

A Complete Axiomatisation of Equivalence for Discrete Probabilistic Programming

Robin Piedeleu^{1*}, Mateo Torres-Ruiz^{1*}, Alexandra Silva², and Fabio Zanasi¹

¹ University College London, UK

² Cornell University, USA

`r.piedeleu@ucl.ac.uk, m.torresruiz@cs.ucl.ac.uk`

Abstract. We introduce a sound and complete equational theory capturing equivalence of discrete probabilistic programs, that is, programs extended with primitives for Bernoulli distributions and conditioning, to model distributions over finite sets of events. To do so, we translate these programs into a graphical syntax of probabilistic circuits, formalised as string diagrams, the two-dimensional syntax of symmetric monoidal categories. We then prove a first completeness result for the equational theory of the conditioning-free fragment of our syntax. Finally, we extend this result to a complete equational theory for the entire language. Our first result gives a presentation of the category of Markov kernels, restricted to objects that are powers of the two-elements set.

1 Introduction

Probabilistic programming languages (PPLs) extend standard languages with two added capabilities: drawing random values from a given distribution, and conditioning on particular variable values. These constructs allow the programmer to define complex statistical models and obtain the probability of an event according to the distribution specified by the program, a task known as *inference*. This process can be understood in Bayesian terms: first-class distributions define a prior over the program’s declared variables; inference then involves computing the posterior distribution, conditional on some chosen values of those variables.

In this paper, we focus on *discrete* probabilistic programs, that is, programs whose associated probability distributions range over a finite set of outcomes. While many existing PPLs manipulate continuous random variables, this generally forces them to approximate the posterior distribution specified by a given program, typically by sampling from it [39, 28, 9]. Discrete probabilistic programs on the other hand lend themselves to *exact* inference, where instead of approximating or sampling from the posterior distribution, it is possible to obtain an exact representation of it, which is interesting for verification purposes.

Several existing languages support this style of inference for discrete probabilistic programs [22, 7, 18]. In this work, we start from a prototypical discrete

* Both authors contributed equally to this research.

PPL that incorporates a primitive for Bernoulli distributions (`flip`) and conditioning ($x := y$). Beyond its probabilistic features, its syntax is that of a simple first-order, non-recursive functional programming language with `let`-expressions for variable declaration and branching with an `if-then-else` constructor. In this respect, it is similar to *Dice* [22] or the *CD-calculus* [36]. While it may appear limited, it is in fact sufficiently expressive to specify arbitrary distributions over tuples of Boolean variables and thus, to encode arbitrary discrete distributions.

The example on the right expresses a simple model, the *von Neumann trick*, as a probabilistic program. Imagine that we are betting on the outcome of a coin flip and cannot trust that the coin is fair. We can use the following protocol to simulate a fair coin: toss the coin twice (`flip p`); if the outcomes are the same (`(first := (not second))`), discard them and start over; otherwise, keep the first outcome and discard the second. This is exactly the program above, where p is an arbitrary parameter strictly between 0 and 1, the unknown probability that the coin lands on heads.

As this example illustrates, probabilistic programs are a compact way to write and communicate probabilistic models: they are modular, compositional, and understood by a large community of practitioners, who can then leverage functional abstraction to build and share increasingly complicated models.

A probabilistic program can often be replaced by an *equivalent* program whose corresponding inference task is simpler to solve. We say that two probabilistic programs are equivalent when they define the same distribution. Von Neumann’s trick provides a first example of a non-trivial program equivalence: the program above is interchangeable with the much simpler program `flip 0.5`.

This notion of equivalence also applies to *partial* programs containing free variables, like the one on the left which is equivalent to the identity function on a Boolean variable.

Program equivalence is essential to ensure that the behaviour of programs remain consistent across different implementations and refinements [24,21]. In probabilistic programming, equivalence checking is crucial for building reliable, efficient, and accurate models, as well as inference algorithms. However, probabilistic features greatly complicate the task of verifying program equivalence.

In this work, we tackle the equivalence problem by presenting a complete equational theory for discrete probabilistic programs. To do so, we translate the simple PPL syntax above into an equally expressive diagrammatic calculus of *probabilistic circuits*. We start from plain Boolean circuits with standard logical gates, and extend them with two additional gates that incorporate probabilistic features: one representing a Bernoulli distribution, another conditioning.

Note that the behaviour of the `if-then-else` construct used in this paper differs from the standard interpretation found in most programming languages. Although our interpretation is natural from the perspective of (probabilistic) circuits, it allows failure in non-executed branches to propagate, breaking equivalences expected to hold in branching constructs (i.e., `(if true then A else B) = A` and `(if false then A else B) = B` will not hold for all A s and B s).

While this behaviour aligns with logical circuits when no failures occur, defining its operational semantics explicitly would require an unusual approach. As we will see, though atypical, this behaviour is a necessary consequence of our choice of semantics, which equates programs who encode the same posterior.

While circuits are often used as informal visual aids, the circuits in this paper are a formal syntax of *string diagrams*, the graphical language of (symmetric) monoidal categories [34]. In recent years, string diagrams have been applied in different fields, including quantum computing [8], electrical engineering [4] and probability theory [13]. They offer several advantages: like programs, we can reason about them algebraically, as syntactic objects with sequential and parallel compositions, and endow them with a compositional semantics; like circuits, the visual representation absorbs structural equivalences of programs and highlights resource-exchange between the different parts of the systems they represent.

The main technical result of our paper is proving that the provided equational theory fully characterises semantic equality of circuits: we show that whenever two circuits denote the same distribution, we can derive this fact using purely equational reasoning on the circuits themselves. To prove this result, we proceed in two steps: we first give a complete equational theory for the conditioning-free fragment of our syntax, which we later extend to the whole language.

Note that these results can be seen as part of a broader program aiming to formulate probability theory in *Markov categories* [13]. In particular, our first main result provides a presentation in terms of generators and equations of the full monoidal subcategory of the category of Markov kernels/stochastic maps [19] on objects that are powers of the two-element set (\mathbb{B}^n for some natural number n). Our contribution is thus one further step on the way to developing an axiomatic account of probability theory in Markov categories.

Outline and main contributions. In Section 2, we give the formal syntax of probabilistic circuits, both as a standard term syntax and as a two-dimensional syntax of string diagrams; we explain its relation with a conventional language for discrete probabilistic programming such as the one used in the previous examples; and we equip it with a compositional semantics which captures the distribution the circuits are intended to represent. In Section 3 we introduce and prove the completeness of an equational theory for the conditioning-free fragment of our circuits. In Section 4 we extend this theory to the full syntax, and prove it complete, thus obtaining a full axiomatisation of discrete probabilistic program equivalence. Section 5 discusses related work and directions for future work.

2 Syntax and Semantics

In this section, we introduce the diagrammatic notation used as syntax for probabilistic programs and define semantics in terms of (sub)distributions.

2.1 Syntax

We define our syntax in two steps: we start with a simple grammar whose constants are circuit primitives with rules for their sequential and parallel composi-

$$\begin{array}{c}
\bullet : 1 \rightarrow 0 \quad \text{---} \bullet : 1 \rightarrow 2 \quad \text{---} \text{---} : 2 \rightarrow 1 \quad \text{---} \text{---} : 1 \rightarrow 1 \\
\text{---} \text{---} : 0 \rightarrow 1 \quad \text{---} \text{---} : 2 \rightarrow 1 \quad \text{---} : 0 \rightarrow 0 \quad \text{---} : 1 \rightarrow 1 \quad \text{---} : 2 \rightarrow 2 \\
\frac{c : \ell \rightarrow m \quad d : m \rightarrow n}{c ; d : \ell \rightarrow n} \quad \frac{c : m_1 \rightarrow n_1 \quad d : m_2 \rightarrow n_2}{c \otimes d : m_1 + m_2 \rightarrow n_1 + n_2}
\end{array}$$

Fig. 1: Typing rules for **ProbCirc** terms. A type is a pair $(m, n) \in \mathbb{N} \times \mathbb{N}$ which we write as $m \rightarrow n$; we write the judgment that c has type $m \rightarrow n$ as $c : m \rightarrow n$.

tion, and then move to a two-dimensional representation of the same circuits as string diagrams. We compare our graphical language with a more conventional programming language syntax, by providing a syntactic translation.

Term language. We consider terms given by the following simple grammar:

$$\text{ProbCirc} \ni c ::= \bullet \mid \text{---} \bullet \mid \text{---} \text{---} \mid \text{---} \text{---} \mid \text{---} \mid \text{---} \mid \text{---} \mid c ; c \mid c \otimes c \quad (1)$$

$$\mid \text{---} \text{---} \quad \forall p \in [0, 1] \quad (2)$$

$$\mid \text{---} \text{---} \quad (3)$$

Though the constants of our language are depicted as diagrams, we treat them as symbols for the moment. Soon, we will consider all **ProbCirc**-terms as *string diagrams* and view the two binary operations of *sequential* ($c ; d$) and *parallel* ($c \otimes d$) composition as those of a monoidal category [34].

We will refer to the fragment of the syntax which contains the rules and constants of line (1) as **BoolCirc**, and the syntax of lines (1) and (2) as **CausCirc**.

Our language does not have variables, making it unnecessary to define alpha-equivalence. However, we need simple typing rules, see Fig. 1. Going forward, we only consider well-typed terms. The type of a term c corresponds to the number of open wires on each side of the diagram of c . A simple induction confirms uniqueness of types: if $c : m \rightarrow n$ and $c : m' \rightarrow n'$, then $m = m'$ and $n = n'$. A circuit with no input (*resp.* output) wires has type $0 \rightarrow n$ (*resp.* $m \rightarrow 0$).

Like in traditional Boolean circuits, our terms are intended to represent networks of logical gates connected by wires carrying Boolean values. We will assign a formal semantics in Section 2.2, but give some intuition for their interpretation now. The purely Boolean fragment of the syntax (**BoolCirc**) includes standard gates for logical conjunction, $\text{---} \text{---}$, which outputs a 1 only when its two inputs are set to 1; logical negation, $\text{---} \text{---}$, which negates its input; a broadcasting gate, $\text{---} \bullet$, that copies its input to two output wires; and a terminating wire, \bullet , that discards its input. Beyond Boolean circuits, there are two additions. Firstly, **CausCirc** introduces $\text{---} \text{---}$, which is the only probabilistic component of our syntax, emitting 1 with probability p and 0 with probability $1 - p$. Note that $\text{---} \text{---}$ has *no input* and that the parameter p is simply a label which is not accessible to the rest of the circuit. Thus we have a generator for each $p \in [0, 1]$. Finally, **ProbCirc** adds a component for explicit conditioning, $\text{---} \text{---}$, which constrains its two input wires to carry equal values and outputs this value. A similar

primitive has appeared in previous work [36]. As we will see, this is the only component whose behaviour is not causal, in the sense that it constrains the values of its input. This is why we call *causal* circuits the conditioning-free fragment of our syntax (CausCirc). Intuitively, these circuits specify a distribution over all possible values of their output wires for any value of their input wires.

Example 2.1. From our generators, we can define other standard Boolean gates as syntactic sugar. For example, OR-gates are defined as

$$\text{OR} := (\neg \rightarrow \otimes \neg \rightarrow); \text{OR}; \neg \rightarrow$$

and multiplexers/*if-then-else* gates are defined as

$$\begin{aligned} \text{MUX} := & (\neg \bullet \otimes \text{---} \otimes \text{---}); (\text{---} \otimes \neg \rightarrow \otimes \text{---} \otimes \text{---}); \\ & (\text{---} \otimes \times \otimes \text{---}); (\text{OR} \otimes \text{OR}); \text{OR} \end{aligned}$$

From terms to string diagrams. As is clear from the previous examples, ProbCirc-terms are not easy to parse for the human eye. Moreover, our main goal is to obtain an equational axiomatisation of semantic equivalence of ProbCirc-terms; when we define their semantics formally, we will see that their interpretation satisfies all the axioms of *symmetric monoidal categories* (SMCs). This licences us to move from terms to *string diagrams*, a common graphical representation of morphisms in SMCs [34,25]. We recall some of the basics below, though we refer the reader to [32] for a more extensive introduction on the topic.

Formally, a string diagram on ProbCirc is defined as an equivalence class of ProbCirc-terms, where the quotient is taken with respect to the reflexive, symmetric and transitive closure of the following axioms:

$$\begin{aligned} c_1 \otimes (c_2 \otimes c_3) &= (c_1 \otimes c_2) \otimes c_3 & (c_1 \otimes c_2); (d_1 \otimes d_2) &= (c_1; d_1) \otimes (c_2; d_2) \\ (c; d); e &= c; (d; e) & c; \text{---} &= c = \text{---}; c & \square \otimes c &= c \otimes \square \\ (\text{---} \otimes c); \times &= \times; (c \otimes \text{---}) & \times; \times &= \text{---} \otimes \text{---} \end{aligned}$$

where c, d, e and c_i, d_i range over ProbCirc-terms of the appropriate type. These axioms state that sequential and parallel composition are associative and unital, that they satisfy a form of interchange law, and that the wire crossings behave as our topological intuition expects—see their diagrammatic translation (4) below.

Let us establish some conventions. We will refer to string diagrams representing BoolCirc-terms as *Boolean circuits*, those representing CausCirc-terms as *causal circuits*, and arbitrary string diagrams for ProbCirc-terms simply as *circuits*. We will depict n parallel wires, for some natural number n , by a labelled-wire ---^n (and omit the label when $n = 1$). Moreover, we will depict a generic circuit $c : m \rightarrow n$ by a labelled grey box, and a Boolean circuit $b : m \rightarrow n$ by a labelled white box:

$$\text{---}^m \text{---}^n c \text{---}^n = m \left\{ \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array} \right\} c \left\{ \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array} \right\}^n \quad \text{---}^m \text{---}^n b \text{---}^n = m \left\{ \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array} \right\} b \left\{ \begin{array}{c} \vdots \\ \vdots \\ \vdots \end{array} \right\}^n$$

Then, we represent $c; d$ as sequential composition, depicted horizontally from left to right (note that the types of the intermediate wires have to match), and $c \otimes d$ as parallel composition, depicted vertically from top to bottom:

$$\frac{\ell}{c; d} \frac{n}{n} = \frac{\ell}{c} \frac{m}{m} \frac{n}{d} \frac{n} \quad \frac{m_1 + m_2}{c_1 \otimes c_2} \frac{n_1 + n_2}{n_1 + n_2} = \frac{\frac{m_1}{c_1} \frac{n_1}{n_1}}{\frac{m_2}{c_2} \frac{n_2}{n_2}}$$

We omit the labels on the wires for readability when they can be inferred unambiguously from the context.

Intuitively, our string diagrams can be formed much like conventional circuits, by wiring the generators of (1)-(3) in sequence or in parallel, crossing wires (with \times) and making wires as long as we want (with ---). As string diagrams, the axioms of SMCs now become near-tautologies:

$$\begin{aligned} & \boxed{c} \text{---} d \text{---} e = c \text{---} \boxed{d \text{---} e} \\ & \boxed{\begin{array}{c} c_1 \\ c_2 \\ c_3 \end{array}} = \begin{array}{c} c_1 \\ c_2 \\ c_3 \end{array} \\ & \boxed{\begin{array}{cc} c_1 & d_1 \\ c_2 & d_2 \end{array}} = \begin{array}{cc} c_1 & d_1 \\ c_2 & d_2 \end{array} \\ & \boxed{c} \text{---} \boxed{} = \boxed{c} \text{---} \phantom{\boxed{}} = \boxed{} \text{---} \boxed{c} \\ & \boxed{} \text{---} \boxed{c} = \phantom{\boxed{}} \text{---} \boxed{c} = \boxed{} \text{---} \phantom{\boxed{}} \\ & \boxed{c} \times = \times \boxed{c} \quad \times \times = \text{---} \text{---} \end{aligned} \quad (4)$$

If we think of the dotted frames as two-dimensional brackets, these identities tell us that the specific bracketing of a term does not matter. This is precisely the advantage of working with string diagrams, rather than terms: they free us from some of the bureaucracy of terms, allowing us to focus on the more structural aspects of our syntax, *i.e.* on how the different components making up a term are wired together. This means in particular that wire crossings obey laws that are topologically obvious, as shown in (4).

Example 2.2. For the **OR**-gate and multiplexers defined in Example 2.1, we have

$$\text{OR-gate} = \text{circuit diagram} \quad \text{multiplexer} = \text{circuit diagram}$$

Even though we have not defined a formal semantics yet, their interpretation is clear from their depiction as circuits: the first will behave like a standard logical **OR**-gate, which outputs the disjunction of its two inputs, and the second like a multiplexer or **if-then-else**-gate, which outputs either its second input if the first is set to 1 or the third input if the first is set to 0. We will call the first input the *guard*, and the next two, the *then*- and *else-branch*, respectively.

We also define generalised gates for n wires, as syntactic sugar, by induction, for **AND**-gates and copying nodes:

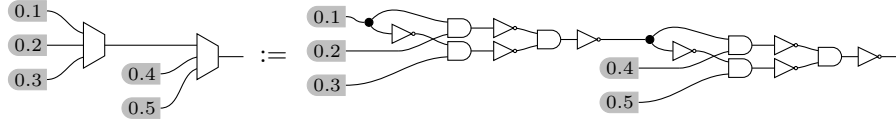
$$\frac{1+n}{1+n} \text{---} \frac{1+n}{1+n} := \frac{n}{n} \text{---} \frac{n}{n} \quad \frac{1+n}{1+n} \text{---} \frac{1+n}{1+n} := \frac{n}{n} \text{---} \frac{n}{n}$$

$$\begin{array}{c}
\frac{}{\Gamma, x : \tau, \Delta \vdash x : \tau} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{fst} \, e : \tau_1} \quad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathbf{snd} \, e : \tau_2} \\
\\
\frac{\Gamma \vdash g : \mathbb{B} \quad \Gamma \vdash e_0 : \tau \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \mathbf{if} \, g \, \mathbf{then} \, e_1 \, \mathbf{else} \, e_0 : \tau} \quad \frac{}{\Gamma \vdash \mathbf{flip} \, p : \mathbb{B}} \quad \frac{}{\Gamma \vdash \mathbf{true} : \mathbb{B}} \\
\\
\frac{\Gamma \vdash e_0 : \tau_0 \quad \Gamma, x : \tau_0 \vdash e_1 : \tau}{\Gamma \vdash \mathbf{let} \, x = e_0 \, \mathbf{in} \, e_1 : \tau} \quad \frac{}{\Gamma, x : \tau, y : \tau, \Delta \vdash (x =: y) : \tau}
\end{array}$$

Fig. 2: Typing rules for a simple language for discrete probabilistic programming. Metavariables x, y range over variable names, and p over reals in $[0, 1]$.

with base cases the corresponding generators; we will use generalised gates for \bullet , \neg , \wedge , \vee , etc. defined in a similar way.

Example 2.3. Consider a simple setting where we flip a coin x with a 0.1 chance of landing heads. If x is heads, we flip a new coin y_1 with a 0.2 chance of heads; otherwise, we flip a third coin y_2 with a 0.3 chance of heads. Finally, based on the outcome of y_i (for either $i = 1$ or $i = 2$), we choose between two more coins z_1 or z_2 : if y_i is heads, z_1 has a 0.4 chance of heads; otherwise, we flip z_2 , which is a fair coin. Now, imagine that we are interested in the probability of the last coin flip landing on heads. This is what the following causal circuit models:



Comparing with a conventional PL syntax. We now explain the relationship of our circuits with the more conventional syntax of the probabilistic programming language used in the introduction. Fig. 2 gives the syntax and typing rules of this simple (first-order) functional PPL: the judgment $\Gamma \vdash e : \tau$ denotes a well-typed partial program e in context Γ , which is a list of pairs $x : \tau$ of free variables with their types, and types are tuples of Booleans. This syntax is closely related to that of *Dice* [22] and of the *CD-calculus* [36, Section 6.3], domain-specific languages for discrete probabilistic programming.

At the level of the purely Boolean fragments, these programming languages and our circuits can express the same maps, though they use **if-then-else** as primitive, rather than conjunction and negation. Indeed both are well known to constitute universal sets of gates for Boolean functions. Instead of wires, a standard syntax uses *variables* to encode circuits inputs and **let**-expressions to encode circuit composition. The latter are not needed to represent plain Boolean functions—the algebraic syntax of Boolean algebra suffices for this purpose [5].

The first probabilistic primitive is **flip** p , which represents a coin flip with probability $p \in [0, 1]$ of landing on **true**, like our own \bullet_p . Note that simply adding **flip** p to the syntax of Boolean algebra would give a syntax that is

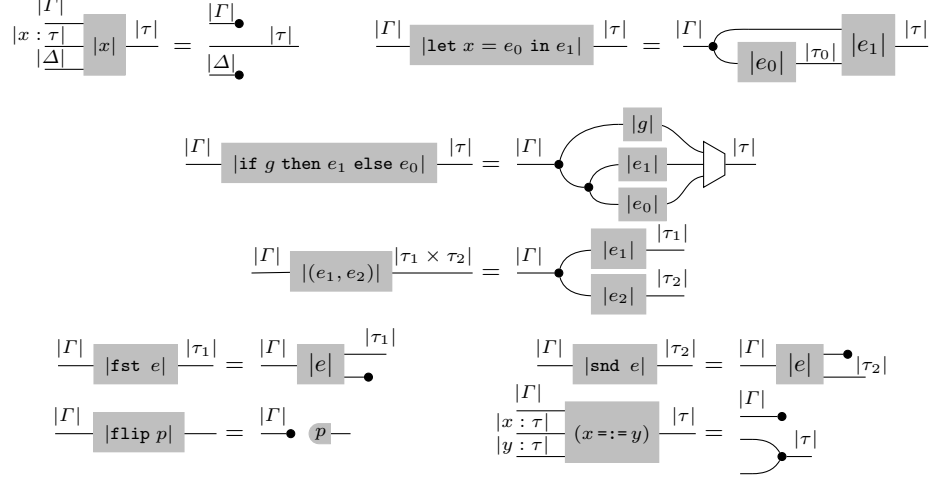


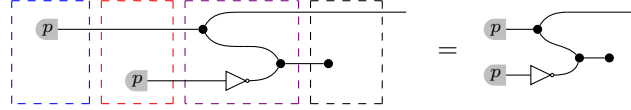
Fig. 3: Diagrammatic translation of programs to circuits. The mapping $|\cdot|$ is defined inductively: on types by $|\tau_1 \times \tau_2| = |\tau_1| + |\tau_2|$ with base case $|\mathbb{B}| = 1$, on contexts by $|Γ, x : \tau| = |Γ| + |\tau|$ with base case $|\emptyset| = 0$, and on (open) terms by the rules above.

insufficient to reuse (or ignore) the outcome of a `flip` p . Indeed there would be no way of binding its outcome to some variable for multiple use, contrary to our diagrammatic syntax, which can broadcast the output of p — to multiple sub-circuits using \multimap (or discard it with \rightarrow). This justifies the need for `let`-expressions: to mimic the non-linear use of probabilistic outcomes. Concretely, `let` $x = e_0$ `in` e_1 binds some term e_0 to the variable x for its use in e_1 .

The second probabilistic primitive is $(x := y)$, interpreted as evidence that the variables x and y are equal; it has the effect of conditioning the posterior distribution the program represents on the events for which these two variables agree. This construct appears in the work of Stein and Staton on exact conditioning [36]. Our circuit \multimap plays a similar role constraining its two inputs to have equal value. Some PPL, like Dice, use `observe` x instead to express conditioning on observations for which x is true [22]. This is equivalent to our choice, since we can simply encode it by setting one side of the $(x := y)$ to `true`. In the other direction, we can encode $(x := y)$ as `observe(not x xor y)`.

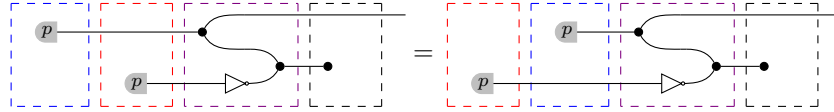
Based on this discussion, we can devise a more formal translation of our symbolic PPL into circuits. A (partial) program $Γ \vdash e : \tau$ will now be translated into a circuit $|e| : |Γ| \rightarrow |\tau|$ where $|Γ|$ is simply the length of the context $Γ$ and, similarly, $|\tau|$ is the length of the corresponding tuple of Booleans. In other words, a partial program with m free variables and whose main expression has type \mathbb{B}^n represents a circuit with m input wires and n output wires. A complete inductive translation on the typing rules is given in Fig. 3. Note that it uses the multiplexer and n -ary gates defined in the main text as syntactic sugar.

Example 2.4. The von Neumann’s trick program from Section 1 can be translated into the following circuit, where each dashed box corresponds to a line:



Intuitively, it is clear that this circuit flips two coins with the same parameter p as in the previous circuit, negates the outcome of one of them, conditions on the two tosses being equal, and asks for the probability that the first flip gave true.

The axioms of SMCs capture non-trivial program transformations: *e.g.*, the first two **let**-expressions can be swapped without changing the diagram³:



This example highlights one advantage of string diagrams—the topological transformations of the two-dimensional representation absorb some, but not all, of the burden of reasoning about program equivalence.

2.2 Semantics

Before establishing the formal semantics of our circuits, we recall some necessary background. In this paper, ‘subdistribution’ will always refer to a *finitely-supported discrete probability subdistribution*, *i.e.*, a map $\varphi : X \rightarrow [0, 1]$ where X is finite and such that $\sum_{x \in X} \varphi(x) \leq 1$. A *distribution* is then a subdistribution φ for which $\sum_{x \in X} \varphi(x) = 1$. We will write subdistributions as formal sums $\sum_x \varphi(x) |x\rangle$, omitting the elements of X for which $\varphi(x) = 0$, and call $\mathcal{D}_{\leq 1} X$ (resp. $\mathcal{D} X$) the set of all subdistributions (resp. distributions) over X .

The symmetric monoidal category of substochastic maps. Our circuits do not map inputs to outputs in a deterministic way. Instead, for each input, they specify a subdistribution over their outputs. Thus, we will interpret them as *substochastic maps* $f : X \rightarrow \mathcal{D}_{\leq 1} Y$, which we write as $f : X \multimap Y$. Currying the Y component, we can think of f as a map $f(-|-) : Y \times X \rightarrow [0, 1]$ or, equivalently, as a matrix whose columns sum to at most 1. We will switch freely between the two perspectives. As with subdistributions, a substochastic map is *stochastic* when $\sum_y f(y|x) = 1$ for all $x \in X$ (or when its columns sum to 1). Note that any map $f : X \rightarrow Y$ can be promoted to a substochastic map $X \multimap Y$ given by $x \mapsto |f(x)\rangle$ for all $x \in X$.

We can also interpret $f : X \multimap Y$ as specifying a *conditional* subdistribution: $f(-|x)$ gives a subdistribution over Y for each $x \in X$. The composition $f;g$ of

³ A fact also known as the commutativity of probabilistic effects in PL parlance.

two substochastic maps $f : X \rightarrowtail Y, g : Y \rightarrowtail Z$ is given by summing over the intermediate variable as follows:

$$(f ; g)(z|x) = \sum_{y \in Y} g(z|y)f(y|x)$$

Note that, if we think of f and g as substochastic matrices, this is the formula for their product in the usual sense. The identity $\text{id}_X : X \rightarrowtail X$ is simply the Dirac delta $\delta_x : x \mapsto |x\rangle$ or, equivalently, the identity matrix. Substochastic maps with these operations define a category, which we call **FinSubStoch** [26]. Moreover, since stochastic maps are stable under composition, they form a subcategory of **FinSubStoch**, which we call **FinStoch** [13].

For two subdistributions $\varphi \in \mathcal{D}_{\leq 1} X$ and $\rho \in \mathcal{D}_{\leq 1} Y$, we can form the product subdistribution $\varphi \otimes \rho \in \mathcal{D}_{\leq 1}(X \times Y)$, given by $(\varphi \otimes \rho)(x, y) = \varphi(x) \cdot \rho(y)$. We will write $|xy\rangle = |x\rangle \otimes |y\rangle$ so that $(\sum_x \varphi(x) |x\rangle) \otimes (\sum_y \rho(y) |y\rangle) = \sum_{x,y} \varphi(x) \rho(y) |xy\rangle$. The same operation can be extended to conditional distributions, that is, to substochastic maps $f_1 : X_1 \rightarrowtail Y_1$ and $f_2 : X_2 \rightarrowtail Y_2$, giving $(f_1 \otimes f_2)(x_1, x_2) = f_1(x_1) \otimes f_2(x_2)$. This makes **FinSubStoch** into a *symmetric monoidal* category, with the Cartesian product of sets as monoidal product, the singleton set $1 = \{\bullet\}$ as unit, and the symmetry $\sigma_Y^X : X \times Y \rightarrowtail Y \times X$ given by $\sigma_Y^X(x, y) = |yx\rangle$. Stochastic maps out of the unit 1 , e.g. $1 \rightarrowtail X$, correspond precisely to distributions over X . Since the product of two distributions is still a distribution, **FinStoch** is a symmetric monoidal subcategory of **FinSubStoch**.

Remark 2.5. There is another way of making **FinStoch** into a SMC, with the monoidal product given by disjoint sum of sets on objects and direct sum of matrices on morphisms. This SMC has already been given a complete axiomatisation in the work of Fritz [12], which is itself a categorical reformulation of the much earlier work of Stone [38] on barycentric algebras. No axiomatisation exists for the monoidal product we consider here.

Copying and deleting. Note that $\mathcal{D}(1) = 1$ and therefore there is only one stochastic map $\epsilon_X : X \rightarrowtail 1$ for any X , given by $\epsilon_X(x) = |\bullet\rangle$. Moreover, there is a canonical diagonal substochastic map inherited from the Cartesian product of sets: $\Delta_X : X \rightarrowtail X \times X$ given by $\Delta_X(x) = |xx\rangle := |x\rangle \otimes |x\rangle$. It is important to note that $f ; \Delta_Y \neq \Delta_X ; (f \times f)$ in general—we say that arbitrary substochastic maps cannot be copied. Intuitively, this makes sense: if f represents some probabilistic process (such as flipping a coin), running f once and reusing its outcome in two different places is different from running f twice (flipping two coins).

Similarly, not every substochastic map $f : X \rightarrowtail Y$ satisfies $f ; \epsilon_Y = \epsilon_X$. Those that do are *stochastic*, since $(f ; \epsilon_Y)(x) = \sum_{y \in Y} f(y|x) = 1 = \epsilon_X(x)$. Such maps are often called *discardable* or *causal* [13, 6], which is why we refer to conditioning-free circuits as ‘causal circuits’. We will later see that they are precisely those circuits whose semantics give a bona-fide stochastic map.

Furthermore, using ϵ , we can recover the usual notion of *marginal* from probability theory. Indeed, post-composing a stochastic map $f : 1 \rightarrowtail X \times Y$ with ϵ_Y

involves summing over all $y \in Y$ as follows: $f;(\text{id}_X \times \epsilon_Y)(x) = \sum_{y \in Y} f(x, y)$. If we think of f as a joint distribution over variables taking values in X and Y , the stochastic map $f;(\text{id}_X \times \epsilon_Y)$ thus corresponds precisely to marginalising over Y to obtain a distribution over X only.

Conditionals and disintegration. Similarly, we can translate the notion of *conditioning* in the language of **FinStoch**. A stochastic map $f : X \rightarrowtail Y$ can be thought of as a conditional distribution $f(-|x)$ for each $x \in X$ and facts about conditionals have their counterpart in **FinStoch**. In particular, given a stochastic map $f : 1 \rightarrowtail X \times Y$, there exists $f_{|X} : X \rightarrowtail Y$ such that $f = f;(\text{id}_X \times \epsilon_Y); \Delta_X;(\text{id}_X \times f_{|X})$. This is a direct translation of the *disintegration* of a joint distribution $\mathbb{P}(x, y) = \mathbb{P}(x)\mathbb{P}(y|x)$ into the product of a marginal and a conditional distribution. In the language of **FinStoch**, $f;(\text{id}_X \times \epsilon_Y)$ corresponds to the marginal $\mathbb{P}(x)$, and $f_{|X}$ to the conditional $\mathbb{P}(y|x)$.

It is known that such disintegrations are *not* unique in general. Indeed, when the marginal distribution $\mathbb{P}(x)$ is not fully-supported ($\mathbb{P}(x) = 0$ for some $x \in X$), then $\mathbb{P}(y|x)$ can be arbitrary. It is clear that any two disintegrations of the same distribution (for the same order of the variables) can have conditionals that differ only on a set of measure zero for the corresponding marginal [6, Section 6].

Proposition 2.6 (Almost-sure uniqueness of disintegrations). *Given two disintegrations $(g, f_{|X})$ and $(g', f'_{|X})$ of the stochastic map $f : A \rightarrowtail X \times Y$, then $g = g' = f;(\text{id}_X \times \epsilon_Y)$ and $f_{|X}(a, x) = f'_{|X}(a, x)$ for all $a \in A$ and $x \in X$ such that $g(x|a) = g'(x|a) \neq 0$.*

Interpreting causal circuits. We are now ready to define the semantics of causal circuits. To do so, we will use the standard Boolean operations of conjunction (written ‘ \wedge ’) and negation (written ‘ \neg ’).

Definition 2.7 (Semantics of causal circuits). *Let $\llbracket \cdot \rrbracket$ be the mapping defined inductively on **CausCirc** by*

$$\begin{aligned} \llbracket \text{---} \bullet \rrbracket (x) &= |xx\rangle & \llbracket \text{---} \bullet \rrbracket (x) &= |\bullet\rangle \\ \llbracket \text{---} \triangleright \rrbracket (x) &= |\neg x\rangle & \llbracket \text{---} p \rrbracket (\bullet) &= p|1\rangle + (1-p)|0\rangle \\ \llbracket \text{---} \rrbracket (x) &= |x\rangle & \llbracket \text{---} \times \rrbracket (x, y) &= |xy\rangle & \llbracket \text{---} \sqcup \rrbracket (x_1, x_2) &= |x_1 \wedge x_2\rangle \\ \llbracket \begin{smallmatrix} \ell & c & m \\ \hline & & \end{smallmatrix} \begin{smallmatrix} m & d & n \\ \hline & & \end{smallmatrix} \rrbracket (z|x) &= \sum_{y \in Y} \llbracket \begin{smallmatrix} m & d & n \\ \hline & & \end{smallmatrix} \rrbracket (z|y) \cdot \llbracket \begin{smallmatrix} \ell & c & m \\ \hline & & \end{smallmatrix} \rrbracket (y|x) \\ \llbracket \begin{smallmatrix} m_1 & c_1 & n_1 \\ \hline m_2 & c_2 & n_2 \end{smallmatrix} \rrbracket (y_1, y_2|x_1, x_2) &= \llbracket \begin{smallmatrix} m_1 & c_1 & n_1 \\ \hline & & \end{smallmatrix} \rrbracket (y_1|x_1) \cdot \llbracket \begin{smallmatrix} m_2 & c_2 & n_2 \\ \hline & & \end{smallmatrix} \rrbracket (y_2|x_2) \end{aligned}$$

Note that the first line of Definition 2.7 is the usual semantics of these circuit gates, given in terms of Boolean operations, lifted to **FinStoch**. The second line defines the semantics of $\text{---} p \text{---}$ to be a Bernoulli distribution with parameter p . The next line forces $\llbracket \cdot \rrbracket$ to map plain wires and wire crossings to identities and symmetries in **FinStoch**. As we see, the image of every primitive lands in **FinStoch**,

i.e., is a (normalised) distribution. Finally, the last two lines guarantee that our interpretation is *compositional*: the semantics of two circuits composed in sequence is their composition in **FinStoch** and the semantics of two circuits in parallel is their monoidal product. Thus, a circuit $c : m \rightarrow n$, with m input wires and n output wires, is interpreted as a stochastic map $\llbracket c \rrbracket : \mathbb{B}^m \rightarrow \mathbb{B}^n$, *i.e.*, as a map $\mathbb{B}^m \rightarrow \mathcal{D}(\mathbb{B}^n)$. As a result, this interpretation does define a symmetric monoidal functor from syntax to semantics.

Proposition 2.8. *The map $\llbracket \cdot \rrbracket$ is a symmetric monoidal functor from **CausCirc** to **FinStoch**.*

Semantics of conditioning. Famously, explicit conditioning complicates the semantics of probabilistic programs. Here, we want the generator \triangleright of our syntax to constrain both its inputs to have the same value. Hence, its interpretation should return the Dirac distribution $|x\rangle$ whenever its two inputs are both equal to x , but what should its output subdistribution be when its two inputs disagree? Following standard practice [7,22,23], we will assign probability 0 to such failed conditions. This suggests extending $\llbracket \cdot \rrbracket$ to all of **ProbCirc** with

$$\llbracket \triangleright \rrbracket (y|x_1, x_2) = \begin{cases} 1 & \text{if } y = x_1 = x_2, \\ 0 & \text{otherwise.} \end{cases} \quad (5)$$

Note that this does define an unnormalised *subdistribution*. As a result, conditioning forces us to leave **FinStoch**, the subcategory of (normalised) distributions.

However, it is always possible to rescale a non-zero subdistribution to obtain a distribution, as has been proposed for the semantics of many PPLs [7,22,11,29]. And indeed, for the purpose of inference, any two closed probabilistic programs that represent the same distribution, should be considered equal. Equivalently, we would like to identify any two circuits $0 \rightarrow n$ whose semantics give proportional subdistributions. While there are other interpretations of conditioning in probabilistic programming [29], this is the perspective we adopt here. Let us make this idea more precise, following the work of Stein and Staton [36].

For any two $\phi, \rho \in \mathcal{D}_{\leq 1}X$, we write $\phi \propto \rho$ if there exists some real number $\lambda > 0$ such that $\phi(x) = \lambda \cdot \rho(x)$. This defines an equivalence relation on $\mathcal{D}_{\leq 1}X$. Moreover, since any non-zero subdistribution can always be normalised, the equivalence classes of \propto are in one-to-one correspondence with distributions over X , plus the uniformly zero subdistribution \perp_X , *i.e.*, $\mathcal{D}_{\leq 1}X / \propto \cong \mathcal{D}X + \{\perp_X\}$. In this sense, conditioning just adds the possibility of failure, denoted by \perp_X , to the standard distributional semantics given by $\llbracket \cdot \rrbracket$.

Thus, our intended semantics should identify circuits of type $0 \rightarrow n$ whose subdistributional semantics differ only by a constant (non-zero) multiplicative factor. To interpret programs with free variables/circuits with input wires, we need to generalise the idea to all substochastic maps. Moreover, we need to do so in a *compositional* way: this means that the semantics for circuits $0 \rightarrow n$ should extend to a semantics for arbitrary circuits $m \rightarrow n$ in a way which is compatible with sequential and parallel composition.

It turns out that there is only one consistent way to do so. First, note that if we identify subdistributions up to a scalar factor, we have to identify all subdistributions over the set $1 = \{\bullet\}$, and therefore all circuits of type $0 \rightarrow 0$. But subdistributions over 1 are precisely the substochastic maps $1 \rightarrow 1$, and these are in one-to-one correspondence with the non-negative reals. This tells us exactly how to extend our equivalence relation \propto over subdistributions to substochastic maps: consider $c : 1 \rightarrow 1$ and $f : X \rightarrow Y$; then c is equivalent to id_1 , the identity map $1 \rightarrow 1$, which implies that $c \times f$ should be equivalent to $\text{id}_1 \times f = f$. Since c was arbitrary, we have that any two substochastic maps that differ by a global nonzero multiplicative factor should be equivalent.

Definition 2.9. *Given two substochastic maps $f, g : X \rightarrow Y$, we write $f \propto g$ if there exists a real number $\lambda > 0$ such that $f(y|x) = \lambda \cdot g(y|x)$ for all $x \in X$ and $y \in Y$. We write $[f]$ for the equivalence class of $f : X \rightarrow Y$.*

Note that λ has to be same scalar factor for all inputs in X (see paragraph preceding Definition 2.9). With this assumption, composition and product of stochastic maps are congruences for \propto , so that stochastic maps up to \propto form a symmetric monoidal category, which we call FinProjStoch , with much of the same structure as FinStoch [36, Theorem 6.4]. FinProjStoch will be our target semantics for ProbCirc , the syntax of all circuits, including those that contain explicit conditioning via \triangleright generators.

Definition 2.10 (Semantics of all circuits). *For each generator g of CausCirc , let $\llbracket g \rrbracket_\propto$ be the equivalence class of $\llbracket g \rrbracket$. In addition, let $\llbracket \triangleright \rrbracket_\propto$ be the equivalence class of the subdistribution defined in (5). Finally, sequential and parallel composition are interpreted as in Definition 2.7.*

Proposition 2.11. *The map $\llbracket \cdot \rrbracket_\propto$ defines a symmetric monoidal functor from ProbCirc to FinProjStoch .*

Props and symmetric monoidal theories. Following standard practice, we can see our circuits as morphisms of a product and permutation category, or *prop*. Formally, a prop is a strict SMC with the natural numbers as the set of objects and addition as the monoidal product. In a prop, all objects are thus monoidal products of a single generating object, *viz.* 1.

Given a set of generating morphisms Σ , we can form the free prop P_Σ as explained in Section 2.1: by quotienting Σ -terms by the laws of SMC. For us here, ProbCirc is the free prop over the set of generators in (1)-(3) and CausCirc the free prop over those in (1)-(2).

Now that we have a formal syntax and semantics, our aim is to reason about semantic equivalence of circuits purely equationally. As we saw, since the syntax of circuits is a prop and its interpretation $\llbracket \cdot \rrbracket$ is a symmetric monoidal functor (Proposition 2.11) into another SMC, the laws of SMCs already capture some semantic equivalences between circuits, but they are not sufficient to derive all valid equivalences. The main aim of this paper is to add enough axioms to obtain a completeness result.

Given a set E consisting of equations between Σ -terms, we write $=_E$ for the smallest congruence with respect to the two compositions $;$ and \otimes containing E . We call the elements of E axioms and the pair (Σ, E) a *symmetric monoidal theory* (or simply, *theory*). Details on the existence and construction of free props on a given theory can be found in [2, Appendix B] or [40]. We say that a theory is *sound* if $c =_E d$ implies $\llbracket c \rrbracket = \llbracket d \rrbracket$ and *complete* when the reverse implication holds. A sound and complete theory is also called an *axiomatisation*. When moreover, for every morphism f of the target semantics there exists a morphism c in P_Σ , the syntax, such that $\llbracket c \rrbracket = f$, we say that a sound and complete theory is a *presentation* of the image of $\llbracket \cdot \rrbracket$.

In what follows we give a sound and complete theory for equivalence of causal circuits (Section 3). Equivalently, this will provide a presentation of $\text{FinStoch}_{\mathbb{B}}$, the SMC of stochastic maps between sets that are powers of \mathbb{B} . For our second main result, we extend this theory to axiomatise the full language of probabilistic circuits, including explicit conditioning (Section 4).

3 An Axiomatisation of Causal Circuits

We now focus on causal circuits, the axiomatisation of which is of independent interest, since it gives a presentation of a monoidal subcategory of FinStoch .

3.1 Equational Theory

We consider CausCirc terms quotiented by the set of equations in Fig. 4 and the laws of symmetric monoidal categories (4). Note that any axiom which uses parameters p, q, \dots implicitly quantifies over $[0, 1]$, unless stated otherwise.

A few comments are in order, to clarify the meaning of the axioms: in the first block, the **A**-axioms define a *cocommutative comonoid* consisting of a *comultiplication* $\multimap : 1 \rightarrow 2$ and a *counit* $\multimap : 1 \rightarrow 0$, together with equations guaranteeing that all different ways of copying a value are equal, that copying followed by discarding is the same as doing nothing, and that two copies are identical, so can be exchanged; in the second block, axioms **B1-B3** define a *commutative monoid* consisting of two generators in our signature, a *multiplication*, given by the Boolean AND-gate $\curlywedge : 2 \rightarrow 1$, and a *unit*, the Boolean constant **true**, represented by $\mathbf{1} : 0 \rightarrow 1$, with their associated axioms: \curlywedge is associative, commutative and its unit is $\mathbf{1}$; in the third block, axioms from Boolean algebra are given, stating the *involution* of negation, the *idempotence* of conjunction, the usual *complementation* axiom and the *distributivity* of conjunction over disjunction; in the fourth block, the axioms allow us to copy (**C**) arbitrary Boolean operations using \multimap . In other words, conjunction, negation, and the two Boolean constants **true** (1) and **false** (0) distribute over \multimap . Semantically, this characterises them as *deterministic* maps without any probabilistic behaviour; in the fifth block, the **D**-axioms allow us to discard (**D**) the result of Boolean operations, using \multimap . Note that axiom **D3** applies more generally to $\mathbf{p} : 0 \rightarrow 1$ for any $p \in [0, 1]$ and encodes the normalisation of the distribution

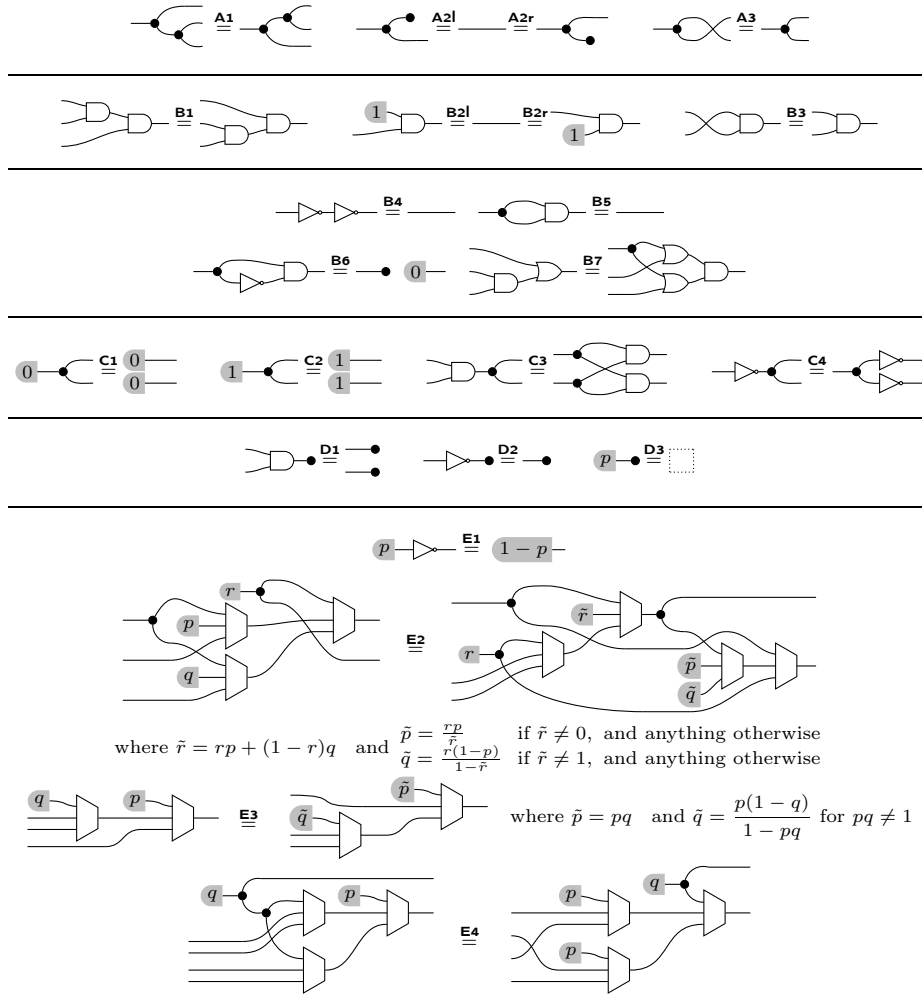


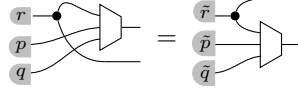
Fig. 4: Axioms for causal circuits.

$\llbracket p- \rrbracket = p|1\rangle + (1-p)|0\rangle$, i.e. the fact that $p + (1-p) = 1$. Semantically, this characterises these as *causal* operations.

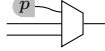
Note that the **A-D** blocks allow us to obtain a diagrammatic form of substitution for the purely Boolean fragment of the theory: it guarantees that any *purely Boolean* expression can be copied (**C**) and discarded (**D**). In addition, they give us algebraic structures that occur in similar diagrammatic calculi: $\overline{\cup}\text{---}$, $\mathbf{1}$ — with $\text{---}\overline{\cup}$, $\text{---}\bullet$ together form a (co)commutative *bimonoid*.

Finally, the last block contains all equations concerning the probabilistic behaviour of **CausCirc** terms. Axiom **E1** is clear: applying a negation after a coin flip with bias p is the same as exchanging the probabilities of the corresponding

distribution, *i.e.*, $p|0\rangle + (1-p)|1\rangle = (1-p)|\neg 0\rangle + p|\neg 1\rangle$. This implies in particular that $|0\rangle$ is the negation of $|1\rangle$, which is a simple Boolean algebra identity. Axiom **E2** generalises a simpler equality, which is the diagrammatic counterpart of two ways of disintegrating a joint distribution of two variables, as $\mathbb{P}(y_0|y_1)\mathbb{P}(y_1) = \mathbb{P}(y_0)\mathbb{P}(y_1|y_0)$:


(6)

E2 further conditions this identity on the value of some input variable (the top left wire). This more general form will be needed to iteratively disintegrate joint distributions over more than two variables, in the proof of completeness. Note that by varying the values of p, q, r in this last diagram, the semantics of the corresponding circuit ranges over all distributions on \mathbb{B}^2 . Axiom **E3** can be understood as form of associativity for convex sums. Indeed, circuits of the form



for some $p \in [0, 1]$, are interpreted as the distribution $p|x_0\rangle + (1-p)|x_1\rangle$, conditional on inputs x_0 and x_1 . That is, circuits of this form simply take the convex sum of their two inputs. This is a binary operation, which is associative *up to reweighing* as axiom **E3** states. Similar axioms appear in previous work that contain convex sum (aka probabilistic choice) as a primitive [38,12,33]. A difference with our work is that convex sum is a derived operation for us. This will allow us to derive laws commonly taken as axioms in related work, most notably the *distributivity* of **if-then-else** over probabilistic choice [33, Section 5]. Axiom **E4** allows us to break redundant correlation between the guard of two **if-then-else** gates: notice that there is one fewer $\rightarrow \cap$ node on the right than on the left. This ability will prove crucial in our proof of completeness, helping us to rewrite complicated circuits into simple trees of convex sums.

3.2 Soundness and Completeness

Using our equational theory, every syntactically provable equality holds in the semantics. The proof of the theorem below is straightforward, as it suffices to verify that each axiom in Fig. 4 is a semantic equality.

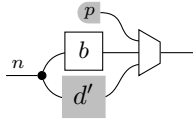
Theorem 3.1 (Soundness). *For circuits $c, d : m \rightarrow n$, if $c = d$ then $\llbracket c \rrbracket = \llbracket d \rrbracket$.*

To prove completeness, we use a normal form argument, a common strategy in completeness proofs for diagrammatic calculi [31,20]. The idea is that normal forms provide a unique syntactic representative for each semantic object in the image of the interpretation functor $\llbracket \cdot \rrbracket$, thereby guaranteeing that, if two circuits denote the same distribution, they will be equal to the same normal form. The proof then relies on a normalisation procedure: an explicit algorithm which rewrites any given circuit into normal form, using only the proposed axioms.

Our normalisation argument proceeds in two high-level steps: 1) we give a procedure to normalise $n \rightarrow 1$ causal circuits; 2) we then reduce the case of general $m \rightarrow n$ causal circuits to multiple applications of the procedure for the $n \rightarrow 1$ case, by syntactically disintegrating a given circuit.

Single-output circuits. The normalisation procedure for $n \rightarrow 1$ circuits begins by rewriting a given circuit into *pre-normal form*. This pre-normal form intuitively represents a diagram as a convex combination of Boolean expressions.⁴

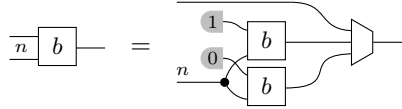
Definition 3.2 (Pre-normal form). *A circuit $d : n \rightarrow 1$ is in pre-normal form if it is in the form defined inductively below:*



where $b : n \rightarrow 1$ is a Boolean circuit and $d' : n \rightarrow 1$ is itself in pre-normal form or is a Boolean circuit (base case).

The core of rewriting a circuit into pre-normal form is to move all occurrences of the probabilistic generators p to the guard of some **if-then-else** gate, and then to eliminate correlations between different branching gates sharing the same p (via \rightarrow nodes). We do the first step through repeated usage of *Shannon expansion*, a well-known consequence of the Boolean algebra axioms (**A-D**).

Lemma 3.3 (Shannon expansion). *For every Boolean circuit $b : 1 + n \rightarrow 1$, we have*



While Shannon expansion can introduce further correlations between the guards of different **if-then-else** gates, we remove these with axiom **E4**, to reach our pre-normal form. This process itself might leave a trail of convex sums, which we need to re-associate in the correct way using axiom **E3** to obtain a circuit in pre-normal form.

Lemma 3.4 (Pre-normalisation). *Every $n \rightarrow 1$ causal circuit is equal to one in pre-normal form.*

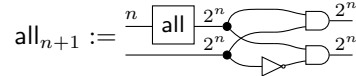
Our pre-normal form is not unique: different circuits may be mapped to the same stochastic map by $\llbracket \cdot \rrbracket$. We thus have to establish a method for showing that any two semantically equivalent circuits in pre-normal form are equal. To do so, we define a suitable *normal form* for single-output circuits. Given some circuit $c : n \rightarrow 1$, its normal form will be a syntactic encoding of the probability

⁴ This can be seen as a generalisation of Birkhoff's theorem for doubly stochastic matrices [3] to the (singly-)stochastic case.

table of $\llbracket c \rrbracket : \mathbb{B}^n \rightarrow \mathbb{B}$: a disjunction of the Bernoulli distribution, in the form of a single \textcircled{p} —, for every possible input. Thus, for any input $x \in \mathbb{B}^n$, we can read the probability of $\llbracket c \rrbracket(x)$ immediately from the normal form of c .

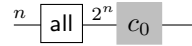
To define the normal form, we will need circuits $\text{all}_n : n \rightarrow 2^n$ which encode all possible 2^n bit vectors of length n , that is, all possible inputs on n wires.

Definition 3.5. Let $\text{all}_n : n \rightarrow 2^n$ be a family of circuits defined by induction on $n \in \mathbb{N}$ with $\text{all}_0 := \textcircled{1}$ — and

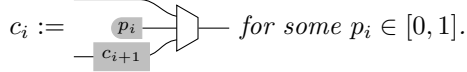


In other words, all_n demultiplexes a n -bit vector into a 2^n -bit vector where exactly one wire can be 1 at the same time, corresponding to the number encoded by the input.

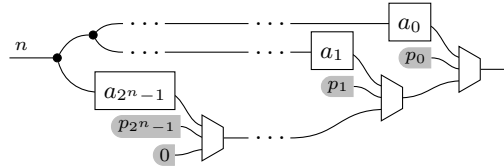
Definition 3.6 (Normal Form for $n \rightarrow 1$ causal circuits). A circuit $n \rightarrow 1$ is in normal form when it is of the form



where $c_0 : 2^n \rightarrow 1$ is a circuit defined recursively as follows: $c_{2^n} := \textcircled{0}$ — and



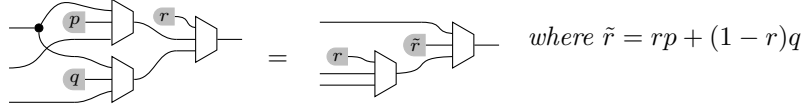
One way to visualise the normal form more easily is as follows:



where each $a_i : n \rightarrow 1$ corresponds to the circuit-encoding of the i -th bit vector of \mathbb{B}^n in the order given by $\text{all}_n : n \rightarrow 2^n$. Since it is simply a syntactic encoding of the conditional probability table of $\llbracket c \rrbracket$ for a given $c : n \rightarrow 1$, this normal form is clearly unique.

It remains to show that any causal circuit of type $n \rightarrow 1$ is equal to one in normal form, by giving a procedure to rewrite it as such using the axioms of our theory. The key idea of this normalisation procedure is to start from the pre-normal form, and progressively aggregate all probabilities coming from the Boolean components of the convex sum that the pre-normal form represents. More specifically, for all possible bit-vector inputs $x \in \mathbb{B}^n$ for a circuit $\llbracket c \rrbracket : \mathbb{B}^n \rightarrow \mathbb{B}$, we need to take a convex sum of all the outputs coming from each of the Boolean components in the pre-normal form at x , weighted by the probability given by the guard of the corresponding **if-then-else** gate. This is done by exploiting a special case of axiom **E2**, obtained by discarding the second output on both the rhs and lhs of the axiom:

Lemma 3.7. *The following equality is derivable for all $r, p, q \in [0, 1]$:*

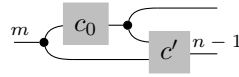


We can then establish our first completeness result:

Lemma 3.8 (Completeness for $n \rightarrow 1$ causal circuits). *Every causal circuit of type $n \rightarrow 1$ is equal to one in normal form.*

Multi-output circuits. For the completeness of $m \rightarrow n$ causal circuits, the main idea is to reduce the problem to the $n \rightarrow 1$ case by reproducing the disintegration of the corresponding distribution syntactically. The normal form is defined as a circuit disintegrated in the order of the output wires, from top to bottom.

Definition 3.9 (Normal form for $m \rightarrow n$ causal circuits). *A circuit $c : m \rightarrow n$ is in normal form when it is of the form*



where $c_0 : m \rightarrow 1$ is in normal form in the sense of Definition 3.6 and $c' : m \rightarrow n - 1$ is in normal form, with the following additional requirement: if $\llbracket c_0 \rrbracket(b|x)$ has probability 0 for $(b, x) \in \mathbb{B} \times \mathbb{B}^m$, then $\llbracket c' \rrbracket(b, x) = |0^{n-1}\rangle$.

The last condition of the above definition is here to deal with the non-uniqueness of disintegrations. As we saw in §2.2, there are several possible disintegrations of a joint distribution $\mathbb{P}(x, y)$ into a marginal $\mathbb{P}(x)$ and a conditional $\mathbb{P}(y|x)$ if the marginal does not have full support—in this case, the value of $\mathbb{P}(y|x)$ for x such that $\mathbb{P}(x) = 0$ can be arbitrary. Because we want our normal forms to represent a given stochastic map uniquely, we need to fix a syntactic convention to handle such cases. Our chosen convention is to place all the probability mass on one arbitrary value, namely 0, so that $\mathbb{P}(y|x) = |0\rangle$.

Proposition 3.10 (Uniqueness of normal forms). *Any two circuits $c, d : m \rightarrow n$ in normal form and such that $\llbracket c \rrbracket = \llbracket d \rrbracket$, are equal.*

We rewrite a given circuit $c : m \rightarrow n + 1$ as the composition of an $m \rightarrow 1$ circuit whose semantics is that of the marginal distribution on the variable corresponding to the first output wire, and a circuit $m + 1 \rightarrow n$ whose semantics is that of the conditional distribution of the variable corresponding to the second wire, conditional on the first. Semantically, this amounts to disintegrating a joint distribution $\mathbb{P}(x_0, \dots, x_n)$ into the product $\mathbb{P}(x_0)\mathbb{P}(x_1, \dots, x_n|x_0)$, by conditioning on the first variable. Once again, axiom **E2** is the engine of the proof: its lhs and rhs represent the two different ways of disintegrating a distribution on two output wires/variables (conditional on some inputs, allowing us to reproduce the

process by going through the normal form while preserving its structure). In this way, starting from a circuit $m \rightarrow n$, we can obtain n circuits of type $m + k \rightarrow 1$, with $0 \leq k \leq n - 1$ by repeatedly computing disintegrations by induction on the number of output wires. By exploiting $n \rightarrow 1$ completeness, we obtain the following theorem.

Theorem 3.11 (Completeness for $m \rightarrow n$ causal circuits). *Every causal circuit of type $m \rightarrow n$ is equal to one in normal form. Thus, for any two causal circuits $c, d : m \rightarrow n$, if $\llbracket c \rrbracket = \llbracket d \rrbracket$ then $c = d$.*

Finally, by characterising the image of $\llbracket \cdot \rrbracket : \text{CausCirc} \rightarrow \text{FinStoch}$, we can conclude that our theory *presents* $\text{FinStoch}_{\mathbb{B}}$, the full monoidal subcategory of FinStoch on objects that are powers of the two-element set, *i.e.*, \mathbb{B}^n for $n \in \mathbb{N}$.

Proposition 3.12. *The image of CausCirc under $\llbracket \cdot \rrbracket$ is $\text{FinStoch}_{\mathbb{B}}$: for every morphism f of $\text{FinStoch}_{\mathbb{B}}$, there exists a causal circuit c such that $\llbracket c \rrbracket = f$.*

Theorem 3.13. *The theory given by the generators in (1)-(2) and the equations of Fig. 4 is a presentation of $\text{FinStoch}_{\mathbb{B}}$.*

Remark 3.14 (Encoding distributions over arbitrary finite sets). A language that can model any distribution over binary variables is also able to encode arbitrary distributions over finite sets, *e.g.*, using what is commonly known as a *one-hot encoding*: a standard way to encode a set X as its copy inside the larger \mathbb{B}^X . To define it, we will identify \mathbb{B}^X with the powerset of X . Given a distribution φ over some finite set X , we can define a distribution $O(\varphi)$ over \mathbb{B}^X by $O(\varphi)(x) = \varphi(x)$ for $x \in X$ and $O(\varphi)(U) = 0$ for non-singleton subsets U of X . On the syntax side, we encode a distribution over X by a circuit with $|X|$ wires, whose semantics is a distribution over $\mathbb{B}^{|X|}$. Thus, each element of X corresponds to a single (output) wire of the associated circuit, but different wires are not allowed to be set to **true** at the same time. One-hot encodings are ubiquitous in computer science. For example, the designers of the language Dice use it to define probabilistic programs over arbitrary finite sets of events [22].

The one-hot encoding can be extended straightforwardly to substochastic maps: for $f : X \rightarrow Y$, let $OH(f)$ be the substochastic map given by $OH(f)(x) = O(f(x))$ for $x \in X$ and $OH(f)(U)$ the zero-everywhere subdistribution if U is not a singleton subset. Thus, we can use our circuits and the associated equational theory to reason about the whole of FinStoch (over arbitrary finite sets rather than powers of the two-element set), with some caveats, which we now explain.

OH does not define a functor from FinSubStoch to itself. While this encoding preserves composition, it does not preserve identities or the monoidal product. Indeed $OH(\text{id}_X)$ is not the identity stochastic map over \mathbb{B}^X . By the definition above, we see that it only acts as the identity over the singletons, and is the zero subdistribution otherwise. While this is not the identity needed to define a functor, $OH(\text{id}_X)$ is an idempotent map. This suggests one way to fix the issue: OH is better thought of as a functor from FinStoch into its Karoubi envelope, whose objects are pairs (X, e) of a set X and an idempotent $e : X \rightarrow X$. The categorical aspects of one-hot encoding are interesting in their own right, but they are not our focus, so we will not pursue this conjecture formally here.

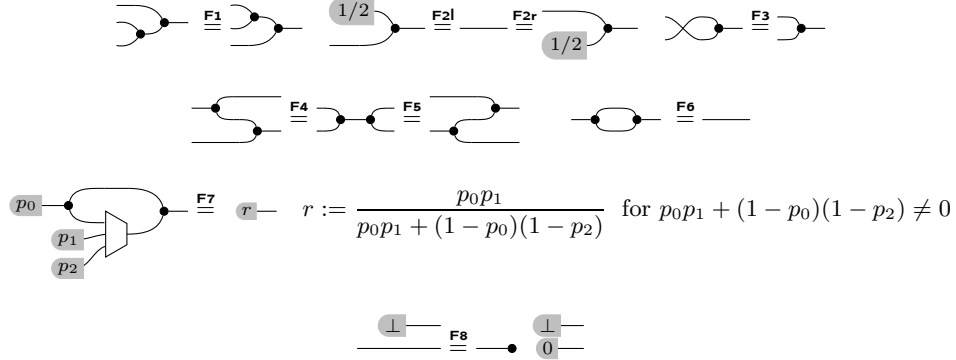


Fig. 5: Axioms for conditioning.

4 Adding conditioning

The power and usefulness of PPLs comes from their ability to specify distributions conditional on observations defined by the program. Our minimal PPL had a construct $(x =: y)$ to condition on the value of x and y being equal, and our circuits have \curvearrowright for the same purpose. In this section, we extend the axiomatisation of Section 3 to all circuits, including those with explicit conditioning.

4.1 Equational theory

The extended equational theory consists of the axioms for causal circuits (Fig. 4) and those given in Fig. 5.

Axioms **F1-F3** make \curvearrowright and $\frac{1}{2}$ into a commutative monoid. The unitality axiom (**F2**) is only sound when we quotient subdistributions by a global multiplicative factor (see Definition 2.9). Had we given our semantics in terms of plain subdistributions (without quotient), these two equalities would not hold. Axioms **F4-F6** state that \curvearrowright , \rightarrow , \curvearrowright , $\frac{1}{2}$ form a *special Frobenius algebra*.

Axiom **F7** encodes the action of conditioning on a distribution over two variables. If we call x_1, x_2 these two variables we can see that the diagram on the right gives this distribution as the product of a Bernoulli with parameter p_0 for x_1 with the distribution of x_2 conditional on x_1 , given by two Bernoulli with parameters p_1 and p_2 . Clearly, x_1 and x_2 are both **true** with probability $p_0 p_1$ and both **false** with probability $(1 - p_0)(1 - p_2)$. Since \curvearrowright has the effect of constraining x_1 and x_2 to be equal, the result is a new Bernoulli distribution with parameter r , whose denominator is the normalisation factor $p_0 p_1 + (1 - p_0)(1 - p_2)$ and whose numerator is the probability that x_1 and x_2 are both **true**.

Axiom **F8** deals with failure, that is, with the special circuit $\frac{1}{0}$. It constrains 0 to be equal to 1, which is unsatisfiable. Any circuit that contains failure is semantically equal to the zero subdistribution, a fact which we will derive using axiom **F8** (Lemma 4.6).

Theorem 4.1 (Soundness). *For $c, d : m \rightarrow n$, if $c = d$ then $\llbracket c \rrbracket_\infty = \llbracket d \rrbracket_\infty$.*

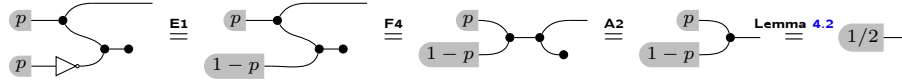
From these axioms, we can derive the action of \triangleright on p -generators: it multiplies the parameters and re-normalises the resulting subdistribution.

Lemma 4.2. *The following equality is derivable for all $p, q \in [0, 1]$ satisfying $pq + (1-p)(1-q) \neq 0$:*

$$\begin{array}{c} p \\ \bullet \\ q \end{array} \text{---} = \text{---} \begin{array}{c} r \end{array} \quad \text{where } r := \frac{pq}{pq + (1-p)(1-q)}$$

We can now prove the correctness of the two examples from the introduction.

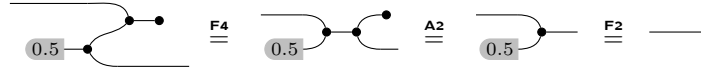
Example 4.3 (von Neumann's trick, correctness). Assume $p \in (0, 1)$. We have



since

$$\frac{p(1-p)}{p(1-p) + (1-p)(1-(1-p))} = \frac{p(1-p)}{p(1-p) + (1-p)p} = \frac{p(1-p)}{2p(1-p)} = \frac{1}{2}$$

Example 4.4. Using the axioms for Frobenius algebras, we can show the partial program from Section 1 (translated into the leftmost diagram below) is equivalent to the identity function on Boolean variables $\mathbf{x} : \mathbb{B} \vdash \mathbf{x} : \mathbb{B}$:



4.2 Completeness

We now show the completeness of the proposed equational theory.

First, we reduce the problem to the case of circuits without input wires, that is, circuits of type $0 \rightarrow n$. This is possible because the Frobenius algebra structure $(\text{---}\bullet, \text{---}\bullet, \triangleright, \text{---}\bullet, \text{---}\bullet)$ with which we can bend wires using $\text{---}\bullet$ and \triangleright . This allows us to transform any $m \rightarrow n$ circuits into one of type $0 \rightarrow m + n$ and vice-versa; moreover, any equality that we can apply to the former also applies to the latter. In categorical terms, we say that our semantics, the SMC FinProjStoch , is *compact-closed*.

Proposition 4.5 (Compactness). *There is a one-to-one mapping $(\cdot)^b$ between diagrams of type $m \rightarrow n$ and those of type $0 \rightarrow m + n$; in addition, for any $c, d : m \rightarrow n$, $c = d$ iff $c^b = d^b$.*

The main idea of the completeness proof is to remove all explicit conditioning from $0 \rightarrow n$ circuits to reduce the completeness of the whole syntax to the causal fragment. The main difference with the causal case is that conditioning introduces the possibility of failure: circuits that contain the unsatisfiable constraint $\text{---}\bullet := \text{---}\bullet$ and whose semantics is the zero subdistribution. We deal with this special case separately, since any two circuits containing failure are equal:

Lemma 4.6 (Failure). *For any two circuits $c, d : m \rightarrow n$, we have*

$$\begin{array}{c} \perp \\ \hline m \quad \boxed{c} \quad n \end{array} = \begin{array}{c} \perp \\ \hline m \quad \boxed{d} \quad n \end{array}$$

Note that this fact is responsible for the unusual behaviour of our **if-then-else** construct: failure in one branch, even if unevaluated, will imply the failure of the whole program. This is an unavoidable consequence of choosing to identify programs that denote the same distribution up to a renormalisation factor.

Theorem 4.7 (Completeness). *For any two circuits $c, d : m \rightarrow n$ with conditioning, $\llbracket c \rrbracket_\infty = \llbracket d \rrbracket_\infty$ iff $c = d$.*

Proof. By compactness (Proposition 4.5), it is enough to prove the statement for $c, d : 0 \rightarrow n$. We want to show that we can always eliminate all conditioning nodes \curvearrowright to rewrite any diagram $c : 0 \rightarrow n$ into normal form. For this, it is enough to show that any diagram $c : 0 \rightarrow n$ composed (on the right, necessarily) with \curvearrowright is equal to some causal, *i.e.* conditioning-free, circuit.

First, we can assume wlog that \curvearrowright is plugged on the first two wires of c ; if it was not, we could simply reorder them and rewrite the resulting diagram into normal form first. Moreover, by Theorem 3.11, there exists circuits c_0, c_1 and c', c'' in normal form such that:

$$\begin{aligned} \begin{array}{c} \curvearrowright \\ \hline c \end{array} \quad n-2 &= \begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c' \end{array} \quad n-2 \end{array} = \begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c_1 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} \\ &= \begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c_1 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} \stackrel{\mathbf{A1}}{=} \begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c_1 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} \\ &\stackrel{\mathbf{F5;F4}}{=} \begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c_1 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} \stackrel{\mathbf{A3}}{=} \begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c_1 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} \\ &\stackrel{\mathbf{F4;F5;F}}{=} \begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c_1 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} \stackrel{\mathbf{A3}}{=} \begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c_1 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} \end{aligned}$$

Since c_0, c_1 are themselves in normal form, there exist weights p_0, q_0, q_1 such that

$$\begin{array}{c} \begin{array}{c} \curvearrowright \\ \hline c_0 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c_1 \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} = \begin{array}{c} p_0 \quad \begin{array}{c} \curvearrowright \\ \hline \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array}$$

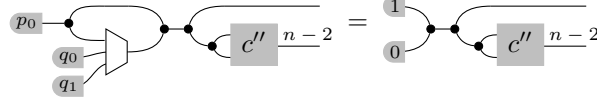
q_0
 q_1

There are two cases to consider. First, if $p_0 q_0 + (1 - p_0)(1 - q_1) \neq 0$, we have

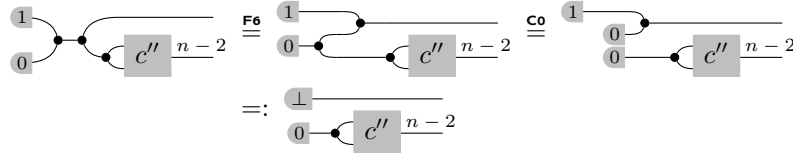
$$\begin{array}{c} p_0 \quad \begin{array}{c} \curvearrowright \\ \hline \end{array} \quad \begin{array}{c} \curvearrowright \\ \hline c'' \end{array} \quad n-2 \end{array} \stackrel{\mathbf{F7}}{=} \begin{array}{c} r \quad \boxed{c''} \quad n-2 \end{array}$$

q_0
 q_1

where $r := \frac{p_0 q_0}{p_0 q_0 + (1-p_0)(1-q_1)}$. Notice that the sub-circuit in the dashed box above is in normal form, so the final circuit is also in normal form. Second, if $p_0 q_0 + (1-p_0)(1-q_1) = 0$, then we either have $p_0 = 1$ and $q_0 = 0$, or $p_0 = 0$ and $q_1 = 1$. Intuitively, in these two cases, the conditioning constraint imposed by \curvearrowright is unsatisfiable, and we get:



Note that this is true in both cases, by commutativity of \curvearrowright . Then,



Since any two circuits containing \perp are equal by Lemma 4.6, the last circuit is equal to one in normal form. Thus, any $c : 0 \rightarrow n$ is equal to a circuit in normal form and the same reasoning as in the proof of Theorem 3.11 shows that any two semantically equivalent circuits $0 \rightarrow n$ are equal to the same circuit in normal form, which concludes the proof.

5 Discussion

In summary, we presented a complete axiomatisation of discrete probabilistic program equivalence. We were able to achieve this result by translating a conventional PL into a calculus of string diagrams, representing Boolean circuits extended with primitives denoting Bernoulli distributions and explicit conditioning, which we equipped with a compositional semantics in terms of subdistributions (up to some re-normalisation factor). Then, we gave a complete equational theory for circuits without conditioning, that is, a presentation of the category of stochastic maps between sets that are powers of the Booleans. Finally, we showed how to extend the preceding axiomatisation to all circuits—and therefore all programs—including those that contain explicit conditioning.

Related work. Our choice of syntax and semantics for conditioning is based on the work of Staton and Stein [35,36] where they define the CD-calculus, a PPL with an exact conditioning operation like $(x := y)$ for both Gaussian and discrete probabilistic programming. In the discrete case, they consider two semantics: the one we use here and another in terms of plain subdistributions (without re-normalisation). While they do study some of the equational properties of the CD-calculus in general and of exact conditioning in particular, they do not give a complete axiomatisation for the discrete case. It should be noted that a complete axiomatisation of the Gaussian case appeared recently [37].

Moreover, a presentation of the SMC of stochastic matrices *with the disjoint sum* as monoidal product was given by Fritz [12], with an axiomatisation similar to that of Stone [38]. This already provides a language expressive enough to encode arbitrary distributions over finite sets⁵. However, unlike our circuits, stochastic matrices with disjoint sum cannot meaningfully express correlation between different variables, making it insufficiently expressive as a language for probabilistic programming. Our work is the first to present (a subcategory of) the same underlying category, but *with the cartesian product* as monoidal product.

Finally, this work fits within the broader programme of axiomatising probability theory in Markov categories [13]. Many papers in the same area use string diagrams to generalise and prove key results in statistics and probability theory [17,15,30,14,27,16], without necessarily having an axiomatisation of their domain of interest. Axioms for conditioning in the setting of *partial* Markov categories have also been used to provide a synthetic account of Bayes theorem [10].

Future work. Our paper opens several new directions. Firstly, we believe that axiomatisations of alternative semantics for conditioning can be achieved with a few simple modifications. In particular, the semantics of conditioning in terms of plain subdistributions (without quotient) seems within reach and would be of independent interest. Secondly, we would like to axiomatise KL-divergence between probabilistic circuits/programs using a *quantitative* equational theory [1,30]. This would require us to replace plain equalities with equalities of the form $c \equiv_\varepsilon d$, to represent the fact that the KL-divergence of $\llbracket c \rrbracket$ relative to $\llbracket d \rrbracket$ is less than ε . Interesting new identities can be expressed in this framework, such as

$$\text{p} \text{---} \bullet \text{---} \left(\begin{array}{c} \text{---} \\ \text{---} \end{array} \right) \equiv_\varepsilon \left(\begin{array}{c} \text{p} \text{---} \\ \text{p} \text{---} \end{array} \right) \quad \text{where } \varepsilon \geq p \ln \left(\frac{p}{p^2} \right) + (1-p) \ln \left(\frac{1-p}{(1-p)^2} \right).$$

Such a quantitative identity measures the information loss of assuming that some phenomenon can be modelled by a single coin flip when the true distribution consists of two independent coin tosses. The same approach could be used to axiomatise other quantitative measures of program divergence, such as the total variation distance [30].

Acknowledgments.

The authors would like to thank Wojciech Róžowski and Dario Stein for many helpful discussions. RP, MTR and AS acknowledge support from ERC grant Autoprobe no. 101002697, ARIA Safeguarded AI program, and EPSRC Standard Grant CLeVer EP/S028641/1. FZ acknowledges support from EPSRC grant EP/V002376/1, MIUR PRIN P2022HXNSC, and ARIA Safeguarded AI program.

⁵ All of them, since every finite set is in bijection with a disjoint sum of a single generating singleton set.

References

1. Bacci, G., Mardare, R., Panangaden, P., Plotkin, G.: Quantitative equational reasoning. *Foundations of Probabilistic Programming* p. 333 (2020)
2. Baez, J.C., Coya, B., Rebro, F.: Props in network theory. *Theory & Applications of Categories* **33** (2018)
3. Birkhoff, G.: Three observations on linear algebra. *Univ. Nac. Tucumán. Revista A.* **5**, 147–151 (1946)
4. Boisseau, G., Sobociński, P.: String diagrammatic electrical circuit theory. *Electronic Proceedings in Theoretical Computer Science* **372**, 178–191 (Nov 2022). <https://doi.org/10.4204/eptcs.372.13>
5. Boole, G.: *The Laws of Thought* (1854). The Open court publishing company, London, (1854)
6. Cho, K., Jacobs, B.: Disintegration and Bayesian inversion via string diagrams. *Mathematical Structures in Computer Science* **29**(7), 938–971 (2019)
7. Claret, G., Rajamani, S.K., Nori, A.V., Gordon, A.D., Borgström, J.: Bayesian inference using data flow analysis. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. p. 92–102. ESEC/FSE 2013, Association for Computing Machinery, New York, NY, USA (2013). <https://doi.org/10.1145/2491411.2491423>
8. Coecke, B., Kissinger, A.: *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press (2017)
9. Dash, S., Kaddar, Y., Paquet, H., Staton, S.: Affine monads and lazy structures for Bayesian programming. *Proc. ACM Program. Lang.* **7**(POPL) (jan 2023). <https://doi.org/10.1145/3571239>
10. Di Lavore, E., Román, M.: Evidential decision theory via partial markov categories. In: *2023 38th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. pp. 1–14. IEEE (2023)
11. Fierens, D., Van den Broeck, G., Renkens, J., Shterionov, D., Gutmann, B., Thon, I., Janssens, G., De Raedt, L.: Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming* **15**(3), 358–401 (Apr 2014). <https://doi.org/10.1017/s1471068414000076>
12. Fritz, T.: A presentation of the category of stochastic matrices. *arXiv preprint arXiv:0902.2554* (2009)
13. Fritz, T.: A synthetic approach to markov kernels, conditional independence and theorems on sufficient statistics. *Advances in Mathematics* **370**, 107239 (aug 2020). <https://doi.org/10.1016/j.aim.2020.107239>
14. Fritz, T., Gonda, T., Houghton-Larsen, N.G., Lorenzin, A., Perrone, P., Stein, D.: Dilations and information flow axioms in categorical probability. *Mathematical Structures in Computer Science* **33**(10), 913–957 (2023)
15. Fritz, T., Gonda, T., Perrone, P.: De finetti’s theorem in categorical probability. *Journal of Stochastic Analysis* **2**(4), 6 (2021)
16. Fritz, T., Gonda, T., Perrone, P., Rischel, E.F.: Representable markov categories and comparison of statistical experiments in categorical probability. *Theoretical Computer Science* **961**, 113896 (2023)
17. Fritz, T., Rischel, E.F.: Infinite products and zero-one laws in categorical probability. *Compositionality* **2** (2020)
18. Gehr, T., Misailovic, S., Vechev, M.: Psi: Exact symbolic inference for probabilistic programs. In: Chaudhuri, S., Farzan, A. (eds.) *Computer Aided Verification*. pp. 62–83. Springer International Publishing (2016)

19. Giry, M.: A categorical approach to probability theory. *Categorical Aspects of Topology and Analysis* pp. 68–85 (1982)
20. Gu, T., Piedeleu, R., Zanasi, F.: A complete diagrammatic calculus for boolean satisfiability. *Electronic Notes in Theoretical Informatics and Computer Science Volume 1-Proceedings of...* (2023). <https://doi.org/10.46298/entics.10481>
21. Hoare, C.A.R., Hayes, I.J., Jifeng, H., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: Laws of programming. *Commun. ACM* **30**(8), 672–686 (aug 1987). <https://doi.org/10.1145/27651.27653>
22. Holtzen, S., Van den Broeck, G., Millstein, T.: Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages* **4**(OOPSLA), 1–31 (2020)
23. Kozen, D.: Semantics of probabilistic programs. *Journal of Computer and System Sciences* **22**(3), 328–350 (1981). [https://doi.org/https://doi.org/10.1016/0022-0000\(81\)90036-2](https://doi.org/https://doi.org/10.1016/0022-0000(81)90036-2)
24. Kozen, D.: Kleene algebra with tests. *ACM Trans. Program. Lang. Syst.* **19**(3), 427–443 (may 1997). <https://doi.org/10.1145/256167.256195>
25. MacLane, S.: *Categories for the Working Mathematician*. Springer New York (1971). <https://doi.org/10.1007/978-1-4612-9839-7>
26. Moss, S., Perrone, P.: Probability monads with submonads of deterministic states. In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science. LICS '22*, Association for Computing Machinery, New York, NY, USA (2022). <https://doi.org/10.1145/3531130.3533355>
27. Moss, S., Perrone, P.: A category-theoretic proof of the ergodic decomposition theorem. *Ergodic Theory and Dynamical Systems* **43**(12), 4166–4192 (2023)
28. Narayanan, P., Carette, J., Romano, W., Shan, C.c., Zinkov, R.: Probabilistic inference by program transformation in Hakaru (system description). In: Kiselyov, O., King, A. (eds.) *Functional and Logic Programming*. pp. 62–79. Springer International Publishing, Cham (2016)
29. Olmedo, F., Gretz, F., Jansen, N., Kaminski, B.L., Katoen, J.P., McIver, A.: Conditioning in probabilistic programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)* **40**(1), 1–50 (2018)
30. Perrone, P.: Markov categories and entropy. *IEEE Transactions on Information Theory* (2023)
31. Piedeleu, R., Zanasi, F.: A string diagrammatic axiomatisation of finite-state automata (2020), <https://arxiv.org/abs/2009.14576>
32. Piedeleu, R., Zanasi, F.: An introduction to string diagrams for computer scientists (2023), <https://arxiv.org/abs/2305.08768>
33. Różowski, W., Kappé, T., Kozen, D., Schmid, T., Silva, A.: Probabilistic guarded kat modulo bisimilarity: Completeness and complexity. In: *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2023)
34. Selinger, P.: *A Survey of Graphical Languages for Monoidal Categories*, p. 289–355. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-12821-9_4
35. Stein, D., Staton, S.: Compositional semantics for probabilistic programs with exact conditioning. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. pp. 1–13. IEEE (2021)
36. Stein, D., Staton, S.: Probabilistic programming with exact conditions. *Journal of the ACM* (2023)
37. Stein, D., Zanasi, F., Samuelson, R., Piedeleu, R.: Graphical quadratic algebra. *arXiv preprint arXiv:2403.02284* (2024)

- 38. Stone, M.H.: Postulates for the barycentric calculus. *Annali di Matematica Pura ed Applicata* **29**, 25–30 (1949)
- 39. Tolpin, D., van de Meent, J.W., Yang, H., Wood, F.: Design and implementation of probabilistic programming language anglican (2016), <https://arxiv.org/abs/1608.05263>
- 40. Zanasi, F.: Interacting Hopf Algebras-the Theory of Linear Systems. Ph.D. thesis, Ecole normale supérieure de Lyon (2015)