Probabilistic Programming Inference via Intensional Semantics

Simon Castellan[1] and Hugo Paquet[2]

[1] Imperial College London, United Kingdom
simon.castellan@phis.me
[2] University of Cambridge, United Kingdom
hugo.paquet@cl.cam.ac.uk

Abstract. We define a new denotational semantics for a first-order probabilistic programming language in terms of *probabilistic event structures*. This semantics is *intensional*, meaning that the interpretation of a program contains information about its behaviour throughout execution, rather than a simple distribution on return values. In particular, occurrences of sampling and conditioning are recorded as explicit events, partially ordered according to the data dependencies between the corresponding statements in the program.
This interpretation is *adequate*: we show that the usual measure-theoretic semantics of a program can be recovered from its event structure representation. Moreover it can be leveraged for MCMC inference: we prove correct a version of single-site Metropolis-Hastings with *incremental recomputation*, in which the proposal kernel takes into account the semantic information in order to avoid performing some of the redundant sampling.

Keywords: Probabilistic programming · Denotational Semantics · Event structures · Bayesian inference.

1 Introduction

Probabilistic programming languages [8] were put forward as promising tools for practitioners of Bayesian statistics. By extending traditional programming languages with primitives for sampling and conditioning, they allow the user to express a wide class of statistical models, and provide a simple interface for encoding inference problems. Although the subject of active research, it is still notoriously difficult to design inference methods for probabilistic programs which perform well for the full class of expressible models.

One popular inference technique, proposed by Wingate et al. [21], involves adapting well-known *Monte-Carlo Markov chain* methods from statistics to probabilistic programs, by manipulating *program traces*. One such method is the Metropolis-Hastings algorithm, which relies on a key *proposal* step: given a program trace x (a sequence $x_1, \ldots, x_n$ of random choices with their likelihood), a proposal for the *next* trace sample is generated by choosing $i \in \{1, \ldots, n\}$

uniformly, resampling x_i, and then continuing to execute the program, only performing additional sampling for those random choices not appearing in x. The variables already present in x are not resampled: only their likelihood is updated according to the new value of x_i. Likewise, some conditioning statements must be re-evaluated in case the corresponding weight is affected by the change to x_i.

Observe that there is some redundancy in this process, since the updating process above will only affect variables and observations when their density directly depends on the value of x_i. This may significantly affect performance: to solve an inference problem one must usually perform a large number of proposal steps. To overcome this problem, some recent implementations, notably [12, 25], make use of *incremental recomputation*, whereby some of the redundancy can be avoided via a form of static analysis. However, as pointed out by Kiselyov [13], establishing the correctness of such implementations is tricky.

Here we address this by introducing a theoretical framework in which to reason about data dependencies in probabilistic programs. Specifically, our first contribution is to define a *denotational semantics* for a first-order probabilistic language, in terms of graph-like structures called *event structures* [22]. In event structures, computational events are partially ordered according to the dependencies between them; additionally they can be equipped with quantitative information to represent probabilistic processes [16, 23]. This semantics is *intensional*, unlike most existing semantics for probabilistic programs, in which the interpretation of a program resembles a probability distribution on output values. We relate our approach to a measure-theoretic semantics [18] through an *adequacy* result.

Our second contribution is the design of a Metropolis-Hastings algorithm which exploits the event structure representation of the program at hand. Some of the redundancy in the proposal step of the algorithm is avoided by taking into account the extra dependency information given by the semantics. We provide a proof of correctness for this algorithm, and argue that an implementation is realistically achievable: we show in particular that all graph structures involved and the associated quantitative information admit a finite, concrete representation.

Outline of the paper. In § 2 we give a short introduction to probabilistic programming. We define our main language of study and its measure-theoretic semantics. In § 3.1, we introduce MCMC methods and the Metropolis-Hastings algorithm in the context of probabilistic programming. We then motivate the need for intensional semantics in order to capture data dependency. In § 4 we define our interpretation of programs and prove adequacy. In § 5 we define an updated version of the algorithm, and prove its correctness. We conclude in § 6.

The proofs of the statements are detailed in the technical report [4]

2 Probabilistic programming

In this section we motivate the need for capturing data dependency in probabilistic programs. Let us start with a brief introduction to probabilistic programming – a more comprehensive account can be found in [8].

2.1 Conditioning and posterior distribution

Let us introduce the problem of inference in probabilistic programming from the point of view of programming language theory.

We consider a first-order programming language enriched with a real number type $\mathbb{R}$ and a primitive `sample` for drawing random values from a given family of standard probability distributions. The language is idealised — but it is assumed that an implementation of the language comprises built-in sampling procedures for those standard distributions. Thus, repeatedly running the program `sample Uniform` $(0, 1)$ returns a sequence of values approaching the true uniform distribution on $[0, 1]$.

Via other constructs in the language, standard distributions can be combined, as shown in the following example program of type $\mathbb{R}$:

```
let x = sample Uniform(0, 1) in
let y = sample Gaussian(x, 2) in
x + y
```

Here the output will follow a probability distribution built out of the usual uniform and Gaussian distributions. Many probabilistic programming languages will offer more general programming constructs: conditionals, recursion, higher-order functions, data types, *etc.*, enabling a wide range of distributions to be expressed in this way. Such a program is sometimes called a *generative model*.

Conditioning. The process of conditioning involves rescaling the distribution associated with a generative model, so as to reflect some bias. Going back to the example above, say we have made some external measurement indicating that $y = 0$, but we would like to account for possible noise in the measurement using another Gaussian. To express this we modify the program as follows:

```
let x = sample Uniform (0, 1) in
let y = sample Gaussian (x, 2) in
observe y (Gaussian (0, 0.01));
x + y;
```

The purpose of the `observe` statement is to increase the occurrence of executions in which y is close to 0; the original distribution, known as the **prior**, must be updated accordingly. The probabilistic weight of each execution is multiplied by an appropriate *score*, namely the **likelihood** of the current value of y in the Gaussian distribution with parameters $(0, 0.01)$. (This is known as a *soft constraint*. Conditioning via *hard constraints*, *i.e.* only giving a nonzero score to executions where y is exactly 0, is not practically feasible.)

The language studied here does not have an `observe` construct, but instead an explicit `score` primitive; this appears already in [19, 18]. So the third line in the program above would instead be `score(pdf-Gaussian (0, 0.01) (y))` where `pdf-Gaussian (0, 0.01)` is the *density* function of the Gaussian distribution. The resulting distribution is not necessarily normalised. We obtain the

posterior distribution by computing the normalising constant, following Bayes' rule:

$$\text{posterior} \propto \text{likelihood} \times \text{prior}.$$

This process is known as Bayesian inference and has ubiquitous applications. The difficulty lies in computing the normalising constant, which is usually obtained as an integral. Below we discuss *approximate* methods for sampling from the posterior distribution; they do not rely on this normalising step.

Measure theory. Because this work makes heavy use of probability theory, we start with a brief account of measure theory. A standard textbook for this is [1]. Recall that a **measurable space** is a set X equipped with a **σ-algebra** Σ_X: a set of subsets of X containing $\emptyset$ and closed under complements and countable unions. Elements of Σ_X are called **measurable sets**. A **measure** on X is a function $\mu : \Sigma_X \to [0, \infty]$, such that $\mu(\emptyset) = 0$ and, for any countable family $\{U_i\}_{i \in I}$ of measurable sets, $\mu(\bigcup_{i \in I} U_i) = \sum_{i \in I} \mu(U_i)$.

An important example is that of the set $\mathbb{R}$ of real numbers, whose σ-algebra $\Sigma_{\mathbb{R}}$ is generated by the intervals $[a, b)$, for $a, b \in \mathbb{R}$ (in other words, it is the smallest σ-algebra containing those intervals). The **Lebesgue measure** on $(\mathbb{R}, \Sigma_{\mathbb{R}})$ is the (unique) measure λ assigning $b - a$ to every interval $[a, b)$ (with $a \leq b$).

Given measurable spaces (X, Σ_X) and (Y, Σ_Y), a function $f : X \to Y$ is **measurable** if for every $U \in \Sigma_Y$, $f^{-1}U \in \Sigma_X$. A measurable function $f : X \to [0, \infty]$ can be *integrated*: given $U \in \Sigma_X$ the **integral** $\int_U f \, d\lambda$ is a well-defined element of $[0, \infty]$; indeed the map $\mu : U \mapsto \int_U f \mathrm{d}\lambda$ is a measure on X, and f is said to be a **density** for μ. The precise definition of the integral is standard but slightly more involved; we omit it.

We identify the following important classes of measures: a measure μ on (X, Σ_X) is a **probability measure** if $\mu(X) = 1$. It is **finite** if $\mu(X) < \infty$, and it is **s-finite** if $\mu = \sum_{i \in I} \mu_i$, a pointwise, countable sum of finite measures.

We recall the usual product and coproduct constructions for measurable spaces and measures. If $\{X_i\}_{i \in I}$ is a countable family of measurable spaces, their **product** $\prod_{i \in I} X_i$ and **coproduct** $\coprod_{i \in I} X_i = \bigcup_{i \in I} \{i\} \times X_i$ as sets can be turned into measurable spaces, where:

- $\Sigma_{\prod_{i \in I} X_i}$ is generated by $\{\prod_{i \in I} U_i \mid U_i \in \Sigma_{X_i} \text{ for all } i\}$, and
- $\Sigma_{\coprod_{i \in I} X_i}$ is generated by $\{\{i\} \times U_i \mid i \in I \text{ and } U_i \in \Sigma_{X_i}\}$.

The measurable spaces in this paper all belong to a well-behaved subclass: call (X, Σ_X) a **standard Borel space** if it either countable and discrete (*i.e.* all $U \subseteq X$ are in Σ_X), or measurably isomorphic to $(\mathbb{R}, \Sigma_{\mathbb{R}})$. Note that standard Borel spaces are closed under countable products and coproducts, and that in a standard Borel space all singletons are measurable.

2.2 A first-order probabilistic programming language

We consider a first-order, call-by-value language $\mathcal{L}$ with types

$$A, B ::= 1 \mid \mathbb{R} \mid \coprod_{i \in I} A_i \mid \prod_{i \in I} A_i$$

where I ranges over nonempty countable sets. The types denote measurable spaces in a natural way: $[\![1]\!]$ is the singleton space, and $[\![\mathbb{R}]\!] = (\mathbb{R}, \Sigma_{\mathbb{R}})$. Products and coproducts are interpreted via the corresponding measure-theoretic constructions: $[\![\prod_{i\in I} A_i]\!] = \prod_{i\in I} [\![A_i]\!]$ and $[\![\coprod_{i\in I} A_i]\!] = \coprod_{i\in I} [\![A_i]\!] = \bigcup_{i\in I} \{i\} \times [\![A_i]\!]$. Moreover, each measurable space $[\![A]\!]$ has a canonical measure $\mu_{[\![A]\!]} : \Sigma_{[\![A]\!]} \to \mathbb{R}$, induced from the Lebesgue measure on $\mathbb{R}$ and the Dirac measure on $[\![1]\!]$ via standard product and coproduct measure constructions.

The terms of $\mathcal{L}$ are given by the following grammar:

$$\begin{aligned} M, N ::= {} & () \mid M; N \mid f \mid \texttt{let}\ a\ \texttt{=}\ M\ \texttt{in}\ N \mid x \\ & \mid (M_i)_{i\in I} \mid \texttt{case}\ M\ \texttt{of}\ \{(i, x) \Rightarrow N_i\}_{i\in I} \\ & \mid \texttt{sample}\ d\ \texttt{(}M\texttt{)} \mid \texttt{score}\ M \end{aligned}$$

and we use standard syntactic sugar to manipulate integers and booleans: $\mathbb{B} = 1+1$, $\mathbb{N} = \sum_{i\in\omega} 1$, and constants are given by the appropriate injections. Conditionals and sequencing can be expressed in the usual way: $\texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 = \texttt{case}\ M\ \texttt{of}\ \{(i, \_) \Rightarrow N_i\}_{i\in\{1,2\}}$, and $M; N = \texttt{let}\ a\ \texttt{=}\ M\ \texttt{in}\ N$, where a does not occur in N. In the grammar above:

- f ranges over measurable functions $[\![A]\!] \to [\![B]\!]$, where A and B are types;
- d ranges over a family of *parametric distributions* over the reals, *i.e.* measurable functions $\mathbb{R}^n \times \mathbb{R} \to \mathbb{R}$, for some $n \in \mathbb{N}$, such that for every $\mathbf{r} \in \mathbb{R}^n$, $\int d(\mathbf{r}, -) = 1$. For the purposes of this paper we ignore all issues related to invalid parameters, arising from *e.g.* a call to `gaussian` with standard deviation $\sigma = 0$. (An implementation could, say, choose to behave according to an alternative distribution in this case.)

The typing rules are as follows:

$$\frac{\Gamma \vdash M : A \qquad \Gamma, a : A \vdash N : B}{\Gamma \vdash \texttt{let}\ a\ \texttt{=}\ M\ \texttt{in}\ N : B} \qquad \frac{\Gamma \vdash M : \mathbb{R}^n \qquad d : \mathbb{R}^n \times \mathbb{R} \to \mathbb{R}}{\Gamma \vdash \texttt{sample}\ d\ \texttt{(}M\texttt{)} : \mathbb{R}}$$

$$\frac{\Gamma \vdash M : \mathbb{R}}{\Gamma \vdash \texttt{score}\ M : 1} \qquad \frac{}{\Gamma, a : A \vdash a : A} \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}$$

$$\frac{\Gamma \vdash M : \sum_{i\in I} A_i \qquad \Gamma, x : A_i \vdash N_i : C}{\Gamma \vdash \texttt{case}\ M\ \texttt{of}\ \{(i, x) \Rightarrow N_i\}_{i\in I} : C} \qquad \frac{\Gamma \vdash M_i : A_i}{\Gamma \vdash (M_i)_{i\in I} : \prod_{i\in I} A_i}$$

$$\frac{f : [\![A]\!] \to [\![B]\!] \text{ measurable} \qquad \Gamma \vdash M : A}{\Gamma \vdash f\ M : B}$$

Among the measurable functions f, we point out the following of interest:

- The usual product projections $\pi_i : [\![\prod_{i\in I} A_i]\!] \to [\![A_i]\!]$ and coproduct injections $\iota_i : [\![A_i]\!] \to [\![\coprod_{i\in I} A_i]\!]$;
- The operators $+, \times : \mathbb{R}^2 \to \mathbb{R}$,
- The tests, eg. $\geq 0 : [\![\mathbb{R}]\!] \to [\![\mathbb{B}]\!]$,
- The constant functions $1 \to A$ of the form $() \mapsto a$ for some $a \in [\![A]\!]$.

Examples for d include $\texttt{uniform} : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}$, $\texttt{gaussian} : \mathbb{R}^2 \times \mathbb{R} \to \mathbb{R}$, ...

2.3 Measure-theoretic semantics of programs

We now define a semantics of probabilistic programs using the measure-theoretic concept of *kernel*, which we define shortly. The content of this section is not new: using kernels as semantics for probabilistic was originally proposed in [14], while the (more recent) treatment of conditioning (`score`) via *s-finite* kernels is due to Staton [18]. Intuitively, kernels provide a semantics of open terms $\Gamma \vdash M : A$ as measures on $[\![A]\!]$ varying according to the values of variables in Γ.

Formally, a **kernel** from (X, Σ_X) to (Y, Σ_Y) is a function $k : X \times \Sigma_Y \to [0, \infty]$ such that for each $x \in X$, $k(x, -)$ is a measure, and for each $U \in \Sigma_Y$, $k(-, U)$ is measurable. (Here the σ-algebra $\Sigma_{[0,\infty]}$ is the restriction of that of $\mathbb{R}+\{\infty\}$.) We say k is **finite** (resp. **probabilistic**) if each $k(x, -)$ is a finite (resp. probability) measure, and it **s-finite** if it is a countable pointwise sum $\sum_{i\in I} k_i$ of finite kernels. We write $k : X \rightsquigarrow Y$ when k is an s-finite kernel from X to Y.

A term $\Gamma \vdash M : A$ will denote an s-finite kernel $[\![M]\!] : [\![\Gamma]\!] \rightsquigarrow [\![A]\!]$, where the context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ denotes the product of its components: $[\![\Gamma]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$.

Notice that any measurable function $f : X \to Y$ can be seen as a *deterministic* kernel $f^\dagger : X \rightsquigarrow Y$. Given two s-finite kernels $k : A \rightsquigarrow B$ and $l : A \times B \rightsquigarrow C$, we define their composition $l \circ k : A \rightsquigarrow C$:

$$(l \circ k)(a, X) = \int_{b\in B} l((a, b), C) \times k(a, \mathrm{d}b).$$

Staton [18] proved that $l \circ k$ is a s-finite kernel.

The interpretation of terms is defined by induction:

- $[\![()]\!]$ is the lifting of $[\![\Gamma]\!] \to 1 : x \mapsto ()$.
- $[\![\texttt{let } a \texttt{ = } M \texttt{ in } N]\!]$ is $[\![N]\!] \circ [\![M]\!]$
- $[\![f\, M]\!] = f^\dagger \circ [\![M]\!]$
- $[\![a]\!](x, X) = \delta_x(X)$, the Dirac distribution $\delta_x(X) = 1$ if $x \in X$ and zero otherwise.
- $[\![\texttt{sample } d \texttt{ (}M\texttt{)}]\!] = \texttt{sam} \circ [\![M]\!]$ where $\texttt{sam}_d : \mathbb{R}^n \rightsquigarrow \mathbb{R}$ is given by $\texttt{sam}_d(\mathbf{r}, X) = \int_{x\in X} d(\mathbf{r}, x)\mathrm{d}x$.
- $[\![\texttt{score } M]\!] = \texttt{sco} \circ [\![M]\!]$ where $\texttt{sco} : [\![R]\!] \to [\![1]\!]$ is $\texttt{sco}(r, X) = r \cdot \delta_{()}(X)$.
- $[\![(M_i)_{i\in I}]\!](\gamma, \prod_{i\in I} X_i) = \prod_{i\in I} [\![M_i]\!](\gamma, X_i)$: this is well-defined since the $\prod X_i$ generate the measurable sets of the product space.
- $[\![\texttt{case } M \texttt{ of } \{(i, x) \Rightarrow N_i\}_{i\in I}]\!] = \texttt{coprod} \circ [\![M]\!]$ where $\texttt{coprod} : \Gamma \times [\![\coprod_{i\in I} A_i]\!] \rightsquigarrow [\![B]\!]$ maps $(\gamma, \{i\} \times X)$ to $[\![N_i]\!](\gamma, X)$.

We observe that when M is a program making no use of conditioning (*i.e.* a generative model), the kernel $[\![M]\!]$ is probabilistic:

Lemma 1. *For $\Gamma \vdash M : A$ without scores, $[\![M]\!](\gamma, [\![A]\!]) = 1$ for each $\gamma \in [\![\Gamma]\!]$.*

2.4 Exact inference

Note that a kernel $1 \rightsquigarrow [\![A]\!]$ is the same as a measure on $[\![A]\!]$. Given a closed program $\vdash M : A$, the measure $[\![M]\!]$ is a combination of the prior (occurrences of

`sample`) and the likelihood (`score`). Because `score` can be called on arbitrary arguments, it may be the case that the measure of the total space (that is, the coefficient $[\![M]\!]([\![A]\!])$, often called the *model evidence*) is 0 or ∞.

Whenever this is *not* the case, $[\![M]\!]$ can be normalised to a probability measure, the posterior distribution. For every $U \in \Sigma_{[\![A]\!]}$,

$$\mathrm{norm}[\![M]\!](U) = \frac{[\![M]\!](U)}{[\![M]\!]([\![A]\!])}.$$

However, in many cases, this computation is intractable. Thus the goal of *approximate inference* is to approach $\mathrm{norm}[\![M]\!]$, the *true posterior*, using a well-chosen sequence of samples.

3 Approximate inference via intensional semantics

3.1 An introduction to approximate inference

In this section we describe the Metropolis-Hastings (MH) algorithm for approximate inference in the context of probabilistic programming. Metropolis-Hastings is a generic algorithm to sample from a probability distribution D on a measurable state space $\mathbb{X}$, of which we know the density $d : \mathbb{X} \to \mathbb{R}$ up to some normalising constant.

MH is part of a family of inference algorithms called *Monte-Carlo Markov chain*, in which the posterior distribution is approximated by a series of samples generated using a Markov chain.

Formally, the MH algorithm defines a Markov chain M on the state space $\mathbb{X}$, that is a probabilistic kernel $M : \mathbb{X} \rightsquigarrow \mathbb{X}$. The correctness of the MH algorithm is expressed in terms of convergence. It says that for almost all $x \in \mathbb{X}$, the distribution $M^n(x, \cdot)$ converges to D as n goes to infinity, where M^n is the n-iteration of M: $M \circ \ldots \circ M$. Intuitively, this means that iterated sampling from M gets closer to D with the number of iterations.

The MH algorithm is itself parametrised by a Markov chain, referred to as the **proposal kernel** $P : \mathbb{X} \rightsquigarrow \mathbb{X}$: for each sampled value $x \in \mathbb{X}$, a proposed value for the next sample is drawn according to $P(x, \cdot)$. Note that correctness only holds under certain assumptions on P.

The MH algorithm assumes that we know how to sample from P, and that its density is known, ie. there is a function $p : \mathbb{X}^2 \to \mathbb{R}$ such that $p(x, \cdot)$ is the density of the distribution $P(x, \cdot)$,

The MH algorithm. On an input state x, the MH algorithm samples from $P(x, \cdot)$ and gets a new sample x'. It then compares the likelihood of x and x' by computing an acceptance ratio $\alpha(x, x')$ which says whether the return state is x' or x. In pseudo-code, for an input state $x \in \mathbb{X}$:

1. Sample a new state x' from the distribution $P(x, \cdot)$

2. Compute the acceptance ratio of x' with respect to x:

$$\alpha(x, x') = \min\left(1, \frac{d(x') \times p(x, x')}{d(x) \times p(x', x)}\right)$$

3. With probability $\alpha(x, x')$, return the new sample x', otherwise return the input state x.

The formula for $\alpha(x, x')$ is known as the Hastings acceptance ratio and is key to the correctness of the algorithm.

Very little is assumed of P, which makes the algorithm very flexible; but of course the convergence rate may vary depending on the choice of P. We give a more formal description of MH in § 5.2.

Single-site MH and incremental recomputation. To apply this algorithm to probabilistic programming, we need a proposal kernel. Given a program M, the execution traces of M form a measurable set $\mathbb{X}_M$. In this setting the proposal is given by a kernel $\mathbb{X}_M \rightsquigarrow \mathbb{X}_M$.

A widely adopted choice of proposal is the *single-site proposal kernel* which, given a trace $x \in \mathbb{X}_M$, generates a new trace x' as follows:

1. Select uniformly one of the random choices s encountered in x.
2. Sample a new value for this instruction.
3. Re-execute the program M from that point onwards and with this new value for s, only ever resampling a variable when the corresponding instruction did not already appear in x.

Observe that there is some redundancy in this process: in the final step, the entire program has to be explored even though only a subset of the random choices will be re-evaluated. Some implementations of Trace MH for probabilistic programming make use of *incremental recomputation*.

We propose in this paper to statically compile a program M to an *event structure* G_M which makes explicit the probabilistic dependences between events, thus avoiding unnecessary sampling.

3.2 Capturing probabilistic dependencies using event structures

Consider the program depicted in Figure 1 in which we are interested in learning the parameters μ and σ of a Gaussian distribution from which we have observed two data points, say v_1 and v_2. For $i = 1, 2$ the function $f_i : \mathbb{R} \to \mathbb{R}$ expresses a soft constraint; it can be understood as indicating how much the sampled value of `x`i matches the observed value v_i.

A *trace* of this program will be of the form

$$\mathsf{Sam}\,\mu \cdot \mathsf{Sam}\,\sigma \cdot \mathsf{Sam}\,x_1 \cdot \mathsf{Sam}\,x_2 \cdot \mathsf{Sco}\,(f_1\,x_1) \cdot \mathsf{Sco}\,(f_2\,x_2) \cdot \mathsf{Rtn}\,(\mu, \sigma),$$

for some μ, σ, x_1, and $x_2 \in \mathbb{R}$ corresponding to sampled values for variables `mu`, `sigma`, `x1` and `x2`.

```
let mu = sample uniform (150, 200) in
let sigma = sample uniform (1, 50) in
let x1 = sample gaussian (mu, sigma) in
let x2 = sample gaussian (mu, sigma) in
score (f1 x1); score (f2 x2);
(mu, sigma)
```

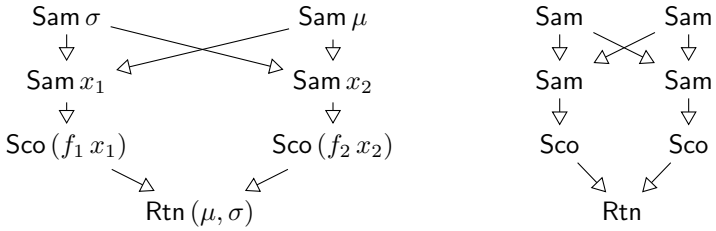
Fig. 1. A simple probabilistic program

A proposal step following the single-site kernel may choose to resample μ; then it must run through the entire trace, checking for potential dependencies to μ, though in this case none of the other variables need to be resampled.

So we argue that viewing a program as tree of traces is not most appropriate in this context: we propose instead to compile a program into a partially ordered structure reflecting the probabilistic dependencies.

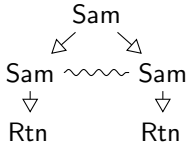
With our approach, the example above would yield the partial order displayed below on the right-hand side. The nodes on the first line corresponds to the sample for μ and σ, and those on the second line to x_1 and x_2. This provides an accurate account of the probabilistic dependencies: whenever $e \leq e'$ (where $\leq$ is the reflexive, transitive closure of $\twoheadrightarrow$), it is the case that e' depends on e.

According to this representation of the program, a trace is no longer a linear order, but instead another partial order, similar to the previous one only annotated with a specific value for each variable. This is displayed below, on the left-hand side; note that the order $\leq$ is drawn top to bottom. There is an obvious erasure map from the trace (left) to the graph (right); this will be important later on.



Conflict and control flow. We have seen that a partial order can be used to faithfully represent the data dependency in the program; it is however not sufficient to accurately describe the control flow. In particular, computational events may live in different *branches* of a conditional statement, as in the following example:

```
let x = sample uniform (0, 5) in
if x ≥ 2 then sample gaussian (3, 1)
else sample uniform (2, 4)
```



The last two samples are independent, but also *incompatible*: in any given trace only one of them will occur. An example of a trace for this program is $\mathsf{Sam}\,1 \cdot \mathsf{Sam}\,3 \cdot \mathsf{Rtn}\,3$.

We represent this information by enriching the partial order with a conflict relation, indicating when two actions are in different branches of a conditional statement. The resulting structure is depicted on the right. Combining partial order and conflict in this way can be conveniently formalised using **event structures** [22]:

Definition 1. *An **event structure** is a tuple $(E, \leq, \#)$ where $(E, \leq)$ is a partially ordered set and $\#$ is an irreflexive, binary relation on E such that*

- *for every $e \in E$, the set $[e] = \{e' \in E \mid e' \leq e\}$ is finite, and*
- *if $e \# e'$ and $e' \leq e''$, then $e \# e''$.*

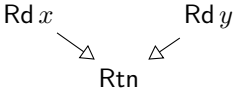
From the partial order $\leq$, we extract **immediate causality** $\rightarrowtriangle$: $e \rightarrowtriangle e'$ when $e < e'$ with no events in between; and from the conflict relation, we extract **minimal conflict** $\sim\!\!\!\sim$: $e \sim\!\!\!\sim e'$ when $e \# e'$ and there are no other conflicts in $[e] \cup [e']$. In pictures we draw $\rightarrowtriangle$ and $\sim\!\!\!\sim$ rather than $\leq$ and $\#$.

A subset $x \subseteq E$ is a **configuration** of E if it is down-closed (if $e' \leq e \in x$ then $e' \in x$) and conflict-free (if $e, e' \in x$ then $\neg(e \# e')$). So in this framework, configurations correspond to exactly to *partial* executions traces of E.

The configuration $[e]$ is the **causal history** of e; we also write $[e)$ for $[e] \setminus \{e\}$. We write $\mathscr{C}(E)$ for the set of all finite configurations of E, a partial order under inclusion. A configuration x is **maximal** if it is maximal in $\mathscr{C}(E)$: for every $x' \in \mathscr{C}(E)$, if $x \subseteq x'$ then $x = x'$. We use the notation $x \overset{e}{\text{—}\!\subset} x'$ when $x' = x \cup \{e\}$, and in that case we say x' **covers** x.

An event structure is **confusion-free** if minimal conflict is transitive, and if any two events e, e' in minimal conflict satisfy $[e) = [e')$.

Compositionality. In order to give semantics to the language in a compositional manner, we must consider arbitrary *open* programs, *i.e.* with free parameters. Therefore we also represent each call to a parameter a as a *read* event, marked $\mathsf{Rd}\,a$. For instance the program $x + y$ with two real parameters will become the event structure



Note that the read actions on x and y are independent in the program (no order is specified), and the event structure respects this independence.

Our dependency graphs are event structures where each event carries information about the syntactic operation it comes from, a **label**, which depends on the typing context of the program:

$$\mathscr{L}^{\mathtt{static}}_{\Gamma \vdash B} ::= \mathsf{Rd}\,a \mid \mathsf{Rtn} \mid \mathsf{Sam} \mid \mathsf{Sco}\,,$$

where a ranges over variables $a : A$ in Γ.

Definition 2. *A **dependency graph** over $\Gamma \vdash B$ is an event structure G along with a labelling map $\mathsf{lbl} : G \to \mathscr{L}^{\mathtt{static}}_{\Gamma\vdash B}$ where any two events $s, s' \in G$ labelled Rtn are in conflict, and all maximal configurations of G are of the form $[r]$ for $r \in G$ a return event.*

The condition on return events ensures that in any configuration of G there is at most one return event. Events of G are called static events.

We use dependency graphs as a causal representation of programs, reflecting the dependency between different parts of the program. In what follows we enrich this representation with runtime information in order to keep track of the dataflow of the program (in §3.3), and the associated distributions (in §3.4).

3.3 Runtime values and dataflow graphs

We have seen how data dependency can be captured by representing a program P as a dependency graph G_P. But observe that this graph does not give any runtime information about the data in P; every event $s \in G_P$ only carries a label $\mathsf{lbl}(s)$ indicating the class of action it belongs to. (For an event labelled $\mathsf{Rd}\, a$, G does not specify the value at a; whereas at runtime this will be filled by an element of $[\![A]\!]$ where A is the type of a.)

To each label, we can associate a measurable space of possible runtime values:

$$\mathscr{Q}(\mathsf{Rd}\, b) = [\![\Gamma(b)]\!] \qquad \mathscr{Q}(\mathsf{Rtn}) = [\![A]\!] \qquad \mathscr{Q}(\mathsf{Sam}) = (\mathbb{R}, \Sigma_{\mathbb{R}}) \qquad \mathscr{Q}(\mathsf{Sco}) = (\mathbb{R}, \Sigma_{\mathbb{R}}).$$

Then, in a particular execution, an event $s \in G_P$ has a value in $\mathscr{Q}(\mathsf{lbl}(s))$, and can be instead labelled by the following expanded set:

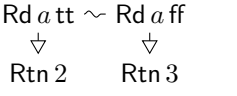
$$\mathscr{L}^{\mathtt{run}}_{\Gamma\vdash B} ::= \mathsf{Rd}\, a\, v \mid \mathsf{Rtn}\, v \mid \mathsf{Sam}\, r \mid \mathsf{Sco}\, r$$

where r ranges over real numbers; in $\mathsf{Rd}\, a\, v$, $a : A \in \Gamma$ and $v \in [\![A]\!]$; and in $\mathsf{Rtn}\, v$, v ranges over elements of $[\![B]\!]$. Notice that there is an obvious forgetful map $\alpha : \mathscr{L}^{\mathtt{run}}_{\Gamma\vdash A} \to \mathscr{L}^{\mathtt{static}}_{\Gamma\vdash A}$, discarding the runtime value. This runtime value can be extracted from a label in $\mathscr{L}^{\mathtt{run}}_{\Gamma\vdash B}$ as follows:

$$\mathbf{q}(\mathsf{Rd}\, b\, v) = v \qquad \mathbf{q}(\mathsf{Rtn}\, v) = v \qquad \mathbf{q}(\mathsf{Sam}\, r) = r \qquad \mathbf{q}(\mathsf{Sco}\, r) = r.$$

In particular, we have $\mathbf{q}(\ell) \in \mathscr{Q}(\alpha(\ell))$.

Such runtime events organise themselves in an event structure E_P, labelled over $\mathscr{L}^{\mathtt{run}}_{\Gamma\vdash B}$, the **runtime graph** of P. Runtime graphs are in general uncountable, and so difficult to represent pictorially. It can be done in some simple, finite cases: the graph for `if` a `then 2 else 3` is depicted on the right. Recall that in dependency graphs conflict was used to represent conditional branches; here instead conflict is used to keep disjoint the possible outcomes of the same static event. (Necessarily, this static event must be a sample or a read, since other actions (return, score) are deterministic.)



Intuitively one can project runtime events to static events by erasing the runtime information; this suggests the existence of a function $\pi_P : E_P \to G_P$. This function will turn out to satisfy the axioms of a *rigid map of event structures*:

Definition 3. *Given event structures $(E, \leq_E, \#_E)$ and $(G, \leq_G, \#_G)$ a function $\pi : E \to G$ is a **rigid map** if*

- *it preserves configurations: for every $x \in \mathscr{C}(E)$, $\pi x \in \mathscr{C}(G)$*
- *it is* locally injective*: for every $x \in \mathscr{C}(E)$ and $e, e' \in x$, if $\pi(e) = \pi(e')$ then $e = e'$.*
- *it preserves dependency: if $e \leq_E e'$ then $\pi(e) \leq_G \pi(e')$.*

In general π is not injective, since many runtime events may correspond to the same static event – in that case however the axioms will require them to be in conflict. The last condition in the definition ensures that all causal dependencies come from G.

Given $x \in \mathscr{C}(G_P)$ we define the possible runtime values for x as the set $\mathscr{Q}(x)$ of functions mapping $s \in x$ to a runtime value in $\mathscr{Q}(\mathsf{lbl}(s))$; in other words $\mathscr{Q}(x) = \prod_{s \in x} \mathscr{Q}(\mathsf{lbl}(s))$. A configuration x' of E_P can be viewed as a trace over $\pi_P\, x'$; hence $\pi_P^{-1}\{x\} := \{x' \in \mathscr{C}(E_P) \mid \pi_P x' = x\}$ is the set of traces of P over x. We can now define dataflow graphs:

Definition 4. *A **dataflow graph** on $\Gamma \vdash B$ is a triple $\mathbf{S} = (E_S, G_S, \pi_S : E_S \to G_S)$ with G_S a dependency graph and E_S a runtime graph, such that:*

- *π_S is a rigid map and $\mathsf{lbl} \circ \pi_S = \alpha \circ \mathsf{lbl} : E_S \to \mathscr{L}^{static}_{\Gamma \vdash B}$*
- *for each $x \in \mathscr{C}(G_S)$, the following function is injective*

$$\begin{aligned} q_x : \pi_S^{-1}\{x\} &\to \mathscr{Q}(x) \\ x' &\mapsto (s \mapsto \mathbf{q}(\mathsf{lbl}(s))) \end{aligned}$$

- *if $e, e' \in E_S$ with $e \rightsquigarrow e'$ then $\pi e = \pi e'$, and moreover e and e' are either both sample or both read events.*

As mentioned above, maximal configurations of E_P correspond to total traces of P, and will be the states of the Markov chain in § 5. By the second axiom, they can be seen as pairs $(x \in \mathscr{C}(G_S), q \in \mathscr{Q}(x))$. Because of the third axiom, E_S is always confusion-free.

Measurable fibres. Rigid maps are convenient in this context because, they allow for reasoning about program traces by organising them as *fibres*. The key property we rely on is the following:

Lemma 2. *If $\pi : E \to G$ is a rigid map of event structures, then the induced map $\pi : \mathscr{C}(E) \to \mathscr{C}(G)$ is a* discrete fibration*: that is, for every $y \in \mathscr{C}(E)$, if $x \subseteq \pi y$ for some $x \in \mathscr{C}(G)$, then there is a unique $y' \in \mathscr{C}(E)$ such that $y' \subseteq y$ and $\pi y' = x$.*

This enables an essential feature of our approach: given a configuration x of the dataflow graph G, the fibre $\pi^{-1}\{x\}$ over it contains all the (possibly partial) program traces over x, *i.e.* those whose path through the program corresponds to that of x. Additionally the lemma implies that every pair of configurations

$x, x' \in \mathscr{C}(G)$ such that $x \subseteq x'$ induces a **restriction map** $r_{x,x'} : \pi^{-1}\{x'\} \to \pi^{-1}\{x\}$, whose action on a program trace over x' is to return its *prefix* over x.

Although there is no measure-theoretic structure in the definition of dataflow graphs, we can recover it: for every $x \in \mathscr{C}(G_S)$, the fibre $\pi_S^{-1}\{x\}$ can be equipped with the σ-algebra induced from $\Sigma_{\mathscr{Q}(x)}$ via q_x; it is generated by sets $q_x^{-1}U$ for $U \in \Sigma_{\mathscr{Q}(x)}$.

It is easy to check that this makes the restriction map $r_{x,x'} : \pi_S^{-1}\{x'\} \to \pi_S^{-1}\{x\}$ measurable for each pair x, x' of configurations with $x \subseteq x'$. (Note that this makes $\mathbf{S}$ a *measurable event structure* in the sense of [16].) Moreover, the map $q_{x,s} : \pi_S^{-1}\{x\} \to \mathscr{Q}(\mathsf{lbl}(s))$ for $s \in x \in \mathscr{C}(G_S)$, mapping $x' \in \pi_S^{-1}\{x\}$ to $\mathbf{q}(\mathsf{lbl}(s'))$ for s' the unique antecedent by π_S of s in x', is also measurable.

We will also make use of the following result:

Lemma 3. *Consider a dataflow* $\mathbf{S}$ *and* $x, y, z \in \mathscr{C}(G_S)$ *with* $x \subseteq y$, $x \subseteq z$, *and* $y \cup z \in \mathscr{C}(G_S)$. *If* $y \cap z = x$, *then the space* $\pi_S^{-1}\{y \cup z\}$ *is isomorphic to the set*

$$\{(u_y, u_z) \in \pi_S^{-1}\{y\} \times \pi_S^{-1}\{z\} \mid r_{x,y}(u_y) = r_{x,z}(u_z)\},$$

with σ*-algebra generated by sets of the form* $\{(u_y, u_z) \in X_y \times X_z \mid X_y \in \Sigma_{\pi_S^{-1}\{y\}}, X_z \in \Sigma_{\pi_S^{-1}\{z\}}$ *and* $r_{x,y}(u_y) = r_{x,z}(u_z)\}$.

(For the reader with knowledge of category theory, this says exactly that the diagram

$$\begin{array}{ccc} \pi_S^{-1}\{y \cup z\} & \xrightarrow{r_{y,y\cup z}} & \pi_S^{-1}\{y\} \\ {\scriptstyle r_{z,y\cup z}}\downarrow & & \downarrow{\scriptstyle r_{x,y}} \\ \pi_S^{-1}\{z\} & \xrightarrow[r_{x,z}]{} & \pi_S^{-1}\{x\} \end{array}$$

is a pullback in the category of measurable spaces.)

3.4 Quantitative dataflow graphs

We can finally introduce the last bit of information we need about programs in order to perform inference: the probabilistic information. So far, in a dataflow graph, we know when the program is sampling, but not from which distribution. This is resolved by adding for each sample event s in the dependency graph a kernel $k_s : \pi^{-1}\{[s)\} \rightsquigarrow \pi^{-1}\{[s]\}$. Given a trace x over $[s)$, k_s specifies a probability distribution according to which x will be extended to a trace over $[s]$. This distribution must of course have support contained in the set $r_{[s),[s]}^{-1}\{x\}$ of traces over $[s]$ of which x is a prefix; this is the meaning of the technical condition in the definition below.

Definition 5. *A* ***quantitative dataflow graph*** *is a tuple* $\mathbf{S} = (E_S, G_S, \pi : E_S \to G_S, (k_s^S))$ *where for each sample event* $s \in G_S$, k_s^S *is a kernel* $\pi^{-1}\{[s)\} \rightsquigarrow \pi^{-1}\{[s]\}$ *satisfying for all* $x \in \pi^{-1}\{[s)\}$,

$$k_s^S(x, \pi^{-1}\{[s]\} \setminus r_{[s),[s]}^{-1}\{x\}) = 0.$$

This axiom stipulates that any extension $x' \in \pi_S^{-1}\{[s]\}$ of $x \in \pi_S^{-1}\{[s)\}$ drawn by k_s must contain x; in effect k_s only samples the runtime value for s.

From graphs to kernels. We show how to collapse a quantitative dataflow graph $\mathbf{S}$ on $\Gamma \vdash B$ to a kernel $[\![\Gamma]\!] \rightsquigarrow [\![B]\!]$. First, we extend the kernel family on sampling events ($k_s^S : \pi^{-1}\{[s)\} \rightsquigarrow \pi^{-1}\{[s]\}$) to a family ($k_s^{S[\gamma]} : \pi^{-1}\{[s)\} \rightsquigarrow \pi^{-1}\{[s]\}$) defined on *all* events $s \in S$, parametrised by the value of the environment $\gamma \in [\![\Gamma]\!]$. To define $k_s^{S[\gamma]}(x, \cdot)$ it is enough to specify its value on the generating set for $\Sigma_{\pi^{-1}\{[s]\}}$. As we have seen this contains elements of the form $q_{[s]}^{-1}(U)$ with $U \in \Sigma_{\mathscr{Q}([s])}$. We distinguish the following cases corresponding to the nature of s:

- If s is a sample event, $k_s^{S[\gamma]} = k_s^S$
- If s is a read on $a : A$, any $x \in \pi^{-1}[s)$ has runtime information $q_{[s)}(x)$ in $\mathscr{Q}([s))$ which can be extended to $\mathscr{Q}([s])$ by mapping s to $\gamma(a)$:
$$k_s^{S[\gamma]}(x, q_{[s]}^{-1}U) = \delta_{q_{[s)}(x)[s:=\gamma(a)]}(U)$$
- If s is a return or a score event: any $x \in \pi^{-1}\{[s)\}$ has at most one extension to $o(x) \in \pi^{-1}\{[s]\}$ (because return and score events cannot be involved in a minimal conflict): $k_s^{S[\gamma]}(x, q_{[s]}^{-1}(U)) = \delta_{q_{[s]}(o(x))}(U)$. If $o(x)$ does not exist, we let $k_s^{S[\gamma]}(x, X) = 0$.

We can now define a kernel $k_{x,s}^{S[\gamma]} : \pi^{-1}\{x\} \rightsquigarrow \pi^{-1}\{x'\}$ for every atomic extension $x \overset{s}{\text{—}\!\subset} x'$ in G_S, ie. when $x' \setminus x = \{s\}$, as follows:

$$k_{x,s}^{S[\gamma]}(y, U) = k_s(r_{[s),x}(y), \{w \in \pi_S^{-1}\{[s]\} \mid (y, w) \in U\}).$$

The second argument to k_s above is always measurable, by a standard measure-theoretic argument based on Lemma 3, as $x \cap [s] = [s)$.

From this definition we derive:

Lemma 4. *If $x \overset{s_1}{\text{—}\!\subset} x_1$ and $x \overset{s_2}{\text{—}\!\subset} x_2$ are concurrent extensions of x (*i.e. *s_1 and s_2 are not in conflict), then $k_{x_1,s_2}^{S[\gamma]} \circ k_{x,s_1}^{S[\gamma]} = k_{x_2,s_1}^{S[\gamma]} \circ k_{x,s_2}^{S[\gamma]}$.*

Given a configuration $x \in \mathscr{C}(G_S)$ and a covering chain $\emptyset \overset{s_1}{\text{—}\!\subset} x_1 \ldots \overset{s_n}{\text{—}\!\subset} x_n = x$, we can finally define a measure on $\pi^{-1}\{x\}$:

$$\mu_x^{S[\gamma]} = k_{x_{n-1},s_n}^{S[\gamma]} \circ \ldots \circ k_{\emptyset,s_1}^{S[\gamma]}(*, \cdot),$$

where $*$ is the only trace over $\emptyset$. The particular covering chain used does not matter by the previous lemma. Using this, we can define the kernel of a quantitative dataflow graph $\mathbf{S}$ as follows:

$$\mathsf{kernel}(\mathbf{S})(\gamma, X) = \sum_{r \in G_S, \mathsf{lbl}(r) = \mathsf{Rtn}} \mu_{[r]}^{S[\gamma]}(q_{[r],r}^{-1}(X)),$$

where the measurable map $q_{[r],r} : \pi^{-1}\{r\} \to [\![B]\!]$ looks up the runtime value of r in an element of the fibre over $[r]$ (defined in § 3.3).

Lemma 5. *$\mathsf{kernel}(\mathbf{S})$ is an s-finite kernel $[\![\Gamma]\!] \rightsquigarrow [\![B]\!]$.*

4 Programs as labelled event structures

We now detail our interpretation of programs as quantitative dataflow graphs. Our interpretation is given by induction, similarly to the measure-theoretic interpretation given in § 2.3, in which composition of kernels plays a central role. In § 4.1, we discuss how to compose quantitative dataflow graphs, and in § 4.2, we define our interpretation.

4.1 Composition of probablistic event structures

Consider two quantitative dataflow graphs, S on $\Gamma \vdash A$, and T on $\Gamma, a : A \vdash B$ where a does not occur in Γ. In what follows we show how they can be composed to form a quantitative dataflow graph $T \odot^a S$ on $\Gamma \vdash B$.

Unlike in the kernel model of § 2.3, we will need two notions of composition. The first one is akin to the usual sequential composition: actions in T must wait on S to return before they can proceed. The second is closer to parallel composition: actions on T which do not depend on a read of the variable a can be executed in parallel with S. The latter composition is used to interpret the let construct. In $\texttt{let}\ a\ \texttt{=}\ M\ \texttt{in}\ N$, we want all the probabilistic actions or reads on other variables which do not depend on the value of a to be in parallel with M. However, in a program such as $\texttt{case}\ M\ \texttt{of}\ \{(i, x) \Rightarrow N_i\}_{i \in I}$ we do not want any actions of N_i to start before the selected branch is known, *i.e.* before the return value of M is known.

By way of illustration, consider the following simple example, in which we only consider runtime graphs, ignoring the rest of the structure for now. Suppose S and T are given by

$$S = \begin{array}{cc} \mathsf{Rd}\, b\, \mathsf{tt} & \sim\ \mathsf{Rd}\, b\, \mathsf{ff} \\ \downarrow & \downarrow \\ \mathsf{Rtn}\, \mathsf{ff} & \mathsf{Rtn}\, \mathsf{tt} \end{array} \qquad T = \begin{array}{ccc} \mathsf{Sam}\ r & & \mathsf{Rd}\, a\, \mathsf{tt} \sim\!\!\sim \mathsf{Rd}\, a\, \mathsf{ff} \\ & \searrow\ \downarrow & \downarrow \\ & \mathsf{Rtn}\, ((), \mathsf{tt}) & \mathsf{Rtn}\, ((), \mathsf{ff}) \end{array}$$

The graph S can be seen to correspond to the program $\texttt{if}\ b\ \texttt{then}\ \mathsf{ff}\ \texttt{else}\ \mathsf{tt}$ and T to the pairing $(\texttt{sample}\ d\ (0), a)$ for any d. Here S is a runtime graph on $b : \mathbb{B} \vdash \mathbb{B}$ and T on $a : \mathbb{B}, b : \mathbb{B} \vdash \mathbb{B}$.

Both notions of compositions are displayed in the diagram below. The sequential composition (left) corresponds to

$$\texttt{if}\ b\ \texttt{then}\ (\texttt{sample}\ d\ (0), \mathsf{ff})\ \texttt{else}\ (\texttt{sample}\ d\ (0), \mathsf{tt})$$

and the parallel composition to $(\texttt{sample}\ d\ (0), \texttt{if}\ b\ \texttt{then}\ \mathsf{ff}\ \texttt{else}\ \mathsf{tt})$:

$$T \odot^a_{\mathrm{seq}} S = \begin{pmatrix} \mathsf{Rd}\, b\, \mathsf{tt} & \sim\ \mathsf{Rd}\, b\, \mathsf{ff} \\ \downarrow & \downarrow \\ \mathsf{Sam}\ r & \mathsf{Sam}\ r \\ \downarrow & \downarrow \\ \mathsf{Rtn}\, \mathsf{ff} & \mathsf{Rtn}\, \mathsf{tt} \end{pmatrix} \qquad T \odot^a_{\mathrm{par}} S = \begin{pmatrix} \mathsf{Sam}\ r & & \mathsf{Rd}\, b\, \mathsf{tt} \sim \mathsf{Rd}\, b\, \mathsf{ff} \\ & \searrow\ \downarrow & \downarrow \\ & \mathsf{Rtn}\, \mathsf{ff} & \mathsf{Rtn}\, \mathsf{tt} \end{pmatrix}$$

Composition of runtime and dependency graphs. Let us now define both composition operators at the level of the event structures. Through the bijection $\mathscr{L}^{\mathtt{static}}_{\Gamma\vdash B} \simeq \mathscr{L}^{\mathtt{run}}_{\Gamma'\vdash 1}$ where $\Gamma'(a) = 1$ for all $a \in \mathrm{dom}(\Gamma)$, we will see dependency graphs and runtime graphs as the same kind of objects, event structures labelled over $\mathscr{L}^{\mathtt{run}}_{\Gamma\vdash A}$.

The two compositions $S \odot^a_{\mathrm{par}} T$ and $S \odot^a_{\mathrm{seq}} T$ are two instances of the same construction, parametrised by a set of labels $D \subseteq \mathscr{L}^{\mathtt{run}}_{\Gamma,a:A\vdash B}$. Informally, D specifies which events of T are to depend on the return value of S in the resulting composition graph. It is natural to assume in particular that D contains all reads on a, and all return events.

Sequential and parallel composition are instances of this construction where D is set to one of the following:

$$D^{\Gamma,a:A\vdash B}_{\mathrm{seq}} = \mathscr{L}^{\mathtt{run}}_{\Gamma,a:A\vdash B} \qquad D^{\Gamma,a:A\vdash B}_{\mathrm{par}} = \{\mathsf{Rd}\, a\, v, \mathsf{Rtn}\, v \in \mathscr{L}^{\mathtt{run}}_{\Gamma,a:A\vdash B}\}.$$

We proceed to describe the construction for an abstract D. Let T be an event structure labelled by $\mathscr{L}^{\mathtt{run}}_{\Gamma,a:A\vdash B}$ and S labelled by $\mathscr{L}^{\mathtt{run}}_{\Gamma\vdash A}$. A configuration $x \in \mathscr{C}(S)$ is a **justification** of $y \in \mathscr{C}(T)$ when

1. if $\mathsf{lbl}(y)$ intersects D, then x contains a return event
2. for all $t \in y$ with label $\mathsf{Rd}\, a\, v$, there exists an event $s \in x$ labelled $\mathsf{Rtn}\, v$.

In particular if $\mathsf{lbl}(y)$ does not intersect D, then any configuration of S is a justification of y. A **minimal justification** of y is a justification that admits no proper subset which is also a justification of y. We now define the event structure $S \cdot_D T$ as follows:

- *Events*: $S \cup \{(x,t) \mid x \in \mathscr{C}(S), t \in T, x \text{ minimal justification for } [t]\}$;
- *Causality*: $\leq_S \ \cup \{(x,t),(x',t') \mid x \subseteq x' \wedge t \leq t'\} \ \cup \ \{s,(x,t) \mid s \in x\}$;
- *Conflict*: the symmetric closure of

$$\#_S \cup \{(x,t),(x',t') \mid x \cup x' \notin \mathscr{C}(T) \vee t \#_B t'\}$$
$$\cup \{s,(x,t) \mid \{s\} \cup x \notin \mathscr{C}(S)\}.$$

Lemma 6. *$S \cdot_D T$ is an event structure, and the following is an order-isomorphism:*

$$\langle \cdot, \cdot \rangle : \{(x,y) \in \mathscr{C}(S) \times \mathscr{C}(T) \mid x \text{ is a justification of } y\} \cong \mathscr{C}(S \cdot_D T).$$

This event structure is not quite what we want, since it still contains return events from S and reads on a from T. To remove them, we use the following general construction. Given a Σ-labelled event structure E and $V \subseteq E$ a set of visible events, its **projection** $E \downarrow V$ has events V and causality, conflict and labelling inherited from E. Thus the composition of S and T is:

$$S \odot^a_D T := S \cdot_D T \downarrow (\{s \in S \mid s \text{ not a return}\} \cup \{(x,t) \mid t \text{ not a read on } a\}).$$

As a result $S \odot^a_D T$ is labelled over $\mathscr{L}^{\mathtt{run}}_{\Gamma\vdash B}$ as needed.

Dataflow information. We now explain how this construction lifts to dataflow graphs. Consider dataflow graphs $S = (E_S, G_S, \pi_S : E_S \to G_S)$ on $\Gamma \vdash A$ and $T = (E_T, G_T, \pi_T : E_T \to E_T)$ on $\Gamma, a : A \vdash B$. Given $D \subseteq \mathscr{L}^{\mathtt{static}}_{\Gamma,a:A\vdash B}$ we define

$$E_{S \cdot_D T} = E_S \cdot_{\alpha^{-1}D} E_T \qquad G_{S \cdot_D T} = G_S \cdot_D G_T$$
$$E_{S \odot^a_D T} = E_S \odot^a_{\alpha^{-1}D} E_T \qquad G_{S \odot^a_D T} = G_S \odot^a_D G_T$$

Lemma 7. *The maps π_S and π_T extend to rigid maps*

$$\pi_{S \cdot_D T} : E_{S \cdot_{\alpha^{-1}D} T} \to G_{S \cdot_D T}$$
$$\pi_{S \odot^a_D T} : E_{S \odot^a_{\alpha^{-1}D} T} \to G_{S \odot^a_D T}$$

Moreover, if $\langle x, y\rangle \in \mathscr{C}(E_{S \cdot_D T})$, $\langle \pi_S\, x, \pi_T\, y\rangle$ is a well-defined configuration of $G_{S \cdot_D T}$. As a result, for $\langle x, y\rangle \in \mathscr{C}(G_{S \cdot_D T})$, we have a injection $\varphi_{x,y} : \pi^{-1}\{\langle x, y\rangle\} \to \pi^{-1}\{x\} \times \pi^{-1}\{y\}$ making the following diagram commute:

$$\begin{array}{ccc} \pi^{-1}\{\langle x, y\rangle\} & \xrightarrow{\varphi_{x,y}} & \pi^{-1}\{x\} \times \pi^{-1}\{y\} \\ q_{\langle x,y\rangle}\downarrow & & \downarrow q_x \times q_y \\ \mathscr{Q}(\langle x, y\rangle) & \xrightarrow[\cong]{} & \mathscr{Q}(x) \times \mathscr{Q}(y) \end{array}$$

In particular, $\varphi_{x,y}$ is measurable and induces the σ-algebra on $\pi^{-1}\{\langle x, y\rangle\}$. We write φ_x for the map $\varphi_{x,\emptyset}$, an isomorphism.

Adding probability. At this point we have defined all the components of dataflow graphs $S \odot^a_D T$ and $S \cdot_D T$. We proceed to make them quantitative.

Observe first that each sampling event of $G_{S \cdot_D T}$ (or equivalently of $G_{S \odot^a_D T}$ – sampling events are never hidden) corresponds either to a sampling event of G_S, or to an event (x, t) where t is a sampling event of G_T. We consider both cases to define a family of kernels $(k_s^{S \cdot_D T})$ between the fibres of $S \cdot_D T$. This will in turn induce a family $(k_s^{S \odot^a_D T})$ on $S \odot^a_D T$.

- If s is a sample event of G_S, we use the isomorphisms $\varphi_{[s)}$ and $\varphi_{[s]}$ of Lemma 7 to define:
$$k_s^{S \odot^a_D T}(v, X) = k_s^S(\varphi^{-1}_{[s)}\, v, \varphi^{-1}_{[s]} X).$$
- If s corresponds to (x, t) for t a sample event of G_T, then for every $X_x \in \Sigma_{\pi_S^{-1}\{x\}}$ and $X_t \in \Sigma_{\pi_T^{-1}\{[t)\}}$ we define
$$k^{S \odot^a_D T}_{(x,t)}(\langle x', y'\rangle, \varphi^{-1}_{x,[t]}(X_x \times X_t)) = \delta_{x'}(X_x) \times k_t^T(y', X_t).$$
By Lemma 7, the sets $\varphi^{-1}_{x,[t]}(X_x \times X_t)$ form a basis for $\Sigma_{\pi^{-1}\{\langle x,[t)\rangle\}}$, so that this definition determines the entire kernel.

So we have defined a kernel $k_s^{S \cdot_D T}$ for each sample event s of $G_{S \cdot_D T}$. We move to the composition $(S \odot_D^a T)$. Recall that the *causal history* of a configuration $z \in \mathscr{C}(G_{S \odot_D^a T})$ is the set $[z]$, a configuration of $G_{S \cdot_D T}$. We see that hiding does not affect the fibre structure:

Lemma 8. *For any $z \in \mathscr{C}(G_{S \odot_D^a T})$, there is a measurable isomorphism $\psi_z : \pi^{-1}_{S \odot_D^a T}\{z\} \cong \pi^{-1}_{S \cdot_D T}\{[z]\}$.*

Using this result and the fact that $G_{S \odot_D^a T} \subseteq G_{S \cdot_D T}$, we may define for each s:

$$k_s^{S \odot_D^a T}(v, X) = k_s^{S \cdot_D T}(\psi_{[s]}(v), \psi_{[s]} X).$$

We conclude:

Lemma 9. *$S \odot_D^a T := (G_{S \odot_D^a T}, E_{S \odot_D^a T}, \pi_{S \odot_D^a T}, (k_s^{S \odot_D^a T}))$ is a quantitative dataflow graph on $\Gamma \vdash B$.*

Multicomposition. By chaining this composition, we can compose on several variables at once. Given quantitative dataflow graphs S_i on $\Gamma \vdash A_i$ and T on $\Gamma, a_1 : A_1, \ldots, a_n : A_n \vdash A$ we define

$$(S_i) \odot_{\text{par}}^{(a_i)} T := S_1 \odot_{\text{par}}^{a_1} (\ldots \odot_{\text{par}}^{a_n} T)$$
$$(S_i) \odot_{\text{seq}}^{(a_i)} T := S_1 \odot_{\text{seq}}^{a_1} (\ldots \odot_{\text{seq}}^{a_n} T)$$

4.2 Interpretation of programs

We now describe how to interpret programs of our language using quantitative dataflow graphs. To do so we follow the same pattern as for the measure-theoretical interpretation given in § 2.3.

Interpretation of functions. Given a measurable function $f : [\![A]\!] \to [\![B]\!]$, we define the quantitative dataflow graph

$$S_f^a = \left(\sum_{v \in [\![A]\!]} \begin{array}{c} \mathsf{Rd}\, a\, v \\ \downarrow \\ \mathsf{Rtn}\,(f\, v) \end{array} \to \begin{array}{c} \mathsf{Rd}\, a \\ \downarrow \\ \mathsf{Rtn} \end{array} \right).$$

We then define $[\![f\, M]\!]_{\mathcal{G}}$ as $[\![M]\!]_{\mathcal{G}} \odot_{\text{par}}^a S_f^a$ where a is chosen so as not to occur free in M.

Probablistic actions. In order to interpret scoring and sampling primitives, we need the following two quantitative dataflow graphs:

$$\mathtt{score} = \left(\sum_{r \in \mathbb{R}} \begin{array}{c} \mathsf{Rd}\, a\, r \\ \downarrow \\ \mathsf{Sco}\, r \\ \downarrow \\ \mathsf{Rtn}\,() \end{array} \to \begin{array}{c} \mathsf{Rd}\, a \\ \downarrow \\ \mathsf{Sco} \\ \downarrow \\ \mathsf{Rtn} \end{array} \right) \qquad \mathtt{sample}_d = \left(\sum_{\mathbf{r} \in \mathbb{R}^n} \begin{array}{c} \mathsf{Rd}\, a\, \mathbf{r} \\ \downarrow \\ \mathsf{Sam}\, s \\ \downarrow \\ \mathsf{Rtn}\,() \end{array} \to \begin{array}{c} \mathsf{Rd}\, a \\ \downarrow \\ \mathsf{Sam} \\ \downarrow \\ \mathsf{Rtn} \end{array} , k_{\mathsf{Sam}} \right)$$

and we define k_{Sam} by integrating the density function d; here we identify $\mathscr{Q}(\{\mathsf{Rd}\, a\, , \mathsf{Sam}\})$ and $\pi^{-1}\{\{\mathsf{Rd}\, a\, , \mathsf{Sam}\}\}$:

$$k_{\mathsf{Sam}}(\{\mathsf{Rd}\, a\, \mathbf{r}\}, U) = \int_{q \in U, q(\mathsf{Rd}\, a) = \mathbf{r}} d(\mathbf{r}, q(\mathsf{Sam}))\mathrm{d}\lambda.$$

We can now interpret scoring and sampling constructs:

$$[\![\texttt{score}\ \ M]\!]_{\mathcal{G}} = [\![M]\!]_{\mathcal{G}} \odot^{a}_{\mathrm{par}} \texttt{score} \qquad [\![\texttt{sample}\ \ d\ \ \texttt{(}M\texttt{)}]\!]_{\mathcal{G}} = [\![M]\!]_{\mathcal{G}} \odot^{a}_{\mathrm{par}} \texttt{sample}_{d}.$$

Interpretation of tuples and variables. Given a family $(a_i)_{i \in I}$, we define the dataflow graph $\texttt{tuple}_{(a_i : A_i)}$ on $a_1 : A_1, \ldots, a_n : A_n \vdash A_1 \times \ldots \times A_n$ as follows. Its set of events is the disjoint union

$$\bigcup_{i \in I, v \in [\![A_i]\!]} \mathsf{Rd}\, a_i\, v + \bigcup_{\mathbf{v} \in [\![A_1 \times \ldots \times A_n]\!]} \mathsf{Rtn}\, \mathbf{v}$$

where the conflict is induced by $\mathsf{Rd}\, a_i\, v \sim\!\!\sim \mathsf{Rd}\, a_i\, v'$ for $v \neq v'$; and causality contains all the pairs $\mathsf{Rd}\, a_i\, v \rightarrowtriangle \mathsf{Rtn}\, (v_1, \ldots, v_n)$ where $v_i = v$. Then we form a quantitative dataflow graph $\texttt{Tuple}_{(a_i : A_i)}$, whose dependency graph is $\texttt{tuple}_{(a_i : 1)}$ (up to the bijection $\mathscr{L}^{\texttt{run}}_{\Gamma \vdash A} \simeq \mathscr{L}^{\texttt{static}}_{\Gamma' \vdash 1}$ where $\Gamma'(a) = 1$ for $a \in \mathrm{dom}(\Gamma)$); and the runtime graph is $\texttt{tuple}_{(a_i : A_i)}$, along with the obvious rigid map between them.

We then define the semantics of $(M_1, \ldots, M_n)$:

$$[\![(M_1, \ldots, M_n)]\!]_{\mathcal{G}} = ([\![M_i]\!]_{\mathcal{G}})_i \odot^{(a_i)}_{\mathrm{par}} \texttt{Tuple}_{\mathtt{a_i : A_i}},$$

where the a_i are chosen free in all of the M_j. This construction is also useful to interpret variables:

$$[\![a]\!]_{\mathcal{G}} = \texttt{Tuple}_{a : A} \qquad \text{where } \Gamma \vdash a : A.$$

Interpretation of pattern matching. Consider now a term of the form $\texttt{case}\, M\, \texttt{of}\ \{(i, a) \Rightarrow N_i\}_{i \in i}$. By induction, we have that $[\![N_i]\!]_{\mathcal{G}}$ is a quantitative dataflow graph on $\Gamma, a : A_i \vdash B$. Let us write $[\![N_i]\!]^{*}_{\mathcal{G}}$ for the quantitative dataflow graph on $\Gamma, a : (\sum_{i \in I} A_i) \vdash B$ obtained by relabelling events of the form $\mathsf{Rd}\, a\, v$ to $\mathsf{Rd}\, a\, (i, v)$, and sequentially precomposing with $\texttt{Tuple}_{a : \sum_{i \in I} A_i}$. This ensures that minimal events in $[\![N_i]\!]^{*}_{\mathcal{G}}$ are reads on a. We then build the quantitative dataflow graph $\sum_{i \in I} [\![N_i]\!]^{*}_{\mathcal{G}}$ on $\Gamma, a : \sum_{i \in I} A_i \vdash B$. This can be composed with $[\![M]\!]_{\mathcal{G}}$:

$$[\![\texttt{case}\ \ M\ \ \texttt{of}\ \ \{(i, a) \Rightarrow N_i\}_{i \in I}]\!]_{\mathcal{G}} = [\![M]\!]_{\mathcal{G}} \odot^{a}_{\mathrm{seq}} \left(\sum_{i \in I} [\![N_i]\!]^{*}_{\mathcal{G}} \right).$$

It is crucial here that one uses *sequential* composition: none of the branches must be evaluated until the outcome of M is known.

Adequacy of composition. We now prove that our interpretation is adequate with respect to the measure-theoretic semantics described in § 2.3. Given any subset $D \subseteq \mathscr{L}^{\mathtt{static}}_{\Gamma,a:A\vdash B}$ containing returns and reads on a, we show that the composition $S \odot^a_D T$ does implement the composition of kernels:

Theorem 1. *For S a quantitative dataflow graph on $\Gamma \vdash A$ and T on $\Gamma, a : A \vdash B$, we have*

$$\mathsf{kernel}(S \odot^a_D T) = \mathsf{kernel}(T) \circ \mathsf{kernel}(S) : [\![\Gamma]\!] \to [\![B]\!].$$

From this result, we can deduce that the semantics in terms of quantitative dataflow graphs is adequate with respect to the measure-theoretic semantics:

Theorem 2. *For every term $\Gamma \vdash M : A$, $\mathsf{kernel}([\![M]\!]_{\mathcal{G}}) = [\![M]\!]$.*

5 An inference algorithm

In this section, we exploit the intensional semantics defined above and define a Metropolis-Hastings inference algorithm. We start, in § 5.1, by giving a concrete presentation of those quantitative dataflow graphs arising as the interpretation of probabilistic programs; we argue this makes them well-suited for manipulation by an algorithm. Then, in § 5.2, we give a more formal introduction to the Metropolis-Hastings sampling methods than that given in § 3.1. Finally, in § 5.3, we build the proposal kernel on which our implementation relies, and conclude.

5.1 A concrete presentation of probabilistic dataflow graphs

Quantitative dataflow graphs as presented in the previous sections are not easy to handle inside of an algorithm: among other things, the runtime graph has an uncountable set of events. In this section we show that some dataflow graphs, in particular those needed for modelling programs, admit a finite representation.

Recovering fibres. Consider a dataflow graph $\mathbf{S} = (E_S, G_S, \pi_S)$ on $\Gamma \vdash B$. It follows from Lemma 3 that the fibre structure of $\mathbf{S}$ is completely determined by the spaces $\pi_S^{-1}\{[s]\}$, for $s \in G_S$, so we focus on trying to give a simplified representation for those spaces.

First, let us notice that if s is a return or score event, given $x \in \pi^{-1}\{x\}$, the value $q_x(s)$ is determined by $q|_{[s)}$. In other words the map $\pi^{-1}\{[s]\} \to \mathscr{Q}([s))$ is an injection. This is due to the fact that minimal conflict in E_S cannot involve return or score events. As a result, E_S induces a partial function $o^S_s : \mathscr{Q}([s)) \rightharpoonup \mathscr{Q}(\mathsf{lbl}(s))$, called the **outcome function**. It is defined as follows:

$$o^S_s(q) = \begin{cases} q_{[s]}(x')(s) & \text{if there exists } x' \in \pi^{-1}\{x'\},\ q_{[s]}(x')|_{[s)} = q, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that x' must be unique by the remark above since its projection to $\mathscr{Q}([s))$ is determined by q. The function o^S is partial, because it might be the case that the event s occurs conditionally on the runtime value on $[s)$.

In fact this structure is all we need in order to describe a dataflow graph:

Lemma 10. *Given G_S a dependency graph on $\Gamma \vdash B$, and partial functions $(o_s) : \mathscr{Q}([s]) \rightharpoonup \mathscr{Q}(\mathsf{lbl}(s))$ for score and return events of S. There exists a dataflow graph $(E_S, G_S, \pi_S : E_S \to G_S)$ whose outcome functions coincide with the o_s. Moreover, there is an order-isomorphism*

$$\mathscr{C}(E_S) \cong \{(x, q) \mid x \in \mathscr{C}(G_S), q \in \mathscr{Q}(x), \forall s \in x, o_s(q|_{[s)}) = q(s)\}.$$

Adding probabilities. To add probabilities, we simply equip each sample event s of G_S with a density function $d_s : \mathscr{Q}([s)) \times \mathbb{R} \rightharpoonup \mathbb{R}$.

Definition 6. *A **concrete quantitative dataflow graph** is a tuple $(G_S, (o_s : \mathscr{Q}([s)) \rightharpoonup \mathscr{Q}(\mathsf{lbl}(s))), (d_s : \mathscr{Q}([s)) \times \mathbb{R} \rightharpoonup \mathbb{R})_{s \in sample(G_S)})$ where $d_s(x, \cdot)$ is normalised.*

Lemma 11. *Any concrete quantitative dataflow graph $\mathcal{S}$ unfolds to a quantitative dataflow graph $\mathtt{unfold}\ \mathcal{S}$.*

We see now that the quantitative dataflow graphs arising as the interpretation of a program must be the unfolding of a concrete quantitative dataflow graph:

Lemma 12. *For any concrete quantitative dataflow graphs $\mathcal{S}$ on $\Gamma \vdash A$ and $\mathcal{T}$ on $\Gamma, a : A \vdash B$, $\mathtt{unfold}\ \mathcal{S} \odot^a_D T\mathtt{unfold}\ \mathcal{T}$ is the unfolding of a concrete quantitative dataflow graph. It follows that for any program $\Gamma \vdash M : B$, $[\![M]\!]_{\mathcal{G}}$ is the unfolding of a concrete quantitative dataflow graph.*

5.2 Metropolis-Hastings

Recall that the Metropolis-Hastings algorithm is used to sample from a density function $d : \mathbb{A} \to \mathbb{R}$ which may not be normalised. Here $\mathbb{A}$ is a measurable *state space*, equipped with a measure λ. The algorithm works by building a Markov chain whose stationary distribution is D, the probability distribution obtained from d after normalisation:

$$\forall X \in \Sigma_{\mathbb{A}}, D(X) = \frac{\int_{x \in X} d(x)}{\int_{x \in \mathbb{A}} d(x)}.$$

Our presentation and reasoning in the rest of this section are inspired by the work of Borgström et al. [2].

Preliminaries on Markov chains. A Markov chain on a measurable state space $\mathbb{A}$ is a probability kernel $k : \mathbb{A} \rightsquigarrow \mathbb{A}$, viewed as a transition function: given a state $x \in \mathbb{A}$, the distribution $k(x, \cdot)$ is the distribution from which a next sample state will be drawn. Usually, each $k(x, \cdot)$ comes with a procedure for sampling: we will treat this as a probabilistic program $M(x)$ whose output is the next state. Given an initial state $x \in \mathbb{A}$ and a natural number $n \in \mathbb{N}$, we have a distribution

$k^n(x,\cdot)$ on $\mathbb{A}$ obtained by iterating k n times. We say that the Markov chain k has **limit** the distribution μ on $\mathbb{A}$ when

$$\lim_{n\to\infty} ||k^n(x,\cdot) - \mu|| = 0 \qquad \text{where } ||\mu_1 - \mu_2|| = \sup_{A\in\Sigma_{\mathbb{A}}} \mu_1(A) - \mu_2(A).$$

For the purposes of this paper, we call a Markov chain $k : \mathbb{A} \to \mathbb{A}$ **computable** when there exists a type A such that $[\![A]\!] = \mathbb{A}$ (up to iso) and an expression *without scores* $x : A \vdash K : A$ such that $[\![K]\!] = k$. (Recall that programs without conditioning denote probabilistic kernels, and are easily sampled from, since all standard distributions in the language are assumed to come with a built-in sampler.)

We will use terms of our language to describe computable Markov chains language, taking mild liberties with syntax. We assume in particular that programs may call each other as subroutines (this can be done via substitutions), and that manipulating finite structures is computable and thus representable in the language.

The Metropolis-Hastings algorithm. Recall that we wish to sample from a distribution with un-normalised density $d : \mathbb{A} \to \mathbb{R}$; d is assumed to be computable. The Markov chain defined by the Metropolis-Hastings algorithm has two parameters: a computable Markov chain $x : A \vdash P : A$, the *proposal kernel*, and a measurable, computable function $p : \mathbb{A}^2 \to \mathbb{R}$ representing the kernel $[\![P]\!]$, *i.e.*

$$[\![P]\!](x, X') = \int_{x'\in X'} p(x,x')\, d\lambda(x').$$

The Markov-chain $\texttt{MH}(P,p,d)$ is defined as

$$\begin{aligned}\texttt{MH}(P,p,d)(x) := {}& \texttt{let } x' = P(x) \texttt{ in}\\ & \texttt{let } \alpha = \min\left(1, \frac{d(x')\times p(x,x')}{d(x)\times p(x',x)}\right) \texttt{ in}\\ & \texttt{let } u = \texttt{sample uniform } (0,1) \texttt{ in}\\ & \texttt{if } u < \alpha \texttt{ then } x' \texttt{ else } x\end{aligned}$$

In words, the Markov chain works as follows: given a start state x, it generates a proposal for the next state x' using P. It then computes an *acceptance ratio* α, which is the probability with which the new sample will be *accepted*: the return state will then either be the original x or x', accordingly.

Assuming P and p satisfy a number of conditions, the algorithm is correct:

Theorem 3. *Assume that P and p satisfies the following properties:*

1. ***Strong irreducibility**: There exists $n \in \mathbb{N}$ such that for all $x \in \mathbb{A}$ and $X \in \Sigma_{\mathbb{A}}$ such that $D(X) \neq \emptyset$ and $d(x) > 0$, there exists $n \in \mathbb{N}$ such that $[\![P]\!]^n(x, X) > 0$.*
2. *$[\![P]\!](x, X') = \int_{x'\in X'} p(x,x')$.*
3. *If $d(x) > 0$ and $p(x,y) > 0$ then $d(y) > 0$.*

4. If $d(x) > 0$ and $d(y) > 0$, then $p(x, y) > 0$ iff $p(y, x) > 0$.

Then, the limit of $\texttt{MH}(P, p, d)$ for any initial state $x \in \mathbb{A}$ with $d(x) > 0$ is equal to D, the distribution obtained after normalising d.

5.3 Our proposal kernel

Consider a closed program $\vdash M : A$ in which every measurable function is a computable one. Then, its interpretation as a concrete quantitative dataflow graph is computable, and we write $\mathcal{S}$ for the quantitative dataflow graph whose unfolding is $[\![M]\!]_{\mathcal{G}}$. Moreover, because M is closed, its measure-theoretic semantics gives a measure $[\![M]\!]$ on $[\![A]\!]$. Assume that $\mathrm{norm}([\![M]\!])$ is well-defined: it is a probability distribution on $[\![A]\!]$. We describe how a Metropolis-Hastings algorithm may be used to sample from it, by reducing this problem to that of sampling from configurations of E_S according to the following density:

$$d_S(x, q) := \left(\prod_{s \in \mathrm{sample}(x)} d_s(q(s)) \right) \left(\prod_{s \in \mathrm{score}(x)} q(s) \right).$$

Lemma 10 induces a natural measure on $\mathscr{C}(E_S)$. We have:

Lemma 13. *For all $X \in \Sigma_{\mathscr{C}(E_S)}$, $\mu^{\mathcal{S}}(X) = \int_{y \in X} d_S(y) \mathrm{d}y$.*

Note that $d_S(x, q)$ is easy to compute, but it is not normalised. Computing the normalising factor is in general intractable, but the Metropolis-Hastings algorithm does not require the density to be normalised.

Let us write $\mu^{\mathcal{S}}_{\mathrm{norm}}(X) = \frac{\mu^{\mathcal{S}}(X)}{\mu^{\mathcal{S}}(\mathscr{C}(E_S))}$ for the normalised distribution. By adequacy, we have for all $X \in \Sigma_{[\![A]\!]}$:

$$\mathrm{norm}[\![M]\!](X) = \mu^{\mathcal{S}}_{\mathrm{norm}}(\texttt{result}^{-1}(X)).$$

where $\texttt{result} : \max \mathscr{C}(E_S) \rightharpoonup [\![A]\!]$ maps a maximal configuration of E_S to its return value, if any. This says that sampling from $\mathrm{norm}[\![M]\!]$ amounts to sampling from $\mu^{\mathcal{S}}_{\mathrm{norm}}$ and only keeping the return value.

Accordingly, we focus on designing a Metropolis-Hastings algorithm for sampling values in $\mathscr{C}(E_S)$ following the (unnormalised) density d_S. We start by defining a proposal kernel for this algorithm.

To avoid overburdening the notation, we will no longer distinguish between a type and its denotation. Since G_S is finite, it can be represented by a type, and so can $\mathscr{C}(G_S)$. Moreover, $\mathscr{C}(E_S)$ is a subset of $\sum_{x \in \mathscr{C}(G_S)} \mathscr{Q}(x)$ which is also representable as the type of pairs $(x \in \mathscr{C}(G_S), q \in \mathscr{Q}(x))$. Operations on G_S and related objects are all computable and measurable so we can directly use them in the syntax. In particular, we will make use of the function $\texttt{ext} : \mathscr{C}(E_S) \to G_S + 1$ which for each configuration $(x, q) \in \mathscr{C}(E_S)$ returns $(1, \mathfrak{s})$ if there exists $x \overset{s}{\multimap}$ with $o_s(q|_{[s)})$ defined, and $(2, *)$ if (x, q) is maximal.

Informally, for $(x, q) \in \mathscr{C}(E_S)$, the algorithm is:

- Pick a sample event $s \in x$, randomly over the set of sample events of x.
- Construct $x_0 := x \setminus \{s' \in x \mid s' \geq s\} \cup \{s\} \in \mathscr{C}(G_S)$.
- Return a maximal extension (x', q') of $(x_0, q|_{x_0})$ by only resampling the sample events of x' which are not in x.

The last step follows the single-site MH principle: sample events in $x \cap x'$ have already been evaluated in x, and are not updated. However, events which are in $x' \setminus x$ belong to conditional branches not explored in x; they must be sampled.

We start by formalising the last step of the algorithm. We give a probabilistic program `complete` which has three parameters: the original configuration (x, q), the current modification (x_0, q_0) and returns a possible maximal extension:

$$
\begin{aligned}
&\texttt{complete}(x, q, x_0, q_0) = \texttt{case ext}(x_0, q_0)\ \texttt{of}\\
&\quad (2, ()) \Rightarrow (x_0, q_0)\\
&\quad (1, s) \Rightarrow\\
&\qquad \texttt{if } s \texttt{ is a return or a score event then}\\
&\qquad\quad \texttt{complete}(x, v, x_0 \cup \{s\}, q_0[s := o_s(q_0)])\\
&\qquad \texttt{else if } s \in x\\
&\qquad\quad \texttt{complete}(x, q, x_0 \cup \{s\}, q_0[s := q(s)])\\
&\qquad \texttt{else}\\
&\qquad\quad \texttt{complete}(x, q, x_0 \cup \{s\}, q_0[s := \texttt{sample } d\ (q_0)])
\end{aligned}
$$

The program starts by trying to extend (x_0, q_0) by calling `ext`. If (x_0, q_0) is already maximal, we directly return it. Otherwise, we get an event s. To extend the quantitative information, there are three cases:

- if s is not a sample event, ie. since S is closed it must be a return or a score event, we use the function o_s.
- if s is a sample event occurring in x, we use the value in q
- if s is a sample event not occurring in x, we sample a value for it.

This program is recursive, but because G_S is finite, there is a static bound on the number of recursive calls; thus this program can be unfolded to a program expressible in our language. We can now define the proposal kernel:

$$
\begin{aligned}
&P_S(x, q) =\\
&\quad \texttt{let } s \texttt{ = sample uniformly over sample events in } x \texttt{ in}\\
&\quad \texttt{let } r \texttt{ = sample } d_s \texttt{ (}q_|[s]\texttt{) in}\\
&\quad \texttt{let } x_0 \texttt{ = } x \setminus \{s' \geq s \mid s' \in x\} \texttt{ in}\\
&\quad \texttt{complete}(x, q, x_0, q[s := r])
\end{aligned}
$$

We now need to compute the density for P_S to be able to apply Metropolis-Hastings. Given $(x, q), (x', q') \in \mathscr{C}(E_S)$, we define:

$$
p_S((x, q), (x', q')) = \sum_{s \in \mathrm{sample}(x)} \left(\frac{q_s(v'|_{[s)})}{|\mathrm{sample}(x)|} \times \prod_{s' \in \texttt{sample}(x' \setminus x)} q_{s'}(v|_{[s')}) \right).
$$

Theorem 4. *The Markov chain P_S and density p satisfy the hypothesis of Theorem 3, as a result for any $(x, q) \in \mathscr{C}(E_S)$ the distribution $[\![\texttt{MH}(d_S, P_S, p_S)^n]\!]((x, q), \cdot)$ tends to μ^P_{norm} as n goes to infinity.*

One can thus sample from $\mathrm{norm}([\![M]\!])$ using the algorithm above, keeping only the return value of the obtained configuration.

Let us re-state the key advantage of our approach: having access to the data dependency information, `complete` requires fewer steps in general, because at each proposal step only a portion of the graph needs exploring.

6 Conclusion

Related work. There are numerous approaches to the semantics of programs with random choice. Among those concerned with statistical applications of probabilistic programming are Staton et al. [19, 18], Ehrhard et al. [7], and Dahlqvist et al. [6]. A game semantics model was announced in [15].

The work of Scibior et al. [17] was influential in suggesting a denotational approach for proving correctness of inference, in the framework of quasi-Borel spaces [9]. It is not clear however how one could reason about data dependencies in this framework, because of the absence of explicit causal information.

Hur et al. [11] gives a proof of correctness for Trace MCMC using new forms of operational semantics for probabilistic programs. This method is extended to higher-order programs with *soft constraints* in Borgström et al. [2]. However, these approaches do not consider incremental recomputation.

To the best of our knowledge, this is the first work adressing formal correctness of incremental recomputation in MCMC. However, methods exist which take advantage of data dependency information to improve the performance of each proposal step in "naive" Trace MCMC. We mention in particular the work on *slicing* by Hur et al. [10]; other approaches include [5], [24]. In the present work we claim no immediate improvement in performance over these techniques, but only a mathematical framework for reasoning about the structures involved.

It is worth remarking that our event structure representation is reminiscent of *graphical model* representation made explicit in some languages. Indeed, for a first-order language such as the one of this paper, Bayesian networks can directly be used as a semantics, see [20]. We claim that the alternative view offered by event structures will allow for an easier extension to higher-order programs, using ideas from game semantics.

Perspectives. This is the start of an investigation into intensional semantics for probabilistic programs. Note that the framework of event structures is very flexible and the semantics presented here is by no means the only possible one. Additionally, though the present work only treats the case of a first-order language, we believe that building on recent advances in probabilistic concurrent game semantics [3, 16] (from which the present work draws much inspiration), we can extend the techniques of this paper to arbitrary higher-order probabilistic programs with recursion.

Acknowledgements. We thank the anonymous referees for helpful comments and suggestions. We also thank Ohad Kammar for suggesting the idea of using causal structures for reasoning about data dependency in this context. This work has been partially sponsored by: EPSRC EP/K034413/1, EP/K011715/1, EP/L00058X/1, EP/N027833/1, EP/N028201/1, and an EPSRC PhD studentship.

References

1. Patrick Billingsley. *Probability and measure*. John Wiley & Sons, 2008.
2. Johannes Borgström, Ugo Dal Lago, Andrew D Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *ACM SIGPLAN Notices*, volume 51, pages 33–46. ACM, 2016.
3. Simon Castellan, Pierre Clairambault, Hugo Paquet, and Glynn Winskel. The concurrent game semantics of probabilistic PCF. In *Logic in Computer Science (LICS), 2018 33rd Annual ACM/IEEE Symposium on, ACM/IEEE*, 2018.
4. Simon Castellan and Hugo Paquet. Probabilistic programming inference via intensional semantics. Technical report available at http://iso.mor.phis.me/publis/esop19.pdf, 2019.
5. Yutian Chen, Vikash Mansinghka, and Zoubin Ghahramani. Sublinear approximate inference for probabilistic programs. *stat*, 1050:6, 2014.
6. Fredrik Dahlqvist, Vincent Danos, Ilias Garnier, and Alexandra Silva. Borel kernels and their approximation, categorically. *arXiv preprint arXiv:1803.02651*, 2018.
7. Thomas Ehrhard, Michele Pagani, and Christine Tasson. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. volume 2, pages 59:1–59:28, 2018.
8. Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.
9. Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *LICS'17, Reykjavik.*, pages 1–12, 2017.
10. Chung-Kil Hur, Aditya V Nori, Sriram K Rajamani, and Selva Samuel. Slicing probabilistic programs. In *ACM SIGPLAN Notices*, volume 49, pages 133–144. ACM, 2014.
11. Chung-Kil Hur, Aditya V Nori, Sriram K Rajamani, and Selva Samuel. A provably correct sampler for probabilistic programs. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 45. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.
12. Oleg Kiselyov. Probabilistic programming language and its incremental evaluation. In *Asian Symposium on Programming Languages and Systems*, pages 357–376. Springer, 2016.
13. Oleg Kiselyov. Problems of the lightweight implementation of probabilistic programming. In *Proceedings of Workshop on Probabilistic Programming Semantics*, 2016.
14. Dexter Kozen. Semantics of probabilistic programs. *J. Comput. Syst. Sci.*, 22(3):328–350, 1981.
15. Luke Ong and Matthijs Vákár. S-finite kernels and game semantics for probabilistic programming. In *POPL'18 Workshop on Probabilistic Programming Semantics (PPS)*, 2018.

16. Hugo Paquet and Glynn Winskel. Continuous probability distributions in concurrent games. *Electronic Notes in Theoretical Computer Science*, 341:321–344, 2018.
17. Adam Ścibior, Ohad Kammar, Matthijs Vákár, Sam Staton, Hongseok Yang, Yufei Cai, Klaus Ostermann, Sean K Moss, Chris Heunen, and Zoubin Ghahramani. Denotational validation of higher-order bayesian inference. *Proceedings of the ACM on Programming Languages*, 2(POPL):60, 2017.
18. Sam Staton. Commutative semantics for probabilistic programming. In *European Symposium on Programming*, pages 855–879. Springer, 2017.
19. Sam Staton, Hongseok Yang, Frank D. Wood, Chris Heunen, and Ohad Kammar. Semantics for probabilistic programming: higher-order functions, continuous distributions, and soft constraints. In *Proceedings of LICS '16, New York, NY, USA, July 5-8, 2016*, pages 525–534, 2016.
20. Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.
21. David Wingate, Andreas Stuhlmüller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778, 2011.
22. Glynn Winskel. Event structures. In *Advances in Petri Nets*, pages 325–392, 1986.
23. Glynn Winskel. Distributed probabilistic and quantum strategies. *Electr. Notes Theor. Comput. Sci.*, 298:403–425, 2013.
24. Yi Wu, Lei Li, Stuart Russell, and Rastislav Bodik. Swift: Compiled inference for probabilistic programming languages. *arXiv preprint arXiv:1606.09242*, 2016.
25. Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. Generating efficient MCMC kernels from probabilistic programs. In *Artificial Intelligence and Statistics*, pages 1068–1076, 2014.