Fine-grained Semantics for Probabilistic Programs

Benjamin Bichsel(✉), Timon Gehr, Martin Vechev

ETH Zürich, Switzerland
{benjamin.bichsel,timon.gehr,martin.vechev}@inf.ethz.ch

Abstract. Probabilistic programming is an emerging technique for modeling processes involving uncertainty. Thus, it is important to ensure these programs are assigned precise formal semantics that also cleanly handle typical exceptions such as non-termination or division by zero. However, existing semantics of probabilistic programs do not fully accommodate different exceptions and their interaction, often ignoring some or conflating multiple ones into a single exception state, making it impossible to distinguish exceptions or to study their interaction.
In this paper, we provide an expressive probabilistic programming language together with a fine-grained measure-theoretic denotational semantics that handles and distinguishes non-termination, observation failures and error states. We then investigate the properties of this semantics, focusing on the interaction of different kinds of exceptions. Our work helps to better understand the intricacies of probabilistic programs and ensures their behavior matches the intended semantics.

1 Introduction

A probabilistic programming language allows probabilistic models to be specified independently of the particular inference algorithms that make predictions using the model. Probabilistic programs are formed using standard language primitives as well as constructs for drawing random values and conditioning. The overall approach is general and applicable to many different settings (e.g., building cognitive models). In recent years, the interest in probabilistic programming systems has grown rapidly with various languages and probabilistic inference algorithms (ranging from approximate to exact). Examples include [11,13,14,25,36,26,27,29] and [10]; for a recent survey, please see [15]. An important branch of recent probabilistic programming research is concerned with providing a suitable semantics for these programs enabling one to formally reason about the program's behaviors [2,3,4,33,34,35].

Often, probabilistic programs require access to primitives that may result in unwanted behavior. For example, the standard deviation σ of a Gaussian distribution must be positive (sampling from a Gaussian distribution with negative standard deviation should result in an error). If a program samples from a Gaussian distribution with a non-constant standard deviation, it is in general undecidable if that standard deviation is guaranteed to be positive. A similar situation occurs for while loops: except in some trivial cases, it is hard to decide if a program terminates with probability one (even harder than checking termination of deterministic programs [20]). However, general while loops are important for many probabilistic programs. As an example, a Markov Chain Monte Carlo sampler is essentially a special probabilistic program, which in practice requires a non-trivial stopping criterion (see e.g. [6] for such a stopping criterion). In addition to offering primitives that may result in such unwanted behavior, many probabilistic programming languages also provide an **observe** primitive that intuitively allows to filter out executions violating some constraint.

Motivation. Measure-theoretic denotational semantics for probabilistic programs is desirable as it enables reasoning about probabilistic programs within the rigorous and general framework of measure theory. While existing research has made substantial progress towards a rigorous semantic foundation of probabilistic programming, existing denotational semantics based on measure theory usually conflate failing **observe** statements (i.e., conditioning), error states and non-termination, often modeling at least some of these as missing weight in a sub-probability measure (we show why this is practically problematic in later examples). This means that even semantically, it is impossible to distinguish these types of exceptions[1]. However, distinguishing exceptions is essential for a solid understanding of probabilistic programs: it is insufficient if the semantics of a probabilistic programming language can only express that *something* went wrong during the execution of the program, lacking the capability to distinguish for example non-termination and errors. Concretely, programmers often want to avoid non-termination and assertion failure, while observation failure is acceptable (or even desirable). When a program runs into an exception, the programmer should be able determine the type of exception, from the semantics.

This Work. This paper presents a clean denotational semantics for a Turing complete first-order probabilistic programming language that supports mixing continuous and discrete distributions, arrays, observations, partial functions and loops. This semantics distinguishes observation failures, error states and non-termination by tracking them as explicit program states. Our semantics allows for fine-grained reasoning, such as determining the termination probability of a probabilistic program making observations from a sequence of concrete values.

In addition, we explain the consequences of our treatment of exceptions by providing interesting examples and properties of our semantics, such as commutativity in the absence of exceptions, or associativity regardless of the presence of exceptions. We also investigate the interaction between exceptions and the **score** primitive, concluding in particular that the probability of non-termination cannot be defined in this case. **score** intuitively allows to increase or decrease the probability of specific runs of a program (for more details, see Section 5.3).

[1] In this paper, we refer to errors, non-termination and observation failures collectively as *exceptions*. For example, a division by zero is an error (and hence and exception), while non-termination is an exception but not an error.

2 Overview

In this section we demonstrate several important features of our probabilistic programming language (PPL) using examples, followed by a discussion involving different kinds of exception interactions.

2.1 Features of Probabilistic Programs

In the following, we informally discuss the most important features of our PPL.

Discrete and continuous primitive distributions. Listing 1 illustrates a simple Gaussian mixture model (the figure only shows the function body). Depending on the outcome of a fair coin flip x (resulting in 0 or 1), y is sampled from a Gaussian distribution with mean 0 or mean 2 (and standard deviation 1). Note that in our PPL, we represent **`gauss`**$(\cdot,\cdot)$ by the more general construct **`sampleFrom`**$_f(\cdot,\cdot)$, with $f : \mathbb{R} \times [0,\infty) \to \mathbb{R} \to \mathbb{R}$ being the probability density function of the Gaussian distribution $f(\mu,\sigma)(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$.

```
y:=0;
if flip(1/2) {
  y=gauss(0,1);
}else{
  y=gauss(2,1);
}
return y;
```

Listing 1. Simple Gaussian mixture

Conditioning. Listing 2 samples two independent values from the uniform distribution on the interval $[0,1]$ and conditions the possible values of x and y on the observation $x+y>1$ before returning x. Intuitively, the first two lines express a-priori knowledge about the uncertain values of x and y. Then, a measurement determines that $x+y$ is greater than 1. We combine this new information with the existing knowledge. Because $x+y>1$ is more likely for larger values of x, the return value has larger weight on larger values. Formally, our semantics handles **`observe`** by introducing an extra program state for observation failure $\lightning$. Hence, the probability distribution after the third line of Listing 2 will put weight $\frac{1}{2}$ on $\lightning$ and weight $\frac{1}{2}$ on those x and y satisfying $x+y>1$.

```
x:=uniform(0,1);
y:=uniform(0,1);
observe(x+y>1);
return x;
```

Listing 2. Conditioning on a continuous distribution

In practice, one will usually condition the output distribution on there being no observation failure ($\lightning$). For discrete distributions, this amounts to computing:

$$Pr[X = x \mid X \neq \lightning] = \frac{Pr[X = x \wedge X \neq \lightning]}{Pr[X \neq \lightning]} = \frac{Pr[X = x]}{1 - Pr[X = \lightning]}$$

where x is the outcome of the program (a value, non-termination or an error) and $Pr[X = x]$ is the probability that the program results in x. Of course, this conditioning only works when the probability of $\lightning$ is not 1. Note that tracking the probability of $\lightning$ has the practical benefit of rendering the (often expensive) marginalization $Pr[X \neq \lightning] = \sum_{x \neq \lightning} Pr[X = x]$ unnecessary.

Other semantics often use sub-probability measures to express failed observations [4,34,35]. These semantics would say that Listing 2 results in a return value between 0 and 1 with probability $\frac{1}{2}$ (and infer that the missing weight of $\frac{1}{2}$ is due to failed observations). We believe one should improve upon this approach as the semantics only implicitly states that the program sometimes fails an observation. Further, this strategy only allows tracking a single kind of exception (in this case, failed observations). This has led some works to conflate observation failure and non-termination [18,34]. We believe there is an important distinction between the two: observation failure means that the program behavior is inconsistent with observed facts, non-termination means that the program did not return a result.

Listing 3 illustrates that it is not possible to condition parts of the program on there being no observation failure. In Listing 3, conditioning the first branch $x := 0;$ **observe**(**flip**($\frac{1}{2}$)) on there being no observation failure yields $Pr[x = 0] = 1$, rendering the observation irrelevant. The same situation arises for the second branch. Hence, conditioning the two branches in isolation yields $Pr[x = 0] = \frac{1}{2}$ instead of $Pr[x = 0] = \frac{2}{3}$.

```
if flip(1/2) {
  x:=0;
  observe(flip(1/2));
}else{
  x:=1;
  observe(flip(1/4));
}
```

Listing 3. The need for tracking $\lightning$

Loops. Listing 4 shows a probabilistic program with a while loop. It samples from the **geometric**($\frac{1}{2}$) distribution, which counts the number of failures (**flip** returns 0) until the first success occurs (**flip** returns 1). This program terminates with probability 1, but it is of course possible that a probabilistic program fails to terminate with positive probability. Listing 5 demonstrates this possibility.

```
n:=0;
while !flip(1/2) {
  n=n+1;
}
return n;
```

Listing 4. Geometric distribution

Listing 5 modifies x until either $x = 0$ or $x = 10$. In each iteration, x is either increased or decreased, each with probability $\frac{1}{2}$. If x reaches 0, the loop terminates. If x reaches 10, the loop never terminates. By symmetry, both termination and non-termination are equally likely. Hence, the program either returns 0 or does not terminate, each with probability $\frac{1}{2}$.

```
x := 5;
while x>0 {
  if x<10 {
    x+=2*flip(1/2)-1;
  }
}
return x;
```

Listing 5. Program that may not terminate

Other semantics often use sub-probability measures to express non-termination [4,23]. Thus, these semantics would say that Listing 5 results in 0 with probability $\frac{1}{2}$ (and nothing else). We propose to track the probability of non-termination explicitly by an additional state $\circlearrowleft$, just as we track the probability of observation failure ($\lightning$).

Partial functions. Many functions that are practically useful are only partial (meaning they are not defined for some inputs). Examples include **uniform**(a, b) (undefined for $b < a$) and $\sqrt{x}$ (undefined for $x < 0$). Listing 6 shows an example program using $\sqrt{x}$. Usually, semantics do not explicitly address partial functions [23,24,28,33] or use partial

```
x:=uniform(-1,1);
x=√x;
return x;
```

Listing 6. Using partial functions

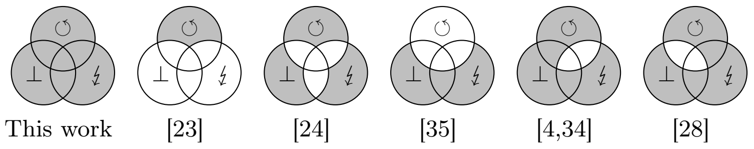

Fig. 1. Visual comparison of the exception handling capabilities of different semantics. For example, $\circlearrowleft$ is filled in [34] because its semantics can handle non-termination. However, the intersection between $\circlearrowleft$ and $\lightning$ is not filled because [34] cannot distinguish non-termination from observation failure.

functions without dealing with failure (e.g. [19] use **`Bernoulli`**(p) without stating what happens if $p \notin [0, 1]$). Most of these languages could use a sub-probability distribution that misses weight in the presence of errors (in these languages, this results in conflating errors with non-termination and observation failures).

We introduce a third exception state $\perp$ that can be produced when partial functions are evaluated outside of their domain. Thus, Listing 6 results in $\perp$ with probability $\frac{1}{2}$ and returns a value from $[0, 1]$ with probability $\frac{1}{2}$ (larger values are more likely). Some previous work uses an error state to capture failing computations, but does not propagate this failure implicitly [34,35]. In particular, if an early expression in a long program may fail evaluating $\sqrt{-4}$, every expression in the program that depends on this failing computation has to check whether an exception has occurred. While it may seem possible to skip the rest of the function in case of a failing computation (by applying the pattern **`if`** $(x = \perp)$ $\{$**`return`** $\perp\}$ **`else`** $\{$rest of function$\}$), this is non-modular and does not address the result of the function being used in other parts of a program.

Although our semantics treat $\perp$ and $\lightning$ similarly, there is an important distinction between the two: $\perp$ means the program terminated due to an error, while $\lightning$ means that according to observed evidence, the program did not actually run.

2.2 Interaction of Exception States

Next, we illustrate the interaction of different exception states. We explain how our semantics handles these interactions when compared to existing semantics. Figure 1 gives an overview of which existing semantics can handle which (interactions of) exceptions. We note that our semantics could easily distinguish more kinds of exceptions, such as division by zero or out of bounds accesses to arrays.

```
x:=0;
while x=0 {
  x=flip(½);
  observe(x=0);
}
```

Listing 7. Mixing loops and observations

Non-termination and observation failure. Listing 7 shows a program that has been investigated in [22]. Based on the observations, it only admits a single behavior, namely always sampling $x = 0$ in the third line. This behavior results in non-termination, but it occurs with probability 0. Hence, the program fails an observation (ending up in state $\lightning$) with probability 1. If we try to condition on not failing any observation (by rescaling appropriately), this results in a division by 0, because the probability of not failing any observation is 0.

The semantics of Listing 7 thus only has weight on $\lightning$, and does not allow conditioning on not failing any observation. This is also the solution that [22] proposes, but in our case, we can formally back up this claim with our semantics.

Other languages handle both non-termination and observation failure by sub-probability distributions, which makes it impossible to conclude that the missing weight is due to observation failure (and not due to non-termination) [4,24,34]. The semantics in [28] cannot directly express that the missing weight is due to observation failure (rather, the semantics are undefined due to a division by zero). However, the semantics enables a careful reader to determine that the missing weight is due to observation failure (by investigating the conditional weakest precondition and the conditional weakest liberal precondition). Some other languages can express neither while loops nor observations [23,33,35].

Assertions and non-termination. For some programs, it is useful to check assumptions explicitly. For example, the implementation of the factorial function in Listing 8 explicitly checks whether x is a valid argument to the factorial function. If $x \notin \mathbb{N}$, the program should run into an error (i.e. only have weight on $\bot$). If $x \in \mathbb{N}$, the program should return $x!$ (i.e. only have weight on $x!$). This example illustrates that earlier exceptions (like failing an assertion) should *bypass* later exceptions (like non-termination, which occurs for $x \notin \mathbb{N}$ if the programmer forgets the first two assertions). This is not surprising, given that this is also the semantics of exceptions in most deterministic languages. Most existing semantics either cannot express Listing 8 ([23,34] have no assertions, [35] has no iteration) or cannot distinguish failing an assertion from non-termination [24,28,33]. The consequence of the latter is that removing the first two assertions from Listing 8 does not affect the semantics. Handling assertion failure by sum types (as e.g. in [34]) could be a solution, but would force the programmer to deal with assertion failure explicitly. Only the semantics in [4] has the expressiveness to implicitly handle assertion errors in Listing 8 without conflating those errors with non-termination.

```
assert(x≥0);
assert(x=⌊x⌋);
fac:=1;
while x≠0 {
  fac=fac*x;
  x=x-1;
}
return fac;
```

Listing 8. Explicitly checking assumptions

Listing 9 shows a different interaction between non-termination and failing assertions. Here, even though the loop condition is always true, the first iteration of the loop will run into an exception. Thus, Listing 9 results in $\bot$ with probability 1. Again, this behavior should not be surprising given the behavior of deterministic languages. For Listing 9, conflating errors with non-termination means the program semantics cannot express that the missing weight is due to an error and not due to non-termination.

```
x:=0;
while 1 {
  x=x/x;
}
```

Listing 9. Guaranteed failure

Observation failure and assertion failure. In our PPL, earlier exceptions bypass later exceptions, as illustrated in Listing 8. However, because we are operating in a probabilistic language, exceptions can occur probabilistically. Listing 10 shows a program that may run into an observation failure, or into an assertion failure, or neither. If it runs into an observation failure (with probability $\frac{1}{2}$), it bypasses the rest of the program, resulting in $\lightning$ with probability $\frac{1}{2}$ and in $\perp$ with probability $\frac{1}{4}$. Conditioning on the absence of observation failures, the probability of $\perp$ is $\frac{1}{2}$.

```
observe(flip(½));
assert(flip(½));
```

Listing 10. Observation or assertion failure

An important observation is that reordering the two statements of Listing 10 will result in a different behavior. This is the case, even though there is no obvious data-flow between the two statements. This is in sharp contrast to the semantics in [34], which guarantee (in the absence of exceptions) that only data flow is relevant and that expressions can be reordered. Our semantics illustrate that even if there is no explicit data-dependency, some seemingly obvious properties (like commutativity) may not hold in the presence of exceptions. Some languages either cannot express Listing 10 ([23,33] lack observations), cannot distinguish observation failure from assertion failure [24] or cannot handle exceptions implicitly [34,35].

Summary. In this section, we showed examples of probabilistic programs that exhibit non-termination, observation failures and errors. Then, we provided examples that show how these exceptions can interact, and explained how existing semantics handle these interactions.

3 Preliminaries

In this section, we provide the necessary theory. Most of the material is standard, however, our treatment of exception states is interesting and important for providing semantics to probabilistic programs in the presence of exceptions. All key lemmas (together with additional definitions and examples) are proven in Appendix A.

Natural numbers, $[n]$, Iverson brackets, restriction of functions. We include 0 in the natural numbers, so that $\mathbb{N} := \{0, 1, \dots\}$. For $n \in \mathbb{N}$, $[n] := \{1, \dots, n\}$. The *Iverson brackets* $[\cdot]$ are defined by $[b] = 1$ if b is true and $[b] = 0$ if b is false. A particular application of the Iverson brackets is to characterize the indicator function of a specific set S by $[x \in S]$. For a function $f : X \to Y$ and a subset of the domain $S \subseteq X$, f restricted to S is denoted by $f_{|S} : S \to Y$.

Set of variables, generating tuples, preservation of properties, singleton set. Let Vars be a set of admissible variable names. We refer to the elements of Vars by x, y, z and x_i, y_i, z_i, v_i, w_i, for $i \in \mathbb{N}$. For $v \in A$ and $n \in \mathbb{N}$, $v!n := (v, \dots, v) \in A^n$ denotes the tuple containing n copies of v. A function $f : A^n \to A$ *preserves a property* if whenever $a_1, \dots, a_n \in A$ have that property, $f(a_1, \dots, a_n) \in A$

has that property. Let $\mathbb{1}$ denote the set which only contains the empty tuple $()$, i.e. $\mathbb{1} := \{()\}$. For sets of tuples $S \subseteq \prod_{i=1}^{n} A_i$, there is an isomorphism $S \times \mathbb{1} \simeq \mathbb{1} \times S \simeq S$. This isomorphism is intuitive and we sometimes silently apply it.

Exception states, lifting functions to exception states. We allow the extension of sets with some symbols that stand for the occurrence of special events in a program. This is important because it allows us to capture the event that a given program runs into specific exceptions. Let $\mathcal{X} := \{\bot, \lightning, \circlearrowleft\}$ be a (countable) set of exception states. We denote by $\overline{A} := A \cup \mathcal{X}$ the set A extended with $\mathcal{X}$ (we require that $A \cap \mathcal{X} = \emptyset$). Intuitively, $\bot$ corresponds to assertion failures, $\lightning$ corresponds to observation failures and $\circlearrowleft$ corresponds to non-termination. For a function $f \colon A \to B$, f *lifted to exception states*, denoted by $\overline{f} \colon \overline{A} \to \overline{B}$ is defined by $\overline{f}(a) = a$ if $a \in \mathcal{X}$ and $\overline{f}(a) = f(a)$ if $a \notin \mathcal{X}$. For a function $f \colon \prod_{i=1}^{n} A_i \to B$, f *lifted to exception states*, denoted by $\overline{f} \colon \prod_{i=1}^{n} \overline{A_i} \to \overline{B}$, propagates the first exception in its arguments, or evaluates f if none of its arguments are exceptions. Formally, it is defined by $\overline{f}(a_1, \dots, a_n) = a_1$ if $a_1 \in \mathcal{X}$, $\overline{f}(a_1, \dots, a_n) = a_2$ if $a_1 \notin \mathcal{X}$ and $a_2 \in \mathcal{X}$, and so on. Only if $a_1, \dots, a_n \notin \mathcal{X}$, we have $\overline{f}(a_1, \dots, a_n) = f(a_1, \dots, a_n)$. Thus, $\overline{f}(\circlearrowleft, a, \bot) = \circlearrowleft$. In particular, we write $\overline{(a, b)}$ for lifting the tupling function, resulting in for example $\overline{(\lightning, \circlearrowleft)} = \lightning$. To remove notation clutter, we do not distinguish the two different liftings $\overline{f} \colon \overline{A} \to \overline{B}$ and $\overline{f} \colon \prod_{i=1}^{n} \overline{A_i} \to \overline{B}$ notationally. Whenever we write $\overline{f}$, it will be clear from the context which lifting we mean. We write $S \overline{\times} T$ for $\{\overline{(s, t)} \mid s \in S, t \in T\}$.

Records. A *record* is a special type of tuple indexed by variable names. For sets $(S_i)_{i \in [n]}$, a record $r \in \prod_{i=1}^{n} (x_i \colon S_i)$ has the form $r = \{x_1 \mapsto v_1, \dots, x_n \mapsto v_n\}$, where $v_i \in S_i$, with the convenient shorthand $r = \{x_i \mapsto v_i\}_{i \in [n]}$. We can access the elements of a record by their name: $r[x_i] = v_i$.

In what follows, we provide the measure theoretic background necessary to express our semantics.

σ-algebra, measurable set, σ-algebra generated by a set, measurable space, measurable functions. Let A be some set. A set $\Sigma_A \subseteq \mathcal{P}(A)$ is called a *σ-algebra on A* if it satisfies three conditions: $A \in \Sigma_A$, Σ_A is closed under complements ($S \in \Sigma_A$ implies $A \backslash S \in \Sigma_A$) and Σ_A is closed under countable unions (for any collection $\{S_i\}_{i \in \mathbb{N}}$ with $S_i \in \Sigma_A$, we have $\bigcup_{i \in \mathbb{N}} S_i \in \Sigma_A$). The elements of Σ_A are called *measurable sets.* For any set A, a trivial σ-algebra on A is its power set $\mathcal{P}(A)$. Unfortunately, the power set often contains sets that do not behave well. To come up with a σ-algebra on A whose sets do behave well, we often start with a set $S \subseteq \mathcal{P}(A)$ that is not a σ-algebra and extend it until we get a σ-algebra. For this purpose, let A be some set and $S \subseteq \mathcal{P}(A)$ a collection of subsets of A. The *σ-algebra generated by S* denoted by $\sigma(S)$ is the smallest σ-algebra that contains S. Formally, $\sigma(S)$ is the intersection of all σ-algebras on A containing S. For a set A and a σ-algebra Σ_A on A, (A, Σ_A) is called a *measurable space.* We often leave Σ_A implicit; whenever it is not mentioned explicitly, it is clear from the context. Table 1 provides the implicit σ-algebras for some common sets. As

Set	**σ-algebra on this set**
$\mathbb{R}$	$\Sigma_{\mathbb{R}} = \mathcal{B} := \sigma(\{[a,b] \subseteq \mathbb{R} \mid a \leq b, a \in \mathbb{R}, b \in \mathbb{R}\})$, the Borel σ-algebra on $\mathbb{R}$ generated by all intervals
S for $S \in \mathcal{B}$	$\Sigma_S = \{T \in \mathcal{B} \mid T \subseteq S\}$
$\prod_{i=1}^n A_i$	$\Sigma_{\prod_{i=1}^n A_i} = \sigma\left(\{\prod_{i=1}^n S_i \mid S_i \in \Sigma_{A_i}\}\right)$
$\prod_{i=1}^n (x_i : A_i)$	$\Sigma_{\prod_{i=1}^n (x_i : A_i)} = \sigma\left(\{\prod_{i=1}^n (x_i : S_i) \mid S_i \in \Sigma_{A_i}\}\right)$
$\overline{A}$	$\Sigma_{\overline{A}} = \{S \cup S' \mid S \in \Sigma_A, S' \in \mathcal{P}(\mathcal{X})\}$

Table 1. Implicit σ-algebras on common sets, for measurable spaces (A, Σ_A), (A_i, Σ_{A_i})

an example, some elements of $\Sigma_{\overline{\mathbb{R}}}$ include $[0,1] \cup \{\perp\}$ and $\{1, 3, \pi\}$. For measurable spaces (A, Σ_A) and (B, Σ_B), a function $f \colon A \to B$ is called *measurable*, if $\forall S \in \Sigma_B \colon f^{-1}(S) \in \Sigma_A$. Here, $f^{-1}(S) := \{a \in A \colon f(a) \in S\}$. If one is familiar with the notion of Lebesgue measurable functions, note that our definition does not include all Lebesgue measurable functions. As a motivation to why we need measurable functions, consider the following scenario. We know the distribution of some variable x, and want to know the distribution of $y = f(x)$. To figure out how likely it is that $y \in S$ for a measurable set S, we can determine how likely it is that $x \in f^{-1}(S)$, because $f^{-1}(S)$ is guaranteed to be a measurable set.

Measures, examples of measures. For a measurable space (A, Σ_A), a function $\mu \colon \Sigma_A \to [0, \infty]$ is called a *measure on A* if it satisfies two properties: null empty set ($\mu(\emptyset) = 0$) and countable additivity (for any countable collection $\{S_i\}_{i \in \mathcal{I}}$ of pairwise disjoint sets $S_i \in \Sigma_A$, we have $\mu\left(\bigcup_{i \in \mathcal{I}} S_i\right) = \sum_{i \in \mathcal{I}} \mu(S_i)$). Measures allow us to quantify the probability that a certain result lies in a measurable set. For example, $\mu([1,2])$ can be interpreted as the probability that the outcome of a process is between 1 and 2.

The *Lebesgue measure* $\lambda \colon \mathcal{B} \to [0, \infty]$ is the (unique) measure that satisfies $\lambda([a,b]) = b - a$ for all $a, b \in \mathbb{R}$ with $a \leq b$. The *zero measure* $\mathbf{0} \colon \Sigma_A \to [0, \infty]$ is defined by $\mathbf{0}(S) = 0$ for all $S \in \Sigma_A$. For a measurable space (A, Σ_A) and some $a \in A$, the *Dirac measure* $\delta_a \colon \Sigma_A \to [0, \infty]$ is defined by $\delta_a(S) = [a \in S]$.

Unfortunately, there are measures that do not satisfy some important properties (for example, they may not satisfy Fubini's theorem, which we discuss later on). The usual way to deal with this is to restrict our attention to σ-finite measures, which are well-known and were studied in great detail. However, σ-finite measures are too restrictive for our purposes. In particular, the s-finite kernels that we introduce later on can induce measures that are not σ-finite. This is why in the following, we work with s-finite measures. Table 2 gives an overview of the different kinds of measures that are important for understanding our work. The expression $1/2 \cdot \delta_1$ stands for the pointwise multiplication of the measure δ_1 by $1/2$: $1/2 \cdot \delta_1 = \lambda S.\, 1/2 \cdot \delta_1(S)$. Here, the λ refers to λ-abstraction and not to the Lebesgue measure. To distinguish the two λs, we always write "$\lambda x.$" (with a dot) when we refer to λ-abstraction. For more details on the definitions and for proofs about the provided examples, see Appendix A.1.

Type of Measure	**Characterization**	**Examples**
probability measure	μ is a measure and $\mu(A) = 1$	$\mu = \delta_1$
sub-probability measure	μ is a measure and $\mu(A) \leq 1$	$\mu = \mathbf{0}$ or $\mu = 1/2 \cdot \delta_1$
σ-finite measure	μ is a measure and $A = \bigcup_{i \in \mathbb{N}} A_i$ for $A_i \in \Sigma_A$ with $\mu(A_i) < \infty$	$\mu = \lambda$
s-finite measure	$\mu = \sum_{i \in \mathbb{N}} \mu_i$ for sub-probability measures μ_i	$\mu(S) = \begin{cases} 0 & \lambda(S) = 0 \\ \infty & \lambda(S) > 0 \end{cases}$
measure	$\mu(\emptyset) = 0$, countable additivity	$\mu(S) = \begin{cases} \lvert S \rvert & S \text{ finite} \\ \infty & \text{otherwise} \end{cases}$

Table 2. Definition and comparison of different measures $\mu\colon \Sigma_A \to [0, \infty]$ on measurable spaces (A, Σ_A). Reading the table top-down, we get from the most restrictive definition to the most permissive definition. For example, any sub-probability measure is also a σ-finite measure. We also provide an example for each type of measure that is not an example of the more restrictive type of measure. For example, the Lebesgue measure λ is σ-finite but not s-finite.

Product of measures, product of measures in the presence of exception states. For s-finite measures $\mu\colon \Sigma_A \to [0, \infty]$ and $\mu'\colon \Sigma_B \to [0, \infty]$, we denote the *product of measures* by $\mu \times \mu'\colon \Sigma_{A \times B} \to [0, \infty]$, and define it by

$$(\mu \times \mu')(S) = \int_{a \in A} \int_{b \in B} [(a, b) \in S] \mu'(db) \mu(da)$$

For s-finite measures $\mu\colon \Sigma_{\overline{A}} \to [0, \infty]$ and $\mu'\colon \Sigma_{\overline{B}} \to [0, \infty]$, we denote the *lifted product of measures* by $\mu \overline{\times} \mu'\colon \Sigma_{\overline{A \times B}} \to [0, \infty]$ and define it using the lifted tupling function: $(\mu \overline{\times} \mu')(S) = \int_{a \in \overline{A}} \int_{b \in \overline{B}} [\overline{(a, b)} \in S] \mu'(db) \mu(da)$. While the product of measures $\mu \times \mu'$ is well known for combining two measures to a joint measure, the concept of a lifted product of measures $\mu \overline{\times} \mu'$ is required to do the same for combining measures that have weight on exception states. Because the formal semantics of our probabilistic programming language makes use of exception states, we always use $\overline{\times}$ to combine measures, appropriately handling exception states implicitly.

Lemma 1. *For measures $\mu\colon \Sigma_A \to [0, \infty]$, $\mu'\colon \Sigma_B \to [0, \infty]$, let $S \in \Sigma_A$ and $T \in \Sigma_B$. Then, $(\mu \times \mu')(S \times T) = \mu(S) \cdot \mu'(T)$.*

For $\mu\colon \Sigma_{\overline{A}} \to [0, \infty]$, $\mu'\colon \Sigma_{\overline{B}} \to [0, \infty]$ and $S \in \Sigma_{\overline{A}}$, $T \in \Sigma_{\overline{B}}$, in general we have $(\mu \overline{\times} \mu')(S \times T) \neq \mu(S) \cdot \mu'(T)$, due to interactions of exception states.

Lemma 2. $\times$ *and* $\overline{\times}$ *for s-finite measures are associative, left- and right-distributive and preserve (sub-)probability and s-finite measures.*

Lebesgue integrals, Fubini's theorem for s-finite measures. Our definition of the Lebesgue integral is based on [31]. It allows integrating functions that sometimes evaluate to ∞, and Lebesgue integrals evaluating to ∞.

Here, (A, Σ_A) and (B, Σ_B) are measurable spaces and $\mu\colon \Sigma_A \to [0,\infty]$ and $\mu'\colon \Sigma_B \to [0,\infty]$ are measures on A and B, respectively. Also, $E \in \Sigma_A$ and $F \in \Sigma_B$. Let $s\colon A \to [0,\infty)$ be a measurable function. s is a *simple function* if $s(x) = \sum_{i=1}^{n} \alpha_i [x \in A_i]$ for $A_i \in \Sigma_A$ and $\alpha_i \in \mathbb{R}$. For any simple function s, the Lebesgue integral of s over E with respect to μ, denoted by $\int_{a\in E} s(a)\mu(da)$, is defined by $\sum_{i=1}^{n} \alpha_i \cdot \mu(A_i \cap E)$, making use of the convention $0 \cdot \infty = 0$. Let $f\colon A \to [0,\infty]$ be measurable but not necessarily simple. Then, the *Lebesgue integral* of f over E with respect to μ is defined by

$$\int_{a\in E} f(a)\mu(da) := \sup\left\{ \int_{a\in E} s(a)\mu(da) \,\middle|\, s : A \to [0,\infty) \text{ is simple}, 0 \leq s \leq f \right\}$$

Here, the inequalities on functions are pointwise. Appendix A.2 lists some useful properties of the Lebesgue integral. Here, we only mention Fubini's theorem, which is important because it entails a commutativity-like property of the product of measures: $(\mu \times \mu')(S) = (\mu' \times \mu)(\mathsf{swap}(S))$, where swap switches the dimensions of S: $\mathsf{swap}(S) = \{(b,a) \mid (a,b) \in S\}$. The proof of this property is straightforward, by expanding the definition of the product of measures and applying Fubini's theorem. As we show in Section 5, this property is crucial for the commutativity of expressions. In the presence of exceptions, it does not hold: $(\mu \overline{\times} \mu')(S) \neq (\mu' \overline{\times} \mu)(\mathsf{swap}(S))$ in general.

Theorem 1 (Fubini's theorem). *For s-finite measures $\mu\colon \Sigma_A \to [0,\infty]$ and $\mu'\colon \Sigma_B \to [0,\infty]$ and any measurable function $f\colon A \times B \to [0,\infty]$,*

$$\int_{a\in A} \int_{b\in B} f(a,b)\mu'(db)\mu(da) = \int_{b\in B} \int_{a\in A} f(a,b)\mu(da)\mu'(db)$$

For s-finite measures $\mu\colon \Sigma_{\overline{A}} \to [0,\infty]$ and $\mu'\colon \Sigma_{\overline{B}} \to [0,\infty]$ and any measurable function $f\colon A \times B \to [0,\infty]$,

$$\int_{a\in \overline{A}} \int_{b\in \overline{B}} \overline{f}(a,b)\mu'(db)\mu(da) = \int_{b\in \overline{B}} \int_{a\in \overline{A}} \overline{f}(a,b)\mu(da)\mu'(db)$$

(Sub-)probability kernels, s-finite kernels, Dirac delta, Lebesgue kernel, motivation for s-finite kernels. In the following, let (A, Σ_A) and (B, Σ_B) be measurable spaces. A *(sub-)probability kernel with source A and target B* is a function $\kappa\colon A \times \Sigma_B \to [0,\infty)$ such that for all $a \in A$: $\kappa(a,\cdot)\colon \Sigma_B \to [0,\infty)$ is a (sub-)probability measure, and $\forall S \in \Sigma_B\colon \kappa(\cdot, S)\colon A \to [0,\infty)$ is measurable. $\kappa\colon A \times \Sigma_B \to [0,\infty]$ is an *s-finite kernel with source A and target B* if κ is a pointwise sum of sub-probability kernels $\kappa_i\colon A \times \Sigma_B \to [0,\infty)$, meaning $\kappa = \sum_{i\in\mathbb{N}} \kappa_i$. We denote the set of s-finite kernels with source A and target B by $A \mapsto B \subseteq A \times \Sigma_B \to [0,\infty]$. Because we only ever deal with s-finite kernels, we often refer to them simply as kernels.

We can understand the Dirac measure as a probability kernel. For a measurable space (A, Σ_A), the *Dirac delta* $\delta\colon A \mapsto A$ is defined by $\delta(a,S) = [a \in S]$. Note that for any a, $\delta(a,\cdot)\colon \Sigma_A \to [0,\infty]$ is the Dirac measure. We often write

$\delta(a)(S)$ or $\delta_a(S)$ for $\delta(a, S)$. Note that we can also interpret $\delta \colon A \mapsto A$ as an s-finite kernel from $A \mapsto B$ for $A \subseteq B$. The *Lebesgue kernel* $\lambda^* \colon A \mapsto \mathbb{R}$ is defined by $\lambda^*(a)(S) = \lambda(S)$, where λ is the Lebesgue measure. The definition of s-finite kernels is a lifting of the notion of s-finite measures. Note that for an s-finite kernel κ, $\kappa(a, \cdot)$ is an s-finite measure for all $a \in A$. In the context of probabilistic programming, s-finite kernels have been used before [34].

Working in the space of sub-probability kernels is inconvenient, because, for example, $\lambda^* \colon \mathbb{R} \mapsto \mathbb{R}$ is not a sub-probability kernel. Even though $\lambda^*(x)$ is σ-finite measure for all $x \in \mathbb{R}$, not all s-finite kernels induce σ-finite measures in this sense. As an example, $(\lambda^*; \lambda^*)(x)$ is not a σ-finite measure for any $x \in \mathbb{R}$ (see Lemma 15 in Appendix A.1). We introduce $(;)$ shortly in Definition 1.

Working in the space of s-finite kernels is convenient because s-finite kernels have many nice properties. In particular, the set of s-finite kernels $A \mapsto B$ is the smallest set that contains all sub-probability kernels with source A and target B and is closed under countable sums.

Lifting kernels to exception states, removing weight from exception states. For kernels $\kappa \colon A \mapsto B$ or kernels $\kappa \colon A \mapsto \overline{B}$, κ *lifted to exception states* $\overline{\kappa} \colon \overline{A} \mapsto \overline{B}$ is defined by $\overline{\kappa}(a) = \kappa(a)$ if $a \in A$ and $\overline{\kappa}(a) = \delta(a)$ if $a \notin A$. When transforming κ into $\overline{\kappa}$, we preserve (sub-)probability and s-finite kernels.

Composing kernels, composing kernels in the presence of exception states.

Definition 1. *Let* $(;) \colon (A \mapsto B) \to (B \mapsto C) \to (A \mapsto C)$ *be defined by* $(f; g)(a)(S) = \int_{b \in B} g(b)(S)\, f(a)(db)$.

Note that $f; g$ intuitively corresponds to first applying f and then g. Throughout this paper, we mostly use $>\!\!=\!\!>$ instead of $(;)$, but we introduce $(;)$ because it is well-known and it is instructive to show how our definition of $>\!\!=\!\!>$ relates to $(;)$.

Lemma 3. $(;)$ *is associative, left- and right-distributive, has neutral element*[2] δ *and preserves (sub-)probability and s-finite kernels.*

Definition 2. *Let* $(>\!\!=\!\!>) \colon (A \mapsto \overline{B}) \to (B \mapsto \overline{C}) \to (A \mapsto \overline{C})$ *be defined by* $(f >\!\!=\!\!> g)(a)(S) = \int_{b \in \overline{B}} \overline{g}(b)(S)\, f(a)(db)$.

We sometimes write $f(a) \gg\!= g$ for $(f >\!\!=\!\!> g)(a)$.

Lemma 4. *For* $f \colon A \mapsto \overline{B}$ *and* $g \colon B \mapsto \overline{C}$, $a \in A$ *and* $S \in \Sigma_{\overline{C}}$,

$$(f >\!\!=\!\!> g)(a)(S) = (f; g)(a)(S) + \sum_{x \in \mathcal{X}} \delta(x)(S) f(a)(\{x\})$$

Lemma 4 shows how $>\!\!=\!\!>$ relates to $(;)$, by splitting $f >\!\!=\!\!> g$ into non-exceptional behavior of f (handled by $(;)$) and exceptional behavior of f (handled by a sum). Intuitively, if f produces an exception state $\star \in \mathcal{X}$, then g is not even evaluated. Instead, this exception is directly passed on, as indicated by $\delta(x)(S)$.

[2] δ is a neutral element of $(;)$ if $(\delta; \kappa) = (\kappa; \delta) = \kappa$ for all kernels κ.

If $f(a)(\mathcal{X}) = 0$ for all $a \in A$, or if $S \cap \mathcal{X} = \emptyset$, then the definitions are equivalent in the sense that $(f; g)(a)(S) = (f \ggg g)(a)(S)$. The difference between $\ggg$ and (;) is the treatment of exception states produced by f. Note that technically, the target $\overline{B}$ of $f\colon A \mapsto \overline{B}$ does not match the source B of $g\colon B \mapsto \overline{C}$. Therefore, to formally interpret $f; g$, we silently restrict the domain of f to $A \times \Sigma_B$.

Lemma 5. *$\ggg$ is associative, left-distributive (but not right-distributive), has neutral element δ and preserves (sub-)probability and s-finite kernels.*

Product of kernels, product of kernels in the presence of exception states. For s-finite kernels $\kappa\colon A \mapsto B$, $\kappa'\colon A \mapsto C$, we define the *product of kernels*, denoted by $\kappa \times \kappa'\colon A \mapsto B \times C$, as $(\kappa \times \kappa')(a)(S) = (\kappa(a) \times \kappa'(a))(S)$. For s-finite kernels $\kappa\colon A \mapsto \overline{B}$ and $\kappa'\colon A \mapsto \overline{C}$, we define the *lifted product of kernels*, denoted by $\kappa \overline{\times} \kappa'\colon A \mapsto \overline{B \times C}$, as $(\kappa \overline{\times} \kappa')(a)(S) = (\kappa(a) \overline{\times} \kappa'(a))(S)$. $\times$ and $\overline{\times}$ allow us to combine kernels to a joint kernel. Essentially, this definition reduces the product of kernels to the product of measures.

Lemma 6. *$\times$ and $\overline{\times}$ for kernels preserve (sub-)probability and s-finite kernels, are associative, left- and right-distributive.*

Binding conventions. To avoid too many parentheses, we make use of some binding conventions, ordering (in decreasing binding strength) $\overline{\times}, \times, ;\,, \ggg, +$.

Summary. The most important concepts introduced in this section are exception states, records, Lebesgue integration, Fubini's theorem and (s-finite) kernels.

4 A Probabilistic Language and its Semantics

We now describe our probabilistic programming language, the typing rules and the denotational semantics of our language.

4.1 Syntax

Let $\mathbb{V} := \mathbb{Q} \cup \{\pi, e\} \subseteq \mathbb{R}$ be a (countable) set of constants expressible in our programs. Let $i, n \in \mathbb{N}$, $r \in \mathbb{V}$, $x \in \text{Vars}$, $\ominus$ a generic unary operator (e.g., $-$ inverts the sign of a value, ! is logical negation mapping 0 to 1 and all other numbers to 0, $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ round down and up respectively), $\oplus$ a generic binary operator (e.g., $+, -, *, /,$ ^ for addition, subtraction, multiplication, division and exponentiation, &&, || for logical conjunction and disjunction, $=, \neq, <, \leq, >, \geq$ to compare values). Let $f\colon A \to \mathbb{R} \to [0, \infty)$ be a measurable function that maps $a \in A$ to a probability density function. We check if f is measurable by uncurrying f to $f\colon A \times \mathbb{R} \to [0, \infty)$. Figure 2 shows the syntax of our language.

Our expressions capture () (the only element of $\mathbb{1}$), r (real numbers), x (variables), $(e_1, \ldots, e_n)$ (tuples), $e[i]$ (accessing elements of tuples for $i \in \mathbb{N}$), $\ominus e$ (unary operators), $e_1 \oplus e_2$ (binary operators), $e_1[e_2]$ (accessing array elements), $e_1[e_2 \mapsto e_3]$ (updating array elements), **array**(e_1, e_2) (creating array of length e_1

$$
\begin{aligned}
e &::= () \mid r \mid x \mid (e_1, \dots, e_n) \mid e[i] \mid \ominus e \mid e_1 \oplus e_2 \mid e_1[e_2] \mid && \text{(Expressions)}\\
&\qquad \texttt{array}(e_1, e_2) \mid e_1[e_2 \mapsto e_3] \mid F(e)\\
F &::= \lambda x.\{P; \texttt{return } e;\} \mid \texttt{flip} \mid \texttt{uniform} \mid \texttt{sampleFrom}_f && \text{(Functions)}\\
P &::= \texttt{skip} \mid x := e \mid x = e \mid P_1; P_2 \mid \texttt{if } e\ \{P_1\} \texttt{ else } \{P_2\} \mid \{P\} \mid && \text{(Statements)}\\
&\qquad \texttt{assert}(e) \mid \texttt{observe}(e) \mid \texttt{while } e\ \{P\}
\end{aligned}
$$

Fig. 2. The syntax of our probabilistic language.

containing e_2 at every index) and $F(e)$ (evaluating function F on argument e). To handle functions $F(e_1, \dots, e_n)$ with multiple arguments, we interpret $(e_1, \dots, e_n)$ as a tuple and apply F to that tuple.

Our functions express $\lambda x.\{P; \texttt{return } e;\}$ (function taking argument x running P on x and returning e), $\texttt{flip}(e)$ (random choice from $\{0, 1\}$, 1 with probability e), $\texttt{uniform}(e_1, e_2)$ (continuous uniform distribution between e_1 and e_2) and $\texttt{sampleFrom}_f(e)$ (sample value distributed according to probability density function $f(e)$). An example for f is the density of the exponential distribution, indexed with rate λ. Formally, $f\colon (0, \infty) \to \mathbb{R} \to [0, \infty)$ is defined by $f(\lambda)(x) = \lambda e^{-\lambda x}$ if $x \geq 0$ and $f(\lambda)(x) = 0$ otherwise. Often, f is partial (e.g., $\lambda \leq 0$ is not allowed). Intuitively, arguments outside the allowed range of f produce the error state $\bot$.

Our statements express $\texttt{skip}$ (no operation), $x := e$ (assigning to a fresh variable), $x = e$ (assigning to an existing variable), $P_1; P_2$ (sequential composition of programs), $\texttt{if } e\ \{P_1\} \texttt{ else } \{P_2\}$ (if-then-else), $\{P\}$ (static scoping), $\texttt{assert}(e)$ (asserting that an expression evaluates to true, assertion failure results in $\bot$), $\texttt{observe}(e)$ (observing that an expression evaluates to true, observation failure results in $\lightning$) and $\texttt{while } e\ \{P\}$ (while loops, non-termination results in $\circlearrowleft$). We additionally introduce syntactic sugar $e_1[e_2] = e_3$ for $e_1 = e_1[e_2 \mapsto e_3]$, $\texttt{if } (e)\ \{P\}$ for $\texttt{if } e\ \{P\} \texttt{ else } \{\texttt{skip}\}$ and $\texttt{func}(e_2)$ for $\lambda x.\{P; \texttt{return } e_1;\}(e_2)$ (using the name $\texttt{func}$ for the function with argument x and body $\{P; \texttt{return } e_1\}$).

4.2 Typing Judgments

Let $n \in \mathbb{N}$. We define types by the following grammar in BNF, where $\tau[]$ denotes arrays over type τ. We sometimes write $\prod_{i=1}^{n} \tau_i$ for the product type $\tau_1 \times \cdots \times \tau_n$.

$$\tau ::= \mathbb{1} \mid \mathbb{R} \mid \tau[] \mid \tau_1 \times \cdots \times \tau_n$$

Note that we also use the type $\tau_1 \mapsto \tau_2$ of kernels with source τ_1 and target τ_2, but we do not list it here to avoid higher-order functions (discussed in Section 4.5).

Formally, a *context* Γ is a set $\{x_i \colon \tau_i\}_{i \in [n]}$ that assigns a type τ_i to each variable $x_i \in$ Vars. In slight abuse of notation, we sometimes write $x \in \Gamma$ if there is a type τ with $x \colon \tau \in \Gamma$. We also write $\Gamma, x \colon \tau$ for $\Gamma \cup \{x \colon \tau\}$ (where $x \notin \Gamma$) and Γ, Γ' for $\Gamma \cup \Gamma'$ (where Γ and Γ' have no common variables).

$$\frac{}{\Gamma \vdash ()\colon \mathbb{1}} \qquad \frac{}{\Gamma \vdash r\colon \mathbb{R}}\ r \in \mathbb{V} \qquad \frac{}{\Gamma \vdash x\colon \tau}\ x\colon \tau \in \Gamma \qquad \frac{\Gamma \vdash e_1\colon \tau_1 \quad \cdots \quad \Gamma \vdash e_n\colon \tau_n}{\Gamma \vdash (e_1, \dots, e_n)\colon \tau_1 \times \cdots \times \tau_n}$$

$$\frac{\Gamma \vdash e\colon \tau_0 \times \cdots \times \tau_{n-1}}{\Gamma \vdash e[i]\colon \tau_i}\ i \in \{0, \dots, n-1\} \qquad \frac{\Gamma \vdash e\colon \mathbb{R}}{\Gamma \vdash \ominus e\colon \mathbb{R}} \qquad \frac{\Gamma \vdash e_1\colon \mathbb{R} \quad \Gamma \vdash e_2\colon \mathbb{R}}{\Gamma \vdash e_1 \oplus e_2\colon \mathbb{R}}$$

$$\frac{\Gamma \vdash e_1\colon \tau[\,] \quad \Gamma \vdash e_2\colon \mathbb{R}}{\Gamma \vdash e_1[e_2]\colon \tau} \qquad \frac{\Gamma \vdash e_1\colon \mathbb{R} \quad \Gamma \vdash e_2\colon \tau}{\Gamma \vdash \texttt{array}(e_1, e_2)\colon \tau[\,]}$$

$$\frac{\Gamma \vdash e_1\colon \tau[\,] \quad \Gamma \vdash e_2\colon \mathbb{R} \quad \Gamma \vdash e_3\colon \tau}{\Gamma \vdash e_1[e_2 \mapsto e_3]\colon \tau[\,]} \qquad \frac{\Gamma \vdash e\colon \tau_1 \quad \vdash F\colon \tau_1 \mapsto \tau_2}{\Gamma \vdash F(e)\colon \tau_2}$$

$$\frac{x\colon \tau_1 \overset{P}{\rightsquigarrow} \Gamma \quad \Gamma \vdash e\colon \tau_2}{\vdash \lambda x.\{P; \texttt{return}\ e;\}\colon \tau_1 \mapsto \tau_2} \qquad \frac{}{\vdash \texttt{flip}\colon \mathbb{R} \mapsto \mathbb{R}} \qquad \frac{}{\vdash \texttt{uniform}\colon \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}}$$

$$\frac{}{\vdash \texttt{sampleFrom}_f\colon \tau \mapsto \mathbb{R}}\ f\colon A \to \mathbb{R} \to [0, \infty), A \in \Sigma_\tau$$

Fig. 3. The typing rules for expressions and functions in our language

$$\frac{}{\Gamma \overset{\texttt{skip}}{\rightsquigarrow} \Gamma} \qquad \frac{\Gamma \vdash e\colon \tau}{\Gamma \overset{x:=e}{\rightsquigarrow} \Gamma, x\colon \tau}\ x \notin \Gamma \qquad \frac{\Gamma \vdash e\colon \tau}{\Gamma \overset{x=e}{\rightsquigarrow} \Gamma}\ x\colon \tau \in \Gamma \qquad \frac{\Gamma \overset{P}{\rightsquigarrow} \Gamma' \quad \Gamma' \overset{Q}{\rightsquigarrow} \Gamma''}{\Gamma \overset{P;Q}{\rightsquigarrow} \Gamma''}$$

$$\frac{\Gamma \overset{P}{\rightsquigarrow} \Gamma'}{\Gamma \overset{\{P\}}{\rightsquigarrow} \Gamma} \qquad \frac{\Gamma \vdash e\colon \mathbb{R} \quad \Gamma \overset{P_1}{\rightsquigarrow} \Gamma' \quad \Gamma \overset{P_2}{\rightsquigarrow} \Gamma'}{\Gamma \overset{\texttt{if}\ e\,\{P_1\}\ \texttt{else}\ \{P_2\}}{\rightsquigarrow} \Gamma'} \qquad \frac{\Gamma \vdash e\colon \mathbb{R}}{\Gamma \overset{\texttt{assert}(e)}{\rightsquigarrow} \Gamma} \qquad \frac{\Gamma \vdash e\colon \mathbb{R}}{\Gamma \overset{\texttt{observe}(e)}{\rightsquigarrow} \Gamma}$$

$$\frac{\Gamma \vdash e\colon \mathbb{R} \quad \Gamma \overset{P}{\rightsquigarrow} \Gamma}{\Gamma \overset{\texttt{while}\ e\,\{P\}}{\rightsquigarrow} \Gamma}$$

Fig. 4. The typing rules for statements

The rules in Figures 3 and 4 allow deriving the type of expressions, functions and statements. To state that an expression e is of type τ under a context Γ, we write $\Gamma \vdash e\colon \tau$. Likewise, $\vdash F\colon \tau \mapsto \tau'$ indicates that F is a kernel from τ to τ'. Finally, $\Gamma \overset{P}{\rightsquigarrow} \Gamma'$ states that a context Γ is transformed to Γ' by a statement P. For $\texttt{sampleFrom}_f$, we intuitively want f to map values from τ to probability density functions. To allow f to be partial, i.e., to be undefined for some values from τ, we use $A \in \Sigma_\tau$ (and hence $A \subseteq [\![\tau]\!]$) as the domain of f (see Section 4.3).

4.3 Semantics

Semantic domains. We assign to each type τ a set $[\![\tau]\!]$ together with an implicit σ-algebra Σ_τ on that set. Additionally, we assign a set $[\![\Gamma]\!]$ to each context $\Gamma = \{x_i\colon \tau_i\}_{i \in [n]}$. Concretely, we have $[\![\mathbb{1}]\!] = \mathbb{1} := \{()\}$ with $\Sigma_{\mathbb{1}} = \{\emptyset, ()\}$, $[\![\mathbb{R}]\!] = \mathbb{R}$ and $\Sigma_{\mathbb{R}} = \mathcal{B}$. The remaining semantic domains are outlined in Figure 5.

$$
\begin{aligned}
[\![\tau[\,]]\!] &= \bigcup_{i\in\mathbb{N}} [\![\tau]\!]^i & \Sigma_{\tau[]} &\text{ is generated by } \bigcup_{i\in\mathbb{N}} \left\{ \prod_{j=1}^{i} S_j \;\middle|\; S_j \in \Sigma_\tau \right\} \\
[\![\tau_1 \times \cdots \times \tau_n]\!] &= \prod_{i=1}^{n} [\![\tau_i]\!] & \Sigma_{\tau_1\times\cdots\times\tau_n} &\text{ is generated by } \left\{ \prod_{i=1}^{n} S_i \;\middle|\; S_i \in \Sigma_{\tau_i} \right\} \\
[\![\Gamma]\!] &= \prod_{i=1}^{n} (x_i \colon [\![\tau_i]\!]) & \Sigma_\Gamma &\text{ is generated by } \left\{ \prod_{i=1}^{n} (x_i \colon S_i) \;\middle|\; S_i \in \Sigma_{\tau_i} \right\}
\end{aligned}
$$

Fig. 5. Semantic domains for types

$$
[\![()]\!]_{\mathbb{1}}(\sigma)(S) = [() \in S] \qquad [\![r]\!]_{\mathbb{R}}(\sigma)(S) = [r \in S] \qquad [\![x]\!]_\tau(\sigma)(S) = [\sigma[x] \in S]
$$

$$
[\![(e_1,\ldots,e_n)]\!]_{\tau_1\times\cdots\times\tau_n} = [\![e_1]\!]_{\tau_1} \overline{\times} \cdots \overline{\times} [\![e_n]\!]_{\tau_n} \qquad [\![e[i]]\!]_{\tau_i} = [\![e]\!]_{\tau_1\times\cdots\times\tau_n} \ggg \lambda t.\delta(t[i])
$$

$$
\begin{aligned}
[\![e_1/e_2]\!]_{\mathbb{R}} &= [\![e_1]\!]_{\mathbb{R}} \overline{\times} [\![e_2]\!]_{\mathbb{R}} \ggg \lambda(x,y).\begin{cases} \delta(x/y) & y \neq 0 \\ \delta(\bot) & y = 0 \end{cases} \\
[\![e_1[e_2]]\!]_\tau &= [\![e_1]\!]_{\tau[]} \overline{\times} [\![e_2]\!]_{\mathbb{R}} \ggg \lambda(t,i).\begin{cases} \delta(t[i]) & i \in \mathbb{N}, i < |t| \\ \delta(\bot) & \text{otherwise} \end{cases} \\
[\![e_1[e_2 \mapsto e_3]]\!]_{\tau[]} &= [\![e_1]\!]_{\tau[]} \overline{\times} [\![e_2]\!]_{\mathbb{R}} \overline{\times} [\![e_3]\!]_\tau \ggg \lambda(t,i,v).\begin{cases} \delta(t[i \mapsto v]) & i \in \mathbb{N}, i < |t| \\ \delta(\bot) & \text{otherwise} \end{cases} \\
[\![\mathsf{array}(e_1,e_2)]\!]_{\tau[]} &= [\![e_1]\!]_{\mathbb{R}} \overline{\times} [\![e_2]\!]_\tau \ggg \lambda(n,v).\begin{cases} \delta(v!n) & n \in \mathbb{N} \\ \delta(\bot) & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 6. The semantics of expressions. $v!n$ stands for the n-tuple $(v,\ldots,v)$. $t[i]$ stands for the i-th element (0-indexed) of the tuple t and $t[i \mapsto v]$ is the tuple t, where the i-th element is replaced by v. $|t|$ is the length of a tuple t. σ stands for a program state over all variables in some Γ, with $\sigma \in [\![\Gamma]\!]$.

Expressions. Figure 6 assigns to each expression e typed by $\Gamma \vdash e\colon \tau$ a probability kernel $[\![e]\!]_\tau \colon [\![\Gamma]\!] \mapsto \overline{[\![\tau]\!]}$. When τ is irrelevant or clear from the context, we may drop it and write $[\![e]\!]$. The formal interpretation of $[\![\Gamma]\!] \mapsto \overline{[\![\tau]\!]}$ is explained in Section 3.[3] Note that Figure 6 is incomplete, but extending it is straightforward. When we need to evaluate multiple terms (as in $(e_1,\ldots,e_n)$), we combine the results using $\overline{\times}$. This makes sure that in the presence of exceptions, the first exception that occurs will have priority over later exceptions. In addition, deterministic functions (like $x+y$) are lifted to probabilistic functions by the Dirac delta (e.g. $\delta(x+y)$) and incomplete functions (like x/y) are lifted to complete functions via the explicit error state $\bot$.

[3] As a quick and intuitive reminder, $\kappa\colon A \mapsto \overline{B}$ means that for every $a \in A$, $\kappa(a)$ will be a distribution over $\overline{B}$, where $\overline{B}$ is B enriched with exception states. Hence, $\kappa(a)$ may have weight on elements of B, on exception states, or on both.

$$\begin{aligned}
[\![\texttt{flip}]\!]_{\mathbb{R}\mapsto\mathbb{R}} &= \lambda p. \begin{cases} p\cdot\delta(1)+(1-p)\cdot\delta(0) & p\in[0,1] \\ \delta(\bot) & \text{otherwise} \end{cases} \\
[\![\texttt{uniform}]\!]_{\mathbb{R}\mapsto\mathbb{R}} &= \lambda(l,r). \begin{cases} \lambda S. \frac{1}{r-l}\lambda([l,r]\cap S) & l<r \\ \delta(\bot) & \text{otherwise} \end{cases} \\
[\![\texttt{sampleFrom}_f]\!]_{\tau\mapsto\mathbb{R}} &= \lambda p. \begin{cases} \lambda S. \int_{x\in\mathbb{R}\cap S} f(p)(x)\lambda(dx) & p\in A \\ \delta(\bot) & p\notin A \end{cases} \\
[\![\lambda x.\{P;\texttt{return}\ e;\}]\!]_{\tau_1\mapsto\tau_2} &= \lambda v.\delta(\{x\mapsto v\}) \ggg [\![P]\!] \ggg [\![e_2]\!]_{\tau_2}
\end{aligned}$$

Fig. 7. The semantics of functions.

$$[\![\texttt{skip}]\!] = \delta \qquad [\![x := e]\!] = [\![x = e]\!] = \delta\overline{\times}[\![e]\!] \ggg \lambda(\sigma,v).\delta(\sigma[x\mapsto v])$$

$$[\![P_1;P_2]\!] = [\![P_1]\!] \ggg [\![P_2]\!] \qquad [\![\{P\}]\!] = [\![P]\!] \ggg \lambda\sigma'.\delta(\sigma'(\Gamma))$$

$$[\![\texttt{if}\ e\ \{P_1\}\ \texttt{else}\ \{P_2\}]\!] = \delta\overline{\times}[\![e]\!]_{\mathbb{R}} \ggg \lambda(\sigma,b). \begin{cases} [\![P_1]\!](\sigma) & b\neq 0 \\ [\![P_2]\!](\sigma) & b=0 \end{cases}$$

$$[\![\texttt{assert}(e)]\!] = \delta\overline{\times}[\![e]\!]_{\mathbb{R}} \ggg \lambda(\sigma,b). \begin{cases} \delta(\sigma) & b\neq 0 \\ \delta(\bot) & b=0 \end{cases}$$

$$[\![\texttt{observe}(e)]\!] = \delta\overline{\times}[\![e]\!]_{\mathbb{R}} \ggg \lambda(\sigma,b). \begin{cases} \delta(\sigma) & b\neq 0 \\ \delta(\lightning) & b=0 \end{cases}$$

Fig. 8. The semantics of programs in our probabilistic language. Here, $\sigma[x\mapsto v]$ results in σ with the value stored under x updated to v. $\sigma'(\Gamma)$ selects only those variables from σ' that occur in Γ, meaning $\{x_i\mapsto v_i\}_{i\in\mathcal{I}}(\{x_i\colon\tau_i\}_{i\in\mathcal{I}'}) = \{x_i\mapsto v_i\}_{i\in\mathcal{I}\cap\mathcal{I}'}$.

Figure 7 assigns to each function F typed by $\vdash F\colon \tau_1\mapsto\tau_2$ a probability kernel $[\![F]\!]_{\tau_1\mapsto\tau_2}\colon [\![\tau_1]\!]\mapsto\overline{[\![\tau_2]\!]}$. In the semantics of **flip**, $\delta(1)\colon \Sigma_{\overline{\mathbb{R}}}\to[0,\infty]$ is a measure on $\overline{\mathbb{R}}$, and $p\cdot\delta(1)$ rescales this measure pointwise. Similarly, the sum $p\cdot\delta(1)+(1-p)\cdot\delta(0)$ is also meant pointwise, resulting in a measure on $\overline{\mathbb{R}}$. Finally, $\lambda p.\, p\cdot\delta(1)+(1-p)\cdot\delta(0)$ is a kernel with source $[0,1]$ and target $\overline{\mathbb{R}}$. For $\texttt{sampleFrom}_f(e)$, remember that $f(p)(\cdot)$ is a probability density function.

Statements. Figure 8 assigns to each statement P with $\Gamma \overset{P}{\rightsquigarrow} \Gamma'$ a probability kernel $[\![P]\!]\colon [\![\Gamma]\!]\mapsto\overline{[\![\Gamma']\!]}$. Note the use of $\overline{\times}$ in $\delta\overline{\times}[\![e]\!]$, which allows evaluating e while keeping the state σ in which e is being evaluated. Intuitively, if evaluating e results in an exception from $\mathcal{X}$, the previous state σ is irrelevant, and the result of $\delta\overline{\times}[\![e]\!]$ will be that exception from $\mathcal{X}$.

While loop. To define the semantics of the while loop **while** e $\{P\}$, we introduce a *kernel transformer* $[\![\texttt{while}\ e\ \{P\}]\!]^{\mathsf{trans}}\colon ([\![\Gamma]\!]\mapsto\overline{[\![\Gamma]\!]})\to([\![\Gamma]\!]\mapsto\overline{[\![\Gamma]\!]})$ that transforms the semantics for n runs of the loop to the semantics for $n+1$ runs of the loop. Concretely,

$$[\![\texttt{while}\ e\ \{P\}]\!]^{\mathsf{trans}}(\kappa) = \delta\overline{\times}[\![e]\!] \mathrel{>\!\!=\!\!=\!\!>} \lambda(\sigma,b).\begin{cases} [\![P]\!](\sigma) \gg= \kappa & b \neq 0 \\ \delta(\sigma) & b = 0 \end{cases}$$

This semantics first evaluates e, while keeping the program state around using δ. If e evaluates to 0, the while loop terminates and we return the current program state σ. If e does not evaluate to 0, we run the loop body P and feed the result to the next iteration of the loop, using κ.

We can then define the semantics of **while** e $\{P\}$ using a special fixed point operator $\mathsf{fix}\colon ((A \mapsto \overline{A}) \to (A \mapsto \overline{A})) \to (A \mapsto \overline{A})$, defined by the pointwise limit $\mathsf{fix}(\Delta) = \lim_{n\to\infty} \Delta^n(\circlearrowleft)$, where $\circlearrowleft := \lambda\sigma.\,\delta(\circlearrowleft)$ and Δ^n denotes the n-fold composition of Δ. $\Delta^n(\circlearrowleft)$ puts all runs of the while loop that do not terminate within n steps into the state $\circlearrowleft$. In the limit, $\circlearrowleft$ only has weight on those runs of the loop that never terminate. $\mathsf{fix}(\Delta)$ is only defined if its pointwise limit exists. Making use of fix, we can define the semantics of the while loop as follows:

$$[\![\texttt{while}\ e\ \{P\}]\!] = \mathsf{fix}\Big([\![\texttt{while}\ e\ \{P\}]\!]^{\mathsf{trans}}\Big)$$

Lemma 7. *For Δ as in the semantics of the while loop, and for each σ and each S, the limit $\lim_{n\to\infty} \Delta^n(\circlearrowleft)(\sigma)(S)$ exists.*

Lemma 7 holds because increasing n may only shift probability mass from $\circlearrowleft$ to other states (we provide a formal proof in Appendix B). Kozen shows a different way of defining the semantics of the while loop [23], using least fixed points. Lemma 8 describes the relation of the semantics of our while loop to the semantics of the while loop of [23]. For more details on the formal interpretation of Lemma 8 and for its proof, see Appendix B.

Lemma 8. *In the absence of exception states, and using sub-probability kernels instead of distribution transformers, the definition of the semantics of the while loop from [23] is equivalent to ours.*

Theorem 2. *The semantics of each expression $[\![e]\!]$ and statement $[\![P]\!]$ is indeed a probability kernel.*

Proof. The proof proceeds by induction. Some lemmas that are crucial for the proof are listed in Appendix C. Conveniently, most functions that come up in our definition are continuous (like $a+b$) or continuous except on some countable subset (like $\frac{a}{b}$) and thus measurable.

4.4 Recursion

To extend our language with recursion, we apply the same ideas as for the while loop. Given the source code of a function F that uses recursion, we define its semantics in terms of a kernel transformer $[\![F]\!]^{\mathsf{trans}}$. This kernel transformer takes

$$\delta\overline{\times}\left[\!\!\left[!\texttt{flip}\left(\frac{1}{2}\right)\right]\!\!\right] >\!\!=\!\!> \lambda(\sigma, b).\begin{cases}\Big(\kappa\overline{\times}[\![1]\!] >\!\!=\!\!> \lambda(x,y).\,\delta\big(x+y\big)\Big)(\sigma) & b \neq 0\\ [\![0]\!](\sigma) & b = 0\end{cases}$$

Fig. 9. Kernel transformer $[\![\texttt{geom}]\!]^{\mathsf{trans}}(\kappa)$ for `geom` given in Listing 11.

semantics for F up to a recursion depth of n and returns semantics for F up to recursion depth $n+1$. Formally, $[\![F]\!]^{\mathsf{trans}}(\kappa)$ follows the usual semantics, but uses κ as the semantics for recursive calls to F (we will provide an example shortly). Finally, we define the semantics of F by $[\![F]\!] := \mathsf{fix}\Big([\![F]\!]^{\mathsf{trans}}\Big)$. Just as for the while loop, $\mathsf{fix}\Big([\![F]\!]^{\mathsf{trans}}\Big)$ is well-defined because stepping from recursion depth n to $n+1$ can only shift probability mass from $\circlearrowleft$ to other states. We note that we could generalize our approach to mutual recursion.

To demonstrate how we define the kernel transformer, consider the recursive implementation of the geometric distribution in Listing 11 (to simplify presentation, Listing 11 uses early return). Given semantics κ for `geom` : $\mathbb{1} \mapsto \mathbb{R}$ up to recursion depth n, we can define the semantics of `geom` up to recursion depth $n+1$, as illustrated in Figure 9.

```
geom(){
  if !flip(1/2){
    return geom()+1;
  }else{
    return 0;
  }
}
```

Listing 11. Geometric distribution

4.5 Higher-order Functions

Our language cannot express higher-order functions. When trying to give semantics to higher-order probabilistic programs, an important step is to define a σ-algebra on the set of functions from real numbers to real numbers. Unfortunately, no matter which σ-algebra is picked, function evaluation (i.e. the function that takes f and x as arguments and returns $f(x)$) is not measurable [1]. This is a known limitation that previous work has looked into (e.g. [35] address it by restricting the set of functions to those expressible by their source code).

A promising recent approach is replacing measurable spaces by quasi-Borel spaces [16]. This allows expressing higher-order functions, at the price of replacing the well-known and well-understood measurable spaces by a new concept.

4.6 Non-determinism

To extend our language with non-determinism, we may define the semantics of expressions, functions and statements in terms of sets of kernels. For an expression e typed by $\Gamma \vdash e : \tau$, this means that $[\![e]\!]_\tau \in \mathcal{P}\left([\![\Gamma]\!] \mapsto [\![\tau]\!]\right)$, where $\mathcal{P}(S)$ denotes the power set of S. Lifting our semantics to non-determinism is mostly straightforward, except for loops. There, $[\![\textbf{while}\ e\ \{P\}]\!]$ contains all kernels of the form $\lim_{n\to\infty}(\Delta_1 \circ \cdots \circ \Delta_n)(\circlearrowleft)$, where $\Delta_i \in [\![\textbf{while}\ e\ \{P\}]\!]^{\mathsf{trans}}$. Previous work has studied non-determinism in more detail, see e.g. [21,22].

5 Properties of Semantics

We now investigate two properties of our semantics: commutativity and associativity. These are useful in practice, e.g. because they enable rewriting programs to a form that allows for more efficient inference [5].

In this section, we write $e_1 \simeq e_2$ when expressions e_1 and e_2 are equivalent (i.e. when $[\![e_1]\!] = [\![e_2]\!]$). Analogously, we write $P_1 \simeq P_2$ for $[\![P_1]\!] = [\![P_2]\!]$.

5.1 Commutativity

In the presence of exception states, our language cannot guarantee commutativity of expressions such as $e_1 + e_2$. This is not surprising, as in our semantics the first exception bypasses all later exceptions.

Lemma 9. *For function* $F()\{\texttt{while}\ 1\ \{\texttt{skip}\}; \texttt{return}\ 0\}$,

$$\frac{1}{0} + F() \not\simeq F() + \frac{1}{0}$$

Formally, this is because if we evaluate $\frac{1}{0}$ first, we only have weight on $\bot$. If instead, we evaluate $F()$ first, we only have weight on $\circlearrowleft$, by an analogous calculation. A more detailed proof is included in Appendix D.

However, the only reason for non-commutativity is the presence of exceptions. Assuming that e_1 and e_2 cannot produce exceptions, we obtain commutativity:

Lemma 10. *If* $[\![e_1]\!](\sigma)(\mathcal{X}) = [\![e_2]\!](\sigma)(\mathcal{X}) = 0$ *for all* σ*, then* $e_1 \oplus e_2 \simeq e_2 \oplus e_1$*, for any commutative operator* $\oplus$.

The proof of Lemma 10 (provided in Appendix D) relies on the absence of exceptions and Fubini's Theorem. This commutativity result is in line with the results from [34], which proves commutativity in the absence of exceptions.

In the analogous situation for statements, we cannot assume commutativity $P_1; P_2 \simeq P_2; P_1$, even if there is no dataflow from P_1 to P_2. We already illustrated this in Listing 10, where swapping two lines changes the program semantics. However, in the absence of exceptions and dataflow from P_1 to P_2, we can guarantee $P_1; P_2 \simeq P_2; P_1$.

5.2 Associativity

A careful reader might suspect that since commutativity does not always hold in the presence of exceptions, a similar situation might arise for associativity of some expressions. As an example, can we guarantee $e_1 + (e_2 + e_3) \simeq (e_1 + e_2) + e_3$, even in the presence of exceptions? The answer is yes, intuitively because exceptions can only change the behavior of a program if the order of their occurrence is changed. This is not the case for associativity. Formally, we derive the following:

Lemma 11. $e_1 \oplus (e_2 \oplus e_3) \simeq (e_1 \oplus e_2) \oplus e_3$*, for any associative operator* $\oplus$.

We include notes on the proof of Lemma 11 in Appendix D, mainly relying on the associativity of $\overline{\times}$ (Lemma 6). Likewise, sequential composition is associative: $(P_1; P_2); P_3 \simeq P_1; (P_2; P_3)$. This is due to the associativity of $>\!\!=\!\!>$ (Lemma 5).

5.3 Adding the `score` Primitive

Some languages include the primitive **score**, which allows to increase or decrease the probability of a certain event (or trace) [34,35].

Listing 12 shows an example program using **score**. Without normalization, it returns 0 with probability $\frac{1}{2}$ and 1 with "probability" $\frac{1}{2} \cdot 2 = 1$. After normalization, it returns 0 with probability $\frac{1}{3}$ and 1 with probability $\frac{2}{3}$. Because **score** allows decreasing the probability of a specific event, it renders **observe** unnecessary. In general, we can replace **observe**(e) by **score**$(e \neq 0)$. However, performing this replacement means losing the explicit knowledge of the weight on $\lightning$.

```
x:=flip(½);
if x=1 {
  score(2);
}
return x;
```

Listing 12. Using **score**

score can be useful to modify the shape of a given distribution. For example, Listing 13 turns the distribution of x, which is a Gaussian distribution, into the Lebesgue measure λ, by multiplying the density of x by its inverse. Hence, the density of x at any location is 1. Note that the distribution over x cannot be described by a probability measure, because e.g. the "probability" that x lies in the interval $[0, 2]$ is 2.

```
x:=gauss(0,1);
score($\sqrt{2\pi}e^{x^2/2}$);
return x;
```

Listing 13. Reshaping a distribution.

Unfortunately, termination in the presence of **score** is not well-defined, as illustrated in Listing 14. In this program, the only non-terminating trace keeps changing its weight, switching between 1 and 2. In the limit, it is impossible to determine the weight of non-termination.

```
i:=0;
while 1 {
  if i=0 {
    score(2);
  }else{
    score(½);
  }
  i=1-i;
}
```

Listing 14. **score** vs non-termination

Hence, allowing the use of the **score** primitive only makes sense after abolishing the tracking of non-termination ($\circlearrowleft$), which can be achieved by only measuring sets that do not contain non-termination. Formally, this means restricting the semantics of expressions e typed by $\Gamma \vdash e : \tau$ to $[\![e]\!]_\tau : \Gamma \mapsto \left([\![\tau]\!] - \{\circlearrowleft\}\right)$. Intuitively, abolishing non-termination means that we ignore non-terminating runs (these result in weight on non-termination). After doing this, we can give well-defined semantics to the **score** primitive.

The typing rule and semantics of **score** are:

$$\frac{\Gamma \vdash e : \mathbb{R}}{\Gamma \xrightarrow{\texttt{score}(e)} \Gamma} \qquad \text{and} \qquad [\![\texttt{score}(e)]\!] = \delta \overline{\times} [\![e]\!]_{\mathbb{R}} \ggg \lambda(\sigma, c).c * \delta(\sigma)$$

After including **score** into our language, the semantics of the language can no longer be expressed in terms of probability kernels as stated in Theorem 2, because the probability of any event can be inflated beyond 1. Instead, the semantics must be expressed in terms of s-finite kernels.

Theorem 3. *After adding the* **score** *primitive and abolishing non-termination, the semantics of each expression* $[\![e]\!]$ *and statement* $[\![P]\!]$ *is an s-finite kernel.*

Proof. As for Theorem 2, the proof proceeds by induction. Most parts of the proof are analogous (e.g. $\ggg$ preserves s-finite kernels instead of probability

Work	Language	Semantics	Typed	Higher-order	Loops	Constraints
We	Imperative	Probability kernels	Typed	First-order	Loops (FP)	Yes
[4]	Functional	Sub-probability kernels	Untyped	Higher-order	Recursion (FP)	Yes
[23]	Imperative	Distribution transformers	Limited	First-order	Loops (LFP)	No
[24]	Imperative	Sub-probability kernels	Limited	First-order	Loops (LFP)	Yes
[28]	Imperative	weakest precondition	Untyped	First-order	Loops (LFP)	Yes
[33]	Declarative	Probability kernels	Limited	First-order	Loops (LFP)	No
[34]	Functional	s-finite kernels	Typed	First-order	Counting measure	**score**(x)
[35]	Functional	Measurable functions	Typed	Higher-order	No	**score**(x)

Table 3. Comparison of existing semantics to ours. When adding **score** to our language (Section 5.3), our semantics use s-finite kernels (not probability kernels).

kernels). For while loops, the limit still exists (Lemma 7 still holds), but it is not bounded from above anymore. The limit indeed corresponds to an s-finite kernel because the limit of strictly increasing s-finite kernels is an s-finite kernel.

In the presence of **score**, we can still talk about the interaction of different exceptions, assuming that we do track different types of exceptions (e.g. division by zero and out of bounds access of arrays). Then, we keep the commutativity and associativity properties studied in the previous sections, because these still hold for s-finite kernels.

```
score(2);
assert(false);
```

Listing 15. Interaction of **score** and **assert**

Listing 15 shows an interaction of **score** with **assert**. As one would expect, our semantics will assign weight 2 to $\perp$ in this case. If the two statements are switched, our semantics will ignore **score**(2) and assign weight 1 to $\perp$. Hence again, commutativity does not hold.

Listing 16 shows a program that keeps increasing the probability of an error. In every loop iteration, there is a "probability" of 1 of running into an error. Overall, Listing 16 results in weight ∞ on state $\perp$.

```
while 1 {
  score(2);
  assert(flip(1/2));
}
```

Listing 16. Interaction of **score**, **assert** and loops

6 Related Work

Kozen provides classic semantics to probabilistic programs [23]. We follow his main ideas, but deviate in some aspects in order to introduce additional features or to make our presentation cleaner. The semantics by Hur et al. [19] is heavily based on [23], so we do not go into more detail here. Table 3 summarizes the comparison of our approach to that of others.

Kernels. Like our work, most modern approaches use kernels (i.e., functions from values to distributions) to provide semantics to probabilistic programs [4,24,33,34]. Borgström et al. [4] use sub-probability kernels on (symbolic) expressions. Staton [34] uses s-finite kernels to capture the semantics of the **score** primitive (when we discuss **score** in Section 5.3, we do the same).

In the classic semantics of [23], Kozen uses distribution transformers (i.e., functions from distributions to distributions). For later work [24], Kozen also switches to sub-probability kernels, which has the advantage of avoiding redundancies. A different approach uses weakest precondition to define the semantics, as in [28]. Staton et al. [35] use a different concept of measurable functions $A \to P(\mathbb{R}_{\geq 0} \times B)$ (where $P(S)$ denotes the set of all probability measures on S).

Typing. Some probabilistic languages are untyped [4,28], while others are limited to just a single type: $\mathbb{R}^n$ [23,24] or $\bigcup_{i=1}^{\infty} \mathbb{N}^i \cup \mathbb{N}^{\infty}$ [33]. Some languages provide more interesting types including sum types, distribution types and tuples [34,35]. We allow tuples and array types, and we could easily account for sum types.

Loops. Because the semantics of while loops is not always straightforward, some languages avoid while loops and recursion altogether [35]. Borgström et al. handle recursion instead of while loops, defining the semantics in terms of a fixed point [4]. Many languages handle while loops by least fixed points [23,24,28,33]. Staton defines while loops in terms of the counting measure [34], which is similar to defining them by a fixed point. We define the semantics of while loops in terms of a fixed point, which avoids the need to prove the least fixed point exists (still, the classic while loop semantics of [23] and our formulation are equivalent).

Most languages do not explicitly track non-termination, but lose probability weight by non-termination [4,23,24,34]. This missing weight can be used to identify the probability of non-termination, but only if other exceptions (such as **`fail`** in [24] or observation failure in [4]) do not also result in missing weight. The semantics of [33] are tailored to applications in networks and lose *non-terminating packet histories* instead of weight (due to a particular least fixed point construction of Scott-continuous maps on algebraic and continuous directed complete partial orders). Some works define non-termination as missing weight in the weakest precondition [28]. Specifically, the semantics in [28] can also explicitly express probability of non-termination *or* ending up in some state (using the separate construct of a weakest liberal precondition). We model non-termination by an explicit state $\circlearrowleft$, which has the advantage that in the context of lost weight, we know what part of that lost weight is due to non-termination.

Kaminski et al. [21] investigate the run-time of probabilistic program with loops and **`fail`** (interpreted as early termination), but without observations. In [21], non-termination corresponds to an infinite run-time.

Error states. Many languages do not consider partial functions (like fractions $\frac{a}{b}$) and thus never run into an exception state [23,24,33]. Olmedo et al. [28] do not consider partial functions, but support the related concept of an explicit **`abort`**. The semantics of **`abort`** relies on missing weight in the final distribution. Some languages handle expressions whose evaluation may fail using sum types [34,35], forcing the programmer to deal with errors explicitly (we discuss the disadvantages of this approach at Listing 6). Formally, a sum type $A + B$ is a disjoint union of the two sets A and B. Defining the semantics of an expression in terms of the sum type $A + \{\perp\}$ allows that expression to evaluate to *either* a value

$a \in A$ *or* to $\bot$. Borgström et al. [4] have a single state **`fail`** expressing exceptions such as dynamically detected type errors (without forcing the programmer to deal with exceptions explicitly). Our semantics also uses sum types to handle exceptions, but the handling is implicit, by defining semantics in terms of $(\gg\!\!=\!\!>)$ (which defines how exceptions propagate in a program) instead of (;).

Constraints. To enforce hard constraints, we use the **`observe`**(e) statement, which puts the program into a special failure state $\lightning$ if it does not satisfy e. We can encode soft constraints by **`observe`**(e), where e is probabilistic (this is a general technique). Borgström et al. [4] allow both soft constraints that reduce the probability of some program traces and hard constraints whose failure leads to the error state **`fail`**. Some languages can handle generalized soft constraints: they can not only decrease the probability of certain traces using soft constraints, but also increase them, using **`score`**(x) [34,35]. We investigate the consequences of adding **`score`** to our language in Section 5.3. Kozen [24] handles hard (and hence soft) constraints using **`fail`** (which results in a sub-probability distribution). Some languages can handle neither hard nor soft constraints [23,33]. Note though that the semantics of ProbNetKAT in [33] can drop certain packages, which is a similar behavior. Olmedo et al. [28] handle hard (and hence soft) constraints by a conditional weakest precondition that tracks both the probability of not failing any observation and the probability of ending in specific states. Unfortunately, this work is restricted to discrete distributions and is specifically designed to handle observation failures and non-termination. Thus, it is not obvious how to adapt the semantics if a different kind of exception is to be added.

Interaction of different exception. Most existing work handles at least some exceptions by sub-probability distributions [4,23,24,33,34]. Then, any missing weight in the final distribution must be due to exceptions. However, this leads to a conflation of all exceptions handled by sub-probability distributions (for the consequences of this, see, e.g., our discussion of Listing 8). Note that semantics based on sub-probability kernels can add more exceptions, but they will simply be conflated with all other exceptions.

Some previous work does not (exclusively) rely on sub-probability distributions. Borgström et al. [4] handle errors implicitly, but still use sub-probability kernels to handle non-termination and **`score`**. Olmedo et al. can distinguish non-termination (which is conflated with exception failure) from failing observations by introducing two separate semantic primitives (conditional weakest precondition and conditional liberal weakest precondition) [28]. Because their solution specifically addresses non-termination, it is non-trivial to generalize this treatment to more than two exception states. By using sum types, some semantics avoid interactions of errors with non-termination or constraint failures, but still cannot distinguish the latter [34,35]. Note that semantics based on sum types can easily add more exceptions (although it is impossible to add non-termination). However, the interaction of different exceptions cannot be observed, because the programmer has to handle exceptions explicitly.

To the best of our knowledge, we are the first to give formal semantics to programs that may produce exceptions in this generality. One work investigates assertions in probabilistic programs, but explicitly disallows non-terminating loops [32]. Moreover, the semantics in [32] are operational, leaving the distribution (in terms of measure theory) of program outputs unclear. Cho et al. [8] investigate the interaction of partial programs and observe, but are restricted to discrete distributions and to only two exception states. In addition, this investigation treats these two exception states differently, making it non-trivial to extend the results to three or more exception states. Katoen et al. [22] investigate the intuitive problems when combining non-termination and observations, but restrict their discussions to discrete distributions and do not provide formal semantics. Huang [17] treats partial functions, but not different kinds of exceptions. In general, we know of no probabilistic programming language that distinguishes more than two different kinds of exceptions. Distinguishing two kinds of exceptions is simpler than three, because it is possible to handle one exception as an explicit exception state and the other one by missing weight (as e.g. in [4]).

Cousot and Monerau [9] provide a trace semantics that captures probabilistic behavior by an explicit randomness source given to the program as an argument. This allows handling non-termination by non-terminating traces. While the work does not discuss errors or observation failure, it is possible to add both. However, using an explicit randomness source has other disadvantages, already discussed by Kozen [23]. Most notably, this approach requires a distribution over the randomness source and a translation from the randomness source to random choices in the program, even though we only care about the distribution of the latter.

7 Conclusion

In this work we presented an expressive probabilistic programming language that supports important features such as mixing continuous and discrete distributions, arrays, observations, partial functions and while-loops. Unlike prior work, our semantics distinguishes non-termination, observation failures and error states. This allows us to investigate the subtle interaction of different exceptions, which is not possible for semantics that conflate different kinds of exceptions. Our investigation confirms the intuitive understanding of the interaction of exceptions presented in Section 2. However, it also shows that some desirable properties, like commutativity, only hold in the absence of exceptions. This situation is unavoidable, and largely analogous to the situation in deterministic languages.

Even though our semantics only distinguish three exception states, it can be trivially extended to handle any countable set of exception states. This allows for an even finer-grained distinction of e.g. division by zero, out of bounds array accesses or casting failures (in a language that allows type casting). Our semantics also allows enriching exceptions with the line number that the exception originated from (of course, this is not possible for non-termination). For an uncountable set of exception states, an extension is possible but not trivial.

References

1. R. J. Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614–630, 1961.
2. G. Barthe, B. Grégoire, J. Hsu, and P.-Y. Strub. Coupling proofs are probabilistic product programs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 161–174, New York, NY, USA, 2017. ACM.
3. G. Barthe, B. Köpf, F. Olmedo, and S. Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 97–110, New York, NY, USA, 2012. ACM.
4. J. Borgström, U. Dal Lago, A. D. Gordon, and M. Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 33–46, New York, NY, USA, 2016. ACM.
5. A. Chaganty, A. Nori, and S. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *Artificial Intelligence and Statistics*, pages 153–160, 2013.
6. D. Chauveau and J. Diebolt. An automated stopping rule for mcmc convergence assessment. *Computational Statistics*, 3(14):419–442, 1999.
7. S. Cheng. A crash course on the lebesgue integral and measure theory, 2008.
8. K. Cho and B. Jacobs. Kleisli semantics for conditioning in probabilistic programming. 2017.
9. P. Cousot and M. Monerau. Probabilistic abstract interpretation. In *European Symposium on Programming*, pages 169–193. Springer, 2012.
10. T. Gehr, S. Misailovic, and M. Vechev. Psi: Exact symbolic inference for probabilistic programs. In *International Conference on Computer Aided Verification*, pages 62–83. Springer, 2016.
11. A. Gelman, D. Lee, and J. Guo. Stan a probabilistic programming language for bayesian inference and optimization. *Journal of Educational and Behavioral Statistics*, 2015.
12. M. Giry. A categorical approach to probability theory. In *Categorical aspects of topology and analysis*, pages 68–85. Springer, 1982.
13. N. D. Goodman, V. K. Mansinghka, D. M. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: A language for generative models. In *In UAI*, pages 220–229, 2008.
14. N. D. Goodman and A. Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2017-5-15.
15. A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, 2014.
16. C. Heunen, O. Kammar, S. Staton, and H. Yang. A convenient category for higher-order probability theory. *CoRR*, abs/1701.02547, 2017.
17. D. E. Huang. On programming languages for probabilistic modeling. `https://danehuang.github.io/papers/dissertation.pdf`, 2017. Accessed: 2017-06-28.
18. C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. Slicing probabilistic programs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 133–144, New York, NY, USA, 2014. ACM.
19. C.-K. Hur, A. V. Nori, S. K. Rajamani, and S. Samuel. A provably correct sampler for probabilistic programs. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 45. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

20. B. L. Kaminski and J.-P. Katoen. On the hardness of almost–sure termination. In *International Symposium on Mathematical Foundations of Computer Science*, pages 307–318. Springer, 2015.
21. B. L. Kaminski, J.-P. Katoen, C. Matheja, and F. Olmedo. Weakest precondition reasoning for expected run–times of probabilistic programs. In *European Symposium on Programming Languages and Systems*, pages 364–389. Springer, 2016.
22. J.-P. Katoen, F. Gretz, N. Jansen, B. L. Kaminski, and F. Olmedo. Understanding probabilistic programs. In *Correct System Design*, pages 15–32. Springer, 2015.
23. D. Kozen. Semantics of probabilistic programs. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, SFCS '79, pages 101–114, Washington, DC, USA, 1979. IEEE Computer Society.
24. D. Kozen. A probabilistic pdl. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 291–297, New York, NY, USA, 1983. ACM.
25. V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *ArXiv e-prints*, Mar. 2014.
26. T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.5, 2013. Microsoft Research Cambridge. http://research.microsoft.com/infernet.
27. P. Narayanan, J. Carette, W. Romano, C.-c. Shan, and R. Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *International Symposium on Functional and Logic Programming*, pages 62–79. Springer, 2016.
28. F. Olmedo, F. Gretz, N. Jansen, B. L. Kaminski, J.-P. Katoen, and A. McIver. Conditioning in probabilistic programming. *ACM Transactions on Programming Languages and Systems*, 2018 (to appear).
29. B. Paige and F. Wood. A compilation target for probabilistic programming languages. In *International Conference on Machine Learning*, pages 1935–1943, 2014.
30. D. Pollard. *A user's guide to measure theoretic probability*, volume 8. Cambridge University Press, 2002.
31. W. Rudin. *Real and complex analysis*. Tata McGraw-Hill Education, 1987.
32. A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. *ACM SIGPLAN Notices*, 49(6):112–122, 2014.
33. S. Smolka, P. Kumar, N. Foster, D. Kozen, and A. Silva. Cantor meets scott: Semantic foundations for probabilistic networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 557–571, New York, NY, USA, 2017. ACM.
34. S. Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, pages 855–879. Springer, 2017.
35. S. Staton, H. Yang, F. Wood, C. Heunen, and O. Kammar. Semantics for probabilistic programming: Higher-order functions, continuous distributions, and soft constraints. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 525–534, New York, NY, USA, 2016. ACM.
36. F. Wood, J. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. *CoRR*, abs/1507.00996, 2015.

A Proofs for preliminaries

In this section, we provide lemmas, proofs and some definitions that were left out or cut short in Section 3. For a more detailed introduction into measure theory, we recommend the book *A crash course on the Lebesgue integral and measure theory* [7].

A.1 Measures

Definition 3. *Let (A, Σ_A) be a measurable space and $\mu\colon \Sigma_A \to [0,\infty]$ a measure on A.*

- *We call μ s-finite if μ can be written as a countable sum $\sum_{i\in\mathbb{N}} \mu_i$ of sub-probability measures μ_i.*
- *We call μ σ-finite if $A = \bigcup_{i\in\mathbb{N}} A_i$ for $A_i \in \Sigma_A$, with $\mu(A_i) < \infty$.*
- *We call μ finite if $\mu(A) < \infty$.*
- *We call μ a sub-probability measure if $\mu(A) \leq 1$.*
- *We call μ a probability measure if $\mu(A) = 1$.*

Note that for a σ-finite measure μ, $\mu(A) = \infty$ is possible, even though $\mu(A_i) < \infty$ for all i. As an example, the Lebesgue measure is σ-finite because $\mathbb{R} = \bigcup_{i\in\mathbb{N}}[-i,i]$ with $\lambda([-i,i]) = 2 * i$, but $\lambda(\mathbb{R}) = \infty$.

Lemma 12. *The following definition of s-finite measures is equivalent to our definition of s-finite measures (the difference is that the μ_is are only required to be finite):*
We call $\mu\colon \Sigma_A \to [0,\infty]$ an s-finite measure if it can be written as $\mu = \sum_{i\in\mathbb{N}} \mu_i$ for finite measures $\mu_i\colon \Sigma_A \to [0,\infty]$.

Proof. Since any sub-probability measure is finite, one direction is trivial. For the other direction, let $\mu = \sum_{i\in\mathbb{N}} \mu'_i$ for finite measures μ'_i. Obviously, $\mu \geq 0$, $\mu(\emptyset) = 0$ and $\mu(\bigcup_{i\in\mathbb{N}} A_i) = \sum_{i\in\mathbb{N}} A_i$ for mutually disjoint $A_i \in \Sigma_A$, so μ is a measure. To show that μ can be written as a sum of sub-probability measures, let $n_i := \lceil \mu'_i(A) \rceil$. Then, $\mu = \sum_{i\in\mathbb{N}} \mu'_i = \sum_{i\in\mathbb{N}} \frac{n_i}{n_i}\mu'_i = \sum_{i\in\mathbb{N}} \sum_{j\in[n_i]} \frac{1}{n_i}\mu'_i$. We let $\mu_i := \frac{1}{n_i}\mu'_i \leq 1$.

Lemma 13. *Any σ-finite measure $\mu\colon \Sigma_A \to [0,\infty]$ is s-finite.*

Proof. Since μ is σ-finite, $A = \bigcup_{i\in\mathbb{N}} A_i$ with $A_i \in \Sigma_A$ and $\mu(A_i) < \infty$. Without loss of generality, assume that the A_i form a partition of A. Then, $\mu(S) = \sum_{i\in\mathbb{N}} \mu(S \cap A_i)$, with $\mu(\cdot \cap A_i) < \infty$. Thus, μ is a countable sum of finite measures.

Definition 4. *The counting measure $c\colon \mathcal{B} \to [0,\infty]$ is defined by*

$$c(S) = \begin{cases} |S| & S \text{ finite} \\ \infty & \text{otherwise} \end{cases}$$

Definition 5. *The infinity measure* $\mu\colon \mathcal{B} \to [0,\infty]$ *is defined by*

$$\mu(S) = \begin{cases} 0 & S = \emptyset \\ \infty & \textit{otherwise} \end{cases}$$

Lemma 14. *Neither the counting measure nor the infinity measure are s-finite.*

Proof. For the counting measure c, assume (toward a contradiction) $c = \sum_{i\in\mathbb{N}} c_i$. We have $\mathbb{R} = \{r \in \mathbb{R} \mid c(\{r\}) > 0\} = \bigcup_{i\in\mathbb{N}}\{r \in \mathbb{R} \mid c_i(\{r\}) > 0\} = \bigcup_{i\in\mathbb{N}}\bigcup_{n\in\mathbb{N}}\{r \in \mathbb{R} \mid c_i(\{r\}) > \frac{1}{n}\}$. Because $\mathbb{R}$ is uncountable, there must be $i, n \in \mathbb{N}$ for which $S := \{r \in \mathbb{R} \mid c_i(\{r\}) > \frac{1}{n}\}$ is uncountable. Thus for any measurable, countably infinite $S' \subseteq S$, $c_i(S') = \infty$, which means that c_i is not finite. Proceed analogously for the infinity measure.

Lemma 15. *The measure* $\mu : \mathcal{B} \to [0,\infty]$ *with* $\mu(S) = \left\{\begin{matrix} 0 & \lambda(S) = 0 \\ \infty & \lambda(S) > 0 \end{matrix}\right\}$ *is s-finite but not* σ*-finite.*

Proof. $\mu = \sum_{i\in\mathbb{N}} \lambda$, and λ is s-finite, so μ is s-finite. Assume (toward a contradiction) that μ is σ-finite. Then $\mathbb{R} = \bigcup_{i\in\mathbb{N}} A_i$ with $A_i \in \mathcal{B}$ and $\mu(A_i) < \infty$. Thus, $\mu(A_i) = 0$ and hence $\mu(\mathbb{R}) = \mu(\bigcup_{i\in\mathbb{N}} A_i) \le \sum_{i\in\mathbb{N}} \mu(A_i) = 0$, a contradiction.

Lemma 16.

$$\begin{aligned} \forall S \in \Sigma_{A\times B}\colon (\mu \times \mu')(S) &= \int_{a\in A} \mu'(\{b \in B \mid (a,b) \in S\})\mu(da) \\ &= \int_{b\in B} \mu(\{a \in A \mid (a,b) \in S\})\mu'(db) \end{aligned}$$

$$\begin{aligned} \forall S \in \Sigma_{\overline{A\times B}}\colon (\mu \overline{\times} \mu')(S) &= \int_{a\in\overline{A}} \mu'(\{b \in \overline{B} \mid \overline{(a,b)} \in S\})\mu(da) \\ &= \int_{b\in\overline{B}} \mu(\{a \in \overline{A} \mid \overline{(a,b)} \in S\})\mu'(db) \end{aligned}$$

Proof.

$$\begin{aligned} (\mu \times \mu')(S) &= \int\limits_{a\in A}\int\limits_{b\in B} [(a,b) \in S]\mu'(db)\mu(da) \\ &= \int\limits_{a\in A}\int\limits_{b\in B} [b \in \{b' \in B \mid (a,b') \in S\}]\mu'(db)\mu(da) \\ &= \int\limits_{a\in A} \mu'(\{b' \in B \mid (a,b') \in S\})\mu(da) \end{aligned}$$

$$\begin{aligned}(\mu \times \mu')(S) &= \int\limits_{a\in A}\int\limits_{b\in B} [(a,b)\in S]\mu'(db)\mu(da) \\ &= \int\limits_{b\in B}\int\limits_{a\in A} [(a,b)\in S]\mu(da)\mu'(db) && \text{Fubini} \\ &= \dots \\ &= \int\limits_{b\in B} \mu(\{a' \in A \mid (a',b)\in S\})\mu'(db)\end{aligned}$$

In the second line, we have used that $(a,b)\in S \iff b \in \{b' \in B \mid (a,b')\in S\}$. The proof works analogously for $\overline{\times}$.

Lemma 17. *Let $\delta\colon A \mapsto A$, $\kappa\colon A \mapsto B$. Then,*

$$(\delta\overline{\times}\kappa)(a)(S) = \kappa(a)(\{b \in \overline{B} \mid \overline{(a,b)} \in S\})$$

Proof.

$$\begin{aligned}(\delta\overline{\times}\kappa)(a)(S) &= \int_{b\in\overline{B}} \delta(a)(\{a' \in A \mid \overline{(a',b)} \in S\})\kappa(a)(db) && \text{Lemma 16} \\ &= \int_{b\in\overline{B}} [\overline{(a,b)}\in S]\kappa(a)(db) \\ &= \kappa(a)(\{b\in\overline{B} \mid \overline{(a,b)}\in S\})\end{aligned}$$

Lemma 1. *For measures $\mu\colon \Sigma_A \to [0,\infty]$, $\mu'\colon \Sigma_B \to [0,\infty]$, let $S \in \Sigma_A$ and $T \in \Sigma_B$. Then, $(\mu\times\mu')(S\times T) = \mu(S)\cdot\mu'(T)$.*

Proof.

$$\begin{aligned}(\mu\times\mu')(S\times T) &= \int_{a\in A} \mu'(\{b\in B \mid (a,b)\in S\times T\})\mu(da) && \text{Lemma 16} \\ &= \int_{a\in A} \mu'\left(\begin{Bmatrix} T & a\in S \\ \emptyset & \text{otherwise}\end{Bmatrix}\right)\mu(da) \\ &= \int_{a\in S} \mu'(T)\mu(da) \\ &= \mu(S) * \mu'(T)\end{aligned}$$

Lemma 2. *$\times$ and $\overline{\times}$ for s-finite measures are associative, left- and right-distributive and preserve (sub-)probability and s-finite measures.*

Proof. Remember that $(\mu\times\mu')(S) = \int_{a\in A}\int_{b\in B}[(a,b)\in S]\mu'(db)\mu(da)$ and that $(\mu\overline{\times}\mu')(S) = \int_{a\in\overline{A}}\int_{b\in\overline{B}}[\overline{(a,b)}\in S]\mu'(db)\mu(da)$. Preservation of (sub-)probability measures is trivial. Distributivity and preservation of s-finite measures are easily established by properties of the Lebesgue integral in Lemma 19.
For associativity, let $\mu\colon \Sigma_A \to [0,\infty]$, $\mu\colon \Sigma_B \to [0,\infty]$ and $\mu\colon \Sigma_C \to [0,\infty]$.

$$((\mu\times\mu')\times\mu'')(S)$$

$$
\begin{aligned}
&= \int\limits_{c\in C} (\mu \times \mu')(\{t \in A \times B \mid (t,c) \in S\})\mu''(dc) && \text{Lemma 16}\\
&= \int\limits_{c\in C} \int\limits_{a\in A} \int\limits_{b\in B} [(a,b) \in \{t \in A \times B \mid (t,c) \in S\}]\mu'(db)\mu(da)\mu''(dc)\\
&= \int\limits_{c\in C} \int\limits_{a\in A} \int\limits_{b\in B} [(a,b,c) \in S]\mu'(db)\mu(da)\mu''(dc)\\
&= \int\limits_{a\in A} \int\limits_{b\in B} \int\limits_{c\in C} [(a,b,c) \in S]\mu''(dc)\mu'(db)\mu(da) && \text{Fubini}\\
&= \int\limits_{a\in A} \int\limits_{b\in B} \int\limits_{c\in C} [(b,c) \in \{t \in B \times C \mid (a,t) \in S\}]\mu''(dc)\mu'(db)\mu(da)\\
&= \int\limits_{a\in A} (\mu' \times \mu'')(\{t \in B \times C \mid (a,t) \in S\})\mu(da)\\
&=(\mu \times (\mu' \times \mu''))(S)\mu(da) && \text{Lemma 16}
\end{aligned}
$$

The proof proceeds analogously for $\overline{\times}$.

Lemma 18. *Let (A, Σ_A) and (B, Σ_B) be measurable spaces. Consider measures $\mu, \mu_1, \mu_2 \colon \Sigma_A \to [0, \infty]$ and $\nu, \nu_1, \nu_2 \colon \Sigma_B \to [0, \infty]$. We assume that $\nu_1 \leq \nu_2$ and $\mu_1 \leq \mu_2$ hold pointwise. Then,*

$$
\begin{aligned}
\mu \overline{\times} \nu_1 &\leq \mu \overline{\times} \nu_2\\
\mu_1 \overline{\times} \nu &\leq \mu_2 \overline{\times} \nu
\end{aligned}
$$

Proof. Let $S \in \Sigma_{A\times B}$ and $\nu_1 \leq \nu_2$. Then, we have

$$
\begin{aligned}
&\nu_1 \leq \nu_2\\
\implies &\underbrace{\int_{b\in\overline{B}} [\overline{(a,b)} \in S]\nu_1(db)}_{=:f(a)} \leq \underbrace{\int_{b\in\overline{B}} [\overline{(a,b)} \in S]\nu_2(db)}_{=:g(a)} && \text{Lemma 19}\\
\implies &\int_{a\in\overline{A}} f(a)\mu(da) \leq \int_{a\in\overline{A}} g(a)\mu(da) && \text{Lemma 19}\\
\implies &(\mu \overline{\times} \nu_1)(S) \leq (\mu \overline{\times} \nu_2)(S)
\end{aligned}
$$

The proof for $\mu_1 \overline{\times} \nu \leq \mu_2 \overline{\times} \nu$ is similar.

A.2 Lebesgue integral

Lemma 19. *Let (A, Σ_A) and (B, Σ_B) be measurable spaces, $E \in \Sigma_A$ and $E' \in \Sigma_B$ measurable sets, $f, f_i, g \colon A \to \mathbb{R}$ and $h \colon A \times B \to \mathbb{R}$ measurable functions, $\mu, \mu_i, \nu \colon \Sigma_A \to [0, \infty]$ and $\mu' \colon \Sigma_B \to [0, \infty]$ measures.*

$$
\int\limits_{a\in E} f(a)\mu(da) \in [0, \infty]
$$

$$
\begin{aligned}
0 \le f \le g \le \infty \implies \int\limits_{a\in E} f(a)\mu(da) &\le \int\limits_{a\in E} g(a)\mu(da) \\
\mu \le \nu \implies \int\limits_{a\in E} f(a)\mu(da) &\le \int\limits_{a\in E} f(a)\nu(da) \\
\sum_{n=1}^{\infty} \int\limits_{a\in E} f_n(a)\mu(da) &= \int\limits_{a\in E} \sum_{n=1}^{\infty} f_n(a)\mu(da) \\
\int\limits_{a\in E} \int\limits_{b\in E'} f(a,b)\mu'(db)\mu(da) &= \int\limits_{b\in E'} \int\limits_{a\in E} f(a,b)\mu'(da)\mu(db) && \mu,\ \mu'\ \sigma\text{-finite} \\
\int\limits_{a\in E} f(a)\left(\sum_{n=1}^{\infty}\mu_i\right)(da) &= \sum_{n=1}^{\infty}\int\limits_{a\in E} f(a)\mu_i(da) \\
\int\limits_{a\in E} f(a)\delta(x)(da) &= f(x) && x \in E
\end{aligned}
$$

Finally, if $f_1 \le f_2 \le \cdots \le \infty$*, we have*

$$\lim_{n\to\infty} \int\limits_{a\in E} f_n(a)\mu(da) = \int\limits_{a\in E} \lim_{n\to\infty} f_n(a)\mu(da)$$

Proof. The following properties can be proven for simple functions and limits of simple functions (this suffices):

$$\int_{a\in E} f(a)\left(\sum_{n=1}^{\infty}\mu_i\right)(da) = \sum_{n=1}^{\infty}\int_{a\in E} f(a)\mu_i(da)$$

$$\mu \le \nu \implies \int_{a\in E} f(a)\mu(da) \le \int_{a\in E} f(a)\nu(da)$$

$\int_{a\in E} f(a)\delta(x)(da) = f(x)$ is straightforward. For the other properties, see [31].

Theorem 1 (Fubini's theorem). *For s-finite measures* $\mu\colon \Sigma_A \to [0,\infty]$ *and* $\mu'\colon \Sigma_B \to [0,\infty]$ *and any measurable function* $f\colon A \times B \to [0,\infty]$*,*

$$\int_{a\in A}\int_{b\in B} f(a,b)\mu'(db)\mu(da) = \int_{b\in B}\int_{a\in A} f(a,b)\mu(da)\mu'(db)$$

For s-finite measures $\mu\colon \Sigma_{\overline{A}} \to [0,\infty]$ *and* $\mu'\colon \Sigma_{\overline{B}} \to [0,\infty]$ *and any measurable function* $f\colon A \times B \to [0,\infty]$*,*

$$\int_{a\in\overline{A}}\int_{b\in\overline{B}} \overline{f}(a,b)\mu'(db)\mu(da) = \int_{b\in\overline{B}}\int_{a\in\overline{A}} \overline{f}(a,b)\mu(da)\mu'(db)$$

Proof. Let $\mu = \sum_{i\in\mathbb{N}}\mu_i$ and $\mu' = \sum_{i\in\mathbb{N}}\mu'_i$ for bounded measures μ_i and μ'_i.

$$\int_{a\in A}\int_{b\in B} f(a,b)\mu'(db)\mu(da)$$

$$
\begin{aligned}
&= \sum_{i,j \in \mathbb{N}} \int_{a \in A} \int_{b \in B} f(a,b) \mu'_j(db) \mu_i(da) && \text{Lemma 19} \\
&= \sum_{i,j \in \mathbb{N}} \int_{b \in B} \int_{a \in A} f(a,b) \mu_i(da) \mu'_j(db) && \text{Fubini for } \sigma\text{-finite measures } \mu_i, \mu'_j \\
&= \int_{b \in B} \int_{a \in A} f(a,b) \mu(da) \mu'(db)
\end{aligned}
$$

The proof in the presence of exception state is analogous.

Lemma 20. *Fubini does not hold for the counting measure* $c\colon \mathcal{B} \to [0,\infty]$ *and the Lebesgue measure* $\lambda\colon \mathcal{B} \to [0,\infty]$ *(because* c *is not s-finite).*

Proof.

$$
\begin{aligned}
\int_{x \in [0,1]} \int_{y \in [0,1]} [x = y] c(dy) \lambda(dx) &= \int_{x \in [0,1]} 1 \lambda(dx) = 1 \\
\int_{y \in [0,1]} \int_{x \in [0,1]} [x = y] \lambda(dx) c(dy) &= \int_{y \in [0,1]} 0 c(dy) = 0
\end{aligned}
$$

A.3 Kernels

Lemma 21. *Let* $\kappa_1, \kappa'_1\colon A \mapsto \overline{B}$ *and* $\kappa_2, \kappa'_2\colon B \mapsto \overline{C}$ *be s-finite kernels.*
If $\kappa_1 \leq \kappa'_1$ *holds pointwise, then*

$$\kappa_1 \mathrel{>\!\!=\!\!>} \kappa_2 \leq \kappa'_1 \mathrel{>\!\!=\!\!>} \kappa_2$$

If $\kappa_2 \leq \kappa'_2$ *holds pointwise, then*

$$\kappa_1 \mathrel{>\!\!=\!\!>} \kappa_2 \leq \kappa_1 \mathrel{>\!\!=\!\!>} \kappa'_2$$

Proof. Assume $\kappa_2 \leq \kappa'_2$. Thus, $\overline{\kappa_2} \leq \overline{\kappa'_2}$. Now, let $a \in A$, $S \in \Sigma_{\overline{C}}$.

$$
\begin{aligned}
(\kappa_1 \mathrel{>\!\!=\!\!>} \kappa_2)(a)(S) &= \int_{b \in \overline{B}} \overline{\kappa_2}(b)(S)\, \kappa_1(a)(db) \\
&\leq \int_{b \in \overline{B}} \overline{\kappa'_2}(b)(S)\, \kappa_1(a)(db) && \overline{\kappa_2} \leq \overline{\kappa'_2}, \text{Lemma 19} \\
&= (\kappa_1 \mathrel{>\!\!=\!\!>} \kappa'_2)(a)(S)
\end{aligned}
$$

The proof for $\kappa_1 \mathrel{>\!\!=\!\!>} \kappa_2 \leq \kappa'_1 \mathrel{>\!\!=\!\!>} \kappa_2$ works analogously.

Lemma 3. $(;)$ *is associative, left- and right-distributive, has neutral element*[4] δ *and preserves (sub-)probability and s-finite kernels.*

[4] δ is a neutral element of $(;)$ if $(\delta; \kappa) = (\kappa; \delta) = \kappa$ for all kernels κ.

Proof. Remember that $(f;g)(a)(S) = \int_{b\in B} g(b)(S)\, f(a)(db)$. Left- and right-distributivity and the neutral element δ follow from properties of the Lebesgue integral in Lemma 19.
Associativity and preservation of (sub-)probability kernels is well known (see for example [12]). For s-finite kernels $f = \sum_{i\in\mathbb{N}} f_i$ and $g = \sum_{i\in\mathbb{N}} g_i$ and $h = \sum_{i\in\mathbb{N}} h_i$, we have (for sub-probability kernels f_i, g_i, h_i)

$$\begin{aligned}(f;g);h = \left(\left(\sum_{i\in\mathbb{N}} f_i\right);\left(\sum_{j\in\mathbb{N}} g_j\right)\right);\sum_{k\in\mathbb{N}} h_k = \sum_{i,j,k\in\mathbb{N}} (f_i;g_j);h_k \\ = \sum_{i,j,k\in\mathbb{N}} f_i;(g_j;h_k) = f;(g;h)\end{aligned}$$

(;) preserves s-finite kernels because for s-finite kernels f and g, we have (for sub-probability kernels f_i, g_i) $f;g = \sum_{i,j\in\mathbb{N}} f_i;g_i$, a sum of kernels.

Lemma 4. *For $f\colon A \mapsto \overline{B}$ and $g\colon B \mapsto \overline{C}$, $a \in A$ and $S \in \Sigma_{\overline{C}}$,*

$$(f \ggg g)(a)(S) = (f;g)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S) f(a)(\{x\})$$

Proof.

$$\begin{aligned}(f \ggg g)(a)(S) &= \int_{b\in\overline{B}} \overline{g}(b)(S)\, f(a)(db) \\ &= \int_{b\in B} \overline{g}(b)(S)\, f(a)(db) + \int_{b\in\mathcal{X}} \overline{g}(b)(S)\, f(a)(db) \\ &= \int_{b\in B} g(b)(S)\, f(a)(db) + \sum_{b\in\mathcal{X}} \overline{g}(b)(S)\, f(a)(\{x\}) \\ &= (f;g)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S) f(a)(\{x\})\end{aligned}$$

Lemma 5. $\ggg$ *is associative, left-distributive (but not right-distributive), has neutral element δ and preserves (sub-)probability and s-finite kernels.*

Proof. Remember that $(f \ggg g)(a)(S) = \int_{b\in\overline{B}} \overline{g}(b)(S)\, f(a)(db)$. Left-distributivity follows from the properties of the Lebesgue integral in Lemma 19. Right-distributivity does not necessarily hold because $\overline{g_1 + g_2}(\bot) \neq \overline{g_1}(\bot) + \overline{g_2}(\bot)$. Associativity for $f\colon A \mapsto \overline{B}$, $g\colon B \mapsto \overline{C}$ and $h\colon C \mapsto \overline{D}$ can be derived by

$$\begin{aligned}&((f \ggg g) \ggg h)(a)(S) \\ =&\Big(\big(f \ggg g\big);h\Big)(a)(S) + \sum_{x\in\mathcal{X}} \delta(x)(S)(f \ggg g)(a)(\{x\}) \\ =&\Big(\Big(f;g + \lambda a'.\lambda S'. \sum_{x\in\mathcal{X}} \delta(x)(S')f(a')(\{x\})\Big);h\Big)(a)(S) +\end{aligned}$$

$$\sum_{x\in\mathcal{X}} \delta(x)(S)(f >\!\!=\!\!> g)(a)(\{x\})$$

$$= (f;g;h)(a)(S) + \underbrace{\Big(\Big(\lambda a'.\lambda S'.\sum_{x\in\mathcal{X}}\delta(x)(S')f(a')(\{x\})\Big);h\Big)(a)(S)}_{=0((;)\text{ integrates over non-exception states})} +$$

$$\sum_{x\in\mathcal{X}} \delta(x)(S)(f >\!\!=\!\!> g)(a)(\{x\})$$

$$= (f;g;h)(a)(S) + \sum_{x\in\mathcal{X}}\delta(x)(S)\Big((f;g)(a)(\{x\}) + \sum_{x'\in\mathcal{X}}\delta(x')(\{x\})f(a)(\{x'\})\Big)$$

$$= (f;g;h)(a)(S) + \sum_{x\in\mathcal{X}}\delta(x)(S)\Big((f;g)(a)(\{x\}) + f(a)(\{x\})\Big)$$

$$= (f;g;h)(a)(S) + \sum_{x\in\mathcal{X}}\delta(x)(S)(f;\lambda a'.\lambda S'.g(a')(S'))(a)(\{x\}) +$$

$$\sum_{x\in\mathcal{X}}\delta(x)(S)f(a)(\{x\})$$

$$= (f;g;h)(a)(S) + \Big(f;\Big(\lambda a'.\lambda S'.\sum_{x\in\mathcal{X}}\delta(x)(S')g(a')(\{x\})\Big)\Big)(a)(S) +$$

$$\sum_{x\in\mathcal{X}}\delta(x)(S)f(a)(\{x\})$$

$$= \Big(f;\Big(g;h + \lambda a'.\lambda S'.\sum_{x\in\mathcal{X}}\delta(x)(S')g(a')(\{x\})\Big)\Big)(a)(S) + \sum_{x\in\mathcal{X}}\delta(x)(S)f(a)(\{x\})$$

$$= \Big(f;\Big(g >\!\!=\!\!> h\Big)\Big)(a)(S) + \sum_{x\in\mathcal{X}}\delta(x)(S)f(a)(\{x\})$$

$$= (f >\!\!=\!\!> (g >\!\!=\!\!> h))(a)(S)$$

Here, we have used Lemma 4, left- and right-distributivity of (;).

To show that $f >\!\!=\!\!> g$ preserves s-finite kernels, let $f\colon A \mapsto \overline{B}$ and $g\colon B \mapsto \overline{C}$ be s-finite kernels. Then, for sub-probability kernels f_i,

$$\begin{aligned}(f >\!\!=\!\!> g)(a)(S) &= (f;g)(a)(S) + \sum_{x\in\mathcal{X}}\delta(x)(S)f(a)(\{x\}) \\ &= (f;g)(a)(S) + \sum_{x\in\mathcal{X}}\sum_{i\in\mathbb{N}}\delta(x)(S)f_i(a)(\{x\})\end{aligned}$$

Note that for each $x \in \mathcal{X}$ and $i \in \mathbb{N}$, $\lambda a.\lambda S.\delta(x)(S)f_i(a)(\{x\})$ is a sub-probability kernel. Thus, $f >\!\!=\!\!> g$ is a sum of s-finite kernels and hence s-finite.

Proving that for sub-probability kernels f and g, $f >\!\!=\!\!> g$ is also a (sub-)probability kernel is trivial, since we only need to show that $(f >\!\!=\!\!> g)(a)(\overline{C}) = 1$ (or ≤ 1).

Lemma 22. *Let (A, Σ_A) and (B, Σ_B) be measurable spaces. Let $f\colon A \times B \to [0,\infty]$ be measurable and $\kappa\colon A \mapsto B$ be a sub-probability kernel. Then, $f'\colon A \to$*

$[0, \infty]$ *defined by*

$$f'(a) := \int_{b \in B} f(a, b)\kappa(a)(db)$$

is measurable.

Proof. See Theorem 20 of [30].

Lemma 23. $\times$ *and* $\overline{\times}$ *preserve (sub-)probability kernels.*

Proof. Let $\kappa \colon A \mapsto B$ and $\kappa' \colon A \mapsto C$ be (sub-)probability kernels. The fact that $(\kappa \times \kappa')(a)(\cdot)$ for all $a \in A$ is a (sub-)probability measure is inherited from Lemma 2. It remains to show that $(\kappa \times \kappa')(\cdot)(S)$ is measurable for all $S \in \Sigma_{B \times C}$, with

$$(\kappa \times \kappa')(a)(S) = \int_{b \in B} \int_{c \in C} [(b, c) \in S]\kappa'(a)(dc)\kappa(a)(db)$$

By Lemma 22, $f' \colon A \times B \to [0, \infty]$ defined by $f'(a, b) = \int_{c \in C} [(b, c) \in S]\kappa'(a)(dc)$ is measurable, using the measurable function $f \colon (A \times B) \times C \to [0, \infty]$ defined by $f((a, b), c) = [(b, c) \in S]$. Again by Lemma 22, $\int_{b \in B} \int_{c \in C} [(b, c) \in S]\kappa'(a)(dc)\kappa(a)(db)$ is measurable.

Proving that for (sub-)probability kernels $\kappa \colon A \mapsto \overline{B}$ and $\kappa' \colon A \mapsto \overline{C}$, $\kappa \overline{\times} \kappa'$ is a (sub-)probability kernel proceeds analogously.

Lemma 6. $\times$ *and* $\overline{\times}$ *for kernels preserve (sub-)probability and s-finite kernels, are associative, left- and right-distributive.*

Proof. Associativity, left- and right-distributivity are inherited from respective properties of the product of measures established by Lemma 2. Sub-probability kernels are preserved by Lemma 23.

S-finite kernels are preserved because $\kappa \times \kappa' = (\sum_{i \in \mathbb{N}} \kappa_i) \times (\sum_{i \in \mathbb{N}} \kappa_i') = \sum_{i,j \in \mathbb{N}} \kappa_i \times \kappa_j'$ (analogously for $\overline{\times}$).

B Proofs for Semantics

Lemma 7. *For* Δ *as in the semantics of the while loop, and for each* σ *and each* S*, the limit* $\lim_{n \to \infty} \Delta^n(\circlearrowleft)(\sigma)(S)$ *exists.*

Proof. In general, $0 \leq \Delta^n(\circlearrowleft)(\sigma)(S) \leq 1$. First, we restrict the allowed arguments for $\lim_{n \to \infty} \Delta^n(\circlearrowleft)(\sigma)(S)$ to only those S with $\circlearrowleft \in S$. We prove by induction that $\Delta^{n+1}(\circlearrowleft) \leq \Delta^n(\circlearrowleft)$, meaning $\forall \sigma \colon \forall S \colon \ \circlearrowleft \in S \implies \Delta^{n+1}(\circlearrowleft)(\sigma)(S) \leq \Delta^n(\circlearrowleft)(\sigma)(S)$. Hence, $\Delta^n(\circlearrowleft)$ is monotone decreasing in n and lower bounded by 0, which means that the limit must exist.

As a base case, we have $\Delta^1(\circlearrowleft)(\sigma)(S) \leq 1 = \delta_{\circlearrowleft}(S) = \Delta^0(\circlearrowleft)(\sigma)(S)$, because $\circlearrowleft \in S$. We proceed by induction with

$$\Delta^{n+1}(\circlearrowleft)(\sigma)(S) = \left(\delta \overline{\times} [\![e]\!] \mathrel{>\!\!=\!\!>} \lambda(\sigma, b). \begin{Bmatrix} [\![P]\!](\sigma) \gg= \Delta^n(\circlearrowleft) & b \neq 0 \\ \delta(\sigma) & b = 0 \end{Bmatrix}\right)(\sigma)(S)$$

$$\leq \left(\delta \overline{\times} [\![e]\!] \ggg \lambda(\sigma, b). \begin{Bmatrix} [\![P]\!](\sigma) \gg\!= \Delta^{n-1}(\circlearrowleft) & b \neq 0 \\ \delta(\sigma) & b = 0 \end{Bmatrix}\right)(\sigma)(S)$$
$$= \Delta^{n}(\circlearrowleft)(\sigma)(S)$$

In the second line, we have used the induction hypothesis. This application is valid because $\kappa_2 \leq \kappa_2'$ implies $\kappa_1 \ggg \kappa_2 \leq \kappa_1 \ggg \kappa_2'$ (Lemma 21).

We proceed analogously when we restrict the allowed arguments for the kernel $\lim_{n\to\infty} \Delta^n(\circlearrowleft)(\sigma)(S)$ to only those S with $\circlearrowleft \notin S$, proving $\Delta^{n+1}(\circlearrowleft) \geq \Delta^n(\circlearrowleft)$ for that case.

Lemma 8. *In the absence of exception states, and using sub-probability kernels instead of distribution transformers, the definition of the semantics of the while loop from [23] is equivalent to ours.*

Definition 6. *In [23], Kozen shows a different way of defining the semantics of the while loop. In our notation, and in terms of probability kernels instead of distribution transformers, that definition becomes*

$$[\![\texttt{while}\ e\ \{P\}]\!] = \sup_{n\in\mathbb{N}} \sum_{k=0}^{n} \left([\![\texttt{filter}(e)]\!] \ggg [\![P]\!]\right)^k \ggg [\![\texttt{filter}(\neg e)]\!]$$

Here, exponentiation is in terms of Kleisli composition, i.e. $\kappa^0 = \delta$ *and* $\kappa^{n+1} = \kappa \ggg \kappa^n$. *The sum and limit are meant pointwise. Furthermore, we define filter by the following expression (note that* $[\![\texttt{filter}(e)]\!]$ *and* $[\![\texttt{filter}(\neg e)]\!]$ *are only sub-probability kernels, not probability kernels).*

$$[\![\texttt{filter}(e)]\!] = \delta \overline{\times} [\![e]\!] \ggg \lambda(\sigma, b). \begin{Bmatrix} \delta(\sigma) & b \neq 0 \\ \mathbf{0} & b = 0 \end{Bmatrix}$$
$$[\![\texttt{filter}(\neg e)]\!] = \delta \overline{\times} [\![e]\!] \ggg \lambda(\sigma, b). \begin{Bmatrix} \delta(\sigma) & b = 0 \\ \mathbf{0} & b \neq 0 \end{Bmatrix}$$

To justify Lemma 8, we prove the more formal Lemma 24. Note that in the presence of exceptions (e.g. P is just $\texttt{assert}(0)$), Definition 6 does not make sense, because if

Lemma 24. *For all* S *with* $S \cap \mathcal{X} = \emptyset$

$$\left(\sum_{k=0}^{n} \left([\![\texttt{filter}(e)]\!] \ggg [\![P]\!]\right)^k \ggg [\![\texttt{filter}(\neg e)]\!]\right)(\sigma)(S) = \Delta^{n+1}(\circlearrowleft)(\sigma)(S)$$

Proof. For $n = 0$, we have

$$\left(\sum_{k=0}^{0} \left([\![\texttt{filter}(e)]\!] \ggg [\![P]\!]\right)^k \ggg [\![\texttt{filter}(\neg e)]\!]\right)(\sigma)(S)$$
$$= \left(\left([\![\texttt{filter}(e)]\!] \ggg [\![P]\!]\right)^0 \ggg [\![\texttt{filter}(\neg e)]\!]\right)(\sigma)(S)$$

$$
\begin{aligned}
&= \Big(\delta >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket\Big)(\sigma)(S) \\
&= \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket(\sigma)(S) \\
&= \left(\delta \overline{\times} \llbracket e \rrbracket >\!\!==\!\!> \lambda(\sigma', b).\begin{Bmatrix} \delta(\sigma') & b = 0 \\ \mathbf{0} & b \neq 0 \end{Bmatrix}\right)(\sigma)(S) \\
&= \left(\delta \overline{\times} \llbracket e \rrbracket >\!\!==\!\!> \lambda(\sigma', b).\begin{Bmatrix} \delta(\sigma') & b = 0 \\ \circlearrowleft(\sigma') & b \neq 0 \end{Bmatrix}\right)(\sigma)(S) && \circlearrowleft \notin S \\
&= \left(\delta \overline{\times} \llbracket e \rrbracket >\!\!==\!\!> \lambda(\sigma', b).\begin{Bmatrix} \delta(\sigma') & b = 0 \\ (\llbracket P \rrbracket >\!\!==\!\!> \circlearrowleft)(\sigma') & b \neq 0 \end{Bmatrix}\right)(\sigma)(S) && S \cap \mathcal{X} = \emptyset \\
&= \Delta^1(\circlearrowleft)
\end{aligned}
$$

For $n \geq 0$, we have

$$
\begin{aligned}
&\left(\sum_{k=0}^{n+1} \Big(\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket\Big)^k >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket\right)(\sigma)(S) \\
&= \left(\left(\sum_{k=0}^{n} \Big(\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket\Big)^{k+1} + (\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> P)^0\right) >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket\right)(\sigma)(S) \\
&= \left(\left(\sum_{k=0}^{n} \Big(\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket\Big)^{k+1} + \delta\right) >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket\right)(\sigma)(S) \\
&= \left(\left(\sum_{k=0}^{n} \Big(\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket\Big)^{k+1}\right) >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket\right)(\sigma)(S) + && \text{since } S \cap \mathcal{X} = \emptyset \\
&\quad (\delta >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket)(\sigma)(S) \\
&= \left(\left(\sum_{k=0}^{n} \Big(\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket\Big)^{k+1}\right) >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket\right)(\sigma)(S) + \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket(\sigma)(S) \\
&= \left(\left(\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket >\!\!==\!\!> \sum_{k=0}^{n} (\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket)^k\right) >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket\right)(\sigma)(S) + \\
&\quad \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket(\sigma)(S) \\
&= \left(\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket >\!\!==\!\!> \left(\sum_{k=0}^{n} (\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket)^k >\!\!==\!\!> \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket\right)\right)(\sigma)(S) + \\
&\quad \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket(\sigma)(S) \\
&= \Big(\llbracket \texttt{\textbf{filter}}(e) \rrbracket >\!\!==\!\!> \llbracket P \rrbracket >\!\!==\!\!> \Delta^{n+1}(\circlearrowleft)\Big)(\sigma)(S) + \llbracket \texttt{\textbf{filter}}(\neg e) \rrbracket(\sigma)(S) \\
&= \left(\delta \overline{\times} \llbracket e \rrbracket >\!\!==\!\!> \lambda(\sigma', b).\begin{Bmatrix} \llbracket P \rrbracket(\sigma') \gg\!= \Delta^{n+1}(\circlearrowleft) & b \neq 0 \\ \delta(\sigma') & b = 0 \end{Bmatrix}\right)(\sigma)(S) \\
&= \Delta^{n+2}(\circlearrowleft)(\sigma)(S)
\end{aligned}
$$

In particular, have have used that left-distributivity does hold in this case since $S \cap \mathcal{X} = \emptyset$.

C Probability kernel

In the following, we list lemmas that are crucial to prove Theorem 2 (restated for convenience).

Theorem 2. *The semantics of each expression* $\llbracket e \rrbracket$ *and statement* $\llbracket P \rrbracket$ *is indeed a probability kernel.*

Lemma 25. *Any measurable function* $f\colon A \to [0,\infty]$ *can be viewed as an s-finite kernel* $f\colon A \mapsto \mathbb{1}$*, defined by* $f(x)(\emptyset) = 0$ *and* $f(x)(\mathbb{1}) = f(x)$*.*

Proof. We prove that f is an s-finite kernel. Let $A_\infty := \{x \in A \mid f(x) = \infty\}$. Since f is measurable, the set A_∞ must be measurable. $f(x)(S) = \sum_{i\in\mathbb{N}}[x \in A_\infty][() \in S] + \sum_{i\in\mathbb{N}} f(x)[i \leq f(x) < i+1][() \in S]$, which is a sum of finite kernels because the sets A_∞ and $\{x \mid i \leq f(x) < i+1\} = f^{-1}([i, i+1))$ are measurable. Note that any sum of finite kernels can be rewritten as a sum of sub-probability kernels.

Lemma 26. *Let* $\kappa'\colon X \mapsto Y$ *and* $\kappa''\colon X \mapsto Y$ *be kernels, and* $f\colon X \to \mathbb{R}$ *measurable. Then,*

$$\kappa(x)(S) = \begin{cases} \kappa'(x)(S) & \text{if } f(x) = 0 \\ \kappa''(x)(S) & \text{otherwise} \end{cases}$$

is a kernel.

Proof. Let $f_{=0}(x) := [f(x) = 0]$, $f_{\neq 0}(x) := [f(x) \neq 0]$. Then, $\kappa = f_{=0} \times \kappa' + f_{\neq 0} \times \kappa''$. Viewing $f_{=0}$ and $f_{\neq 0}$ as kernels $X \mapsto \mathbb{1}$ immediately gives the desired result.

Lemma 27. *Let* (A, Σ_A) *and* (B, Σ_B) *be measurable spaces. Let* $\{A_i\}_{i\in\mathcal{I}}$ *be a partition of* A *into measurable sets, for a countable set of indices* $\mathcal{I}$*. Consider a function* $f\colon A \to B$*. If* $f_{|A_i}\colon A_i \to B$ *is measurable for each* $i \in \mathcal{I}$*, then* f *is measurable.*

Lemma 28. *Let* $f\colon A \to B$ *be measurable. Then* $\kappa\colon A \mapsto B$ *with* $\kappa(a) = \delta(f(a))$ *is a kernel.*

The following lemma is important to show that the semantics of the while loop is a probability kernel.

Lemma 29. *Suppose* $\{\kappa_n\}_{n\in\mathbb{N}}$ *is a sequence of (sub-)probability kernels* $A \mapsto B$*. Then, if the limit* $\kappa = \lim_{n\to\infty} \kappa_n$ *exists, it is also a (sub-)probability kernel. Here, the limit is pointwise in the sense* $\forall a \in A\colon \forall S \in \Sigma_B\colon \kappa(a, S) = \lim_{n\to\infty} \kappa_n(a)(S)$*.*

Proof. For every $a \in A$, $\kappa(a, \cdot)$ is a measure, because the pointwise limit of finite measures is a measure. For every $S \in \Sigma_B$, $\kappa(\cdot, S)$ is measurable, because the pointwise limit of measurable functions $f_n\colon A \to \mathbb{R}$ (with $\mathcal{B}$ as the σ-algebra on $\mathbb{R}$) is measurable.

D Proofs for consequences

In this section, we provide some proofs of consequences of our semantics, explained in Section 5.

Lemma 9. *For function* $F()\{\texttt{while}\ 1\ \{\texttt{skip}\}; \texttt{return}\ 0\}$,

$$\frac{1}{0} + F() \not\simeq F() + \frac{1}{0}$$

Proof. If we evaluate $\frac{1}{0}$ first, we will only have weight on $\bot$.

$$\begin{aligned}
&\left[\!\!\left[\frac{1}{0} + F()\right]\!\!\right] \\
=&\left[\!\!\left[\frac{1}{0}\right]\!\!\right] \overline{\times} [\![F()]\!] \mathrel{>\!\!=\!\!>} \lambda(x,y).\delta(x+y) \\
=&\delta(\bot)\overline{\times}[\![F()]\!] \mathrel{>\!\!=\!\!>} \lambda(x,y).\delta(x+y) \\
=&\delta(\bot) \mathrel{>\!\!=\!\!>} \lambda(x,y).\delta(x+y) \\
=&\delta(\bot)
\end{aligned}$$

If instead, we first evaluate $F()$, we only have weight on $\circlearrowleft$, by an analogous calculation.

Lemma 10. *If* $[\![e_1]\!](\sigma)(\mathcal{X}) = [\![e_2]\!](\sigma)(\mathcal{X}) = 0$ *for all* σ, *then* $e_1 \oplus e_2 \simeq e_2 \oplus e_1$, *for any commutative operator* $\oplus$.

Proof.

$$\begin{aligned}
[\![e_1 \oplus e_2]\!](\sigma)(S) &= [\![e_1]\!]\overline{\times}[\![e_2]\!] \mathrel{>\!\!=\!\!>} \lambda(x,y).\delta(x \oplus y) \\
&= \int_{z \in \overline{\mathbb{R}\times\mathbb{R}}} \overline{\lambda(x,y).\delta(x \oplus y)}(z)(S)([\![e_1]\!]\overline{\times}[\![e_2]\!])(\sigma)(dz) \\
&= \int_{(x,y)\in\mathbb{R}\times\mathbb{R}} \delta(x \oplus y)(S)([\![e_1]\!] \times [\![e_2]\!])(\sigma)(d(x,y)) \\
&= \int_{(y,x)\in\mathbb{R}\times\mathbb{R}} \delta(y \oplus x)(S)([\![e_2]\!] \times [\![e_1]\!])(\sigma)(d(y,x)) \\
&= [\![e_2 \oplus e_1]\!](\sigma)(S)
\end{aligned}$$

Here, we crucially rely on the absence of exceptions (for the third equality) and Fubini's Theorem (for the fourth equality).

Lemma 11. $e_1 \oplus (e_2 \oplus e_3) \simeq (e_1 \oplus e_2) \oplus e_3$, *for any associative operator* $\oplus$.

Proof. The important steps of the proof are the following.

$$\begin{aligned}
[\![e_1 \oplus (e_2 \oplus e_3)]\!] &= [\![e_1]\!]\overline{\times}[\![e_2 \oplus e_3]\!] \mathrel{>\!\!=\!\!>} \lambda(x,s).\delta(x \oplus s) \\
&= [\![e_1]\!]\overline{\times}\Big([\![e_2]\!]\overline{\times}[\![e_3]\!] \mathrel{>\!\!=\!\!>} \lambda(y,z).\delta(y \oplus z)\Big) \mathrel{>\!\!=\!\!>} \lambda(x,s).\delta(x \oplus s)
\end{aligned}$$

$$
\begin{aligned}
&= [\![e_1]\!] \overline{\times} \Big([\![e_2]\!] \overline{\times} [\![e_3]\!] \Big) \mathrel{>\!\!=\!\!>} \lambda(x,(y,z)).\delta(x \oplus y \oplus z) \\
&= \Big([\![e_1]\!] \overline{\times} [\![e_2]\!] \Big) \overline{\times} [\![e_3]\!] \mathrel{>\!\!=\!\!>} \lambda((x,y),z).\delta(x \oplus y \oplus z) \\
&= [\![(e_1 \oplus e_2) \oplus e_3]\!]
\end{aligned}
$$

Here, we make crucial use of associativity for the lifted product of measures in Lemma 6.