_{+ (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket}$$

This concludes the proof.

Theorem Appendix A.2. Provided P is a distribution, that is, is_final_dist(P).

$$\begin{split} \left(\mathcal{F}_{P}^{b}\right)^{0}(\mathring{\mathbf{1}}) &= \mathring{\mathbf{1}} \\ \left(\mathcal{F}_{P}^{b}\right)^{1}(\mathring{\mathbf{1}}) &= \lambda(s,s'). \ \Sigma_{\infty}s_{0}.\llbracket b(s)\rrbracket * \overline{P}(s,s_{0}) + \llbracket \neg \ b(s)\rrbracket * \llbracket s' = s \rrbracket \end{split}$$

If
$$n > 1$$
, then

$$\begin{array}{l} > 1, \ then \\ & \left(\mathcal{F}_{P}^{b}\right)^{n} (\mathring{1}) = \\ & \left(\begin{pmatrix} \sum_{\infty} s_{n-1}. \llbracket b(s) \rrbracket * \overline{P}(s,s_{n-1}) * \\ \left(\sum_{\infty} s_{n-2}. \llbracket b(s_{n-1}) \rrbracket * \overline{P}(s_{n-1},s_{n-2}) * \\ \left(\vdots * \\ \left(\sum_{\infty} s_{0}. \llbracket b(s_{1}) \rrbracket * \overline{P}(s_{1},s_{0}) \right) \end{pmatrix} \right) + \\ & \lambda(s,s'). \begin{pmatrix} \sum_{\infty} s_{i-1}. \llbracket b(s) \rrbracket * \overline{P}(s,s_{i-1}) * \\ \left(\sum_{i=1}^{n-1} \left(\sum_{i=1}^{n-1} \left[\sum_{i=1}^{n$$

We note that the part marked with (diff) is the only difference of this $(\mathcal{F}_P^b)^n$ (1) from $(\mathcal{F}_P^b)^n$ (0).

PROOF. We show below that $(\mathcal{F}_P^b)^n$ (1) for n=0 to 3 satisfies the theorem.

$$(\mathcal{F}_P^b)^0(\mathring{1}) = \lambda(s,s').0 = \mathring{1}$$

$$(\mathcal{F}_P^b)^1(\mathring{1})$$

$$= \{ \text{Defintion (loop function)} \}$$

$$\text{if }_c b \text{ then } (P:_{\hat{p}} \mathring{1}) \text{ else } \mathbb{I}_p$$

$$= \{ \text{Law } (\mathcal{F}_P^b \text{ altdef}) \}$$

$$\mathbb{b} \| * \overline{(P:_{\hat{p}} \mathring{1})} + \mathbb{b} \| * \mathbb{I} \mathbb{I} \|$$

$$= \{ P:_{\hat{p}} \mathring{1} = \underline{\sum_{\infty} v_0 \bullet \overline{P}[v_0/\mathbf{v}']} + \mathbb{b} \| * \mathbb{I} \mathbb{I} \|$$

$$= \{ \text{Expand as a function form, use } s \text{ and } s' \text{ for initial and final observation states} \}$$

$$\{ \text{Substitution and Definition (skip), Theorem 5.8, and Law (multiplication of constant)} \}$$

$$\lambda(s,s'). \ \underline{\sum_{\infty} s_0. [b(s)] * \overline{P}(s,s_0) + \mathbb{b} [s] * \mathbb{s}' = s]}$$

$$(\mathcal{F}_P^b)^2(\mathring{1})$$

$$= \{ \mathcal{F}^2(\mathring{1}) = \mathcal{F}(\mathcal{F}^1(\mathring{1})) \text{ and same as previous proof} \}$$

$$\lambda(s,s'). \ \left(\frac{[b(s)] * (\underline{\sum_{\infty} s_0. [b(s_1)] * \overline{P}(s_1,s_0) + [-b(s_1)] * [s' = s_1]}}{(\underline{\sum_{\infty} s_0. [b(s_1)] * \overline{P}(s_1,s_0) + [-b(s_1)] * [s' = s_1]}} \right)$$

$$= \{ \text{Theorem 5.8 where the proof of } is_prob \text{ is omitted} \}$$

$$\{ \text{Law (multiplication of constant) and proof of summable is omitted} \}$$

$$\lambda(s,s').\underbrace{\begin{pmatrix} \Sigma_{\infty}s_1 \bullet \llbracket b(s) \rrbracket * \overline{P}(s,s_1) * \left(\Sigma_{\infty}s_0 \cdot \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) \right) + \\ \Sigma_{\infty}s_1 \bullet \llbracket b(s) \rrbracket * \overline{P}(s,s_1) * \left(\llbracket \neg b(s_1) \rrbracket * \llbracket s' = s_1 \rrbracket \right) + \\ \llbracket \neg b(s) \rrbracket * \llbracket s' = s \rrbracket \end{pmatrix}}$$

Assume $(\mathcal{F}_P^b)^n(\mathring{1})$ satisfies the theorem, we show below that $(\mathcal{F}_P^b)^{n+1}(\mathring{1})$ also satisfies the theorem.

$$\left(\mathcal{F}_{P}^{b}\right)^{n+1}(\mathring{1})$$

= { Assumption,
$$\mathcal{F}^{n+1}(\mathring{1}) = \mathcal{F}(\mathcal{F}^n(\mathring{1}))$$
, and same as previous proof }

$$\lambda(s,s'). \begin{cases} \begin{bmatrix} b(s) \\ \end{bmatrix} * \\ \begin{pmatrix} \begin{bmatrix} \sum_{\infty} s_n \bullet \overline{P}(s,s_n) * \\ \\ \\ \end{bmatrix} & \begin{bmatrix} \sum_{\infty} s_{n-1} . \llbracket b(s_n) \rrbracket * \overline{P}(s_n,s_{n-1}) * \\ \\ \\ \vdots & \\ \\ \end{bmatrix} & \begin{bmatrix} \sum_{\infty} s_{n-2} . \llbracket b(s_n) \rrbracket * \overline{P}(s_{n-1},s_{n-2}) * \\ \\ \vdots & \\ \\ \end{bmatrix} & \begin{bmatrix} \sum_{\infty} s_{i-2} . \llbracket b(s_n) \rrbracket * \overline{P}(s_1,s_0) \end{pmatrix} \\ \\ & \begin{bmatrix} \sum_{\infty} s_{i-1} . \llbracket b(s_n) \rrbracket * \overline{P}(s_n,s_{i-1}) * \\ \\ \vdots & \\ \\ \end{bmatrix} & \begin{bmatrix} \sum_{i=1} \begin{pmatrix} \sum_{\infty} s_{i-1} . \llbracket b(s_n) \rrbracket * \overline{P}(s_n,s_{i-1}) * \\ \\ \vdots & \\ \\ \end{bmatrix} & \begin{bmatrix} \sum_{i=1} \begin{pmatrix} \sum_{\infty} s_{i-2} . \llbracket b(s_{i-1}) \rrbracket * \overline{P}(s_1,s_0) * \llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket \end{pmatrix} \end{pmatrix} \end{pmatrix} \\ = \{ \text{ Expand finite summation, Law (multiplication of constant), same as previous proof } \} \end{cases}$$

= { Expand finite summation, Law (multiplication of constant), same as previous proof }

$$\lambda(s,s'). \begin{pmatrix} \left(\sum_{\infty} s_n. \llbracket b(s) \rrbracket * \overline{P}(s,s_n) * \\ \left(\sum_{\infty} s_{n-1}. \llbracket b(s_n) \rrbracket * \overline{P}(s_n,s_{n-1}) * \\ \left(\vdots * \\ \left(\sum_{\infty} s_0. \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) \right) \right) \end{pmatrix} + \\ \left(\sum_{i=1}^{n} \left(\sum_{\infty} s_{i-1}. \llbracket b(s) \rrbracket * \overline{P}(s,s_{i-1}) * \\ \left(\sum_{i=1}^{n} \left(\sum_{\infty} s_{i-2}. \llbracket b(s_{i-1}) \rrbracket * \overline{P}(s_{i-1},s_{i-2}) * \\ \left(\vdots * \\ \left(\sum_{\infty} s_0. \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) * \llbracket \neg b(s_0) \rrbracket * \llbracket s' = s_0 \rrbracket \right) \right) \right) \\ + (\llbracket \neg b(s) \rrbracket) * \llbracket s' = s \rrbracket$$

This concludes the proof.

Theorem Appendix A.3.

$$\forall n : \mathbb{N} \mid n \geq 1 \bullet \mathcal{ID}(n, b, P) = \lambda(s, s'). \begin{pmatrix} \Sigma_{\infty} s_{n-1}. \llbracket b(s) \rrbracket * \overline{P}(s, s_{n-1}) * \\ \sum_{\infty} s_{n-2}. \llbracket b(s_{n-1}) \rrbracket * \overline{P}(s_{n-1}, s_{n-2}) * \\ \vdots * \\ (\Sigma_{\infty} s_{0}. \llbracket b(s_{1}) \rrbracket * \overline{P}(s_{1}, s_{0})) \end{pmatrix}$$

PROOF. If n = 1, then

$$\mathcal{ID}(1, b, P) =$$

= { Defintions (iteration difference) and \mathcal{F}_0 }

if $_{c}$ b then $(P;_{p} 1)$ else 0

= { Theorem 5.15 Law 3 and Theorem 5.13 Law 5, and Theorem 5.8 }

$$\llbracket b \rrbracket * \overline{\left(P;_{p} \mathring{1}\right)}$$

 $= \{ P :_{p} \mathring{1} = \Sigma_{\infty} v_{0} \bullet \overline{P}[v_{0}/\mathbf{v}'] \}$

{ Expand as a function form, use s and s' for initial and final observation states}

$$\lambda(s,s')$$
. $\Sigma_{\infty} s_0 \cdot \llbracket b(s) \rrbracket * P(s,s_0)$

Assume $\mathcal{ID}(n, b, P)$ satisfies the theorem, we show below that $\mathcal{ID}(n+1, b, P)$ also satisfies the theorem.

$$\mathcal{ID}(n+1,b,P) =$$

```
if _c b then (P;_p \mathcal{ID}(n,b,P)) else 0
            = { Theorem 5.15 Law 3 and Theorem 5.13 Law 5, and Theorem 5.8 }
                \llbracket b \rrbracket * (P;_{p} \mathcal{ID}(n,b,P))
           = { Definition (sequential composition) }
              \llbracket b \rrbracket * \left( \underbrace{\left( \Sigma_{\infty} v_0 \bullet \overline{P}[v_0/\mathbf{v}'] * \overline{\left( \mathcal{I} \mathcal{D}(n, b, P) \right)}[v_0/\mathbf{v}] \right)} \right)
            = { Theorem 5.8 where the proof of is\_prob is omitted }
               \llbracket b \rrbracket * \left( \Sigma_{\infty} v_0 \bullet \overline{P}[v_0/\mathbf{v}'] * (\mathcal{ID}(n, b, P))[v_0/\mathbf{v}] \right)
            = { Law (multiplication of constant) and proof of summable is omitted }
               (\Sigma_{\infty} v_0 \bullet \llbracket b \rrbracket * \overline{P}[v_0/\mathbf{v}'] * (\mathcal{ID}(n, b, P))[v_0/\mathbf{v}])
            = { Assumption, expand as a function form, use s and s' for initial and final observation states }
              \lambda(s,s'). \begin{pmatrix} \Sigma_{\infty} s_n \cdot \llbracket b(s) \rrbracket * \overline{P}(s,s_n) * \\ \sum_{\infty} s_{n-1} \cdot \llbracket b(s_n) \rrbracket * \overline{P}(s_n,s_{n-1}) * \\ \vdots * \\ (\Sigma_{\infty} s_0 \cdot \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0)) \end{pmatrix}
This concludes the proof of Theorem Appendix A.3.
       We now show the proof of Theorem 6.5: \forall n : \mathbb{N} \bullet \mathcal{F}_P^{b^n}(\mathring{1}) - \mathcal{F}_P^{b^n}(\mathring{0}) = \mathcal{ID}(n, b, P)
PROOF. For n = 0,
              \mathcal{F}_{P}^{b^{0}}(\mathring{1}) - \mathcal{F}_{P}^{b^{0}}(\mathring{0})
            = { Theorems Appendix A.1 and Appendix A.2 }
            = \{ \text{ Theorem } 5.10 \}
            = { Definition (iteration difference) }
               \mathcal{ID}(0, b, P)
For n=1,
               \mathcal{F}_{P}^{b^{1}}(\mathring{1}) - \mathcal{F}_{P}^{b^{1}}(\mathring{0})
           = { Theorems Appendix A.1 and Appendix A.2 }
                \lambda(s, s'). \ \Sigma_{\infty} s_0. \llbracket b(s) \rrbracket * \overline{P}(s, s_0) + \llbracket \neg \ b(s) \rrbracket * \llbracket s' = s \rrbracket - \lambda(s, s'). \ \llbracket \neg \ b(s) \rrbracket * \llbracket s' = s \rrbracket
           = \{ \text{ Definition } 4.8 \}
               \lambda(s,s').\left(\underline{\Sigma_{\infty}s_0.\llbracket b(s)\rrbracket*\overline{P}(s,s_0)+\llbracket\neg\ b(s)\rrbracket*\llbracket s'=s\rrbracket}-\underline{\llbracket\neg\ b(s)\rrbracket*\llbracket s'=s\rrbracket}\right)
           = \{ \text{ Definition } 4.6 \}
               \lambda(s,s').\left(\max\left(0,\overline{\left(\Sigma_{\infty}s_{0}.\llbracket b(s)\rrbracket\ast\overline{P}(s,s_{0})+\llbracket\lnot\ b(s)\rrbracket\ast\llbracket s'=s\rrbracket\right)}-\overline{\left(\llbracket\lnot\ b(s)\rrbracket\ast\llbracket s'=s\rrbracket\right)}\right)
            = { Theorem 5.8 where the proof of is\_prob is omitted }
```

= { Defintions (iteration difference) and \mathcal{F}_0 }

$$\lambda(s,s'). \left(\max\left(0,\left(\Sigma_{\infty}s_{0}.\llbracket b(s)\rrbracket*\overline{P}(s,s_{0})+\llbracket \neg b(s)\rrbracket*\llbracket s'=s\rrbracket\right)-(\llbracket \neg b(s)\rrbracket*\llbracket s'=s\rrbracket)\right)\right)$$

$$=\left\{\forall\,s\bullet\Sigma_{\infty}s_{0}.\llbracket b(s)\rrbracket*\overline{P}(s,s_{0})\geq0\text{ because }P\text{ is a distribution, Theorem 5.4, and Definition 5.4}\right.\right\}$$

$$\lambda(s,s').\ \underline{\Sigma_{\infty}s_{0}}.\llbracket b(s)\rrbracket*\overline{P}(s,s_{0})$$

$$=\left\{\text{Theorem Appendix A.3}\right\}$$

$$\mathcal{ID}(1,b,P)$$
For $n>1$, we assume $\mathcal{F}_{P}^{b\,n}(\mathring{1})-\mathcal{F}_{P}^{b\,n}(\mathring{0})=\mathcal{ID}(n,b,P)$, then
$$\mathcal{F}_{P}^{b\,n+1}(\mathring{1})-\mathcal{F}_{P}^{b\,n+1}(\mathring{0})$$

$$=\left\{\text{Theorems Appendix A.1 and Appendix A.2, and same as previous proof}\right\}$$

$$\lambda(s,s').\left(\begin{array}{c} \sum_{\infty}s_{n}.\llbracket b(s)\rrbracket*\overline{P}(s,s_{n})*\\ \sum_{\infty}s_{n-1}.\llbracket b(s_{n})\rrbracket*\overline{P}(s_{n},s_{n-1})*\\ \vdots*\\ \left(\sum_{\infty}s_{0}.\llbracket b(s_{1})\rrbracket*\overline{P}(s_{1},s_{0})\right)\end{array}\right)$$

$$=\left\{\text{Theorem Appendix A.3}\right\}$$

$$\mathcal{ID}(n+1,b,P)$$
This concludes the proof of Theorem 6.5.

This concludes the proof of Theorem 6.5.

Appendix A.2. Proof of Theorem 7.15 Proof.

 $\lambda n \bullet \overline{\mathcal{ID}(n, c = tl, cflip)}(s, s')$

$$\begin{array}{c} = \{ \text{ Theorem Appendix A.3 where } b = (c = tl) \text{ and } P = cflip \} \\ \\ \lambda \, n \, \bullet \, \boxed{ \begin{pmatrix} \sum_{\infty} s_{n-1}. \llbracket b(s) \rrbracket * \overline{P}(s,s_{n-1}) * \\ \sum_{\infty} s_{n-2}. \llbracket b(s_{n-1}) \rrbracket * \overline{P}(s_{n-1},s_{n-2}) * \\ \vdots * \\ \left(\sum_{\infty} s_0. \llbracket b(s_1) \rrbracket * \overline{P}(s_1,s_0) \right) \end{pmatrix} } \\ (s,s') \\ = \{ \text{ Definition 6.4 where } cstate \text{ only has one variable } c \text{ and so } s \text{ is replaced by } c \} \\ \{ \overline{cflip} = 1/2 * \llbracket c' = hd \rrbracket + 1/2 * \llbracket c' = tl \rrbracket \text{ according to Law } (cflip \text{ altdef}) \text{ and Theorem 5.9} \} \\ \end{array}$$

$$\lambda \, n \, \bullet \, \overline{\left(\lambda(c,c'). \begin{pmatrix} \Sigma_{\infty} s_{n-1}. \llbracket c = tl \rrbracket * (1/2 * \llbracket s_{n-1} = hd \rrbracket + 1/2 * \llbracket s_{n-1} = tl \rrbracket) * \\ \sum_{\infty} s_{n-2}. \llbracket s_{n-1} = tl \rrbracket * (1/2 * \llbracket s_{n-2} = hd \rrbracket + 1/2 * \llbracket s_{n-2} = tl \rrbracket) * \\ \left(\vdots * \\ (\Sigma_{\infty} s_{0}. \llbracket s_{1} = tl \rrbracket * (1/2 * \llbracket s_{0} = hd \rrbracket + 1/2 * \llbracket s_{0} = tl \rrbracket)) \right) \right) \right) \right) } (c,c')$$

= { Every summation variable such as s_{n-1} only when it is equal to tl, then $[s_{n-1} = tl] = 1$. }

{ Otherwise, it is 0 and the whole summation is also 0 because $a * \cdots * 0 * \cdots * b = 0$.}

{ All Σ_{∞} are removed because of only one state tl satisfying $s_i = tl$ }

$$\{ All [s_i = hd] = 0 \}$$

$$\lambda n \bullet \overline{\left(\lambda(c,c'). \begin{pmatrix} \llbracket c = tl \rrbracket * (1/2*1)* \\ 1*(1/2*1)* \\ \vdots * \\ (1*(1/2*1)) \end{pmatrix}\right)} (c,c')$$

$$= \{ \text{ Rewrite } \}$$

$$\lambda \, n \bullet \overline{\left(\underline{\lambda(c,c')}. \llbracket c = tl \rrbracket * (1/2)^n\right)}(c,c')$$

$$= \{ \text{ Theorem 5.8 where } is_prob(\lambda(c,c'). \llbracket c = tl \rrbracket * (1/2)^n) \}$$

$$\lambda \, n \bullet \llbracket c = tl \rrbracket * (1/2)^n$$

So if c = hd, then

$$\left(\lambda n \bullet \overline{\mathcal{ID}(n, c = tl, cflip)}(s, s') = \lambda n \bullet 0\right)$$

Otherwise,

$$\left(\lambda \, n \bullet \overline{\mathcal{ID}(n, c = tl, cflip)}(s, s') = \lambda \, n \bullet (1/2)^n\right)$$

We conclude that $\forall (c, c') : cstate \times cstate$ such that

$$\left(\lambda n \bullet \overline{\mathcal{ID}(n, c = tl, cflip)}(c, c')\right) \xrightarrow{n \to \infty} 0$$

according to Definition 3.4.

Appendix A.3. The necessity of $\mathcal{FS}_{\uparrow}(\lambda n \bullet \mathcal{I}(n, b, P, \mathring{0}))$

We show below to establish the continuity Theorem 6.12, $\mathcal{FS}_{\uparrow}(\lambda n \bullet \mathcal{I}(n, b, P, \mathring{0}))$ is required. Here we omit the details about type conversion to make the discussion clearer.

We aim to prove a continuity theorem below and then our proof later shows that without the premise, we cannot prove such theorem.

Theorem Appendix A.4. If the final state of P is a distribution, then $\mathcal{F}_P^b(X)$ is continuous. That is, for an non-empty countable increasing chain $S_0 \leq S_1 \leq S_2 \leq ...$ of type [s] prfun and bound above (so has a supremum), then

$$\mathcal{F}_{P}^{b}\left(\left|\begin{array}{c} n \bullet S_{n} \right.\right) = \left|\begin{array}{c} n \bullet \mathcal{F}_{P}^{b}(S_{n}) \end{array}\right.$$
 (\mathcal{F}_{P}^{b} continuous)

PROOF. We define

$$F_1(X) \cong \mathbf{if}_c \ b \ \mathbf{then} \ X \ \mathbf{else} \ \mathbb{I}_p$$

 $F_2(X) \cong P :_n X$

So $\mathcal{F}_P^b(X) \cong F_1(F_2(X))$. According to [73, Lemma 5.3], if both F_1 and F_2 are continuous, then $\mathcal{F}_P^b(X)$ is also continuous. It is trivial to show that F_1 is continuous, so our goal is to prove F_2 is continuous. That is,

$$F_2\left(\bigsqcup n \bullet S_n\right) = \bigsqcup n \bullet F_2(S_n) \tag{F_2 continuous}$$

Because S_n is an increasing chain and bound above, according to the monotone sequence theorem, the limit of S_n is just its supremum. That is,

$$\forall (s, s') \bullet (\lambda \ n \bullet S_n(s, s')) \xrightarrow{n \to \infty} \left(\bigsqcup n \bullet S_n \right) (s, s')$$

where (s, s') denotes the initial and final observations. According to Definition 3.4,

$$\forall (s, s') \bullet \forall \epsilon : \mathbb{R} > 0 \bullet \exists M : \mathbb{N} \bullet \forall l \ge M \bullet \left(\left(\bigsqcup n \bullet S_n \right) (s, s') - S_l(s, s') \right) < \epsilon$$

$$(S_n \text{ supremum as limit})$$

According to Theorem 5.16 Law 7, F_2 is monotonic. It is trivial to show that $F_2(S_n)$ is also an increasing chain and bound above. According to the monotone sequence theorem,

$$\forall (s, s') \bullet (\lambda \, n \bullet F_2(S_n)(s, s')) \xrightarrow{n \to \infty} \left(\bigsqcup n \bullet F_2(S_n) \right) (s, s') \tag{F_2(S_n) supremum as limit}$$

Therefore, according to the unique sequence limit theorem (if exists), to prove $(F_2 \text{ continuous})$, we need to prove that

$$\forall (s, s') \bullet (\lambda \, n \bullet F_2(S_n)(s, s')) \xrightarrow{n \to \infty} F_2\left(\bigsqcup n \bullet S_n\right)(s, s') \tag{F_2(\bigsqcup n \bullet S_n) as limit}$$

According to Definition 3.4, this is equal to prove

$$\forall (s,s') \bullet \forall \varepsilon : \mathbb{R} > 0 \bullet \exists N : \mathbb{N} \bullet \forall l \geq N \bullet \mid F_2(S_l)(s,s') - F_2\left(\bigsqcup n \bullet S_n \right)(s,s') \mid < \varepsilon$$

$$= \{ F_2 \text{ is monotonic and } S_l \leq \left(\bigsqcup n \bullet S_n \right), \text{ so } F_2(S_l)(s,s') \leq F_2\left(\bigsqcup n \bullet S_n \right)(s,s') \}$$

$$\forall (s,s') \bullet \forall \varepsilon : \mathbb{R} > 0 \bullet \exists N : \mathbb{N} \bullet \forall l \geq N \bullet F_2\left(\bigsqcup n \bullet S_n \right)(s,s') - F_2(S_l)(s,s') < \varepsilon$$

We can rewrite the non-quantifier part above to

$$|F_{2}(S_{l})(s,s') - F_{2}\left(\bigsqcup n \bullet S_{n}\right)(s,s')| < \varepsilon$$

$$= \{F_{2} \text{ is monotonic and } S_{l} \leq \left(\bigsqcup n \bullet S_{n}\right), \text{ so } F_{2}(S_{l})(s,s') \leq F_{2}\left(\bigsqcup n \bullet S_{n}\right)(s,s')\}$$

$$F_{2}\left(\bigsqcup n \bullet S_{n}\right)(s,s') - F_{2}(S_{l})(s,s') < \varepsilon$$

$$= \{\text{Definitions of } F_{2} \text{ and } ;_{p} \text{ (sequential composition)}\}$$

$$\Sigma_{\infty}s_{0} \bullet P(s,s_{0}) * \left(\bigsqcup n \bullet S_{n}\right)(s_{0},s') - \Sigma_{\infty}s_{0} \bullet P(s,s_{0}) * S_{l}(s_{0},s') < \varepsilon$$

$$= \{\text{Theorem 3.7 Law (subtraction) and proof of summable is omitted }\}$$

$$\Sigma_{\infty}s_{0} \bullet P(s,s_{0}) * \left(\left(\bigsqcup n \bullet S_{n}\right)(s_{0},s') - S_{l}(s_{0},s')\right) < \varepsilon$$

So our goal is to prove

$$\forall (s, s') \bullet \forall \varepsilon : \mathbb{R} > 0 \bullet \exists N : \mathbb{N} \bullet \forall l \ge N \bullet \left(\Sigma_{\infty} s_0 \bullet P(s, s_0) * \left(\left(\bigsqcup n \bullet S_n \right) (s_0, s') - S_l(s_0, s') \right) \right) < \varepsilon$$

The key step in proving the goal above is to supply a witness for N. Based on $(S_n \text{ supremum as limit})$, we can choose a N to make $((\lfloor n \cdot S_n)(s_0, s') - S_l(s_0, s'))$ any small (say $\epsilon(s_0)$), but this cannot guarantees

$$(\Sigma_{\infty} s_0 \bullet P(s, s_0) * \epsilon(s_0)) < \varepsilon$$

because this is an infinite sum. In other words, the question is to find a N such that this summation converges to a value less than any number ε . The approach we use in this paper is to assume $\mathcal{FS}_{\uparrow}(\lambda n \bullet \mathcal{I}(n, b, P, \mathring{0}))$, and then we can construct such N.

We omit further details of this proof.

References

- S. Thrun, W. Burgard, D. Fox, Probabilistic Robotics (Intelligent Robotics and Autonomous Agents), The MIT Press, 2005.
- [2] A. D. Gordon, T. A. Henzinger, A. V. Nori, S. K. Rajamani, Probabilistic programming, in: Future of Software Engineering Proceedings, FOSE 2014, Association for Computing Machinery, New York, NY, USA, 2014, p. 167–181. doi:10.1145/2593882.2593900.

URL https://doi.org/10.1145/2593882.2593900

- [3] Wikipedia, Monty Hall problem Wikipedia, the free encyclopedia, http://en.wikipedia.org/w/index.php?title=Monty%20Hall%20problem&oldid=1137277370, [Online; accessed 09-February-2023] (2023).
- [4] E. C. R. Hehner, A probability perspective, Formal Aspects Comput. 23 (4) (2011) 391-419. doi:10.1007/s00165-010-0157-0.
 URL https://doi.org/10.1007/s00165-010-0157-0
- [5] G. Grimmett, D. Welsh, Probability: An Introduction, Oxford University Press, Clarendon Press, 1986.
- [6] J. Hurd, Formal verification of probabilistic algorithms, Tech. Rep. UCAM-CL-TR-566, University of Cambridge, Computer Laboratory (May 2003). doi:10.48456/tr-566.
 URL https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-566.pdf
- [7] A. McIver, C. Morgan, Abstraction, Refinement and Proof for Probabilistic Systems, Monographs in Computer Science, Springer, 2005. doi:10.1007/b138392.
- [8] A. McIver, C. Morgan, B. L. Kaminski, J.-P. Katoen, A New Proof Rule for Almost-Sure Termination, Proc. ACM Program. Lang. 2 (POPL) (dec 2017). doi:10.1145/3158121. URL https://doi.org/10.1145/3158121
- [9] K. Chatterjee, H. Fu, P. Novotný, Termination Analysis of Probabilistic Programs with Martingales, Cambridge University Press, 2020, p. 221–258.
- [10] A. McIver, C. Morgan, Abstraction, Refinement and Proof for Probabilistic Systems, Springer New York, NY, 2005, Ch. Introduction to pGCL: Its logic and its model, pp. 3–36. doi:10.1007/0-387-27006-X_1. URL https://doi.org/10.1007/0-387-27006-X_1
- [11] C. Morgan, A. McIver, pGCL: formal reasoning for random algorithms, South African Computer Journal 22 (1999) 14-27. URL http://hdl.handle.net/10500/24296
- [12] E. Dijkstra, A Discipline of Programming, Prentice-Hall Series in Automa, Prentice-Hall, 1976. URL https://books.google.co.uk/books?id=MsUmAAAAMAAJ
- [13] M. J. C. Gordon, T. F. Melham (Eds.), Introduction to HOL: A theorem proving environment for higher order logic, Cambridge University Press, 1993. URL http://www.cs.ox.ac.uk/tom.melham/pub/Gordon-1993-ITH.html
- [14] J. Hurd, A. McIver, C. Morgan, Probabilistic guarded commands mechanized in HOL, Theor. Comput. Sci. 346 (1) (2005) 96-112. doi:10.1016/j.tcs.2005.08.005.
- [15] B. L. Kaminski, Advanced weakest precondition calculi for probabilistic programs, Ph.D. thesis, RWTH Aachen University, Germany (2019). URL http://publications.rwth-aachen.de/record/755408
- [16] F. Olmedo, F. Gretz, N. Jansen, B. L. Kaminski, J.-P. Katoen, A. Mciver, Conditioning in probabilistic programming, ACM Trans. Program. Lang. Syst. 40 (1) (jan 2018). doi:10.1145/3156018. URL https://doi.org/10.1145/3156018
- [17] B. L. Kaminski, J.-P. Katoen, C. Matheja, F. Olmedo, Weakest Precondition Reasoning for Expected Run-Times of Probabilistic Programs, Springer Berlin Heidelberg, 2016, pp. 364–389. doi:10.1007/978-3-662-49498-1_15.
- [18] B. L. Kaminski, J.-P. Katoen, C. Matheja, F. Olmedo, Weakest Precondition Reasoning for Expected Runtimes of Randomized Algorithms, J. ACM 65 (5) (aug 2018). doi:10.1145/3208102.
- [19] G. Barthe, T. Espitau, M. Gaboardi, B. Grégoire, J. Hsu, P.-Y. Strub, An Assertion-Based Program Logic for Probabilistic Programs, Springer International Publishing, 2018, pp. 117–144. doi:10.1007/978-3-319-89884-1_5.
- [20] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, P.-Y. Strub, EasyCrypt: A Tutorial, Springer International Publishing, 2014, pp. 146–166. doi:10.1007/978-3-319-10082-1_6.
- [21] P. Schröer, K. Batz, B. L. Kaminski, J.-P. Katoen, C. Matheja, A Deductive Verification Infrastructure for Probabilistic Programs, Proc. ACM Program. Lang. 7 (OOPSLA2) (oct 2023). doi:10.1145/3622870. URL https://doi.org/10.1145/3622870
- [22] E. C. R. Hehner, Probabilistic predicative programming, in: D. Kozen, C. Shankland (Eds.), Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings, Vol. 3125 of Lecture Notes in Computer Science, Springer, 2004, pp. 169–185. doi:10.1007/978-3-540-27764-4_10. URL https://doi.org/10.1007/978-3-540-27764-4_10
- [23] K. Ye, A. Cavalcanti, S. Foster, A. Miyazawa, J. Woodcock, Probabilistic modelling and verification using RoboChart and PRISM, Softw. Syst. Model. 21 (2) (2022) 667–716. doi:10.1007/s10270-021-00916-8. URL https://doi.org/10.1007/s10270-021-00916-8
- [24] K. Ye, S. Foster, J. Woodcock, Automated reasoning for probabilistic sequential programs with theorem proving, in: U. Fahrenberg, M. Gehrke, L. Santocanale, M. Winter (Eds.), Relational and Algebraic Methods in Computer Science, Springer International Publishing, Cham, 2021, pp. 465–482.
- [25] E. C. Hehner, Specifications, programs, and total correctness, Science of Computer Programming 34 (3) (1999) 191-205. doi:https://doi.org/10.1016/S0167-6423(98)00027-6. URL https://www.sciencedirect.com/science/article/pii/S0167642398000276
- [26] F. Dahlqvist, A. Silva, D. Kozen, Semantics of Probabilistic Programming: A Gentle Introduction, in: G. Barthe, J.-P. Katoen, A. Silva (Eds.), Foundations of Probabilistic Programming, Cambridge University Press, 2020, p. 1–42. doi:10.1017/9781108770750.002.
- [27] A. McIver, C. Morgan, Correctness by construction for probabilistic programs, in: T. Margaria, B. Steffen (Eds.), Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I, Vol.

- 12476 of Lecture Notes in Computer Science, Springer, 2020, pp. 216–239. doi:10.1007/978-3-030-61362-4_{1}-{2}. URL https://doi.org/10.1007/978-3-030-61362-4_12
- [28] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (10) (1969) 576–580. doi:10.1145/ 363235.363259.
- [29] E. C. R. Hehner, Predicative programming part i, Commun. ACM 27 (2) (1984) 134-143. doi:10.1145/69610.357988. URL https://doi.org/10.1145/69610.357988
- [30] D. Kozen, Semantics of probabilistic programs, Journal of Computer and System Sciences 22 (3) (1981) 328-350. doi: https://doi.org/10.1016/0022-0000(81)90036-2. URL https://www.sciencedirect.com/science/article/pii/0022000081900362
- [31] D. Kozen, A probabilistic PDL, Journal of Computer and System Sciences 30 (2) (1985) 162-178. doi:https://doi.org/10.1016/0022-0000(85)90012-1.
 URL https://www.sciencedirect.com/science/article/pii/0022000085900121
- [32] J. He, C. Morgan, A. McIver, Deriving probabilistic semantics via the 'weakest completion', in: J. Davies, W. Schulte, M. Barnett (Eds.), Formal Methods and Software Engineering, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 131–145
- [33] J. Woodcock, A. Cavalcanti, A Tutorial Introduction to Designs in Unifying Theories of Programming, in: E. A. Boiten, J. Derrick, G. Smith (Eds.), Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings, Vol. 2999 of Lecture Notes in Computer Science, Springer, 2004, pp. 40–66. doi:10.1007/ 978-3-540-24756-2\dangle4. URL https://doi.org/10.1007/978-3-540-24756-2_4
- [34] C. Morgan, Proof rules for probabilistic loops, in: Proceedings of the BCS-FACS 7th Conference on Refinement, BCS Learning & Development Ltd, 1996, p. 10. doi:10.14236/ewic/RW1996.10.
- [35] D. Cock, Verifying Probabilistic Correctness in Isabelle with pGCL, in: F. Cassez, R. Huuck, G. Klein, B. Schlich (Eds.), Proceedings Seventh Conference on Systems Software Verification, SSV 2012, Sydney, Australia, 28-30 November 2012, Vol. 102 of EPTCS, 2012, pp. 167–178. doi:10.4204/EPTCS.102.15.
- [36] F. Gretz, J.-P. Katoen, A. McIver, Operational versus weakest pre-expectation semantics for the probabilistic guarded command language, Performance Evaluation 73 (2014) 110-132, special Issue on the 9th International Conference on Quantitative Evaluation of Systems. doi:https://doi.org/10.1016/j.peva.2013.11.004.

 URL https://www.sciencedirect.com/science/article/pii/S0166531613001429
- [37] H. Jifeng, K. Seidel, A. McIver, Probabilistic models for the guarded command language, Science of Computer Programming 28 (2) (1997) 171-192, formal Specifications: Foundations, Methods, Tools and Applications. doi:https://doi.org/10.1016/S0167-6423(96)00019-6.
 URL https://www.sciencedirect.com/science/article/pii/S0167642396000196
- [38] J. Woodcock, A. Cavalcanti, S. Foster, A. Mota, K. Ye, Probabilistic Semantics for RoboChart, in: P. Ribeiro, A. Sampaio (Eds.), Unifying Theories of Programming, Springer International Publishing, Cham, 2019, pp. 80–105.
- [39] P. Audebaud, C. Paulin-Mohring, Proofs of randomized algorithms in coq, Sci. Comput. Program. 74 (8) (2009) 568-589. doi:10.1016/j.scico.2007.09.002. URL https://doi.org/10.1016/j.scico.2007.09.002
- [40] The Coq development team, The Coq Proof Assistant, coq.inria.fr, Accessed: 2021-05-20 (2020).
- [41] L. H. Ramshaw, Formalizing the analysis of algorithms., Ph.D. thesis, Stanford University, Stanford, CA, USA, aAI8001994 (1979).
- [42] J. den Hartog, E. De Vink, Verifying Probabilistic Programs Using a Hoare like Logic, International journal of foundations of computer science 13 (3) (2002) 315–340, imported from DIES. doi:10.1142/S012905410200114X.
- [43] R. Chadha, L. Cruz-Filipe, P. Mateus, A. Sernadas, Reasoning about probabilistic sequential programs, Theor. Comput. Sci. 379 (1-2) (2007) 142–165. doi:10.1016/j.tcs.2007.02.040.
- [44] R. Rand, S. Zdancewic, VPHL: A verified partial-correctness logic for probabilistic programs, in: D. R. Ghica (Ed.), The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015, Vol. 319 of Electronic Notes in Theoretical Computer Science, Elsevier, 2015, pp. 351-367. doi: 10.1016/j.entcs.2015.12.021.
- [45] G. Barthe, B. Grégoire, S. Zanella Béguelin, Formal certification of code-based cryptographic proofs, in: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09, Association for Computing Machinery, New York, NY, USA, 2009, p. 90–101. doi:10.1145/1480881.1480894. URL https://doi.org/10.1145/1480881.1480894
- [46] B. L. Kaminski, J. Katoen, C. Matheja, On the hardness of analyzing probabilistic programs, Acta Informatica 56 (3) (2019) 255–285. doi:10.1007/s00236-018-0321-1.
- [47] C. Morgan, A. McIver, K. Seidel, Probabilistic predicate transformers, ACM Transactions on Programming Languages and Systems (TOPLAS) 18 (3) (1996) 325–353.
- [48] A. Chakarov, S. Sankaranarayanan, Expectation Invariants for Probabilistic Program Loops as Fixed Points, in: M. Müller-Olm, H. Seidl (Eds.), Static Analysis, Springer International Publishing, Cham, 2014, pp. 85–100.
- [49] B. L. Kaminski, J. Katoen, On the Hardness of Almost-Sure Termination, in: G. F. Italiano, G. Pighizzini, D. Sannella (Eds.), Mathematical Foundations of Computer Science 2015 40th International Symposium, MFCS 2015, Milan, Italy, August 24-28, 2015, Proceedings, Part I, Vol. 9234 of Lecture Notes in Computer Science, Springer, 2015, pp. 307-318. doi:10.1007/978-3-662-48057-1_{2}{4}. URL https://doi.org/10.1007/978-3-662-48057-1_24
- [50] O. Bournez, F. Garnier, Proving positive almost-sure termination, in: J. Giesl (Ed.), Term Rewriting and Applications,

- Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 323–337.
- [51] J. Esparza, A. Gaiser, S. Kiefer, Proving Termination of Probabilistic Programs Using Patterns, in: P. Madhusudan, S. A. Seshia (Eds.), Computer Aided Verification 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings, Vol. 7358 of Lecture Notes in Computer Science, Springer, 2012, pp. 123-138. doi: 10.1007/978-3-642-31424-7_{1}_{4}. URL https://doi.org/10.1007/978-3-642-31424-7_14
- [52] A. Chakarov, S. Sankaranarayanan, Probabilistic Program Analysis with Martingales, in: N. Sharygina, H. Veith (Eds.), Computer Aided Verification 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings, Vol. 8044 of Lecture Notes in Computer Science, Springer, 2013, pp. 511-526. doi: 10.1007/978-3-642-39799-8_{4}. URL https://doi.org/10.1007/978-3-642-39799-8_34
- [53] L. M. Ferrer Fioriti, H. Hermanns, Probabilistic termination: Soundness, completeness, and compositionality, SIGPLAN Not. 50 (1) (2015) 489?501. doi:10.1145/2775051.2677001. URL https://doi.org/10.1145/2775051.2677001
- [54] S. Agrawal, K. Chatterjee, P. Novotný, Lexicographic ranking supermartingales: An efficient approach to termination of probabilistic programs, Proc. ACM Program. Lang. 2 (POPL) (dec 2017). doi:10.1145/3158122. URL https://doi.org/10.1145/3158122
- [55] M. Huang, H. Fu, K. Chatterjee, New Approaches for Almost-Sure Termination of Probabilistic Programs, in: S. Ryu (Ed.), Programming Languages and Systems 16th Asian Symposium, APLAS 2018, Wellington, New Zealand, December 2-6, 2018, Proceedings, Vol. 11275 of Lecture Notes in Computer Science, Springer, 2018, pp. 181-201. doi:10.1007/978-3-030-02768-1_{1}_{1}. URL https://doi.org/10.1007/978-3-030-02768-1_11
- [56] K. Chatterjee, H. Fu, P. Novotný, R. Hasheminezhad, Algorithmic Analysis of Qualitative and Quantitative Termination Problems for Affine Probabilistic Programs, ACM Trans. Program. Lang. Syst. 40 (2) (may 2018). doi:10.1145/3174800. URL https://doi.org/10.1145/3174800
- [57] A. McIver, C. Morgan, Abstraction, Refinement and Proof for Probabilistic Systems, Monographs in Computer Science, Springer, 2005, Ch. Introduction to pGCL, pp. 3–35. doi:10.1007/b138392. URL https://doi.org/10.1007/b138392
- [58] J. M. Spivey, The Z Notation: A Reference Manual, 2nd, Prentice-Hall, 1992.
- [59] J. C. P. Woodcock, J. Davies, Using Z—Specification, Refinement, and Proof, Prentice-Hall, 1996.
- [60] C. A. R. Hoare, J. He, Unifying Theories of Programming, Prentice-Hall, 1998.
- [61] C. A. R. Hoare, Programs are predicates, Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences 312 (1984) 475 489.
- [62] A. Tarski, On the calculus of relations, J. Symb. Log. 6 (3) (1941) 73-89. doi:10.2307/2268577.
- [63] S. Foster, J. Baxter, A. Cavalcanti, J. Woodcock, F. Zeyda, Unifying semantic foundations for automated verification tools in isabelle/utp, Science of Computer Programming 197 (2020) 102510. doi:https://doi.org/10.1016/j.scico. 2020.102510. URL https://www.sciencedirect.com/science/article/pii/S0167642320301192
- [64] J. N. Foster, Bidirectional programming languages, Ph.D. thesis, University of Pennsylvania (2009).
- [65] S. Foster, F. Zeyda, J. Woodcock, Unifying Heterogeneous State-Spaces with Lenses, in: A. Sampaio, F. Wang (Eds.), Theoretical Aspects of Computing ICTAC 2016 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings, Vol. 9965 of Lecture Notes in Computer Science, 2016, pp. 295-314. doi: 10.1007/978-3-319-46750-4_{1}_{7}.

 URL https://doi.org/10.1007/978-3-319-46750-4_17
- [66] E. C. R. Hehner, Predicative programming, part II, Commun. ACM 27 (2) (1984) 144-151. doi:10.1145/69610.357990. URL https://doi.org/10.1145/69610.357990
- [67] E. C. R. Hehner, A Practical Theory of Programming, Springer New York, 1993. doi:10.1007/978-1-4419-8596-5.
- [68] S. Abramsky, A. Jung, Domain theory, in: Handbook of Logic in Computer Science (Vol. 3): Semantic Structures, Oxford University Press, Inc., USA, 1995, p. 1–168.
- [69] J. Hölzl, F. Immler, B. Huffman, Type Classes and Filters for Mathematical Analysis in Isabelle/HOL, in: S. Blazy, C. Paulin-Mohring, D. Pichardie (Eds.), Interactive Theorem Proving, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 279–294.
- [70] A. Tarski, A Lattice-Theoretical Fixpoint Theorem and its Applications, Pacific Journal of Mathematics 5 (2) (1955) 285–309.
- [71] D. E. Knuth, Two Notes on Notation, The American Mathematical Monthly 99 (5) (1992) 403-422.
 URL http://www.jstor.org/stable/2325085
- [72] C. GUNTER, D. SCOTT, Chapter 12 semantic domains, in: J. VAN LEEUWEN (Ed.), Formal Models and Semantics, Handbook of Theoretical Computer Science, Elsevier, Amsterdam, 1990, pp. 633–674. doi:https://doi.org/10.1016/ B978-0-444-88074-1.50017-2.
- URL https://www.sciencedirect.com/science/article/pii/B9780444880741500172
- [73] H. R. Nielson, F. Nielson, Semantics with Applications: An Appetizer, Springer London, 2007. doi:10.1007/978-1-84628-692-6.
- [74] C. Daws, Symbolic and parametric model checking of discrete-time markov chains, in: Z. Liu, K. Araki (Eds.), Theoretical Aspects of Computing - ICTAC 2004, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 280–294.
- [75] E. M. Hahn, H. Hermanns, L. Zhang, Probabilistic reachability for parametric markov models, Int. J. Softw. Tools

- Technol. Transf. 13 (1) (2011) 3-19. doi:10.1007/s10009-010-0146-x. URL https://doi.org/10.1007/s10009-010-0146-x
- [76] A. Miyazawa, P. Ribeiro, W. Li, A. Cavalcanti, J. Timmis, J. Woodcock, Robochart: modelling and verification of the functional behaviour of robotic applications, Softw. Syst. Model. 18 (5) (2019) 3097–3149. doi:10.1007/s10270-018-00710-z.
 URL https://doi.org/10.1007/s10270-018-00710-z
- [77] A. Cavalcanti, W. Barnett, J. Baxter, G. Carvalho, M. C. Filho, A. Miyazawa, P. Ribeiro, A. Sampaio, RoboStar Technology: A Roboticist's Toolbox for Combined Proof, Simulation, and Testing, in: A. Cavalcanti, B. Dongol, R. Hierons, J. Timmis, J. Woodcock (Eds.), Software Engineering for Robotics, Springer International Publishing, Cham, 2021, pp. 249–293. doi:10.1007/978-3-030-66494-7_9. URL https://doi.org/10.1007/978-3-030-66494-7_9
- [78] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall Int., 1985.
- [79] A. W. Roscoe, Understanding Concurrent Systems, Texts in Computer Science, Springer, 2011.
- [80] J. Baxter, P. Ribeiro, A. Cavalcanti, Sound reasoning in tock-CSP, Acta Informatica (Apr. 2021). doi:10.1007/s00236-020-00394-3.
 URL https://eprints.whiterose.ac.uk/174356/
- [81] A. Cavalcanti, J. Baxter, G. Carvalho, Roboworld: Where can my robot work?, in: R. Calinescu, C. S. Păsăreanu (Eds.), Software Engineering and Formal Methods, Springer International Publishing, Cham, 2021, pp. 3–22.
- [82] S. Foster, J. J. Huerta y Munive, G. Struth, Differential hoare logics and refinement calculi for hybrid systems with isabelle/hol, in: U. Fahrenberg, P. Jipsen, M. Winter (Eds.), Relational and Algebraic Methods in Computer Science, Springer International Publishing, Cham, 2020, pp. 169–186.
- [83] A. L. C. Cavalcanti, A. C. A. Sampaio, A. Miyazawa, P. Ribeiro, M. S. C. Filho, A. Didier, W. Li, J. Timmis, Verified simulation for robotics, Science of Computer Programming 174 (2019) 1-37. doi:10.1016/j.scico.2019.01.004. URL https://www-users.cs.york.ac.uk/%7Ealcc/publications/papers/CSMRCD19.pdf
- [84] C. Morgan, A. McIver, K. Seidel, J. W. Sanders, Refinement-oriented probability for csp, Form. Asp. Comput. 8 (6) (1996) 617-647. doi:10.1007/BF01213492. URL https://doi.org/10.1007/BF01213492
- [85] C. Morgan, Of probabilistic wp and csp—and compositionality, in: A. E. Abdallah, C. B. Jones, J. W. Sanders (Eds.), Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2005, pp. 220–241. doi: 10.1007/11423348_12. URL https://doi.org/10.1007/11423348_12
- [86] M. Núñez, D. de Frutos, L. Llana, Acceptance trees for probabilistic processes, in: I. Lee, S. A. Smolka (Eds.), CONCUR '95: Concurrency Theory, Springer Berlin Heidelberg, Berlin, Heidelberg, 1995, pp. 249–263.
- [87] F. C. Gómez, D. de Frutos Escrig, V. V. Ruiz, A sound and complete proof system for probabilistic processes, in: M. Bertran, T. Rus (Eds.), Transformation-Based Reactive Systems Development, Springer Berlin Heidelberg, Berlin, Heidelberg, 1997, pp. 340–352.
- [88] M. Kwiatkowska, G. Norman, A fully abstract metric-space denotational semantics for reactive probabilistic processes, Electronic Notes in Theoretical Computer Science 13 (1998) 182, comprox III, Third Workshop on Computation and Approximation. doi:https://doi.org/10.1016/S1571-0661(05)80222-1. URL https://www.sciencedirect.com/science/article/pii/S1571066105802221
- [89] S. Georgievska, S. Andova, Probabilistic CSP: Preserving the Laws via Restricted Schedulers, in: J. B. Schmitt (Ed.), Measurement, Modelling, and Evaluation of Computing Systems and Dependability and Fault Tolerance, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 136–150.
- [90] H. Hansson, B. Jonsson, A calculus for communicating systems with time and probabilities, in: [1990] Proceedings 11th Real-Time Systems Symposium, 1990, pp. 278–287. doi:10.1109/REAL.1990.128759.
- [91] A. Giacalone, C. Jou, S. A. Smolka, Algebraic reasoning for probabilistic concurrent systems, in: M. Broy, C. B. Jones (Eds.), Programming concepts and methods: Proceedings of the IFIP Working Group 2.2, 2.3 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel, 2-5 April, 1990, North-Holland, 1990, pp. 443–458.
- [92] W. Yi, K. G. Larsen, Testing probabilistic and nondeterministic processes, in: Proceedings of the IFIP TC6/WG6.1 Twelth International Symposium on Protocol Specification, Testing and Verification XII, North-Holland Publishing Co., NLD, 1992, p. 47–61.
- [93] R. Vanglabbeek, S. Smolka, B. Steffen, Reactive, generative, and stratified models of probabilistic processes, Information and Computation 121 (1) (1995) 59-80. doi:https://doi.org/10.1006/inco.1995.1123. URL https://www.sciencedirect.com/science/article/pii/S0890540185711236
- [94] S. Andova, Probabilistic process algebra, Ph.D. thesis, Mathematics and Computer Science (2002). doi:10.6100/ IR561343.
- [95] K. G. Larsen, A. Skou, Bisimulation through probabilistic testing, Inf. Comput. 94 (1) (1991) 1–28. doi:10.1016/ 0890-5401(91)90030-6.
- [96] B. Bloom, A. R. Meyer, A remark on bisimulation between probabilistic processes, in: A. R. Meyer, M. A. Taitslin (Eds.), Logic at Botik '89, Springer Berlin Heidelberg, Berlin, Heidelberg, 1989, pp. 26–40.
- [97] B. Jonsson, W. Yi, K. G. Larsen, Chapter 11 probabilistic extensions of process algebras**this chapter is dedicated to the fond memory of linda christoff., in: J. Bergstra, A. Ponse, S. Smolka (Eds.), Handbook of Process Algebra, Elsevier Science, Amsterdam, 2001, pp. 685-710. doi:https://doi.org/10.1016/B978-044482830-9/50029-1. URL https://www.sciencedirect.com/science/article/pii/B9780444828309500291

- [98] S. hwey Wu, S. A. Smolka, E. W. Stark, Composition and behaviors of probabilistic i/o automata, Theoretical Computer Science 176 (1) (1997) 1-38. doi:https://doi.org/10.1016/S0304-3975(97)00056-X. URL https://www.sciencedirect.com/science/article/pii/S030439759700056X
- [99] A. Hartmanns, H. Hermanns, In the quantitative automata zoo, Science of Computer Programming 112 (2015) 3-23, fundamentals of Software Engineering (selected papers of FSEN 2013). doi:https://doi.org/10.1016/j.scico.2015.08.009.
 URL https://www.sciencedirect.com/science/article/pii/S0167642315002580
- [100] K. Seidel, Probabilistic communicating processes, Theoretical Computer Science 152 (2) (1995) 219-249. doi:https://doi.org/10.1016/0304-3975(94)00286-0. URL https://www.sciencedirect.com/science/article/pii/0304397594002860
- [101] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. W. Roscoe, FDR3 A Modern Refinement Checker for CSP, in: Tools and Algorithms for the Construction and Analysis of Systems, 2014, pp. 187–201.
- [102] A. Miyazawa, A. Cavalcanti, P. Ribeiro, K. Ye, W. Li, J. Woodcock, J. Timmis, RoboChart Reference Manual, Tech. rep., University of York, www.cs.york.ac.uk/circus/publications/techreports/reports/robochart-reference.pdf (2020).
- [103] M. Z. Kwiatkowska, G. Norman, D. Parker, PRISM 4.0: Verification of probabilistic real-time systems, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 585-591. doi:10.1007/978-3-642-22110-1_{4}{7}. URL https://doi.org/10.1007/978-3-642-22110-1_47
- [104] Y. Murray, D. A. Anisi, M. Sirevåg, P. Ribeiro, R. S. Hagag, Safety assurance of a high voltage controller for an industrial robotic system, in: G. Carvalho, V. Stolz (Eds.), Formal Methods: Foundations and Applications, Springer International Publishing, Cham, 2020, pp. 45–63.
- [105] P. Cousot, R. Cousot, Constructive versions of Tarski's fixed point theorems, Pacific Journal of Mathematics 81 (1) (1979) 43–57.