An Adequate While-Language for Stochastic Hybrid Computation

Renato Neves
nevrenato@di.uminho.pt
University of Minho & INESC-TEC
Braga, Portugal

José Proença
jose.proenca@fc.up.pt
University of Porto & CISTER
Porto, Portugal

Juliana Souza
juliana.p.souza@inesctec.pt
University of Minho & INESC-TEC
Braga, Portugal

Abstract

We introduce a language for formally reasoning about programs that combine differential constructs with probabilistic ones. The language harbours, for example, such systems as adaptive cruise controllers, continuous-time random walks, and physical processes involving multiple collisions, like in Einstein's Brownian motion.

We furnish the language with an operational semantics and use it to implement a corresponding interpreter. We also present a complementary, denotational semantics and establish an adequacy theorem between both cases.

CCS Concepts

• **Theory of computation → Semantics and reasoning**.

Keywords

Theory of Programming, Program Semantics, Hybrid Computation, Probabilistic Computation

ACM Reference Format:

Renato Neves, José Proença, and Juliana Souza. 2025. An Adequate While-Language for Stochastic Hybrid Computation. In *Proceedings of The 27th International Symposium on Principles and Practice of Declarative Programming (PPDP'25).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Motivation. This paper aims at combining two lines of research in programming theory – hybrid and probabilistic programming. Both paradigms are rapidly proliferating and have numerous applications (see *e.g.* [5, 14, 17, 34]), however, despite increasing demand, their combination from a programming-oriented perspective is rarely considered.

Examples abound on why such a combination is crucial, and even extremely simple cases attest this. Let us briefly analyse one such case. The essence of hybrid programming is the idea of mixing differential constructs with classical program operations, as a way to model and analyse computational devices that closely interact with physical processes, such as movement, energy, and electromagnetism. One of the simplest tasks in this context is to move a stationary particle (for illustrative purposes one can regard it as a vehicle) from position zero to position, say, three. A simple hybrid program that implements this task is the following one.

```
p := 0 ;
v := 0 ;
p' = v, v' = 1  for x  ;
p' = v, v' = -1 for y
```

Observe the use of variable assignments which set the vehicle's position (p) and velocity (v) to zero, and note as well the presence of the sequencing operator (;). Most notably, the last two instructions are the aforementioned differential constructs which in this case describe the continuous dynamics of the vehicle at certain stages of the program's execution. Specifically `p' = v, v'= 1 for x` states that the vehicle will *accelerate at the rate of* $1^m/_{s^2}$ *for* x *seconds* while the last instruction states that it will *deccelerate at the rate of* $-1^m/_{s^2}$ *for* y *seconds*. The goal then is to 'solve' the program for x and y so that the vehicle moves and subsequently stops precisely at position three.

Now, the reader has probably noticed that the program just described is an idealised version of reality: there will be noise in the vehicle's actuators, which will cause fluctuations in the acceleration rates, and the switching time between one continuous dynamics to the other is not expected to be precisely x seconds but a value close to it. In face of this issue, it is natural to introduce uncertainty factors in the previous program and change the nature of the question *"will my vehicle be at position three at* x+y *seconds?"* to a more probabilistic one, where one asks about probabilities of reaching the desired position instead.

Remarkably the alternative approach of simply considering *discretisations* of hybrid programs combined with probabilistic constructs (*i.e.* using *purely* probabilistic programming) does not work in general – choosing suitables sizes for the discrete steps can be as hard or even harder than taking the continuous variant, particularly when so-called Zeno behaviour is present [35].

Contributions. We contribute towards a programming theory of stochastic hybrid computation – *i.e.* the combination of hybrid with probabilistic programming. Following traditions of programming theory, we first introduce a simple while-language on which to study stochastic hybrid computation. Our language extends the original while-language [36, 38] merely with the kind of differential construct just seen and with random sampling [24]. Its simplicity is intended, so that one can focus on the core essence of stochastic hybrid computation, but we will see that despite such it already covers a myriad of interesting and well-known examples.

We then furnish the language with an operational semantics, so that we can formally reason about stochastic hybrid programs. Among other things, we use the semantics to extend *Lince* – an existing interpreter of hybrid programs [14, 27] – to an interpreter of stochastic hybrid ones. We will show how this interpreter can

PPDP'25, Rende, Italy

ACM ISBN 978-1-4503-XXXX-X/2018/06
https://doi.org/XXXXXXX.XXXXXXX

be used to automatically present statistical information about the program under analysis. All examples in this paper have been animated using our extended Lince, available online at https://arcatools.org/lince.

Finally, following the basic motto in programming theory that different semantic approaches are necessary to fully understand the computational paradigm at hand, we introduce a compositional, denotational semantics for our language. In a nutshell, for a given initial state σ a program denotation $[\![\mathtt{p}]\!]$ will correspond to a *Markov process* [32] – intuitively its outputs are given in the form of a probability distribution and are time-dependent. The semantics is built in a systematic way, using basic principles of monad theory [13, 28], measure theory, and functional analysis [2, 9, 32], to which we can then recur (via the semantics) to derive different results about stochastic hybrid computation. We end our contributions with the proof of an adequacy theorem between the operational semantics and the denotational counterpart.

Related work. Whilst research on probabilistic programming is extensive (see *e.g.* [3–5, 17, 24]), work on marrying it with hybrid programming is much scarcer and mostly focussed on (deductive) verification. The two core examples in this line of research are [33] and [34]. The former presents an extension of the process algebra CSP that harbours both probabilistic and differential constructs. Among other things it furnishes this extension with a process-algebra like, transition-based semantics – which although quite interesting for verification purposes is less amenable to operational perspectives involving *e.g.* implementations. It presents moreover a corresponding proof system for reasoning about certain kinds of Hoare triples. The latter *op. cit.* [34] extends the well-known *differential dynamic logic* with probabilistic constructs. This logic is based on a Kleene-algebraic approach which while has resulted in remarkable progress w.r.t. verification, it is also known to have fundamental limitations in the context of hybrid programming, particularly in the presence of non-terminating behaviour which is frequent in this domain (see details for example in [14, 20, 21]).

Document structure. Section 2 introduces our stochastic language, its operational semantics, and corresponding interpreter. Section 3 recalls a series of measure-theoretic foundations for developing our denotational semantics – which is then presented in Section 4 together with the aforementioned adequacy theorem. Section 5 discusses future work and concludes.

We assume from the reader knowledge of probability theory [9], monads [13, 19, 28], and category theory [26]. A monad will often be denoted in the form of a Kleisli triple, *i.e.* $(T, \eta^T, (-)^{\star^T})$, and whenever no ambiguities arise we will omit the superscripts in the unit and Kleisli operations. Similarly we will sometimes denote a monad just by its functorial component T. Still about notation, we will denote respectively by inl and inr the left $X \longrightarrow X + Y$ and right $Y \longrightarrow X + Y$ injections into a coproduct. We will denote measurable spaces by the letters X, Y, Z... and whenever we need to explicitly work with the underlying σ-algebras we will use (X, Σ_X), (Y, Σ_Y), (Z, Σ_Z), and so forth.

2 The language and its operational semantics

We now introduce our stochastic hybrid language. In a nutshell, it extends the hybrid language in [14, 27] with an instruction for sampling from the uniform continuous distribution over the unit interval $[0, 1]$.

We start by postulating a finite set $\{\mathtt{x}_1, \ldots, \mathtt{x}_n\}$ of variables and a stock of *partial* functions $\mathtt{f} : \mathbb{R}^n \rightharpoonup \mathbb{R}$ that contains the usual arithmetic operations, trigonometric ones, and so forth. As usual partiality will be crucial for handling division by 0, logarithms, and square roots, among other things. We then define expressions and Boolean conditions via the following standard BNF grammars,

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{f(e,\ldots,e)}$$
$$\mathtt{b} ::= \mathtt{e <= e} \mid \mathtt{b \&\& b} \mid \mathtt{b \mid\mid b} \mid \mathtt{tt} \mid \mathtt{ff}$$

where x is a variable in the stock of variables and f is a function in the stock of partial functions. Programs are then built according to the two BNF grammars below.

$$\mathtt{a} ::= \mathtt{x_1' = e\ ,\ldots,\ x_n' = e\ for\ e} \mid \mathtt{x := e} \mid \mathtt{x := unif(0,1)}$$
$$\mathtt{p} ::= \mathtt{a} \mid \mathtt{p\ ;\ p} \mid \mathtt{if\ b\ then\ p\ else\ p} \mid \mathtt{while\ b\ do\ \{\ p\ \}}$$

The first grammar formally describes the three forms that an atomic program a can take: *viz.* a differential operation – expressing a system's continuous dynamics – that will 'run' for e time units, an assignment, and the aforementioned sampling operation. The second grammar formally describes the usual program constructs of imperative programming [36, 38].

We proceed by introducing some syntactic sugar relative to the differential operations and sampling. We will use this sugaring later on to provide further intuitions about the language.

First, observe that the language supports *wait calls* by virtue of the instruction $\mathtt{x_1' = 0\ ,\ldots,\ x_n' = 0\ for\ e}$. The latter states that the system is halted for e time units, as no variable can change during this time period. The operation will be denoted by the more suggestive notation `wait e`. Next, although we have introduced merely sampling from the uniform distribution over $[0, 1]$, it is well-known that other kinds of sampling can be encoded from it. For example, for two real numbers $\mathtt{a} \leq \mathtt{b}$ one can effectively sample from the uniform distribution over [a, b] via the sequence of instructions,

```
x := unif(0,1) ; x := (b - a) * x + a
```

For simplicity, we abbreviate such sequence to `x := unif(a,b)`. In the same spirit, it will be useful to sample from the exponential distribution with a given rate `lambda` > 0, in which case we write

```
x := unif(0,1) ; x := - ln(x)/lambda
```

and abbreviate this program to `x := exp(lambda)`. Next, in order to sample from normal distributions we resort for example to the Box-Muller method [8]. Specifically we write

```
x1 := unif(0,1) ;
x2 := unif(0,1) ;
x  := sqrt(-2 * (ln x1)) * cos(2 * pi * x2)
```

and suggestively abbreviate the program to `x := normal(0,1)`. The latter amounts to sampling from the normal distribution with mean 0 and standard deviation 1. Note that sampling from a normal distribution with mean m and standard deviation s is then given by,

```
x := normal(0,1) ; x := m + s * x
```

We abbreviate this last program to `x := normal(m,s)`. We encode Bernoulli trials in our language standardly. Specifically Bernoulli trials, denoted by `bernoulli(r,p,q)`, with `r` $\in [0,1]$ and `p`, `q` two programs, are encoded as,

```
x := unif(0,1) ; if x <= r then p else q
```

which denotes the evaluation of `p` with probability `r` and `q` with probability `1 - r`. Finally, we will also resort to the usual syntactic sugar constructs in imperative programming, *e.g.* `x := x + 1` as `x++`, and so forth.

We are ready to introduce a series of examples written in our programming language. In order to render their descriptions more lively we complement the latter with analysis information given by the aforementioned interpreter. The interpreter as well as the examples are available online at http://arcatools.org/lince.

Example 2.1 (Approximations of normal distributions via random walks). We start with a very simple case that does not involve any differential operation. Specifically we approximate the standard normal distribution via a random walk – a very common procedure in probabilistic programming [3, 23]. Note that this is different from the previous sampling operation `x := normal(0,1)`, in that it does not involve any trigonometric or logarithmic operation; furthermore the resulting distribution will always be discrete. The general idea is to write down the program below.

```
x := 0 ; c := 0 ;
while c <= n do {
      bernoulli(1/2, x++, x--) ; c++
} ;
x := x/sqrt(n)
```

Then by an appeal to the central limit theorem [9] one easily sees that the program approximates the expected normal distribution. The parameter `n` refers to the degree of precision, getting a perfect result as `n` $\longrightarrow \infty$.

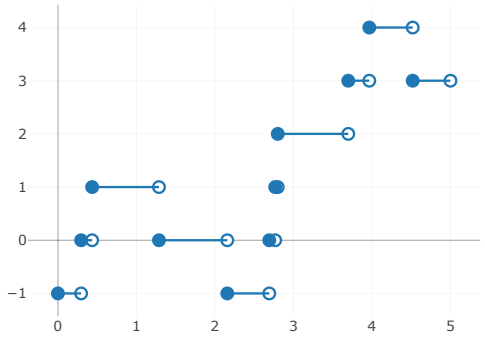

Figure 1: An execution sample of a continuous-time random walk in which the waiting time is given by sampling from the uniform distribution on $[0,1]$.

Example 2.2 (Continuous-time random walks). We now shift our focus from random walks to continuous-time ones [23] – a natural generalisation that introduces uncertainty in the number of steps a walker performs in a given time interval. These are particularly useful for studying anomalous diffusion patterns (*i.e.* the mean squared displacement), with applications ranging from biology to finance [23]. A very simple example in our language is as follows.

```
x := 0 ;
while tt {
      bernoulli(1/2, x++, x--) ;
      d := unif(0,1) ;
      wait d
}
```

The prominent feature is that the walker now waits – according to the uniform distribution on $[0,1]$ – before jumping. A helpful intuition from Nature is to think for example of a pollinating insect that jumps from one flower to another.

Whilst we do not aim at fully exploring the example here, a quick inspection tells that the average waiting time will be $\frac{1}{2}$ and thus the diffusion pattern of this stochastic process grows linearly in time. In accordance to this, Figure 1 presents an execution sample w.r.t. the first 5 time units (horizontal axis) of the process: the plot was given by our interpreter, and indeed shows 10 jumps performed during this period. Alternatively, a more complex diffusion pattern arises by setting the waiting time to be `wait(d * abs(x))`: *i.e.* it now increases in proportion to the distance from the origin, which means that the diffusion pattern w.r.t. time need not to be linear anymore.

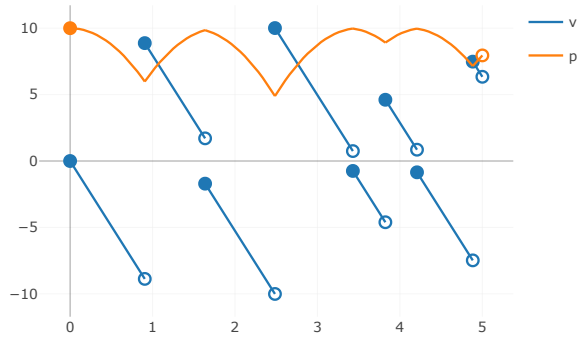

Figure 2: An execution sample of the ball's position (p) and velocity (v) during the first 5 time units.

Example 2.3 (A ball and random kicks). Let us now consider one of the classical examples in hybrid systems theory, *viz.* a bouncing ball [30, 35]. As usual we express the ball's continuous dynamics via a system of differential equations (those of motion) and jumps as assignments that 'reset' velocity. In fact we will consider a variant in which there is no ground for the ball to bounce off. Instead it will be kicked randomly, as encoded in the following program.

```
p := 10 ; v := 0 ;
while tt do {
      d := unif(0,1) ;
      p' = v, v' = -9.8 for d ;
      v := -v
}
```

In a nutshell, the ball moves according to the system of differential equations until it is kicked up (or down) as dictated by `v := -v`. The duration between jumps is random, again with a mean time of 1/2. Figure 2 presents an execution sample of the bouncing ball during the first 5 time units.

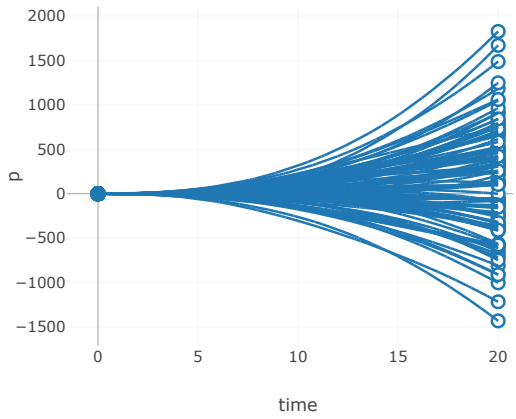

Figure 3: Multiple execution samples of the particle's position overlayed, in order to depict how the position's probability mass spreads over space w.r.t time.

Example 2.4 (Einstein's Brownian motion). Since the last example already moved us so close to it, we might as well consider Einstein's thesis for the cause of Brownian motion [11]. Essentially he posited that the (apparent) erratic motion of a particle suspended in a fluid is due to invisible collisions with atoms and molecules in the liquid. In the one-dimensional setting, one can describe a particular instance of this stochastic process as follows.

```
p := 0 ; v := 0 ; a := 0 ;
while tt do {
      d := exp(lambda) ;
      bernoulli (1/2, a--, a++) ;
      p' = v, v' = a for d
}
```

Note that random collisions are here encoded in the form a Bernoulli trial, and that their frequency is given by the Poisson process prescribed by the sampling operation. Each collision causes a bump in the acceleration (which will either be incremented or decremented). Figure 3 then presents multiple execution samples overlayed on top of each other, in order to depict of the probability mass of the particle's position spreads over space w.r.t. time.

Example 2.5 (Positioning of a particle). We now revisit the task of moving a stationary particle from position $0m$ to position $3m$, using acceleration rates $a = 1m/s^2$ and $a = -1m/s^2$. Recall that the respective program consists in accelerating (with rate $a\ m/s^2$) and then decelerating ($-a\ m/s^2$) the particle for prescribed durations x and y. Now, since a and $-a$ have the same magnitude we can safely assume that x = y. Such durations are then analytically deduced via the observation that,

$$dist = \int_0^t v_a(x)\,dx \ + \int_0^t v_{-a}(x)\,dx$$

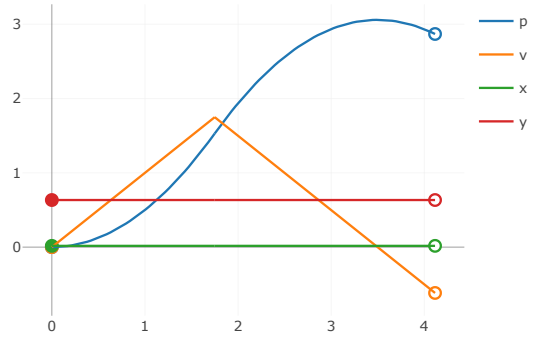

Figure 4: Execution sample of the particle's position (p) and velocity (v).

where $v_a(x) = a \cdot x$ and $v_{-a}(x) = v_a(t) + -a \cdot x$ are respectively the particle's velocity functions w.r.t. the time intervals $[0, t)$ and $[t, 2 \cdot t]$. In this context, t is the value (*i.e.* the duration x) that we wish to find out (see further details in [27]). Then observe that the value $dist$ corresponds to the area of a triangle with basis $2 \cdot t$ and height $v_a(t)$. This geometric shape yields the equations,

$$\begin{cases} dist & = \frac{1}{2} \cdot (2 \cdot t) \cdot v_a(t) \quad \textbf{(area)} \\ v_a(t) & = a \cdot t \quad \textbf{(height)} \end{cases} \qquad \Longrightarrow \qquad t = \sqrt{\frac{dist}{a}}$$

Finally note that for $dist = 3$ and $a = 1$ we obtain $t = \sqrt{3}$. Thus, ideally we would like to set the durations of both acceleration and deceleration to $\sqrt{3}$. This would then give rise to a total duration of $2\sqrt{3}$ and the particle would stop precisely at position 3. But in reality we should expect a small error in the durations of such instructions. To this effect, in the program below we add an uncertainty factor to the calculated durations x and y.

```
x := exp(2) + sqrt(3);
y := exp(2) + sqrt(3);
p' = v, v' = 1  for x ;
p' = v, v' = -1 for y
```

Figure 4 presents an execution sample where we see the effects of the small errors in the durations: specifically the program at some point exceeds position 3 and then terminates slightly below it. Note also that the expected shape of the velocity function is lost.

Example 2.6 (Adaptive cruise controller). Lastly we consider a scenario in which a particle tries to follow another one as closely as possible and without crashing into it. For illustration purposes we consider that the following particle (henceforth, the follower) starts 50 meters behind the other one (henceforth, the leader) and that it is stationary. It will be able to either accelerate or brake with forces *e.g.* $2m/s^2$ and $-2m/s^2$, respectively, during 1 time unit each time. The leader, on the other hand, starts with its velocity at $10m/s$ and cannot accelerate or brake.

The follower's choice of whether to accelerate or brake each time is determined by checking whether, in case of choosing to accelerate during one time unit, a 'safe braking distance' from the leader is maintained – a braking distance is determined 'safe' if the follower's braking trajectory does not intersect that of the leader. Technically this amounts to finding the roots of the pointwise difference of

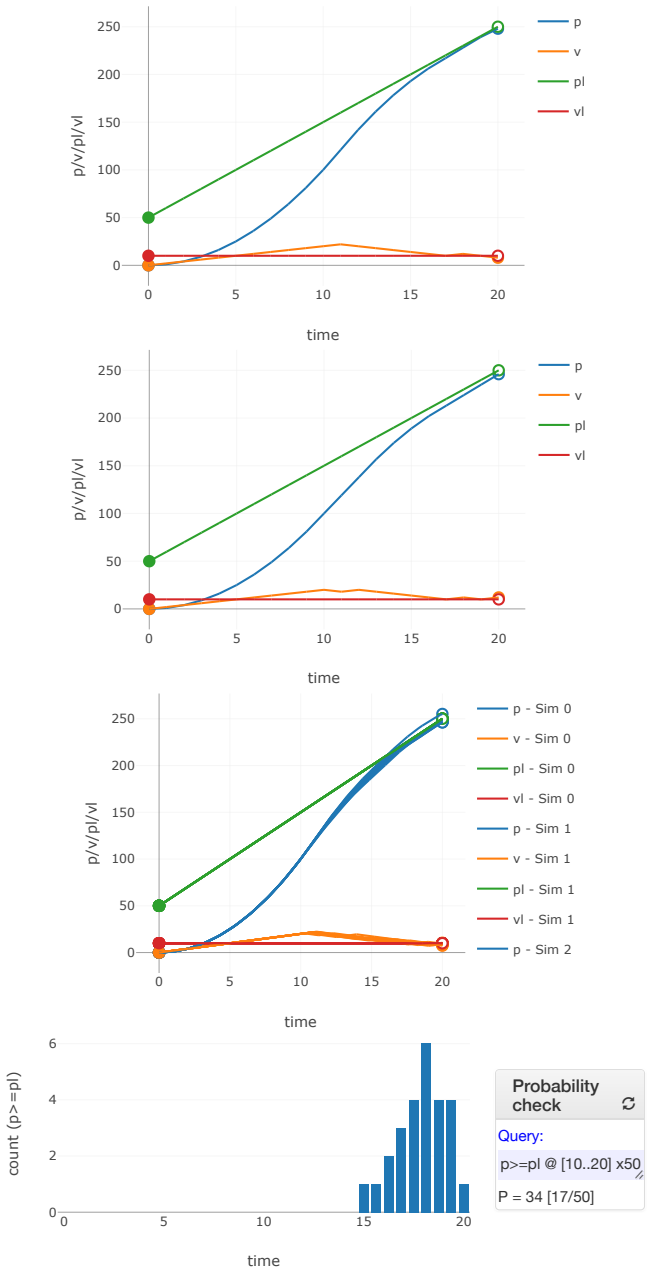

Figure 5: Multiple execution samples of different variants of an adaptive cruise controller. Labels p and v denote the follower's position and velocity while pl and vl indicate the leader's position and velocity. An histogram and a probability checker that count how many times p≥pl in 50 runs with probabilistic waiting times.

both trajectories (*i.e.* finding the roots of a quadratic equation): the absence of roots amounts to the absence of intersections (see further details in [27]). The overall idea of the scenario just described is encoded by the following program, where the operation `safe(p,v,pl,vl)` informs whether roots were found or not.

```
p := 0 ; v := 0 ; pl := 50 ; vl := 10 ;
while tt {
  if safe(p,v,pl,vl)
  then p' = v, v' =  2, pl' = vl, vl' = 0 for 1
  else p' = v, v' = -2, pl' = vl, vl' = 0 for 1
}
```

Let us now add some uncertainty to the leader: it will be able to uniformly take any acceleration in the range $[-1, 1]$. This means that for *complete safety*, the function `safe(p,v,pl,vl)` needs to be tweaked to assume the *worst possible scenario*: *i.e.* while the follower's braking trajectory will be the same, the leader's trajectory is now assumed to be the one that results from choosing acceleration $-1m/s^2$ (and not $0m/s^2$, as before). The resulting program is then as follows.

```
p := 0 ; v := 0 ; pl := 50 ; vl := 10 ; a := 0;
while tt {
  a := unif(-1,1) ;
  if safe(p,v,pl,vl)
  then p' = v, v' =  2, pl' = vl, vl' = a for 1
  else p' = v, v' = -2, pl' = vl, vl' = a for 1
}
```

Yet another option for introducing uncertainty is to consider the fact that the waiting times will be given by an exponential distribution, as follows.

```
p := 0 ; v := 0 ; pl := 50 ; vl := 10 ;
while tt {
  x := exp(lambda) ; x++ ;
  if safe(p,v,pl,vl)
  then p' = v, v' =  2, pl' = vl, vl' = 0 for x
  else p' = v, v' = -2, pl' = vl, vl' = 0 for x
}
```

The function `safe(p,v,pl,vl)` would then need to be tweaked again – but remarkably now with no hope for complete safety, as in theory `x` can take any value from $[1, \infty)$ and thus no worst-case scenario exists.

Figure 5 (top) presents an execution sample in which the system is completely deterministic (*i.e.* the leader's velocity is constant with 100% certainty) and thus the follower gets as close as possible to the leader. On the other hand, Figure 5 (middle) presents an execution sample in which the follower assumes the worst-case scenario just described and thus cannot get as close to the leader. Finally Figure 5 (bottom plot) presents several execution samples overlayed in which the original `safe` function is used and the respective durations are given by the exponential distribution `1 + exp(8)`. It shows that, while collisions are improbable they do occur. This low probability of collision is quantified at the bottom of Figure 5, counting how many times `p >= pl` holds in 50 runs over time (bottom-left) and anywhere in the interval [10,20] (bottom-right).

Operational semantics. The section's remainder is devoted to introducing an operational semantics for the language – not only such is a basis for formal reasoning about stochastic hybrid programs it is also the engine of the interpreter that we have been showcasing thus far. In a nutshell, the semantics marries Kozen's operational semantics for a probabilistic language [24] with the

semantics of hybrid programs that was presented in [14, 27]. We will need some preliminaries.

We take the Hilbert cube $[0,1]^\omega$ as the source of randomness. Operationally speaking this means that sampling will amount to drawing values from an element of $[0,1]^\omega$ (a stream) that is fixed *a priori*. For example, sampling once will amount to taking the head of this element and sampling n times will amount to taking the respective prefix of size n. Next we assume that the semantics of expressions e and Boolean conditions b are given by partial maps $[\![\mathsf{e}]\!] : \mathbb{R}^n \rightharpoonup \mathbb{R}$ and $[\![\mathsf{b}]\!] : \mathbb{R}^n \rightharpoonup \{\mathtt{tt}, \mathtt{ff}\}$. These can be defined in the usual way. Now, since we are in the context of imperative programming we will recur to the notion of a store $\sigma : \{\mathsf{x}_1, \ldots, \mathsf{x}_n\} \to \mathbb{R}$ (also known as memory or environment) [36, 38]. It assigns a real number to any given variable in the language. For a store σ, we will use the notation $\sigma[\mathsf{x} \mapsto v]$ to denote the store that is exactly like σ except for the fact that x is now assigned value v. Finally we assume that any system of differential equations in our language induces a partial map $\phi : \mathbb{R}^n \times \mathbb{R}_{\geq 0} \rightharpoonup \mathbb{R}^n$ – which in our context will be regarded as the respective (partial) solution. For simplicity we denote an operation $\mathsf{x}_1\mathtt{'} = \mathsf{e}_1, \ldots, \mathsf{x}_n\mathtt{'} = \mathsf{e}_n$ `for` e by the simpler expression $\mathtt{diff}(\mathsf{e}_1, \ldots, \mathsf{e}_n, \mathsf{e})$.

The rules of our semantics are then presented in Figure 6. They dictate what the next computational step will be when evaluating a program p with initial store σ w.r.t. time instant t. As alluded before each evaluation is associated with a source of randomness $s \in [0,1]^\omega$ from which p draws sampling results. Note from the rules that such computational steps lead to one of three possible outcomes: *viz.* an error flag *err*, an output store, or a resumption (*i.e.* an updated evaluation stack of programs, store, time instant, and source of randomness) which can then be evaluated in the next step (the empty stack is denoted by *skip*).

The subtlest feature of our semantics is that, not only will the evaluation stack of programs tend to decrease (with the exception of constructs such as while loops, which may temporarily increase it), also the time instant t at the beginning of the evaluation will tend to decrease along the computational steps performed. Intuitively this means that the evaluation is 'moving forward in time' until reaching the target time instant t that we wish to evaluate. Most notably, when it detects that such time instant was reached it forces the termination of the evaluation, even if the evaluation stack of programs is currently non-empty. Such is expressed by the rule **(seq-stop$^{\rightarrow}$)**, and is crucial for evaluating non-terminating programs, like the ones described in Example 2.2, Example 2.3, Example 2.4, and Example 2.6. We illustrate this feature next with a simple example, but more details can also be found in [14].

Example 2.7. Consider the following non-terminating program,

```
x := 0 ; while tt { x++ ; wait 1 }
```

Although the loop involved does not terminate, one can always evaluate the program in a finite amount of steps for any given time instant. Let us see what happens, for example, at time instant $1 + 1/2$. First, for simplicity we denote the loop simply by p and the store $\sigma : \{\mathsf{x}\} \to \mathbb{R}$ that is defined as $\sigma(\mathsf{x}) = v$ by $\mathsf{x} \mapsto v$. We then deduce the following sequence of small-step transitions which arise from the rules in Figure 6. The last transition arises precisely due to rules **(diff-stop$^{\rightarrow}$)** and **(seq-stop$^{\rightarrow}$)**. Henceforth we will call events such as the one just described *time-based terminations*, in order to distinguish from those ordinary terminations that originate in emptying the program evaluation stack. We will see later on that this subtle aspect can be neatly handled in the denotational context via an exception monad.

$$\begin{aligned}
\mathtt{x:=\ 0\ ;\ p},\ \sigma,\ 1+1/2,\ s &\to \mathsf{p},\ (\mathsf{x}\mapsto 0),\ 1+1/2,\ s\\
&\to \mathsf{x{+}{+}\ ;\ wait\ 1\ ;\ p},\ (\mathsf{x}\mapsto 0),\ 1+1/2,\ s\\
&\to \mathtt{wait\ 1\ ;\ p},\ (\mathsf{x}\mapsto 1),\ 1+1/2,\ s\\
&\to \mathsf{p},\ (\mathsf{x}\mapsto 1),\ 1/2,\ s\\
&\to \mathsf{x{+}{+}\ ;\ wait\ 1\ ;\ p},\ (\mathsf{x}\mapsto 1),\ 1/2,\ s\\
&\to \mathtt{wait\ 1\ ;\ p},\ (\mathsf{x}\mapsto 2),\ 1/2,\ s\\
&\to \mathsf{x}\mapsto 2
\end{aligned}$$

Let us briefly mention how our interpreter uses this semantics to provide (overlayed) execution samples of a given stochastic hybrid program p. The basic idea is simple: we first generate an entropy source *i.e.* a sample $s \in [0,1]^\omega$ and then use it to compute the execution chain $\mathsf{p}, \sigma, t, s \to \ldots$ for multiple time instants t, corresponding to different snapshots of the program's behavioural trajectory. In order to obtain overlayed execution samples one just repeats this process multiple times, *i.e.* with different samples.

We now introduce a big-step operational semantics in Figure 7, which abstracts from intermediate computational steps in the context of the small-step variant. Although in programming theory big-step semantics have multiple applications [36, 38], here we use it to connect small-step to the denotational counterpart in Section 4. In other words, the big-step variant is a midpoint between small-step and denotational semantics. Since this big-step semantics is based on the same ideas as small-step, we skip its explanation.

We conclude the section by showing that the small-step and big-step semantics agree, in the sense that they give rise to the same input-output relation. Technically, we factor in the reflexive-transitive closure of the small-step relation as follows. First, we call 'terminal' those tuples arising from steps (in the small-step semantics) that are of the form $skip, \sigma, t, s$, or σ, or *err*. Then we build an 'input-output relation' ($\Rightarrow$) via the reflexive-transitive closure of the small-step relation, as detailed in Figure 8. Finally,

Theorem 2.8. *For every program* p*, store* σ*, time instant* t*, and source of randomness* s*, we have the following following equivalence:*

$$\mathsf{p}, \sigma, t, s \Downarrow v \quad \textit{iff} \quad \mathsf{p}, \sigma, t, s \Rightarrow v$$

Proof. The right-to-left direction follows by induction on the length of small-step reduction sequence and Lemma 2.9. The left-to-right direction follows by induction over big-step derivations. □

Lemma 2.9. *Given a program* p*, a store* σ*, time instant* t*, and a source of randomness* s*, the following is true:*

1. *if* $\mathsf{p}, \sigma, t, s \to \mathsf{p}', \sigma', t', s'$ *and* $\mathsf{p}', \sigma', t', s' \Downarrow skip, \sigma'', t'', s''$ *then we have* $\mathsf{p}, \sigma, t, s \Downarrow skip, \sigma'', t'', s''$*;*
2. *if* $\mathsf{p}, \sigma, t, s \to \mathsf{p}', \sigma', t', s'$ *and* $\mathsf{p}', \sigma', t', s' \Downarrow \sigma''$ *then we have* $\mathsf{p}, \sigma, t, s \Downarrow \sigma''$*;*
3. *if* $\mathsf{p}, \sigma, t, s \to \mathsf{p}', \sigma', t', s'$ *and* $\mathsf{p}', \sigma', t', s' \Downarrow err$ *then we have* $\mathsf{p}, \sigma, t, s \Downarrow err$*;*

Proof. Follows straightforwardly by induction over the rules concerning the small-step semantics. □

$$\textbf{(asg-rnd}^{\rightarrow}\textbf{)}\quad \mathtt{x := unif(0,1)}, \sigma, t, (h:s) \rightarrow \mathit{skip}, \sigma[\mathtt{x} \mapsto h], t, s$$

$$\textbf{(asg}^{\rightarrow}\textbf{)}\quad \mathtt{x := e}, \sigma, t, s \rightarrow \mathit{skip}, \sigma[\mathtt{x} \mapsto [\![\mathtt{e}]\!](\sigma)], t, s \qquad ([\![\mathtt{e}]\!](\sigma) \text{ defined})$$

$$\textbf{(asg-err}^{\rightarrow}\textbf{)}\quad \mathtt{x := e}, \sigma, t, s \rightarrow \mathit{err} \qquad ([\![\mathtt{e}]\!](\sigma) \text{ undefined})$$

$$\textbf{(diff-stop}^{\rightarrow}\textbf{)}\quad \mathtt{diff(e_1,\ldots,e_n,e)}, \sigma, t, s \rightarrow \phi(\sigma, t) \qquad ([\![\mathtt{e}]\!](\sigma) > t)$$

$$\textbf{(diff-skip}^{\rightarrow}\textbf{)}\quad \mathtt{diff(e_1,\ldots,e_n,e)}, \sigma, t, s \rightarrow \mathit{skip}, \phi(\sigma, [\![\mathtt{e}]\!](\sigma)), t - [\![\mathtt{e}]\!](\sigma), s \qquad (0 \leq [\![\mathtt{e}]\!](\sigma) \leq t)$$

$$\textbf{(diff-err}^{\rightarrow}\textbf{)}\quad \mathtt{diff(e_1,\ldots,e_n,e)}, \sigma, t, s \rightarrow \mathit{err} \qquad ([\![\mathtt{e}]\!](\sigma) < 0 \text{ or } [\![\mathtt{e}]\!](\sigma) \text{ undefined})$$

$$\textbf{(if-true}^{\rightarrow}\textbf{)}\quad \mathtt{if\ b\ then\ p\ else\ q}, \sigma, t, s \rightarrow \mathtt{p}, \sigma, t, s \qquad ([\![\mathtt{b}]\!](\sigma) = \mathtt{tt})$$

$$\textbf{(if-false}^{\rightarrow}\textbf{)}\quad \mathtt{if\ b\ then\ p\ else\ q}, \sigma, t, s \rightarrow \mathtt{q}, \sigma, t, s \qquad ([\![\mathtt{q}]\!](\sigma) = \mathtt{ff})$$

$$\textbf{(if-err}^{\rightarrow}\textbf{)}\quad \mathtt{if\ b\ then\ p\ else\ q}, \sigma, t, s \rightarrow \mathit{err} \qquad ([\![\mathtt{b}]\!](\sigma) \text{ undefined})$$

$$\textbf{(wh-true}^{\rightarrow}\textbf{)}\quad \mathtt{while\ b\ do\ \{\,p\,\}}, \sigma, t, s \rightarrow \mathtt{p\ ;\ while\ b\ do\ \{\,p\,\}}, \sigma, t, s \qquad ([\![\mathtt{b}]\!](\sigma) = \mathtt{tt})$$

$$\textbf{(wh-false}^{\rightarrow}\textbf{)}\quad \mathtt{while\ b\ do\ \{\,p\,\}}, \sigma, t, s \rightarrow \mathit{skip}, \sigma, t, s \qquad ([\![\mathtt{b}]\!](\sigma) = \mathtt{ff})$$

$$\textbf{(wh-err}^{\rightarrow}\textbf{)}\quad \mathtt{while\ b\ do\ \{\,p\,\}}, \sigma, t, s \rightarrow \mathit{err} \qquad ([\![\mathtt{b}]\!](\sigma) \text{ undefined})$$

$$\textbf{(seq-stop}^{\rightarrow}\textbf{)}\quad \frac{\mathtt{p}, \sigma, t, s \rightarrow \sigma'}{\mathtt{p\ ;\ q}, \sigma, t, s \rightarrow \sigma'} \qquad \textbf{(seq-skip}^{\rightarrow}\textbf{)}\quad \frac{\mathtt{p}, \sigma, t, s \rightarrow \mathit{skip}, \sigma', t', s'}{\mathtt{p\ ;\ q}, \sigma, t, s \rightarrow \mathtt{q}, \sigma', t', s'}$$

$$\textbf{(seq-err}^{\rightarrow}\textbf{)}\quad \frac{\mathtt{p}, \sigma, t, s \rightarrow \mathit{err}}{\mathtt{p\ ;\ q}, \sigma, t, s \rightarrow \mathit{err}} \qquad \textbf{(seq}^{\rightarrow}\textbf{)}\quad \frac{\mathtt{p}, \sigma, t, s \rightarrow \mathtt{p}', \sigma', t', s'}{\mathtt{p\ ;\ q}, \sigma, t, s \rightarrow \mathtt{p}'\ \mathtt{;\ q}, \sigma', t', s'}$$

Figure 6: Small-step operational semantics

3 Measure theory

This section briefly recalls a series of results about measure theory [2, 10, 32], focus being on those that form the backbone of the semantics described in Section 4.

Our main working category will be Meas, *i.e.* that of measurable spaces and measurable functions. Recall that it has both (infinite) products and coproducts [1, Section 21]. Recall as well that it is distributive, *i.e.* for all measurable spaces X, Y, Z there exists an isomorphism,

$$\mathrm{dist} : X \times (Y + Z) \to X \times Y + X \times Z$$

Now, let Top be the category of topological spaces and continuous maps, and recall that it has (infinite) products and coproducts as well [1, Section 21]. There exists a functor $B : \mathrm{Top} \to \mathrm{Meas}$ that sends any given topological space to the measurable space with the same carrier and equipped with the respective Borel σ-algebra [2, Section 4.4]. In particular when treating a subset of real numbers as a measurable space we will be tacitly referring to the respective Borel σ-algebra. It is well-known that B preserves finite products of second-countable topological spaces [16, Definition 6.3.7], which is the case for example of Polish spaces (see [2, Chapter 3] and [16, Theorem 6.3.44]). This property is key for our semantics: it will allow us to treat solutions of systems of differential equations – which are continuous functions and thus live in Top – as measurable functions. Further details about this crucial aspect are available in the following section.

We proceed by briefly recalling basic results about measures – a more detailed description is available for example in [2, Chapter 10], [3, Chapter 1], and [32, Chapter 2].

Definition 3.1. For a measurable space (X, Σ_X) a measure is a function $\mu : \Sigma_X \to \mathbb{R}$ such that $\mu(U) \geq 0$ for all measurable sets U, $\mu(\emptyset) = 0$ and moreover it is σ-additive, *i.e.*

$$\mu\left(\bigcup_{i=1}^{\infty} U_i\right) = \sum_{i=1}^{\infty} \mu(U_i)$$

where $(U_i)_{i \in \omega}$ is any family of pairwise disjoint measurable sets. As usual, a measure is called a subdistribution if $\mu(X) \leq 1$ and a distribution if $\mu(X) = 1$.

For a measurable space X the set of measures $\mathrm{M}(X)$ forms a vector space via pointwise extension. It also forms a normed space when equipped with the total variation norm,

$$\|\mu\| = \sup\left\{\sum_{i=1}^{n} \|\mu(U_i)\| \mid \{U_1, \ldots, U_n\} \text{ measurable partition of X}\right\}$$

In particular, for a positive measure μ we have $\|\mu\| = \mu(X)$. Note that $\mathrm{M}(X)$ is also a Banach space by virtue of the reals numbers forming a Banach space, specifically the limit of a Cauchy sequence is built via pointwise extension.

$$\textbf{(asg-rnd)}\quad \frac{}{\texttt{x := unif(0,1)}, \sigma, t, (h : s) \Downarrow \mathit{skip}, \sigma[\texttt{x} \mapsto h], t, s}$$

$$\textbf{(asg-skip)}\quad \frac{[\![\texttt{e}]\!](\sigma) \text{ defined}}{\texttt{x := e}, \sigma, t, s \Downarrow \mathit{skip}, \sigma[\texttt{x} \mapsto [\![\texttt{e}]\!](\sigma)], t, s} \qquad \textbf{(asg-err)}\quad \frac{[\![\texttt{e}]\!](\sigma) \text{ undefined}}{\texttt{x := e}, \sigma, t, s \Downarrow \mathit{err}}$$

$$\textbf{(diff-skip)}\quad \frac{0 \le [\![\texttt{e}]\!](\sigma) \le t}{\texttt{diff}(\texttt{e}_1, \ldots, \texttt{e}_n, , \texttt{e}), \sigma, t, s \Downarrow \mathit{skip}, \phi(\sigma, [\![\texttt{e}]\!](\sigma)), t - [\![\texttt{e}]\!](\sigma), s}$$

$$\textbf{(diff-stop)}\quad \frac{[\![\texttt{e}]\!](\sigma) > t}{\texttt{diff}(\texttt{e}_1, \ldots, \texttt{e}_n, , \texttt{e}), \sigma, t, s \Downarrow \phi(\sigma, t)} \qquad \textbf{(diff-err)}\quad \frac{[\![\texttt{e}]\!](\sigma) < 0 \text{ or } [\![\texttt{e}]\!](\sigma) \text{ undefined}}{\texttt{diff}(\texttt{e}_1, \ldots, \texttt{e}_n, , \texttt{e}) \Downarrow \mathit{err}}$$

$$\textbf{(seq-skip)}\quad \frac{\texttt{p}, \sigma, t, s \Downarrow \mathit{skip}, \sigma', t', s' \qquad \texttt{q}, \sigma', t', s' \Downarrow v}{\texttt{p ; q}, \sigma, t, s \Downarrow v}$$

$$\textbf{(seq-stop)}\quad \frac{\texttt{p}, \sigma, t, s \Downarrow \sigma'}{\texttt{p ; q}, \sigma, t, s \Downarrow \sigma'} \qquad \textbf{(seq-err)}\quad \frac{\texttt{p}, \sigma, t, s \Downarrow \mathit{err}}{\texttt{p ; q}, \sigma, t, s \Downarrow \mathit{err}}$$

$$\textbf{(if-true)}\quad \frac{[\![\texttt{b}]\!](\sigma) = \texttt{tt} \qquad \texttt{p}, \sigma, t, s \Downarrow v}{\texttt{if b then p else q}, \sigma, t, s \Downarrow v}$$

$$\textbf{(if-false)}\quad \frac{[\![\texttt{b}]\!](\sigma) = \texttt{ff} \qquad \texttt{q}, \sigma, t, s \Downarrow v}{\texttt{if b then p else q}, \sigma, t, s \Downarrow v} \qquad \textbf{(if-err)}\quad \frac{[\![\texttt{b}]\!](\sigma) \text{ undefined}}{\texttt{if b then p else q}, \sigma, t, s \Downarrow \mathit{err}}$$

$$\textbf{(wh-true)}\quad \frac{[\![\texttt{b}]\!](\sigma) = \texttt{tt} \qquad \texttt{p ; while b do \{ p \}}, \sigma, t, s \Downarrow v}{\texttt{while b do \{ p \}}, \sigma, t, s \Downarrow v}$$

$$\textbf{(wh-false)}\quad \frac{[\![\texttt{b}]\!](\sigma) = \texttt{ff}}{\texttt{while b do \{ p \}}, \sigma, t, s \Downarrow \mathit{skip}, \sigma, t, s} \qquad \textbf{(wh-err)}\quad \frac{[\![\texttt{b}]\!](\sigma) \text{ undefined}}{\texttt{while b do \{ p \}}, \sigma, t, s \Downarrow \mathit{err}}$$

Figure 7: Big-step operational semantics

$$\frac{\texttt{p}, \sigma, t, s \to v}{\texttt{p}, \sigma, t, s \Rightarrow v}\ (v \text{ terminal}) \qquad \frac{\texttt{p}, \sigma, t, s \to \texttt{p}', \sigma', t', s' \qquad \texttt{p}', \sigma', t', s' \Rightarrow v}{\texttt{p}, \sigma, t, s \Rightarrow v}$$

Figure 8: Big-step semantics via the reflexive-transitive closure of the small-step relation

Take a measure $\mu \in \mathrm{M}(X)$ and a measure $\nu \in \mathrm{M}(Y)$. Then there exists the so-called *tensor* or *product measure* $\mu \otimes \nu \in \mathrm{M}(X \times Y)$, which is defined by the equation $\mu \otimes \nu(U \times V) = \mu(U)\nu(V)$ on all measurable rectangles $U \times V \in \Sigma_{X \times Y}$. Specifically the latter extends standardly to all measurable sets by an appeal to Carathéodory's extension, and moreover the extension is unique if μ and ν are subdistributions (see *e.g.* [2, Lemma 10.33]). Another useful fact is that for any subdistributions $\mu, \nu \in \mathrm{M}(X)$ and $\rho \in \mathrm{M}(Y)$ the equation below holds.

$$(\mu + \nu) \otimes \rho = \mu \otimes \rho + \nu \otimes \rho$$

The product measure construction just described also applies to countable families of distributions $(\mu_i)_{i \in \omega}$ in $\mathrm{M}(X)$.

Next, for a measurable space X we will denote by $\mathrm{G}(X)$ the set of subdistributions. The construct $\mathrm{G}(-)$ thus defined forms the Giry monad in Meas when every $\mathrm{G}(X)$ is equipped with the σ-algebra generated by the evaluation maps,

$$\mathrm{eval}_U : \mathrm{G}(X) \to \mathbb{R} \qquad \mu \mapsto \mu(U) \qquad (U \subseteq X \text{ measurable})$$

[31]. This last clause is equivalent to stating that for any map $f : X \to \mathrm{G}(Y)$ between measurable spaces, if $\mathrm{eval}_U \cdot f$ is measurable for all measurable subsets U, then f will be measurable as well. The respective Kleisli morphisms $f : X \to \mathrm{G}(Y)$ are typically called Markov kernels and their Kleisli extension $f^\star : \mathrm{G}(X) \to \mathrm{G}(Y)$ is given by Lebesgue integration [2],

$$f^\star(\mu) = U \mapsto \int_{x \in X} f(x)(U)\, d\mu(x)$$

For every measurable space X the unit $\delta : X \to \mathrm{G}(X)$ of this monad is given by the Dirac delta $\delta_x \in \mathrm{G}(X)$ with $x \in X$, *i.e.*

$$\delta_x(U) = \begin{cases} 1 & \text{if } x \in U \\ 0 & \text{otherwise} \end{cases}$$

In particular, the functorial action of $\mathrm{G}(-) : \mathrm{Meas} \to \mathrm{Meas}$ is the pushforward measure operation. We will often abuse notation by denoting a linear combination $\sum_i p_i \cdot \delta_{x_i}$ simply by $\sum_i p_i \cdot x_i$ with $x_i \in X$. The symbol "$-$" in the notation $\mathrm{G}(-)$ denotes a placeholder for the functor in abstraction, *i.e.*, before specifying its argument.

Let us recall useful properties about the Giry monad. First it is commutative when equipped with the double-strength operation $\mathrm{G}(X) \times \mathrm{G}(Y) \to \mathrm{G}(X \times Y)$ defined via the product measure [37]. The fact that such operation is measurable follows from [2, Lemma 4.11]. Second for any bounded measurable map $f : X \to \mathbb{R}$ and measures $\mu, \nu \in \mathrm{G}(X)$ Lebesgue integration satisfies the conditions,

$$\int f d(\mu + \nu) = \int f d\mu + \int g d\nu \qquad \int f d(s \cdot \mu) = s \cdot \left(\int f d\mu \right)$$

for any scalar $s \in \mathbb{R}$. Thus we immediately conclude that the Kleisli extension of a Markov kernel will always be linear. Third if the codomain of f restricts to $[0, 1]$ we obtain,

$$\int f d\mu \leq \mu(X) = \|\mu\|$$

This entails that the Kleisli extension of a Markov kernel will be bounded and thus continuous (even contractive) w.r.t. the metric induced by the total variation norm. This provides a number of tools from functional analysis. For example one can analyse how $f^\star$ acts on a measure μ by a series of approximations μ_n to μ. Not only this, the set of maps $\mathsf{Meas}(\mathrm{G}(X), \mathrm{G}(Y))$ can be equipped with the metric induced by the operator norm,

$$\|T\| = \sup \{ \|T(\mu)\| \mid \mu \in \mathrm{G}(X), \|\mu\| \leq 1 \}$$

which moves us beyond classical program equivalence by allowing to compare programs in terms of distances and not just equality (see for example [7]). More details about this last aspect will be given later on.

Another useful fact is that for a given measure $\mu \in \mathrm{G}(X)$ any measurable subset $U \subseteq X$ gives rise to a new measure $\mu(U \cap -)$. Moreover for any bounded measurable map $f : X \to \mathbb{R}$ and measure $\mu \in \mathrm{G}(X)$ we obtain,

$$\int_X f d\mu(X \cap -) = \int_U f d\mu(U \cap -) + \int_{\overline{U}} f d\mu(\overline{U} \cap -)$$

where $\overline{U}$ represents the complement of U. Another useful property of the Giry monad is that for every measurable space X the space $\mathrm{G}(X)$ inherits the usual order on the real numbers, via pointwise extension. What is more, the induced order has a bottom element (the zero-mass measure) and it is ω-complete, by virtue of the completeness property of the real numbers. Remarkably, an ω-increasing sequence of measures $(\mu_n)_{n \in \omega}$ in $\mathrm{G}(X)$ is Cauchy and $\sup_{n \in \omega} \mu_n = \lim_{n \to \infty} \mu_n$, thanks to the monotone convergence theorem (see *e.g.* [2, Theorem 11.18] or [32, Theorem 3.6]). This is helpful to jump between domain theory and functional analysis whenever necessary.

Next, observe that the aforementioned order extends to Markov kernels via pointwise extension. It has a bottom element (the map constant on the zero-mass measure) and it is ω-complete. The last property follows directly from the definition of the σ-algebra of $\mathrm{G}(X)$ for every X and from the fact that the pointwise supremum of real-valued measurable functions is measurable. Also, it follows from the monotone convergence theorem that the equation,

$$(\sup_{i \in \omega} f_i)^\star = \sup_{i \in \omega} f_i^\star$$

holds for any increasing sequence $(f_i)_{i \in \omega}$ of Markov kernels. This will be crucial for the interpretation of while-loops in the following section. Finally it follows from the fact that multiplication preserves suprema that,

$$(\sup_{i \in \omega} \mu_i) \otimes \nu = \sup_{i \in \omega} \mu_i \otimes \nu$$

for any increasing sequence of subdistributions $(\mu_i)_{i \in \omega}$ in $\mathrm{G}(X)$ and $\nu \in \mathrm{G}(Y)$.

4 Denotational semantics

We now introduce a denotational, measure-theoretic semantics for our stochastic language. In a nutshell, it extends Kozen's well-known probabilistic semantics [24] with a mechanism for handling the time-based terminations that were described in Section 2. The extension boils down to the following categorical construction.

Any object E in a category C with binary coproducts induces a monad $E + (-)$ which intuitively gives semantics to exception handling [29]. It follows from the universal property of coproducts that any monad T in C combines with $E + (-)$. In other words we have a new monad $T \otimes E$ which handles at the same time effects arising from T and exceptions. Concretely, the functorial action of this new monad is given by $T(E + (-))$, the unit $\eta^{T \otimes E}$ by the composition $\eta^T \cdot \mathrm{inr} : X \to T(E + X)$, and the Kleisli lifting $(-)^{\star_{T \otimes E}}$ by the equation,

$$f^{\star_{T \otimes E}} = [\eta^T \cdot \mathrm{inl}, f]^{\star_T}$$

Time-based terminations will be handled precisely via one such monad $\mathrm{G} \otimes E$ in Meas – in other words these terminations are technically seen as exceptions, in the sense that they also inhibit the execution of subsequent computations and are merely propagated forward along the evaluation. Specifically the denotation $[\![\mathtt{p}]\!]$ of a program $\mathtt{p}$ will be a Markov kernel $X \to \mathrm{G}(E + X)$ in which elements of E denote time-based terminations. The space X will be in particular the product $\mathbb{R}^n \times \mathbb{R}_{\geq 0}$ whilst E will be $\mathbb{R}^n$ (thus analogously to Section 2, n is the cardinality of our stock of variables and possible outputs are either elements of $\mathbb{R}^n \times \mathbb{R}_{\geq 0}$ or $\mathbb{R}^n$). Consequently, for every $(\sigma, t) \in \mathbb{R}^n \times \mathbb{R}_{\geq 0}$ we have $[\![\mathtt{p}]\!](\sigma, t)$ as a subdistribution which assigns probabilities to the outputs of $\mathtt{p}$ w.r.t. time instant t and initial state σ. For any given initial state $\sigma \in \mathbb{R}^n$, one can also see a denotation $[\![\mathtt{p}]\!]$ as inducing a Markov process $[\![\mathtt{p}]\!](\sigma, -)$ which intuitively means that the subdistribution of outputs evolves over time. Note as well that the possibility of the total mass of $[\![\mathtt{p}]\!](\sigma, t)$ being strictly lower than 1 for a given input $(\sigma, t) \in \mathbb{R}^n \times \mathbb{R}_{\geq 0}$ reflects the possibility of divergence and/or errors in the evaluation of expressions and Boolean conditions.

Lastly in order to interpret while-loops, we observe that the combined monad $\mathrm{G} \otimes E$ inherits the order of G, and moreover,

$$(\sup_{i \in \omega} f_i)^{\star_{\mathrm{G} \otimes E}} = \sup_{i \in \omega} f_i^{\star_{\mathrm{G} \otimes E}} \tag{1}$$

for any increasing sequence $(f_i)_{i \in \omega}$ of Markov kernels. This last equation follows from the Scott-continuity of co-pairing on its second argument.

We are finally ready to introduce our denotational semantics. It is defined in Figure 9, via induction on the syntactic structure of programs. It assumes, as usual, that the semantics of expressions $\mathtt{e}$ and Boolean conditions $\mathtt{b}$ are given by *measurable* partial maps $[\![\mathtt{e}]\!] : \mathbb{R}^n \rightharpoonup \mathbb{R}$ and $[\![\mathtt{b}]\!] : \mathbb{R}^n \rightharpoonup \{\mathtt{tt}, \mathtt{ff}\}$. The measurability of each interpretation clause in Figure 9 is then straightforward to verify. Indeed, the only somewhat complicated case is the first clause,

$$[\![\texttt{diff}(\texttt{e}_1,\ldots,\texttt{e}_n,\texttt{e})]\!] = (\sigma,t) \mapsto \begin{cases} 1\cdot\phi(\sigma,t) & \text{if } [\![\texttt{e}]\!](\sigma) > t \\ 1\cdot(\phi(\sigma,[\![\texttt{e}]\!](\sigma)),t-[\![\texttt{e}]\!](\sigma)) & \text{if } 0 \le [\![\texttt{e}]\!](\sigma) \le t \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\texttt{x} := \texttt{e}]\!] = (\sigma,t) \mapsto \begin{cases} 1\cdot(\sigma[\texttt{x} \mapsto [\![\texttt{e}]\!](\sigma)],t) & \text{if } [\![\texttt{e}]\!](\sigma) \text{ is well-defined} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\texttt{x}_i := \texttt{unif}(\texttt{0},\texttt{1})]\!] = (\sigma,t) \mapsto \sigma[\texttt{x}_i \mapsto \lambda] \otimes (1\cdot t)$$

$$[\![\texttt{if b then p else q}]\!] = (\sigma,t) \mapsto \begin{cases} [\![\texttt{p}]\!](\sigma,t) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{tt} \\ [\![\texttt{q}]\!](\sigma,t) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{ff} \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\texttt{p ; q}]\!] = [\![\texttt{q}]\!]^\star \cdot [\![\texttt{p}]\!]$$

$$[\![\texttt{while b do p}]\!] = \mathrm{lfp}\left(k \mapsto (\sigma,t) \mapsto \begin{cases} k^\star\cdot[\![\texttt{p}]\!](\sigma,t) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{tt} \\ 1\cdot(\sigma,t) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{ff} \\ 0 & \text{otherwise} \end{cases}\right)$$

Figure 9: Denotational semantics

which crucially relies on two related properties. First, the fact that every continuous map $\phi : \mathbb{R}^n \times \mathbb{R}_{\ge 0} \to \mathbb{R}^n$ is measurable as a map $B(\phi) : B(\mathbb{R})^n \times B(\mathbb{R}_{\ge 0}) \to B(\mathbb{R})^n$ (which we commented on in the last section). Second, by an analogous reasoning, the fact that the subtraction $\mathbb{R} \times_{\mathsf{Meas}} \mathbb{R} \to \mathbb{R}$ map is measurable. This entails in particular that the strictly greater relation $(>)$ is measurable as a function $\mathbb{R} \times_{\mathsf{Meas}} \mathbb{R} \to 1+1$, by virtue of $(-\infty, 0)$ being a measurable subset of $\mathbb{R}$. Next, observe our slight abuse of notation in $\sigma[\texttt{x}_i \mapsto \lambda]$ which abbreviates the product measure,

$$1\cdot\sigma(\texttt{x}_1) \otimes \ldots \otimes 1\cdot\sigma(\texttt{x}_{i-1}) \otimes \lambda \otimes 1\cdot\sigma(\texttt{x}_{i+1}) \otimes \ldots \otimes 1\cdot\sigma(\texttt{x}_n)$$

where λ is the uniform distribution on $[0,1]$. Finally the last clause interprets while-loops via Kleene's least fixpoint construction. The fact that the map from which we take the least fixpoint is Scott-continuous follows straightforwardly from our previous observations and in particular Equation (1).

The section's remainder is devoted to proving adequacy of our denotational semantics w.r.t. the operational counterpart that was described in Section 2. In order to achieve this – and following the same steps as [24] – we will recur to an auxiliary semantics, which reframes our operational semantics as a *measurable* map. We will see that such is necessary in order to sensibly extend the input-output relation induced by the operational semantics to a probabilistic setting – and thus subsequently connect the latter to the denotational semantics, as intended. Let us thus proceed by presenting this auxiliary semantics.

We will need some preliminaries. Recall that any object E in a category C with binary coproducts induces a monad $E + (-)$. We take the particular case in which $\mathsf{C} = \mathsf{Meas}$ and $E = 1$. We then equip the Kleisli morphisms of this monad with the partial order that is induced from the notion of a flat domain [12]. It is easy to see that this order is ω-complete by an appeal to the following well-known theorem [2, Theorem 4.27].

Theorem 4.1. *Consider an increasing sequence of measurable maps $(f_i)_{i\in\omega} : X \to 1 + Y$. Their supremum w.r.t. the order of flat domains is also measurable.*

Observe then that for any increasing sequence of measurable maps $(f_i)_{i\in\omega} : X \to 1 + Y$ we have,

$$(\sup_{i\in\omega} f_i)^\star = \sup_{i\in\omega} f_i^\star$$

thanks to Scott-continuity of co-pairing on its second argument. Denoting this monad by $(-)_\perp$, observe that its Kleisli category $\mathsf{Meas}_{(-)_\perp}$ is isomorphic to PMeas, *i.e.* that of measurable spaces and partial measurable maps. PMeas has binary coproducts by general categorical results [28]. We then take as the interpretation domain of our auxiliary semantics the monad $E + (-)$ in PMeas where $E = \mathbb{R}^n$. In order to interpret while-loops via this monad, note that it inherits the ω-complete order of $(-)_\perp$ and furthermore,

$$(\sup_{i\in\omega} f_i)^\star = \sup_{i\in\omega} f_i^\star \tag{2}$$

The operational semantics in functional form is now presented in Figure 10. The measurability of each interpretation clause is once again straightforward to verify, and similarly for the fact that the map from which we take the least fixpoint is Scott-continuous, thanks to Equation (2). The symbol $*$ represents undefinedness. Observe that although the functional version of the big-step semantics and the denotational semantics may look similar, the former yields a unique final configuration for a given input, whereas the latter produces a probability distribution over such configurations, enabling probabilistic interpretations.

Finally, the following theorem establishes the aforementioned connection between the operational semantics in Figure 7 and the functional semantics that we have just presented.

Theorem 4.2. *Consider a program* p*, an environment σ, a time instant t, and an entropy source s. Then the following implications hold:*

$$\begin{aligned} &\texttt{p},\sigma,t,s \Downarrow \mathit{skip},\sigma',t',s' && \Rightarrow && (\!|\texttt{p}|\!)(\sigma,t,s) = (\sigma',t',s') \\ &\texttt{p},\sigma,t,s \Downarrow \sigma' && \Rightarrow && (\!|\texttt{p}|\!)(\sigma,t,s) = \sigma' \\ &\texttt{p},\sigma,t,s \Downarrow \mathit{err} && \Rightarrow && (\!|\texttt{p}|\!)(\sigma,t,s) = * \end{aligned}$$

$$(\!|\texttt{diff}(\texttt{e}_1,\ldots,\texttt{e}_n,\texttt{e})|\!) = (\sigma,t,s) \mapsto \begin{cases} \phi(\sigma,t) & \text{if } [\![\texttt{e}]\!](\sigma) > t \\ (\phi(\sigma,[\![\texttt{e}]\!](\sigma)), t - [\![\texttt{e}]\!](\sigma), s) & \text{if } 0 \le [\![\texttt{e}]\!](\sigma) \le t \\ * & \text{otherwise} \end{cases}$$

$$(\!|\texttt{x} := \texttt{e}|\!) = (\sigma,t,s) \mapsto \begin{cases} (\sigma[\texttt{x} \mapsto [\![\texttt{e}]\!](\sigma)], t, s) & \text{if } [\![\texttt{e}]\!](\sigma) \text{ is well-defined} \\ * & \text{otherwise} \end{cases}$$

$$(\!|\texttt{x}_i := \texttt{unif}(\texttt{0},\texttt{1})|\!) = (\sigma,t,(h:s)) \mapsto (\sigma[\texttt{x}_i \mapsto h], t, s)$$

$$(\!|\texttt{p} \;;\; \texttt{q}|\!) = (\!|\texttt{q}|\!)^\star \cdot (\!|\texttt{p}|\!)$$

$$(\!|\texttt{if b then p else q}|\!) = (\sigma,t,s) \mapsto \begin{cases} (\!|\texttt{p}|\!)(\sigma,t,s) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{tt} \\ (\!|\texttt{q}|\!)(\sigma,t,s) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{ff} \\ * & \text{otherwise} \end{cases}$$

$$(\!|\texttt{while b do p}|\!) = \mathrm{lfp}\left(k \mapsto (\sigma,t,s) \mapsto \begin{cases} k^\star \cdot (\!|\texttt{p}|\!)(\sigma,t,s) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{tt} \\ (\sigma,t,s) & \text{if } [\![\texttt{b}]\!](\sigma) = \texttt{ff} \\ * & \text{otherwise} \end{cases} \right)$$

Figure 10: Functional version of the big-step semantics in Figure 7.

Moreover the following implications also hold:

$$(\!|\texttt{p}|\!)(\sigma,t,s) = (\sigma',t',s') \quad \Rightarrow \quad \texttt{p},\sigma,t,s \Downarrow \mathit{skip},\sigma',t',s'$$
$$(\!|\texttt{p}|\!)(\sigma,t,s) = \sigma' \quad \Rightarrow \quad \texttt{p},\sigma,t,s \Downarrow \sigma'$$

Proof. The proof is laborious but straightforward. The first three implications follow by induction over the big-step derivations trees, a close inspection of Kleisli composition, and the fixpoint equation concerning while-loops. The last two implications follow by structural induction of programs, again a close inspection of Kleisli composition, and the proof that for all $i \in \omega$ the following implications hold,

$$f_i(\sigma,t,s) = (\sigma',t',s') \quad \Rightarrow \quad \texttt{while b do p},\sigma,t,s \Downarrow \mathit{skip},\sigma',t',s'$$
$$f_i(\sigma,t,s) = \sigma' \quad \Rightarrow \quad \texttt{while b do p},\sigma,t,s \Downarrow \sigma'$$

where the maps $(f_i)_{i\in\omega} : X \rightharpoonup E + X$ are the components of the supremum involved in Kleene's least fixpoint construction. □

Note that the previous theorem excludes the implication,

$$(\!|\texttt{p}|\!)(\sigma,t,s) = * \quad \Rightarrow \quad \texttt{p},\sigma,t,s \Downarrow \mathit{err}$$

This is simply because the equation $(\!|\texttt{p}|\!)(\sigma,t,s) = *$ may arise from divergence (and not necessarily from an evaluation error) which the operational semantics cannot track. On the other hand, the theorem entails that if $(\!|\texttt{p}|\!)(\sigma,t,s) = *$ and the operational semantics evaluates the tuple $\texttt{p},\sigma,t,s$ to a value then this value is necessarily *err*.

We are now ready to extend the auxiliary semantics $(\!|-|\!)$ to a probabilistic setting. Such hinges on the fact that the pushforward measure construction $(-)_\star$ is functorial on partial measurable maps. This yields the composite functor,

$$\mathsf{PMeas}_E \xrightarrow{(-)^\star} \mathsf{PMeas} \xrightarrow{(-)_\star} \mathsf{Meas}$$

where PMeas_E is the Kleisli category of the monad $E+(-)$ in PMeas and $(-)^\star$ is the respective Kleisli extension. More concretely we obtain the inference rule,

$$\frac{f : X \rightharpoonup E + Y}{(f^\star)_\star : \mathrm{G}(E+X) \to \mathrm{G}(E+Y)}$$

which, intuitively, entails that for any $(\!|\texttt{p}|\!)$ the function $((\!|\texttt{p}|\!)^\star)_\star$ moves probability masses of the measure given as input according to the operational semantics. This is analogous to what happens with the denotational semantics; and our adequacy theorem will render such analogy precise.

We are ready to formulate our adequacy theorem. In order to keep notation simple we will abbreviate the space $\mathbb{R}^n \times \mathbb{R}_{\ge 0}$ to X and, as before, the space $\mathbb{R}^n$ to E. Note that any measure $\mu \in \mathrm{G}(E+X)$ can be decomposed into $\mu_{|E} \in \mathrm{G}(E)$ and $\mu_{|X} \in \mathrm{G}(X)$, where $\mu_{|E}(U) = \mu(U + \emptyset)$ for all measurable subsets $U \subseteq E$ and analogously for $\mu_{|X}$. These restriction operations are linear, commute with suprema, and furthermore $\mu = \mathrm{inl}_\star(\mu_{|E}) + \mathrm{inr}_\star(\mu_{|X})$. We will often abuse notation and elide the left and right injections in the previous measure decomposition.

Theorem 4.3 (Adequacy). *Consider a program* p *and let* λ *be the uniform distribution on* $[0,1]$. *For any measure* $\mu \in \mathrm{G}(E+X)$, *the following equation holds.*

$$((\!|\texttt{p}|\!)^\star)_\star\left(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega}\right) = [\![\texttt{p}]\!]^\star(\mu)_{|E} + [\![\texttt{p}]\!]^\star(\mu)_{|X} \otimes \lambda^\omega$$

In particular the equation below holds for all measurable subsets $U \subseteq E$ *and* $V \subseteq X$.

$$((\!|\texttt{p}|\!)^\star)_\star\left(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega}\right)(U + V \times [0,1]^\omega) = [\![\texttt{p}]\!]^\star(\mu)(U+V)$$

Proof. The proof is obtained via structural induction. The base cases follow straightforwardly although laborious. The case of sequential composition also follows straightfowardly, thanks to the functorial laws and the Kleisli extension laws. We then focus on the case of conditionals. We will need to decompose the measure μ in the cases that it satisfies and does not satisfy b: *i.e.* we will denote $\mu(\emptyset + [\![\texttt{b}]\!] \cap -)$ by ν and $\mu(\emptyset + \overline{[\![\texttt{b}]\!]} \cap -)$ by ρ, where slightly

overloading of notation we set $[\![\mathsf{b}]\!] = \{(\sigma, t) \in X \mid [\![\mathsf{b}]\!](\sigma) = \mathtt{tt}\}$. Then,

$$
\begin{aligned}
&(\!|\texttt{if b then p else q}|\!)^\star)_\star\left(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega}\right)\\
&= \{\mu_{|X} = (\nu + \rho)_{|X}\}\\
&(\!|\texttt{if b then p else q}|\!)^\star)_\star\left(\mu_{|E} + (\nu + \rho)_{|X} \otimes \lambda^{\otimes\omega}\right)\\
&= \{\text{Addition of measures distributes over tensor}\}\\
&(\!|\texttt{if b then p else q}|\!)^\star)_\star\left(\mu_{|E} + \nu_{|X} \otimes \lambda^{\otimes\omega} + \rho_{|X} \otimes \lambda^{\otimes\omega}\right)\\
&= \{\text{Linearity + Semantics definition }\}\\
&\mu_{|E} + ((\!|\mathsf{p}|\!)^\star)_\star\left(\nu_{|X} \otimes \lambda^{\otimes\omega}\right) + ((\!|\mathsf{q}|\!)^\star)_\star\left(\rho_{|X} \otimes \lambda^{\otimes\omega}\right)\\
&= \{\text{Induction hypothesis }\}\\
&\mu_{|E} + [\![\mathsf{p}]\!]^\star(\nu)_{|E} + [\![\mathsf{p}]\!]^\star(\nu)_{|X} \otimes \lambda^{\otimes\omega} + [\![\mathsf{q}]\!]^\star(\rho)_{|E} + [\![\mathsf{q}]\!]^\star(\rho)_{|X} \otimes \lambda^{\otimes\omega}\\
&= \{\text{Addition of measures distributes over tensor}\}\\
&\mu_{|E} + ([\![\mathsf{p}]\!]^\star(\nu) + [\![\mathsf{q}]\!]^\star(\rho))_{|E} + ([\![\mathsf{p}]\!]^\star(\nu) + [\![\mathsf{q}]\!]^\star(\rho))_{|X} \otimes \lambda^{\otimes\omega}\\
&= \{\text{Semantics definition}\}\\
&[\![\texttt{if b then p else q}]\!]^\star(\mu)_{|E} + ([\![\mathsf{p}]\!]^\star(\nu) + [\![\mathsf{q}]\!]^\star(\rho))_{|X} \otimes \lambda^{\otimes\omega}\\
&= \{\text{Semantics definition}\}\\
&[\![\texttt{if b then p else q}]\!]^\star(\mu)_{|E} + [\![\texttt{if b then p else q}]\!]^\star(\mu)_{|X} \otimes \lambda^{\otimes\omega}
\end{aligned}
$$

Lastly we focus on the case of while-loops. Thus let $(f_i)_{i\in\omega}$ be the family of maps involved in Kleene's fixpoint construction w.r.t. the semantics $(\!|-|\!)$ and analogously for $(g_i)_{i\in\omega}$ and the semantics $[\![-]\!]$. We need to show that for all $i \in \omega$,

$$(f_i^\star)_\star\left(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega}\right) = g_i^\star(\mu)_{|E} + g_i^\star(\mu)_{|X} \otimes \lambda^{\omega} \tag{3}$$

This is obtained via induction over the natural numbers. Note that the reasoning is similar to the case concerning conditionals so we omit this step. Then,

$$
\begin{aligned}
&(\!|\texttt{while b do p}|\!)^\star)_\star(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega})\\
&= \{\text{ Semantics definition }\}\\
&((\sup_{i\in\omega} f_i)^\star)_\star(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega})\\
&= \{\text{ Equation (2) }\}\\
&((\sup_{i\in\omega} f_i^\star))_\star(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega})\\
&= \{\text{ Monotone convergence theorem }\}\\
&(\sup_{i\in\omega}(f_i^\star)_\star)(\mu_{|E} + \mu_{|X} \otimes \lambda^{\otimes\omega})\\
&= \{\text{ Equation (3) }\}\\
&\sup_{i\in\omega} g_i^\star(\mu)_{|E} + g_i^\star(\mu)_{|X} \otimes \lambda^{\otimes\omega}\\
&= \{\text{Addition commutes with sup. + Tensor distribute over sup. }\}\\
&\sup_{i\in\omega} g_i^\star(\mu)_{|E} + \left(\sup_{i\in\omega} g_i^\star(\mu)_{|X}\right) \otimes \lambda^{\otimes\omega}\\
&= \{\text{ Equation (1) + Semantics definition }\}\\
&[\![\texttt{while b do p}]\!]^\star(\mu)_{|E} + [\![\texttt{while b do p}]\!]^\star(\mu)_{|X} \otimes \lambda^{\otimes\omega}
\end{aligned}
$$

□

5 Conclusions and future work

This paper provides a basis towards a programming framework of stochastic hybrid systems. It is rooted not only on an operational semantics, which we used in the implementation of an interpreter, but also on a compositional, measure-theoretic counterpart, with which one can formally reason about program equivalence and approximating behaviour, among other things. These contributions open up several interesting research lines. We briefly detail next the ones we are currently exploring.

First, the fact that we committed ourselves to a denotational semantics based on monads will now allow us to capitalise on the more general, extensive theory of monad-based program semantics. This includes for example the extension of our semantics with additional computational effects [28], with higher-order features and different evaluation mechanisms [22, 25], and corresponding logics as well as predicate transformer perspectives [15, 18]. We are already working on the last two topics [15, 18], for not only they potentially offer a complementary tool in our framework of stochastic hybrid programs they would also facilitate a more natural connection between our work and previous results on deductive verification of stochastic hybrid systems [33, 34].

Second, recall from Section 3 and Section 4 that in our semantics program denotations are *contractive* operators $\mathrm{G}(E+X) \to \mathrm{G}(E+X)$. Thus following the observations in [6, 7], such forms the basis of a corresponding theory of *metric* program equivalence. Concretely, instead of comparing two program denotations in terms of *equality* we are able to systematically compare them in terms of distances. In the setting of stochastic hybrid programming this is a much more desirable approach, as in practice it is unrealistic to expect that two programs match their outputs with exact precision. Not only this, reasoning about program distances is extremely important for computationally simulating such programs, since we are limited to certain finite precision aspects and thus can frequently only approximate idealised behaviours.

The works [6, 7] can inclusively be used as basis for a deductive metric equational system w.r.t. our programming language. Very briefly, the corresponding metric $d(-,=)$ would be induced by the operator norm in conjunction with the total variation norm (both are detailed in Section 3). Then [6, 7] would lead us on the analysis of how the different program constructs interact with this metric. For example, it is immediate from the *op. cit.* that for any programs, p, p' and q, q', the following rule holds:

$$\frac{d([\![\mathsf{p}]\!], [\![\mathsf{p}']\!]) \le \epsilon_1 \qquad d([\![\mathsf{q}]\!], [\![\mathsf{q}']\!]) \le \epsilon_2}{d([\![\mathsf{p}\ ;\ \mathsf{q}]\!], [\![\mathsf{p}'\ ;\ \mathsf{q}']\!]) \le \epsilon_1 + \epsilon_2}$$

We leave a full acount of such a metric equational system to future work.

Acknowledgments

This work is financed by National Funds through FCT - Fundação para a Ciência e a Tecnologia, I.P. (Portuguese Foundation for Science and Technology) within the project IBEX, with reference 10.54499/PTDC/CCI-COM/4280/2021. This work is also supported by the CISTER Research Unit (UIDP/UIDB/04234/2020), financed by National Funds through FCT/MCTES (Portuguese Foundation for Science and Technology).

References

[1] Jiří Adámek, Horst Herrlich, and George E. Strecker. 2009. *Abstract and Concrete Categories - The Joy of Cats.* Dover Publications.

[2] Charalambos D. Aliprantis and Kim C. Border. 2006. *Infinite Dimensional Analysis: a Hitchhiker's Guide.* Springer. doi:10.1007/3-540-29587-9

[3] Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. 2020. *Foundations of probabilistic programming.* Cambridge University Press.

[4] Ryan Culpepper and Andrew Cobb. 2017. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In *European Symposium on Programming*. Springer, 368–392.

[5] Fredrik Dahlqvist and Dexter Kozen. 2019. Semantics of higher-order probabilistic programs with conditioning. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–29.

[6] Fredrik Dahlqvist and Renato Neves. 2023. A complete V-equational system for graded lambda-calculus. *Electronic Notes in Theoretical Informatics and Computer Science* 3 (2023).

[7] Fredrik Dahlqvist and Renato Neves. 2023. The syntactic side of autonomous categories enriched over generalised metric spaces. *Logical Methods in Computer Science* 19 (2023).

[8] Luc Devroye. 1986. *Non-Uniform Random Variate Generation.* Springer-Verlag.

[9] R. M. Dudley. 2002. *Real Analysis and Probability* (2 ed.). Cambridge University Press. doi:10.1017/CBO9780511755347

[10] Richard M Dudley. 2018. *Real analysis and probability.* Chapman and Hall/CRC.

[11] Albert Einstein. 1905. Über die von der molekularkinetischen Theorie der Wärme geforderte Bewegung von in ruhenden Flüssigkeiten suspendierten Teilchen. *Annalen der physik* 4 (1905).

[12] Gerhard Gierz, Karl Heinrich Hofmann, Klaus Keimel, Jimmie D. Lawson, Michael W. Mislove, and Dana S. Scott. 1980. *A compendium of continuous lattices.* Springer-Verlag, Berlin. xx + 371 pages.

[13] Michèle Giry. 1982. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, B. Banaschewski (Ed.). Lecture Notes in Mathematics, Vol. 915. Springer Berlin Heidelberg, 68–85. doi:10.1007/BFb0092872

[14] Sergey Goncharov, Renato Neves, and José Proença. 2020. Implementing Hybrid Semantics: From Functional to Imperative. In *Theoretical Aspects of Computing - ICTAC 2020 - 17th International Colloquium, Macau, China, November 30 - December 4, 2020, Proceedings (Lecture Notes in Computer Science, Vol. 12545)*, Violet Ka I Pun, Volker Stolz, and Adenilso Simão (Eds.). Springer, 262–282. doi:10.1007/978-3-030-64276-1_14

[15] Sergey Goncharov and Lutz Schröder. 2013. A relatively complete generic Hoare logic for order-enriched effects. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*. IEEE, 273–282.

[16] Jean Goubault-Larrecq. 2013. *Non-Hausdorff topology and domain theory: Selected topics in point-set topology.* Vol. 22. Cambridge University Press.

[17] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. 2017. A convenient category for higher-order probability theory. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 1–12.

[18] Wataru Hino, Hiroki Kobayashi, Ichiro Hasuo, and Bart Jacobs. 2016. Healthiness from Duality. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science* (New York, NY, USA) *(LICS '16)*. Association for Computing Machinery, New York, NY, USA, 682–691. doi:10.1145/2933575.2935319

[19] Dirk Hofmann, Gavin J Seal, and Walter Tholen. 2014. *Monoidal Topology: A Categorical Approach to Order, Metric, and Topology.* Vol. 153. Cambridge University Press.

[20] Peter Höfner. 2009. *Algebraic calculi for hybrid systems.* Ph. D. Dissertation. University of Augsburg.

[21] Peter Höfner and Bernhard Möller. 2011. Fixing Zeno gaps. *Theoretical Computer Science* 412, 28 (2011), 3303 – 3322. Festschrift in Honour of Jan Bergstra.

[22] GA Kavvos. 2025. Adequacy for Algebraic Effects Revisited. *Proceedings of the ACM on Programming Languages* 9, OOPSLA1 (2025), 927–955.

[23] J. Klafter and I.M. Sokolov. 2011. Continuous-time random walks. In *First Steps in Random Walks: From Tools to Applications*. Oxford University Press. doi:10.1093/acprof:oso/9780199234868.003.0003

[24] Dexter Kozen. 1979. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE, 101–114.

[25] Paul Blain Levy. 2022. Call-by-push-value. *ACM SIGLOG News* 9, 2 (2022), 7–29. doi:10.1145/3537668.3537670

[26] Saunders Mac Lane. 1998. *Categories for the working mathematician.* Vol. 5. springer.

[27] Pedro Mendes, Ricardo Correia, Renato Neves, and José Proença. 2024. Formal Simulation and Visualisation of Hybrid Programs. In *Proceedings Sixth International Workshop on Formal Methods for Autonomous Systems, FMAS@iFM 2024, Manchester, UK, 11th and 12th of November 2024 (EPTCS, Vol. 411)*, Matt Luckcuck and Mengwei Xu (Eds.). 20–37. doi:10.4204/EPTCS.411.2

[28] Eugenio Moggi. 1989. Computational Lambda-Calculus and Monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*. IEEE Computer Society, 14–23.

[29] Eugenio Moggi. 1991. Notions of computation and monads. *Information and computation* 93, 1 (1991), 55–92.

[30] Renato Neves. 2018. *Hybrid programs.* Ph. D. Dissertation. University of Minho. https://repositorium.sdum.uminho.pt/handle/1822/56808

[31] Prakash Panangaden. 1999. The category of Markov kernels. *Electronic Notes in Theoretical Computer Science* 22 (1999), 171–187.

[32] Prakash Panangaden. 2009. *Labelled Markov Processes.* Imperial College Press.

[33] Yu Peng, Shuling Wang, Naijun Zhan, and Lijun Zhang. 2015. Extending hybrid CSP with probability and stochasticity. In *Dependable Software Engineering: Theories, Tools, and Applications: First International Symposium, SETTA 2015, Nanjing, China, November 4-6, 2015, Proceedings 1*. Springer, 87–102.

[34] André Platzer. 2011. Stochastic differential dynamic logic for stochastic hybrid programs. In *International Conference on Automated Deduction*. Springer, 446–460.

[35] André Platzer. 2018. *Logical foundations of cyber-physical systems.* Vol. 662. Springer.

[36] John C Reynolds. 1998. *Theories of programming languages.* Cambridge University Press. doi:10.1017/CBO9780511626364

[37] Tetsuya Sato. 2018. The Giry monad is not strong for the canonical symmetric monoidal closed structure on Meas. *Journal of Pure and Applied Algebra* 222, 10 (2018), 2888–2896.

[38] Glynn Winskel. 1993. *The formal semantics of programming languages - an introduction.* MIT Press. doi:10.7551/mitpress/3054.001.0001