Fixed-Parameter Tractable Inference for Discrete Probabilistic Programs, via String Diagram Algebraisation

Benedikt Peterseim
Universiteit Twente, Enschede, The Netherlands

Milan Lopuhaä-Zwakenberg
Universiteit Twente, Enschede, The Netherlands

Abstract

Discrete probabilistic programs (DPPs) provide a highly expressive formalism for compactly defining arbitrary finite probabilistic models. This expressivity comes at a price: DPP inference is PSPACE-hard. In this work, we show that DPP inference only takes polynomial time for programs that are "structurally simple". More precisely, inference can be performed in polynomial time when the *primal graph* of each function appearing in the probabilistic program has bounded treewidth, and the inverse *acceptance probability* is at most exponential in the size of the probabilistic program. Existing algorithms do not achieve this performance guarantee.

Our method relies on finding suitable decompositions, *algebraisations*, of the string diagrams underlying DPPs, employing existing algorithms for tree decompositions. This is independent of the probabilistic setting of DPPs and has direct applications to many problems, such as evaluating queries on relational databases and cybersecurity risk assessment via attack trees.

2012 ACM Subject Classification Mathematics of computing → Probabilistic inference problems; Theory of computation → Fixed parameter tractability; Theory of computation → Categorical semantics

Keywords and phrases probabilistic programming, string diagrams, treewidth, fixed-parameter tractability, probabilistic inference, parameterised complexity, hypergraph categories

Digital Object Identifier 10.4230/LIPIcs.LICS.2026.76

Related Version *Full version including appendix*: `https://arxiv.org/abs/2604.25321` [27]

Funding This work is partially supported by the European Research Council under grant agreement No 101187945 (RUBICON).

Acknowledgements We thank the anomymous reviewers for their thoughtful, thorough feedback.

1 Introduction

1.1 Discrete probabilistic programs (DPPs)

Probabilistic programming languages are declarative formal languages for specifying probabilistic models. We focus on *discrete* probabilistic programs (DPPs) as introduced in [14], with Boolean variables. Thus we allow exactly the following constructs:

1. let expressions, or *assignments*;
2. the basic Boolean connectives $\wedge, \vee, \neg$;
3. a biased coin flip function `flip`(p), for each $p \in [0,1] \cap \mathbb{Q}$;
4. `observe` expressions, conditioning on an event being true;
5. function definitions.

Extensions to more general finite types and constructs (such as bounded iteration) can be reduced to this Boolean core [14]. An example of a DPP is given in Figure 1. DPPs are highly expressive and include, for example, *fault trees*, which are ubiquitous in reliability

```
test(z) :=
  let t_⊤|⊤ = flip(0.99);
  let t_⊤|⊥ = flip(0.02);
  let t_TP = z ∧ t_⊤|⊤;
  let t_FP = ¬z ∧ t_⊤|⊥;
  let t = t_TP ∨ t_FP;
  t

hasDisease( ) :=
  let x = flip(10^−4);
  let t_1 = test(x);
  let t_2 = test(x);
  let y = t_1 ∧ ¬t_2;
  observe(y);
  x
```

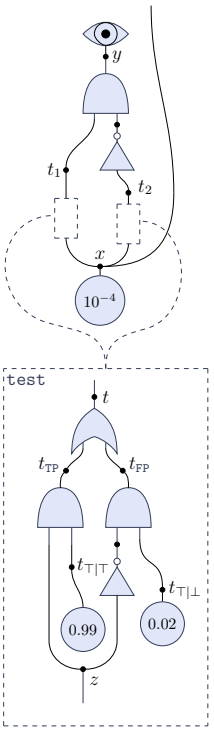


Figure 1 The event that a patient has an infectious disease given one positive and one negative test result, as a discrete probabilistic program `hasDisease` (left) and a hierarchical string diagram (right). The disease occurs in 10^{-4} of the population; the test has a 99% true positive rate (TP) and a 2% false positive rate (FP).

engineering [31, 33], as well as *Bayesian networks* [24, 25] which are used for probabilistic reasoning in many fields including healthcare [23] and cybersecurity [39]. Further applications of discrete probabilistic programming, and its role for probabilistic programming in general, are described in detail in [14].

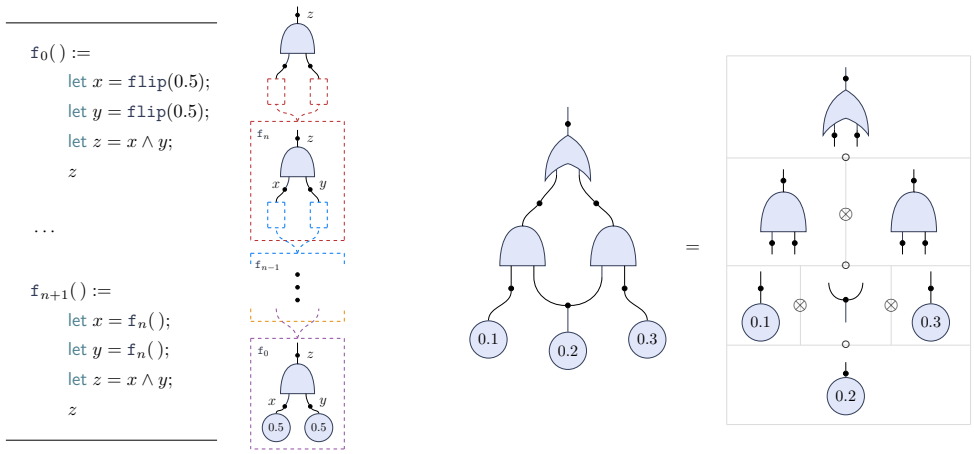
1.2 The Boolean probabilistic inference problem

The main subject of this paper is probabilistic inference: given a DPP with no inputs and a single output, what is the probability that this DPP returns *true*? We cannot hope to efficiently compute this exactly: there exist programs of length $O(n)$ whose probability of returning *true* is 2^{-2^n}, which requires exponentially many digits to express; see Figure 2a. Instead, we look at the approximate problem:

▶ **Problem 1.** *Given a discrete probabilistic program* `f` *with no inputs and a single Boolean output, and a natural number d, compute the probability that* `f` *returns* true*, up to d fractional binary digits.*

Problem 1 is PSPACE-hard [14], even when restricting to deterministic DPPs, showing that this specific hardness result has little to do with probability and instead stems from the expressive power of function definitions. However, without function definitions, probabilistic inference remains NP-hard even when restricted to fault trees [20] and Bayesian networks.

For Bayesian networks, however, inference is known to be *fixed-parameter tractable* [19], i.e. there are inference algorithms with polynomial time complexity given that a certain parameter is bounded. The relevant parameter is the *treewidth* of the so-called moral graph



(a) (b)

Figure 2 (a) Family $(\texttt{f}_n)$ of DPPs. The probability that $\texttt{f}_n$ returns *true* is doubly exponentially small in n. The weighted-model-counting-based inference algorithm of `Dice` requires 2^n fresh variables to compile $\texttt{f}_n$ to a weighted Boolean formula [14, Section 4.3], causing exponential blowup before any weighted model count is performed. (b) Decomposition of a dot diagram via parallel ($\otimes$) and sequential ($\circ$) composition, equivalently the term $\vee\circ(\wedge\otimes\wedge)\circ(\texttt{flip}(0.1)\otimes\texttt{copy}\otimes\texttt{flip}(0.3))\circ\texttt{flip}(0.2)$. The operations $\circ$ and $\otimes$ correspond to matrix multiplication and Kronecker product; see Section 3.

of the Bayesian network, which measures how far away the network is from being "tree-like". Similar results are known for fault trees [20] and the related problem of weighted model counting [6, 5, 9]. However, such results do not exist for DPPs in general.

1.3 Contributions

Our main contribution is a fixed-parameter tractable algorithm for inference in DPPs. The key parameter is the maximal treewidth k of the so-called *primal graph* of the functions in the DPP.

▶ **Main result 2** (Theorem 55 paraphrased). *There exists an algorithm that computes the probability that a DPP returns* true *up to d digits in $O(2^{O(k)}d^2n^3\log^3(p_{\text{acc}}^{-1}))$ time, where n measures the overall size of the DPP,*[1] *and p_{acc} is the* acceptance probability*, the probability that all* `observed` *values are true.*[2]

Existing methods do not achieve these performance guarantees: the state-of-the-art inference algorithm of `Dice` [14] handles DPPs by compiling them into a weighted propositional formula and applying weighted model counting to the result. Through function calls this propositional formula can grow exponentially in size; see Figure 2a. By contrast, we essentially compute the semantics of each called function separately and then combine. Moreover, even when restricting to DPPs corresponding to Bayesian networks or fault trees, our algorithm does not simply reduce to established methods. Instead, we rely on *string diagram algebraisation*.

[1] More precisely, $n = LMN$, where L is the maximum number of distinct functions called within any function, M is the number of functions in the DPP, and N is the maximum number of assignments in any function. In particular, if l is the number of lines in the DPP, then n is in $O(l^3)$, and hence, our algorithm takes polynomial time in the number of lines of the given DPP, for fixed k, d and p_{acc}.

[2] The dependency on p_{acc} can be removed, for example, if the depth of the call graph of the DPP is bounded (or at most poly-logarithmic in n), or if there are only boundedly many `observe`d events.

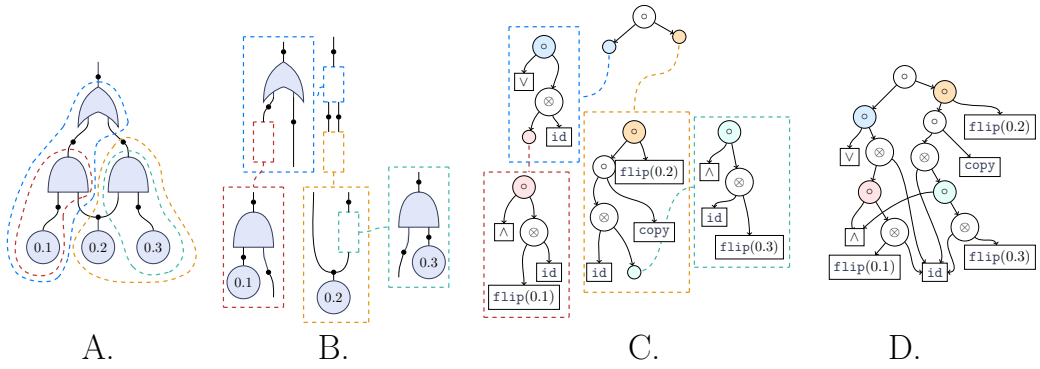


Figure 3 The algorithm of Theorem 32: A. Compute a branch decomposition of the dot diagram. B. Recursively extract sub-diagrams to obtain a hierarchical dot diagram. C. Algebrise each sub-diagram separately as a DAG. D. Compose DAGs via substitution. The resulting term has a *width* of 3 and a DAG-size of 17.

1.4 Idea of proof

Our algorithm operates on the string diagram – or more precisely, *dot diagram* [17] – representing the abstract syntax of a DPP. Dot diagrams can be decomposed into a *term* consisting of atomic parts via parallel and sequential decomposition; see Figure 2. Such a term is called an *algebraisation*, and there are generally many algebraisations of a dot diagram. To perform inference, each subterm is translated into a substochastic matrix, and $\circ$ and $\otimes$ translate to matrix composition and Kronecker product, respectively. A subterm with m inputs and n outputs corresponds to a $2^m \times 2^n$-matrix; thus we need algebraisations with low *width*, where the width is the maximum number of inputs or outputs over all subterms.

Unfortunately, existing approaches to string diagram algebraisation are not designed towards minimising algebraisation width: the Frobenius decomposition of [38] is aimed towards getting a decomposition *fast*. The algorithm used by the Python toolkit *DisCoPy* [7, 36] to compute the semantics of a string diagram also essentially uses an algebraisation algorithm, but only implicitly, and without optimising for low width. A generalisation of graph-theoretic width notions to monoidal categories, *monoidal width*, has been studied in [8], but without exploiting this connection for algorithmic purposes.[3] To address these shortcomings we develop an algorithm that finds algebraisations with low width. Specifically, we prove:

▶ **Main result 3** (Theorem 32 paraphrased). *There is an algorithm that takes as input a string diagram of size n whose primal graph has treewidth k, and outputs an algebraisation of width $O(k)$ and size $O(nk^2)$, in $O(2^{O(k)}n)$ time.*

The idea of the proof is to take existing algorithms for finding a so-called *tree decomposition* of the primal graph, convert it to a *branch decomposition*, and translate this to an algebraisation (see Figure 3). The treewidth of a graph is the minimal width of its tree decompositions. Finding the optimal tree decomposition is NP-hard [1]; instead, we employ approximation algorithms that find a sufficiently good decomposition in $O(2^{O(k)}n)$ time [18].

[3] A more detailed discussion of the similarities and differences between the width of an algebraisation and monoidal width is given in the appendix of the full version of this paper [27]. Summarised briefly, our notion of algebraisation can be understood in terms of the *monoidal decompositions* of [8], but the respective notions of width differ, making it difficult to establish a deeper relation.

1.5 Further applications of algebraisation

Since Theorem 32 works on the level of string diagrams, it has applications beyond DPPs. By taking values in matrices over arbitrary semirings rather than stochastic matrices, our results can be applied to *algebraic model counting* [15], which in turn covers Boolean satisfiability, weighted model counting, and shortest path problems. Our results also apply to other problems that can be phrased in terms of string diagrams, such as evaluating conjunctive queries on relational databases [4], or quantitative cybersecurity risk assessment via attack trees [26]. In a different direction, the term representation that our algorithm computes can also be used for visualising string diagrams in a recursive manner [30].

2 Syntax: dot diagrams

We first define the syntax of our DPPs, in terms of *dot diagrams* [17]. On this level of generality, these are neither necessarily probabilistic or Boolean, but instead defined over an arbitrary *monoidal signature*.

Notations for lists. If X is a set, X^* denotes the set of lists (i.e. finite, ordered sequences) of elements of X. We write lists as tuples $\mathbf{x} = (x_1, \dots, x_n)$ and write $|\mathbf{x}|$ for its length n. The set of all its elements is $\mathsf{set}(\mathbf{x})$.

2.1 Monoidal signatures

Monoidal signatures record a number of *sorts* and *function symbols*. The difference to ordinary (multi-sorted) algebraic signatures is that function symbols may have multiple outputs.

▶ **Definition 4.** *A* monoidal signature *is a pair*

$$\Sigma = (\Sigma_0, \Sigma(\,\cdot\,,\,\cdot\,))$$

consisting of

1. *a set* Σ_0*, the set of* sorts*;*
2. *for all* $\mathbf{S}, \mathbf{T} \in \Sigma_0^*$*, a set* $\Sigma(\mathbf{S}, \mathbf{T})$*.*

The elements f *of* $\Sigma(\mathbf{S}, \mathbf{T})$ *are called* function symbols *with* domains $\mathbf{S}$ *and* codomains $\mathbf{T}$*; this is denoted* $f\colon \mathbf{S} \to \mathbf{T}$*. Irrespective of domains and codomains we also write* $f \in \Sigma$*. A* morphism of monoidal signatures $F\colon \Sigma \to \Sigma'$ *is a map* $F\colon \Sigma_0 \to \Sigma'_0$*, together with a family of maps,*

$$\Sigma(\mathbf{S}, \mathbf{T}) \to \Sigma'(F(\mathbf{S}), F(\mathbf{T})),$$

each component of which we also denote by F*.*

▶ **Example 5.** The *signature of discrete probabilistic programs* is the signature $\mathsf{BoolStochSig}$, where

1. $\mathsf{BoolStochSig}$ has a single sort, denoted by $\mathbb{B}$, so $\mathsf{BoolStochSig}_0 = \{\mathbb{B}\}$.
2. The function symbols of $\mathsf{BoolStochSig}$ are ($p \in [0,1] \cap \mathbb{Q}$):

$$\wedge\colon (\mathbb{B},\mathbb{B}) \to (\mathbb{B}), \qquad \vee\colon (\mathbb{B},\mathbb{B}) \to (\mathbb{B}),$$
$$\neg\colon (\mathbb{B}) \to (\mathbb{B}), \qquad \texttt{observe}\colon (\mathbb{B}) \to (\,),$$
$$\texttt{flip}(p)\colon (\,) \to (\mathbb{B}).$$

2.2 Variable sets and assignments

Having defined our set of function symbols, we now define variables and variable assignments.

▶ **Definition 6** (Variable set). *Let Σ be a monoidal signature. A* variable set *over Σ is a pair (X, sort_X) consisting of a finite set X and $\mathsf{sort}_X \colon X \to \Sigma_0$. We write sort instead of sort_X when X is clear, and we write X instead of (X, sort_X) for the variable set. A* morphism of variable sets $X \to Y$ *is a map $\phi \colon X \to Y$ such that $\mathsf{sort}_Y \circ \phi = \mathsf{sort}_X$.*

▶ **Definition 7** (Assignment). *Let Σ be a monoidal signature and X be a variable set over Σ. An* assignment *over X is a triple $(\mathbf{y}, f, \mathbf{x})$, where $\mathbf{x}, \mathbf{y} \in X^*$ and $f \in \Sigma(\mathsf{sort}_*(\mathbf{x}), \mathsf{sort}_*(\mathbf{y}))$. $\mathbf{x}$ and $\mathbf{y}$ are the* input *and* output *variables, respectively, and we will write*

$$\mathsf{let}\ y_1, \ldots, y_n = f(x_1, \ldots, x_m)$$

to denote the assignment $((y_1, \ldots, y_n), f, (x_1, \ldots, x_m))$. This will be rendered as "$f(x_1, \ldots, x_m)$" when $n = 0$, and "$\mathsf{let}\ y_1, \ldots, y_n = f$" when $m = 0$. Moreover, we write $\mathsf{A}_\Sigma(X)$ for the set of all assignments over the variable set X.

The pushforward of a morphism of variable sets $\phi \colon X \to Y$ is the map

$$\mathsf{A}_\Sigma(\phi) \colon\ \mathsf{A}_\Sigma(X) \to \mathsf{A}_\Sigma(Y), \quad (\mathbf{y}, f, \mathbf{x}) \mapsto (\phi_*(\mathbf{y}), f, \phi_*(\mathbf{x})),$$

where the $\phi_* \colon X^* \to Y^*$ is the pushforward of lists.

▶ **Example 8.** Let $\Sigma := \mathsf{BoolSig}$, let $X := \{x, y, z\}$ with $\mathsf{sort}_X(\,\cdot\,) := \mathbb{B}$. Then $((z), \wedge, (x, y))$ is an assignment over X, representing the "line of code" $\mathsf{let}\ z = x \wedge y$.

Note that we do not make any assumptions about how often a variable may appear on either side of the assignment. We will see how a variable occurring multiple times on the left hand side of the assignment can be interpreted, once we have defined the notion of a *dot diagram* over Σ.

2.3 Dot diagrams

After function symbols, variables and assignments, we can now define programs, which for us are *dot diagrams*. These were introduced in [17] as hypergraphs whose vertices are called *boxes* and whose hyperedges are called *dots*. Our definition looks more directly like one of "straight-line programs" but is equivalent; the original definition corresponds to passing to the "data flow graph".

▶ **Definition 9** (Dot diagram). *Let Σ be a monoidal signature. A* dot diagram $\mathtt{f}$ *over Σ or Σ-diagram is a tuple $(X_\mathtt{f}, A_\mathtt{f}, \mathsf{in}_\mathtt{f}, \mathsf{out}_\mathtt{f})$ where*

1. *$X_\mathtt{f}$ is a variable set;*
2. *$A_\mathtt{f} \in \mathsf{A}_\Sigma(X_\mathtt{f})^*$ is a list of assignments; and*
3. *$\mathsf{in}_\mathtt{f}, \mathsf{out}_\mathtt{f} \in X_\mathtt{f}^*$ are lists of* input *and* output *variables;*
4. *for all $x \in X_\mathtt{f}$, x is either an in- or output variable, or x appears in some assignment from $A_\mathtt{f}$, i.e. there exists some $(\mathbf{y}, g, \mathbf{x}) \in A_\mathtt{f}$ such that $x \in \mathsf{set}(\mathbf{y})$ or $x \in \mathsf{set}(\mathbf{x})$.*

In the following, we will write

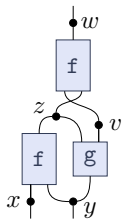
$$\begin{aligned}&\mathtt{f}(x_1, \ldots, x_n) :=\\ &\qquad a_1;\\ &\qquad \ldots\\ &\qquad a_k;\\ &\qquad y_1, \ldots, y_m\end{aligned}$$

instead of $\mathtt{f} := (X_\mathtt{f}, (a_1, \ldots, a_k), (x_1, \ldots, x_n), (y_1, \ldots, y_m))$ to define a dot diagram $\mathtt{f}$. Dot diagrams over Σ form a monoidal signature which we denote by $\mathrm{Dot}(\Sigma)$.

Two dot diagrams $\mathtt{f}, \mathtt{g} \in \mathrm{Dot}(\Sigma)$ *are* equivalent*, written* $\mathtt{f} \equiv \mathtt{g}$*, if there exists an isomorphism of variable sets* $\phi\colon X_{\mathtt{f}} \to X_{\mathtt{g}}$ *such that that* $\phi_*(\mathsf{in}_{\mathtt{f}}) = \mathsf{in}_{\mathtt{g}}$*,* $\phi_*(\mathsf{out}_{\mathtt{f}}) = \mathsf{out}_{\mathtt{g}}$*, and* $\mathsf{set}(\mathsf{A}_\Sigma(\phi)(A_{\mathtt{f}})) = \mathsf{set}(A_{\mathtt{g}})$*. We write* $\underline{\mathtt{f}}$ *for the equivalence class of* $\mathtt{f}$*, and* $\underline{\mathrm{Dot}}(\Sigma)$ *for the set of* Σ*-diagrams modulo equivalence.*

▶ **Example 10.** Let Σ be the monoidal signature with $\Sigma_0 = \{\mathbb{S}\}$ and two function symbols $\mathtt{f}\colon \mathbb{S} \otimes \mathbb{S} \to \mathbb{S}$ and $\mathtt{g}\colon \mathbb{S} \to \mathbb{S} \otimes \mathbb{S}$. Let $X := \{x, y, z, v, w\}$ be the variable set over Σ with $\mathsf{sort}_X(\,\cdot\,) := \mathbb{S}$. Then below (left) is an example of a dot diagram $\mathtt{h}$ over Σ with variable set X, two input variables x, y, one output variable w, and three assignments. On the right is its graphical representation where every assignment is a *box* and every variable is a *dot*.

$\mathtt{h}(x, y) :=$
  let $z = \mathtt{f}(x, y)$;
  let $z, v = \mathtt{g}(y)$;
  let $w = \mathtt{f}(v, z)$;
  w

$\mathtt{h} =$ 

▶ **Example 11.** Up to "syntactic sugar", the function `test` from Figure 1 is a BoolStochSig-diagram. Its set of variables is $X_{\mathtt{test}} = \{z, \bar{z}, t_{\top|\top}, t_{\top|\bot}, t_{\mathsf{TP}}, t_{\mathsf{FP}}, t\}$ and $\mathsf{sort}_X(\,\cdot\,) := \mathbb{B}$. To "de-sugar" it into a dot diagram, we need to write the assignment let $t_{\mathsf{FP}} = \neg z \wedge t_{\top|\bot}$ as let $\bar{z} = \neg z$; let $t_{\mathsf{FP}} = \bar{z} \wedge t_{\top|\bot}$. The resulting dot diagram has one input variable z and one output variable t.

On the other hand, the DPP `hasDisease` from Figure 1 is not simply a dot diagram, since it consists of two functions where one is called by the other.

2.4 Hierarchical dot diagrams

Hierarchical dot diagrams allow for "function definitions": the function symbols appearing within any assignments may themselves be defined as a dot diagram. Hierarchical dot diagrams are our generalisation of DPPs.

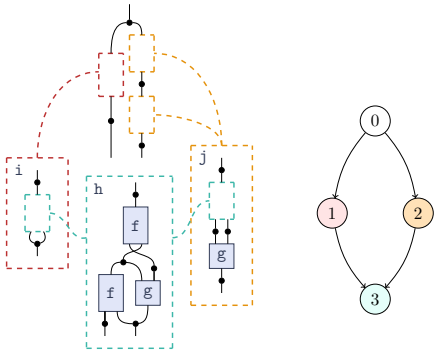
▶ **Definition 12.** *Let* Σ *be a monoidal signature. A* hierarchical dot diagram *over* Σ *(short: hierarchical* Σ*-diagram) is a tuple,*

$$\mathtt{f} = (V(\mathtt{f}), E(\mathtt{f}), R_{\mathtt{f}}, (\Sigma_v)_{v \in V(\mathtt{f})}, (\mathtt{f}_v)_{v \in V(\mathtt{f})}, \mathrm{defn}_{\mathtt{f}}),$$

such that

1. $(V(\mathtt{f}), E(\mathtt{f}))$ *is a connected, rooted directed acyclic graph with root* $R_{\mathtt{f}}$*, called the* call graph *of* $\mathtt{f}$*.*
2. *Each* Σ_v *is a monoidal signature with the same set of sorts* Σ_0 *as* Σ*, and* $\mathtt{f}_v$ *is a dot diagram over* $\Sigma \sqcup_{\Sigma_0} \Sigma_v$*, i.e. the signature whose set of sorts is* Σ_0 *and whose set of function symbols is given by the disjoint union of those of* Σ *and* Σ_v*.*
3. $\mathrm{defn}_{\mathtt{f},v}\colon \Sigma_v \to \mathrm{ch}(v)$ *is a bijection from the set of function symbols in* Σ_v *to the set of children of* v*, such that the map* $g \mapsto \mathtt{f}_{\mathrm{defn}_{\mathtt{f},v}(g)}$ *is a morphism of signatures* $\Sigma_v \to \bigsqcup_{u \in \mathrm{ch}(v)} \mathrm{Dot}(\Sigma \sqcup_{\Sigma_0} \Sigma_u)$*.*
4. *The number of children of any vertex* $v \in V(\mathtt{f})$ *(equivalently, the number of function symbols in* Σ_v*) is at most* $|A_{\mathtt{f}_v}|$*.*

A Σ*-diagram* $\mathtt{g}$ appears in *a hierarchical* Σ*-diagram* $\mathtt{f}$ *if there exists some* $v \in V(\mathtt{f})$ *such that* $\mathtt{f}_v = \mathtt{g}$*.*



(a) (b)

Figure 4 (a) Example hierarchical Σ-diagram k described in Example 13; (b) the call graph of k.

Condition (4) is only necessary for more convenient complexity bounds, ensuring that hierarchical dot diagrams do not contain too many "unused" function symbols.

▶ **Example 13.** Let Σ and h be given as in Example 10. Figure 4 shows an example k of a hierarchical dot diagram over Σ and its call graph. The vertices of the call graph are $V(\mathtt{k}) = \{0, 1, 2, 3\}$ and its root is $R_{\mathtt{k}} = 0$. The signatures at each node are

$$\Sigma_0 = \{\mathtt{i} : \mathbb{S} \to \mathbb{S}, \mathtt{j} : \mathbb{S} \to \mathbb{S}\},$$
$$\Sigma_1 = \Sigma_2 = \{\mathtt{h} \otimes \mathtt{h} : \mathbb{S} \to \mathbb{S}\}, \quad \Sigma_3 = \varnothing.$$

The dot diagram at the vertex 3 is $\mathtt{k}_3 = \mathtt{h}$; the dot diagrams at the other vertices are given as shown in Figure 4. Finally, the functions $\mathrm{defn}_{\mathtt{k},v}$ are given by

$$\mathrm{defn}_{\mathtt{k},0}(\mathtt{i}) = 1, \qquad \mathrm{defn}_{\mathtt{k},0}(\mathtt{j}) = 2,$$
$$\mathrm{defn}_{\mathtt{k},1}(\mathtt{h}) = 3, \qquad \mathrm{defn}_{\mathtt{k},2}(\mathtt{h}) = 3.$$

Intuitively, the nodes in the call graph represent subroutines in k, and putting $\mathrm{defn}_{\mathtt{k},1}(\mathtt{h}) = 3$ means that that the function symbol h called in subroutine 1 is defined by subroutine 3.

The discrete probabilistic programs from the introduction, such as the program hasDisease from Figure 1, can also be understood as hierarchical dot diagrams, and this is how we *define* discrete probabilistic programs.

▶ **Definition 14** (DPP). *A* discrete probabilistic program (DPP) *is a hierarchical dot diagram over the monoidal signature* BoolStochSig.

3 Semantics: hypergraph categories

Dot diagrams and their hierarchical variant are purely syntactic notions. We now discuss their semantics. For DPPs these will be (sub)stochastic matrices; in the general case, these are morphisms in a *hypergraph category*. For the subsequent discussion, we assume some familiarity with basic category theory, and monoidal categories in particular.

3.1 Strict hypergraph categories

We now define strict hypergraph categories, which are strict monoidal categories with additional structure to model structural operations in DPPs and dot diagrams in general; see [10] for a general discussion of hypergraph categories and their history.

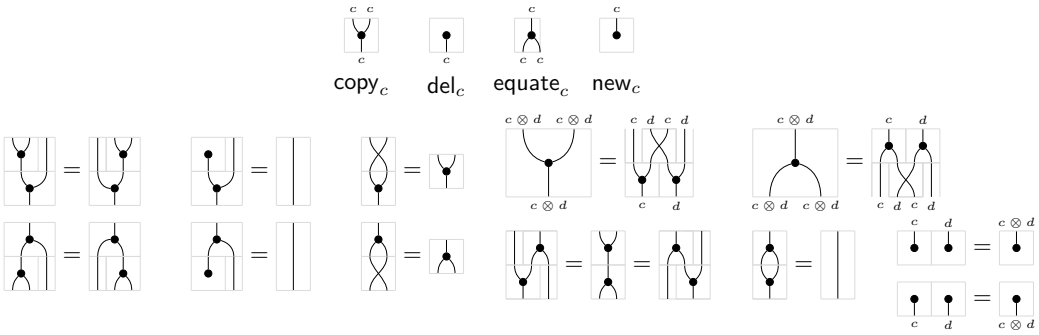


Figure 5 The axioms of strict hypergraph categories. We use standard string diagram notation for symmetric monoidal categories, drawing light grey boxes around morphisms to avoid confusion with our representation of dot diagrams.

▶ **Definition 15.** *A* strict hypergraph category *is a strict symmetric monoidal category C, together with four families of morphisms*

$$\mathsf{copy}_c \colon c \to c \otimes c, \quad \mathsf{del}_c \colon c \to I_C,$$
$$\mathsf{equate}_c \colon c \otimes c \to c, \quad \mathsf{new}_c \colon I_C \to c,$$

where c ranges over the objects of C and I_C is the tensor unit. These morphisms are required to satisfy the equations shown in Figure 5. A strict hypergraph functor *is a strict symmetric monoidal functor that preserves* $\mathsf{copy}_\bullet$, $\mathsf{del}_\bullet$, $\mathsf{equate}_\bullet$, *and* $\mathsf{new}_\bullet$.

3.2 Matrix hypergraph categories

In our general framework, we will define the semantics of dot diagrams as a morphism of strict hypergraph categories. However, we are particularly interested in *matrix hypergraph categories*, as these include the substochastic matrices that will be the semantics of DPPs. We take the slightly more general viewpoint of matrices over arbitrary (commutative, unital) semirings, as these also cover further applications such as cybersecurity risk metrics in attack trees [26].

▶ **Definition 16.** *A semiring is a set R with two binary operations $+$ and $\cdot$, and two elements 0 and 1, such that $(R, +, 0)$ and $(R, \cdot, 1)$ are commutative unital monoids, and $\cdot$ distributes over $+$.*

▶ **Definition 17.** *Let R be a semiring. We define the hypergraph category Mat_R as having natural numbers as objects, and $m \times n$-matrices over R as morphisms $n \to m$. The composition of morphisms in Mat_R is given by matrix multiplication; the tensor product is given on objects by the product of natural numbers, $n \otimes m := nm$, and on morphisms by the Kronecker product of matrices. The braiding is defined on generators as,*

$$\mathsf{swap}_{n,m} \cdot (e_i \otimes e_j) := e_j \otimes e_i,$$

for all $i \in \{1, \dots n\}$, $j \in \{1, \dots, m\}$, where e_i is the i-th standard basis vector. Finally, the hypergraph category structure is given by,

$$\mathsf{copy}_n \cdot e_i := e_i \otimes e_i, \quad \mathsf{del}_n \cdot e_i := (1)$$
$$\mathsf{equate}_n \cdot (e_i \otimes e_j) := \delta_{ij} e_i, \quad \mathsf{new}_n \cdot e_1 := \mathbf{1}_n,$$

where $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ otherwise, and $\mathbf{1}_n \in R^n$ denotes the vector all of whose entries are 1. The axioms of hypergraph categories can easily be checked by a direct calculation.

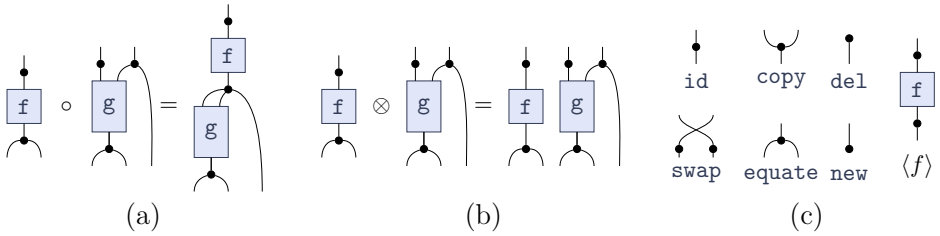


Figure 6 (a) Composition and (b) tensor product of dot diagrams; (c) special dot diagrams.

An important sub-hypergraph category of $\mathsf{Mat}_{\mathbb{Q}}$ is the following, which will be the semantic target category for DPPs.

▶ **Definition 18** (BoolSubStoch). *A substochastic matrix is a real matrix all of whose columns sum to a value of at most* 1*. The hypergraph category* BoolSubStoch *has natural numbers of the form* 2^n *as objects (*$n \in \mathbb{N}$*), and substochastic* $2^m \times 2^n$*-matrices as morphisms* $2^n \to 2^m$*. The symmetric monoidal and hypergraph structures are given in the same way as for* $\mathsf{Mat}_{\mathbb{Q}}$*.*

The idea is that the entry ij of a substochastic matrix records the probability that a DPP returns the Boolean vector $\mathsf{bin}(i)$, the binary digits of i, *and* that all `observe`d events hold, on the input $\mathsf{bin}(j)$. If instead we want `observe` expressions to properly *condition* on an event, we also need to identify two substochastic matrices if they differ by a positive factor; see [34, Section 6.2-6.3] for a thorough discussion of the denotational semantics of discrete probabilistic programs, as well as [35] for a general treatment of the categorical semantics of probabilistic programs.

3.3 The hypergraph category of dot diagrams

Dot diagrams (up to equivalence) themselves also form a strict hypergraph category $\underline{\mathsf{Dot}}(\Sigma)$; in fact, the *free* strict hypergraph category over a monoidal signature Σ (see Figure 6):

- Its objects are given by lists of sorts from Σ;
- its morphisms are (equivalence classes of) dot diagrams;
- The (sequential) composite of two dot diagrams `f`, `g` is given by gluing the output variables of `f` to the input variables of `g`, merging the corresponding dots;
- the tensor product is given by disjoint union;
- `copy`, `del`, `equate` `new` are as in Figure 6.

Precise definitions of the composition and tensor product of dot diagrams are given in the appendix of the full version of this paper [27]. They are equivalent to those of [17], which represent dot diagrams as cospans of labelled hypergraphs. We also introduce special dot diagrams for `id`, `swap`, and $\langle f \rangle$ for each $f \in \Sigma$, as in Figure 6. Note that, strictly speaking, these are equivalence classes of dot diagrams.

3.4 Interpretations and semantics

We can now define the semantics of dot diagrams as morphisms in strict hypergraph categories. We first have to define how to *interpret* function symbols.

▶ **Definition 19** (Interpretation). *Let* Σ *be a monoidal signature and let* C *be a hypergraph category. Then* C *defines a monoidal signature whose sorts are the objects of* C *and whose function symbols* $\mathbf{S} \to \mathbf{T}$ *are given by all morphisms* $S_1 \otimes \cdots \otimes S_m \to T_1 \otimes \cdots \otimes T_n$ *in* C*. An* interpretation *of* Σ *in* C *is a morphism of monoidal signatures* $\mathcal{I} \colon \Sigma \to C$*.*

▶ **Example 20.** The *substochastic matrix interpretation* is the interpretation $\mathcal{S}\colon \mathsf{BoolStochSig} \to \mathsf{BoolSubStoch}$ of the signature of DPPs in the hypergraph category of substochastic matrices, which is defined as follows: the sole sort $\mathbb{B}$ of BoolStochSig is mapped to the object 2^1 of BoolSubStoch. On function symbols $\mathcal{S}$ is given by:

$$\mathcal{S}(\wedge) := \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad \mathcal{S}(\vee) := \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \end{pmatrix}, \quad \mathcal{S}(\neg) := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$
$$\mathcal{S}(\texttt{flip}(p)) := \begin{pmatrix} 1-p \\ p \end{pmatrix}, \qquad \mathcal{S}(\texttt{observe}) := \begin{pmatrix} 0 & 1 \end{pmatrix}.$$

With the interpretation of function symbols in place, the semantics of dot diagrams are the unique extension of the interpretation to dot diagrams. The following result makes this precise. The proof is given in [17, Theorem 4.6], by reduction to the corresponding statement for traced monoidal categories [16, Theorem 5.5.10].[4] For a more recent, published reference, see [3, Corollary 4.2].[5]

▶ **Theorem 21.** *Let Σ be a monoidal signature, let C be a strict hypergraph category, and let $\mathcal{I}\colon \Sigma \to C$ be an interpretation. Then there exists a unique strict hypergraph functor*

$$[\![\cdot]\!]_{\mathcal{I}}\colon \underline{\mathrm{Dot}}(\Sigma) \to C$$

such that for all function symbols f in Σ, we have $[\![\langle f \rangle]\!]_{\mathcal{I}} = \mathcal{I}(f)$, where $\langle f \rangle$ is the dot diagram (modulo equivalence) associated to f. In other words, $\underline{\mathrm{Dot}}(\Sigma)$ is the free strict hypergraph category on Σ.

▶ **Definition 22** (Semantics). *Let Σ be a monoidal signature, let C be a hypergraph category, and let $\mathcal{I}\colon \Sigma \to C$ be an interpretation. Moreover, let $\mathtt{f}$ be a dot diagram. The semantics of $\mathtt{f}$ under $\mathcal{I}$ is $[\![\underline{\mathtt{f}}]\!]_{\mathcal{I}}$, where $[\![\cdot]\!]_{\mathcal{I}}$ is the unique strict hypergraph functor from Theorem 21.*

Hierarchical dot diagrams will receive semantics in Definition 50.

4 Algebraisations

From the previous section it follows (ignoring hierarchical structure for the moment) that the semantics of a DPP with 0 inputs and 1 output is a $2^1 \times 2^0$-substochastic matrix. In Section 6.2, we will relate this to the acceptance probability of Problem 1. For now, we will turn to the problem of *computing* this matrix. For this, we define *hypergraph terms*, which are (non-unique) algebraic representations of dot diagrams. The process of finding a "good" hypergraph term is called *algebraisation*, which is the topic of Section 5.

4.1 Hypergraph terms

Hypergraph terms are formal composites and tensor products of function symbols and constant symbols representing the special morphisms in a hypergraph category.

[4] The characterisation of free traced monoidal categories was stated earlier in [13], but the proof outline given there seems to merely state *what* there is to prove, without saying *how*.

[5] Note, however, that according to [11, Remark 4.3], the proof given there – which is essentially the same as the one in [40] – contains two substantial gaps. Nevertheless, the idea of proof is substantial, relying on a Frobenius-type decomposition similar to the one we will also use in Section 5.2; hence these gaps can presumably be closed using suitable inductive arguments similar to the ones in [11].

▶ **Definition 23.** *Let Σ be a monoidal signature. The monoidal signature of* hypergraph terms *over Σ is given by*

$$\mathrm{Term}(\Sigma) := (\mathrm{Term}(\Sigma)_0, \mathrm{Term}(\Sigma)(\,\cdot\,,\,\cdot\,)),$$

where the set of sorts $\mathrm{Term}(\Sigma)_0 := \Sigma_0$ is the same as that of Σ and $\mathrm{Term}(\Sigma)(\,\cdot\,,\,\cdot\,)$ is defined inductively as follows:

1. *if $f \in \Sigma(\mathbf{S}, \mathbf{T})$, then $f \in \mathrm{Term}(\Sigma)(\mathbf{S}, \mathbf{T})$;*
2. *if $\mathbf{S}, \mathbf{T} \in \Sigma_0^*$, then*

$$\begin{aligned} \mathrm{id}_{\mathbf{T}} &\in \mathrm{Term}(\Sigma)(\mathbf{T}, \mathbf{T}), & \mathrm{swap}_{\mathbf{S},\mathbf{T}} &\in \mathrm{Term}(\Sigma)(\mathbf{ST}, \mathbf{TS}),\\ \mathrm{copy}_{\mathbf{T}} &\in \mathrm{Term}(\Sigma)(\mathbf{T}, \mathbf{TT}), & \mathrm{del}_{\mathbf{T}} &\in \mathrm{Term}(\Sigma)(\mathbf{T}, ()),\\ \mathrm{equate}_{\mathbf{T}} &\in \mathrm{Term}(\Sigma)(\mathbf{TT}, \mathbf{T}), & \mathrm{new}_{\mathbf{T}} &\in \mathrm{Term}(\Sigma)((), \mathbf{T}); \end{aligned}$$

3. *if $s \in \mathrm{Term}(\Sigma)(\mathbf{T}, \mathbf{U})$ and $t \in \mathrm{Term}(\Sigma)(\mathbf{S}, \mathbf{T})$, then*

$$s \circ t \in \mathrm{Term}(\Sigma)(\mathbf{S}, \mathbf{U}).$$

4. *if $s \in \mathrm{Term}(\Sigma)(\mathbf{S}, \mathbf{T})$ and $t \in \mathrm{Term}(\Sigma)(\mathbf{U}, \mathbf{V})$, then*

$$s \otimes t \in \mathrm{Term}(\Sigma)(\mathbf{SU}, \mathbf{TV});$$

A hypergraph term *is any function symbol $t \in \mathrm{Term}(\Sigma)$ of this signature.*

We write $s \subseteq t$ when s is a sub-term of t; this relationship is defined in the standard inductive way. The same applies to the substitution *of a finite family of hypergraph terms $(s_i)_{i\in I}$ for a family of function symbols $(g_i)_{i\in I}$ (from Σ) in a hypergraph term t, written $t[g_i \mapsto s_i]$.*

For computational purposes, we assume that hypergraph terms are represented as labelled directed acyclic graphs (DAGs) with maximal sharing, i.e. the nodes of the DAG underlying a hypergraph term represent unique sub-terms of t; see Figure 7, discussed below.

▶ **Example 24.** Let Σ be the monoidal signature with a single sort $\mathbb{S}$ and two function symbols $\mathtt{f} \colon \mathbb{S} \otimes \mathbb{S} \to \mathbb{S}$ and $\mathtt{g} \colon \mathbb{S} \otimes \mathbb{S} \to \mathbb{S}$. Then

$$t = \mathtt{f} \circ (\mathrm{equate}_{(\mathbb{S})} \otimes \mathrm{id}_{(\mathbb{S})}) \circ (\mathtt{f} \otimes \mathtt{g}) \circ (\mathrm{id}_{(\mathbb{S})} \otimes \mathrm{copy}_{(\mathbb{S})}) \in \mathrm{Term}(\Sigma)(\mathbb{S}^3, \mathbb{S})$$

and $s := \mathtt{f} \circ \mathrm{swap}_{(\mathbb{S},\mathbb{S})} \in \mathrm{Term}(\Sigma)(\mathbb{S}^2, \mathbb{S})$ are hypergraph terms over Σ (see Figure 7). Then

$$\begin{aligned} t[\mathtt{f} \mapsto s] = {} & (\mathtt{f} \circ \mathrm{swap}_{(\mathbb{S},\mathbb{S})}) \circ (\mathrm{equate}_{(\mathbb{S})} \otimes \mathrm{id}_{(\mathbb{S})})\\ & \circ ((\mathtt{f} \circ \mathrm{swap}_{(\mathbb{S},\mathbb{S})}) \otimes \mathtt{g}) \circ (\mathrm{id}_{(\mathbb{S})} \otimes \mathrm{copy}_{(\mathbb{S})}), \end{aligned}$$

as shown in Figure 7 (b).

4.2 Semantics of hypergraph terms

The semantics of hypergraph terms are defined in the obvious inductive way.

▶ **Definition 25.** *Let Σ be a monoidal signature, let C be a strict hypergraph category, and let $\mathcal{I} \colon \Sigma \to C$ be an interpretation. The* semantics *$[\![t]\!]_{\mathcal{I}}$ of a hypergraph term t under $\mathcal{I}$ is defined recursively as follows:*

$$\begin{aligned} [\![f]\!]_{\mathcal{I}} &:= \mathcal{I}(f), & [\![\mathrm{id}_T]\!]_{\mathcal{I}} &:= \mathsf{id}_{\mathcal{I}(T)}, & [\![s \circ t]\!]_{\mathcal{I}} &:= \mathcal{I}(s) \circ \mathcal{I}(t)\\ [\![\mathrm{swap}_{S,T}]\!]_{\mathcal{I}} &:= \mathsf{swap}_{\mathcal{I}(S),\mathcal{I}(T)}, & [\![s \otimes t]\!]_{\mathcal{I}} &:= \mathcal{I}(s) \otimes \mathcal{I}(t),\\ [\![\mathrm{copy}_T]\!]_{\mathcal{I}} &:= \mathsf{copy}_{\mathcal{I}(T)}, & [\![\mathrm{del}_T]\!]_{\mathcal{I}} &:= \mathsf{del}_{\mathcal{I}(T)},\\ [\![\mathrm{equate}_T]\!]_{\mathcal{I}} &:= \mathsf{equate}_{\mathcal{I}(T)}, & [\![\mathrm{new}_T]\!]_{\mathcal{I}} &:= \mathsf{new}_{\mathcal{I}(T)}. \end{aligned}$$

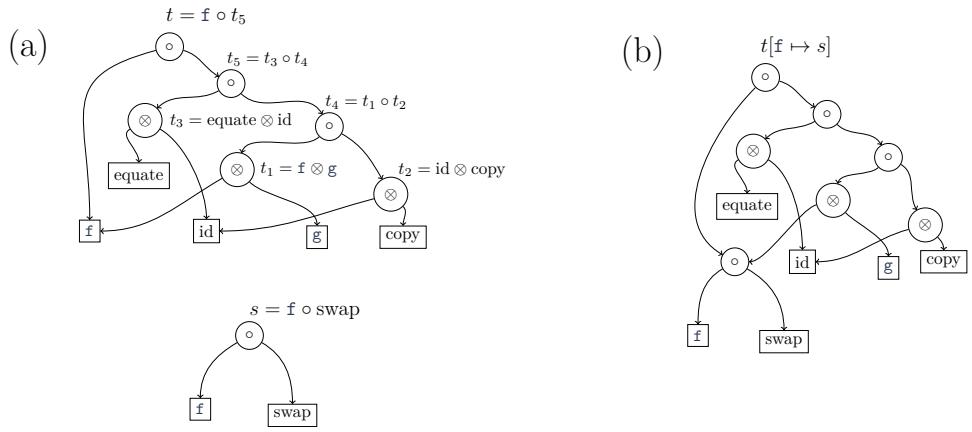


Figure 7 (a) DAGs representing the terms t and s from Example 24; (b) the substitution of s for $\mathtt{f}$ in t.

Two strict hypergraph terms s, t are equivalent *if for every strict hypergraph category C and every interpretation $\mathcal{I}\colon\ \Sigma \to C$, we have $[\![s]\!]_{\mathcal{I}} = [\![t]\!]_{\mathcal{I}}$. We write $\underline{t}$ for the equivalence class of a hypergraph term t under this equivalence relation. Finally, we denote the set of hypergraph terms $\mathbf{S} \to \mathbf{T}$ modulo equivalence by $\underline{\mathrm{Term}}(\Sigma)(\mathbf{S}, \mathbf{T})$, and accordingly, we write $\underline{\mathrm{Term}}(\Sigma)$ for the resulting hypergraph category.*

As for $\underline{\mathrm{Dot}}(\Sigma)$, the hypergraph category $\underline{\mathrm{Term}}(\Sigma)$ is the free strict hypergraph category on Σ. In more detail, this means the following.

▶ **Theorem 26.** *Let Σ be a signature, let C be a strict hypergraph category, and let $\mathcal{I} : \Sigma \to C$ be an interpretation. Then*

$$[\![\cdot]\!]_{\mathcal{I}}\colon\ \underline{\mathrm{Term}}(\Sigma) \to C$$

is the unique strict hypergraph functor such that for all function symbols $f \in \Sigma$, we have $[\![f]\!]_{\mathcal{I}} = \mathcal{I}(f)$.

4.3 From terms to dot diagrams

By the essential uniqueness of the free strict hypergraph category, we have an isomorphism of hypergraph categories $\underline{\mathrm{Dot}}(\Sigma) \cong \underline{\mathrm{Term}}(\Sigma)$. The isomorphism in the direction from hypergraph terms to dot diagrams can easily be described explicitly, as follows.

▶ **Definition 27.** *Let Σ be a monoidal signature. Let $\mathcal{I}_{\mathsf{dot}}\colon \Sigma \to \underline{\mathrm{Dot}}(\Sigma)$ be the interpretation that is the identity on function symbols and assigns to each function symbol f in Σ its corresponding dot diagram $\langle f \rangle$. The* associated dot diagram *(modulo equivalence) of a hypergraph term t is,*

$$\underline{\mathsf{dot}}(t) := [\![t]\!]_{\mathcal{I}_{\mathsf{dot}}}$$

i.e. its semantics in $\underline{\mathrm{Dot}}(\Sigma)$ under $\mathcal{I}_{\mathsf{dot}}$.

4.4 Algebraisations of dot diagrams

Going in the other direction, from dot diagrams to hypergraph terms, is what we call *algebraisation*; the resulting term that equivalently presents a morphism in the free hypergraph category is called *an* algebraisation.

▶ **Definition 28.** *An* algebraisation *of a dot diagram* $\mathtt{f}$ *is a hypergraph term* t *such that* $\underline{\mathsf{dot}}(t) = \underline{\mathtt{f}}$.

4.5 Quantifying the complexity of terms

We consider two ways to quantify the complexity of a hypergraph term t. The first one, the *DAG-size*, corresponds to the size of the directed acyclic graph representing t. The second one, the *width* of a hypergraph term, corresponds to the largest number of in- and outputs of any intermediate result reached when evaluating t bottom-up. The width of t can be used, for example, to bound the time complexity of evaluating the semantics of t in a matrix category, where composition is given by matrix multiplication and the tensor product is given by the Kronecker product.

▶ **Definition 29.** *Let* Σ *be a signature and let* $t : (S_1, \dots, S_n) \to (T_1, \dots, T_n)$ *be a hypergraph term over* Σ. *The* DAG-size $|t|$ *of* t *is the number of distinct sub-terms of* t. *The* width *of* t *is the maximum,*

$$\mathsf{width}(t) := \max_{s \subseteq t} |\mathrm{dom}(s)| + |\mathrm{codom}(s)|,$$

over the number of inputs plus the number of outputs of sub-terms of t.

Note that $\mathsf{width}(t)$ is *not* easily read from the DAG representations of Figure 7: one also needs to take the number of inputs and outputs at each vertex into account.

5 An algebraisation algorithm

The goal of this section is to describe an algorithm that converts a dot diagram over a monoidal signature into a hypergraph term of low width and small DAG-size. The main high-level steps of this algorithm are shown in Figure 3. The width and DAG-size of the algebraisation that our algorithm finds depend on the *treewidth* of the so-called *primal graph* of $\mathtt{f}$. Subsequently, a "graph" is always a finite, undirected graph which may contain loops or multiple edges.

▶ **Definition 30** (Primal graph)**.** *Let* $\mathtt{f}$ *be a* Σ*-diagram over a monoidal signature* Σ. *The* primal graph $\mathtt{f}'$ *of* $\mathtt{f}$ *is the graph whose set of vertices is the set* $X_{\mathtt{f}}$ *of variables of* $\mathtt{f}$*, with exactly one edge between any two distinct variables that both appear either a) in some common assignment, b) or as inputs, c) or as outputs.*

The following definition is standard; see [2] for an exposition to tree decompositions and treewidth, including their history and applications.

▶ **Definition 31.** *Let* G *be a graph. A* tree decomposition *of* G *is a tree* $\mathcal{T}$ *together with a family* $(X_t)_{t \in V(\mathcal{T})}$ *of subsets* $X_t \subseteq V(G)$*, called* bags*, such that:*

1. *The bags of* $\mathcal{T}$ *cover* G*:*

$$\bigcup_{t \in V(\mathcal{T})} X_t = G,$$

2. *For all adjacent* $v, w \in V(G)$*, then there exists some* $t \in \mathcal{T}$ *such that* $v, w \in X_t$.
3. *For all* $v \in V(G)$*, the set* $\{t \in V(\mathcal{T}) \mid v \in X_t\}$ *induces a connected subtree of* $\mathcal{T}$.

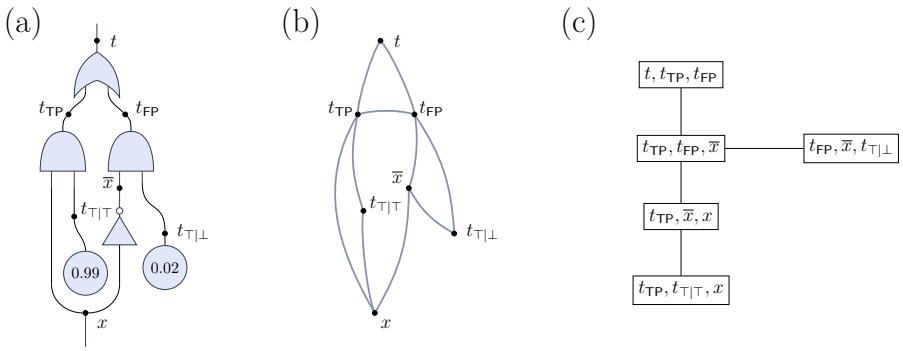


Figure 8 (a) Dot diagram `test` from Figure 1; (b) its primal graph `test′`; (c) tree decomposition of `test′` of width 2.

The width *of a tree decomposition $\mathcal{T}$ is the cardinality of its largest bag, minus one,*

$$\mathsf{width}(\mathcal{T}) := \max_{t \in \mathcal{T}} |X_t| - 1.$$

The treewidth $\mathsf{tw}(G)$ *of G is the minimum width of any tree decomposition of G.*

The main result for this section shows that the treewidth of the primal graph of a Σ-diagram `f` is a crucial parameter for both the width and the DAG-size of the algebraisation of `f`, as well as for the time it takes to find this algebraisation. This version of our algebraisation algorithm employs a specific algorithm for low-width tree-decompositions [18] for the purpose of a clean theoretical description. However, any algorithm for computing tree decompositions (or, more directly, *branch decompositions of hypergraphs*) may be used.

▶ **Theorem 32.** *Let `f` be a Σ-diagram over some monoidal signature Σ, let $k := \mathsf{tw}(\mathtt{f}')$ be the treewidth of the primal graph of `f`, and let $n := 1 + |A_{\mathtt{f}}|$ be one plus the number of assignments in `f`.*

Then `f` has an algebraisation t with $\mathsf{width}(t) = O(k)$ and DAG-size $|t| = O(nk^2)$. Moreover, this algebraisation can be computed in $O(2^{O(k)}n)$ time.

The rest of this section is devoted to describing the algebraisation algorithm in detail, and showing its correctness. In particular, we will conclude with the proof of Theorem 32 and its generalisation to hierarchical dot diagrams, Theorem 49. The proof consists of the following steps:

Sec. 5.1 First, we analyse algebraisations of diagrams without any assignments;

Sec. 5.2 We then consider "inefficient" algebraisations of general diagrams (disregarding width for the moment);

Sec. 5.3 In preparation for the next step, we introduce *unfolding* hierarchical dot diagrams into single large dot diagrams;

Sec. 5.4 We refactor large dot diagrams into hierarchical dot diagrams, with "small" component dot diagrams;

Sec. 5.5 Finally, we put everything together in a divide-and-conquer fashion, by using the "inefficient" algebraisation on each small component.

By the time we are done, we get the analogous result for hierarchical dot diagrams (Theorem 49) practically for free. All omitted proofs can be found the appendix of the full version of this paper [27].

5.1 Algebrising pure wiring

We first algebrise *pure wirings*: dot diagrams with no assignments at all. We distinguish three special types: *permutation-type*, *copy-and-delete-type*, and *equate-and-select-type* dot diagrams.

▶ **Definition 33.** *Let Σ be a monoidal signature. A Σ-diagram $\mathtt{f}$ is a* pure wiring *if $|A_{\mathtt{f}}| = 0$. If, moreover, $\mathsf{in}_{\mathtt{f}}$ and $\mathsf{out}_{\mathtt{f}}$ both do not contain any duplicates, are of the same length and $\mathsf{set}(\mathsf{in}_{\mathtt{f}}) = \mathsf{set}(\mathsf{out}_{\mathtt{f}})$, then $\mathtt{f}$ is of* permutation type*.*

A copy-and-delete-type *Σ-diagram is a pure wiring such that both $\mathsf{in}_{\mathtt{f}}$ and $\mathsf{out}_{\mathtt{f}}$ are sorted according to some (common, arbitrary) order on $X_{\mathtt{f}}$, every variable in $\mathsf{out}_{\mathtt{f}}$ is also in $\mathsf{in}_{\mathtt{f}}$, and $\mathsf{in}_{\mathtt{f}}$ contains no duplicates.*

Dually, an equate-and-select-type *Σ-diagram is a pure wiring such that $\mathsf{in}_{\mathtt{f}}$ and $\mathsf{out}_{\mathtt{f}}$ are sorted according to some (common, arbitrary) order on $X_{\mathtt{f}}$, and $\mathsf{out}_{\mathtt{f}}$ contains no duplicates.*

Copy-and-delete-type dot diagrams are trivial to algebrise: they are a tensor product of i-fold copying dot diagrams, where each i is the multiplicity of the corresponding variable in the list of outputs. Algebraisations for equate-and-select-type dot diagrams are obtained in the same way. The case of algebrising permutation-type wirings is less obvious. In essence, it amounts to a parallel sorting algorithm that uses only adjacent swaps, leading to the following lemma.

▶ **Lemma 34.** *Let Σ be a monoidal signature, let σ be a permutation-type Σ-diagram, and let $k := |\mathsf{in}_\sigma| = |\mathsf{out}_\sigma|$ be the number of inputs (or equivalently outputs) of σ. Then there exists an algebraisation t of σ with $|t| = O(k^2)$ and $\mathsf{width}(t) \leq 2k$. Moreover, such an algebraisation can be computed in time $O(k^2)$.*

To algebrise general pure wirings, we need the following lemma and its corollary, allowing us to write any pure wiring as a composition of permutation-type, copy-and-delete-type, and equate-and-select-type wirings. This can be shown by passing to the side of hypergraph terms, and then proving the corresponding decomposition property by structural induction.

▶ **Lemma 35.** *Let $\mathtt{f}$ be a pure wiring over Σ. Then there exist permutation-type Σ-diagrams $\sigma_1, \sigma_2, \sigma_3$, a copy-and-select-type Σ-diagram c, and an equate-and-delete-type Σ-diagram e such that $\mathtt{f} = \sigma_3 \circ \mathsf{e} \circ \sigma_2 \circ \mathsf{c} \circ \sigma_1$.*

▶ **Corollary 36.** *Let $\mathtt{f}$ be a pure wiring over Σ, and let $k := \max\{|\mathsf{in}_{\mathtt{f}}|, |\mathsf{out}_{\mathtt{f}}|\}$. Then there exists an algebraisation t of $\mathtt{f}$ with $|t| = O(k^2)$ and $\mathsf{width}(t) \leq 2k$. In addition, such an algebraisation can be computed in time $O(k^2)$.*

5.2 Inefficient algebraisations

We give a method for obtaining algebraisations of general dot diagrams that are cleanly described, at the cost of resulting in a large width. Later, we minimise this damage by only applying this to small dot diagrams as part of our divide-and-conquer approach.

The width-inefficient algorithm is based on the observation that any dot diagram can be decomposed into a pure wiring, followed by a tensor product of function symbols and identities, followed by a pure wiring; see Figure 9. This is the subject of the next lemma, which is essentially a consequence of [38, Proposition 5.4], where an essentially equivalent decomposition is called the *Frobenius decomposition*. Before we can state it, we need to define the *width* of a dot diagram, which we will use to bound the time complexity of computing such a decomposition.

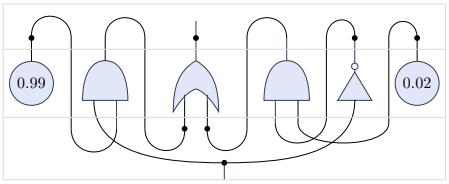


Figure 9 Frobenius decomposition of the dot diagram `test` from Figure 1: pure wiring, followed by a tensor product of function symbols and identities, followed by pure wiring.

▶ **Definition 37.** *Let* $\mathtt{f}$ *be a dot diagram. Let* n_i, m_i *be the number of inputs (outputs, respectively) of the* i*-th assignment of* $\mathtt{f}$*, for* $i \in \{1, \dots, |A_{\mathtt{f}}|\}$*. Let* $N := \max_{i \in \{1,\dots,|A_{\mathtt{f}}|\}} n_i$ *and* $M := \max_{i \in \{1,\dots,|A_{\mathtt{f}}|\}} m_i$*. The* width *of* $\mathtt{f}$ *is*

$$\mathsf{width}(\mathtt{f}) := \max\{|\mathsf{in}_{\mathtt{f}}|, |\mathsf{out}_{\mathtt{f}}|, M, N\},$$

the maximum number of in- or outputs over $\mathtt{f}$ *and its assignments.*

The following are proven in the appendix of the full version of this paper [27].

▶ **Lemma 38** (Frobenius decomposition)**.** *Let* Σ *be a monoidal signature and let* $\mathtt{f}$ *be a* Σ*-diagram. For each* $a \in A_{\mathtt{f}}$*, write* $a = (\mathsf{let}\ \mathbf{y}_a = g_a(\mathbf{x}_a))$*, and let* $\mathbf{S}_a, \mathbf{T}_a \in \Sigma_0^*$ *be the list of (co-)domains of* $g_a : \mathbf{S}_a \to \mathbf{T}_a$*. Then there exist pure wirings* $\mathsf{w}_1, \mathsf{w}_2$ *such that*

$$\mathtt{f} \equiv \mathsf{w}_1 \circ \bigotimes_{a \in A_{\mathtt{f}}} (g_a \otimes \mathsf{id}_{\mathbf{S}_a} \otimes \mathsf{id}_{\mathbf{T}_a}) \circ \mathsf{w}_2,$$

where $\equiv$ *denotes equivalence of dot diagrams. Moreover, the wirings* $\mathsf{w}_1, \mathsf{w}_2$ *can be computed in* $O(|A_{\mathtt{f}}| \cdot \mathsf{width}(\mathtt{f}))$ *time.*

▶ **Corollary 39.** *Let* $\mathtt{f}$ *be a* Σ*-diagram. Let* $w := \mathsf{width}(\mathtt{f})$ *be the width of* $\mathtt{f}$ *and let* $n := |A_{\mathtt{f}}| \geq 1$*. Then there exists an algebraisation* $t \in \mathrm{Term}(\Sigma)$ *of* $\mathtt{f}$ *such that* $\mathsf{width}(t) \leq 6nw$ *and* $|t| = O(n^2 w^2)$*. Moreover, such* t *can be computed in* $O(n^2 w^2)$ *time.*

If every function symbol has at least one in- or output, the width resulting from the Frobenius decomposition is at least n for a dot diagram with n assignments, which clearly far from optimal. We will therefore only use it on small dot diagrams.

5.3 Unfolding hierarchical dot diagrams

The next step of our divide-and-conquer approach is to refactor arbitrary dot diagrams into hierarchical dot diagrams, in which each dot diagram is small. To explain this, we first need the opposite notion: *unfolding* a hierarchical dot diagram into a large regular dot diagram via recursive substitution.

▶ **Definition 40.** *Let* Σ *be a monoidal signature and* $\mathtt{f}$ *be a hierarchical* Σ*-diagram. Let* $(g_i \colon \mathbf{S}_i \to \mathbf{T}_i)_{i \in I} \in \Sigma^I$ *be a finite family of function symbols, and let* $(\mathtt{g}_i \colon \mathbf{S}_i \to \mathbf{T}_i)_{i \in I}$ *be a finite family of* Σ*-diagrams over the same index set. The* substitution *of* $(\mathtt{g}_i)_{i \in I}$ *for* $(g_i)_{i \in I} \in \Sigma^I$ *in* $\mathtt{f}$*, written*

$$\mathtt{f}[g_i \mapsto \mathtt{g}_i]_{i \in I},$$

is defined as follows. The set of variables of $\mathtt{f}[g_i \mapsto \mathtt{g}_i]_{i \in I}$ *is the following pushout,*

$$X_{\mathtt{f}\,[g_i \mapsto \mathtt{g}_i]_{i \in I}} := \left(X_{\mathtt{f}} \sqcup \bigsqcup_{i \in I} \bigsqcup_{\mathsf{let}\ \mathbf{y} = g_i(\mathbf{x})} X_{\mathtt{g}_i} \right) / \sim,$$

where $\sqcup$ denotes disjoint union, and $\sim$ is the equivalence relation that identifies the input (resp. output) variables of each (copy of) $\mathtt{g}_i$ with the input (resp. output) variables of the corresponding assignment in which g_i occurs. The lists of in- and output variables of $\mathtt{f}[g_i \mapsto \mathtt{g}_i]_{i\in I}$ are the the pushforward of those of $\mathtt{f}$ under the inclusion $X_{\mathtt{f}} \to X_{\mathtt{f}\,[g_i\mapsto \mathtt{g}_i]_{i\in I}}$. Finally, the list of assignments of $\mathtt{f}[g_i \mapsto \mathtt{g}_i]_{i\in I}$ is given by replacing each assignment of the form

$$\mathsf{let}\ \mathbf{y} = g_i(\mathbf{x})$$

in $A_{\mathtt{f}}$ by the list $A_{\mathtt{g}_i}$, with each variable in $A_{\mathtt{f}}$ or $A_{\mathtt{g}_i}$ replaced by its image in $X_{\mathtt{f}[g_i\mapsto \mathtt{g}_i]_{i\in I}}$.

Unfolding a hierarchical Σ-diagram can now be defined by recursively substituting the definitions of all dot diagrams appearing therein along its call graph.

▶ **Definition 41.** *Let $\mathtt{f}$ be a hierarchical Σ-diagram. Define*

$$\mathsf{unfold}'(\mathtt{f}, v) := \begin{cases} \mathtt{f}_v & \text{if } v \text{ is a leaf,} \\ \mathtt{f}_v[\mathsf{defn}_{\mathtt{f}}^{-1}(u) \mapsto \mathsf{unfold}'(\mathtt{f}, u)]_{u\in \mathrm{ch}(v)} & \text{otherwise.} \end{cases}$$

Unfolding $\mathtt{f}$ is defined as $\mathsf{unfold}(\mathtt{f}) := \mathsf{unfold}'(\mathtt{f}, R_{\mathtt{f}})$.

To extend the algebraisation algorithm of Corollary 39 to hierarchical dot diagrams, we need the fact that algebraisation commutes with substitution.

▶ **Lemma 42.** *Let $\mathtt{f}$ be a Σ-diagram, let $(g_i : \mathbf{S}_i \to \mathbf{T}_i)_{i\in I}$ be a finite family of function symbols in Σ, and let $(\mathtt{g}_i : \mathbf{S}_i \to \mathbf{T}_i)_{i\in I}$ be a family of Σ-diagram over the same index set. Moreover, suppose that t is an algebraisation of $\mathtt{f}$ and that s_i is an algebraisation of $\mathtt{g}_i$, for each $i \in I$. Then $t[g_i \mapsto s_i]_{i\in I}$ is an algebraisation of $\mathtt{f}[g_i \mapsto \mathtt{g}_i]_{i\in I}$.*

Proof. This can be shown by induction on the structure of t. In the base case, when t is consists of a single symbol, the claim is trivial. If, on the other hand, $t = t_1 \circ t_2$ or $t = t_1 \otimes t_2$, this follows quickly using the induction hypothesis and the definitions of the composition and tensor product of Σ-diagrams. ◀

▶ **Lemma 43.** *Let Σ be a monoidal signature and let $\mathtt{f}$ be a hierarchical Σ-diagram. Let $N := \max_{v\in V(\mathtt{f})} |A_{\mathtt{f}_v}| > 1$ be the maximum number of assignments of any dot diagram appearing in $\mathtt{f}$, and let $K := \max_{v\in V(\mathtt{f})} \mathsf{width}(\mathtt{f}_v)$ be the maximum width of any dot diagram appearing in $\mathtt{f}$.*

Then there exists an algebraisation $t \in \mathrm{Term}(\Sigma)$ of $\mathsf{unfold}(\mathtt{f})$ with $|t| = O(|V(\mathtt{f})|N^2K^2)$ and $\mathsf{width}(t) \le 6NK$. Moreover, it can be computed in $O(|V(\mathtt{f})|N^3K^2)$ time.

5.4 Algebraisation via branch decomposition

To turn a dot diagram into a hierarchical dot diagram in which only "small" auxiliary dot diagrams appear, we rely on the notion of a *branch decomposition* of a hypergraph. By a *hypergraph* we mean an undirected hypergraph allowing for loops and multiple edges:

▶ **Definition 44.** *A* hypergraph *G consists of a set $V(G)$ of* vertices*, a set $E(G)$ of* hyperedges*, and a relation $I(G) \subseteq V(G) \times E(G)$. A vertex v is* incident *with e if $(v, e) \in I(G)$.*

Dot diagrams have an associated hypergraph, the *dependency hypergraph*, whose vertices are the variables of the dot diagram.

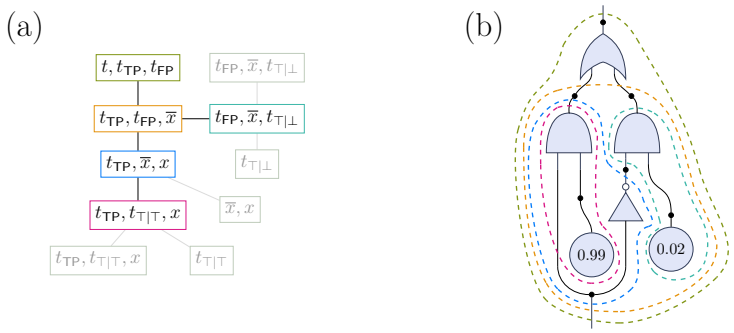


Figure 10 (a) A tree decomposition of `test`′ extended by redundant (light grey) bags to turn it into a branch decomposition of Dep(`f`); (b) branch decomposition visualised as a hierarchical clustering of the boxes of `test`.

▶ **Definition 45.** *Let* Σ *be a signature and let* `f` *be a* Σ*-diagram. The* dependency hypergraph $\mathrm{Dep}(\mathtt{f})$ *of* `f` *is defined by*

$$V(\mathrm{Dep}(\mathtt{f})) = X_{\mathtt{f}},$$

$$E(\mathrm{Dep}(\mathtt{f})) = \mathsf{set}(A_{\mathtt{f}}) \cup \{\mathsf{in}_{\mathtt{f}}, \mathsf{out}_{\mathtt{f}}\},$$

and a vertex x *is incident with* e *in* $\mathrm{Dep}(\mathtt{f})$ *if and only if either*

1. $e = (\mathsf{let}\ \mathbf{y} = f(\mathbf{x}))$ *is an assignment and* $x \in \mathbf{x}$ *or* $x \in \mathbf{y}$*, or*
2. $e = \mathsf{in}_{\mathtt{f}}$ *and* $x \in \mathsf{in}_{\mathtt{f}}$*, or*
3. $e = \mathsf{out}_{\mathtt{f}}$ *and* $x \in \mathsf{out}_{\mathtt{f}}$*.*

By an *unrooted binary tree*, we mean a tree (i.e. a connected acyclic undirected graph) in which each vertex has degree either 1 or 3. The notion of a branch decomposition of a hypergraph is due to Robertson and Seymour [29].

▶ **Definition 46.** *Let* G *be a hypergraph. A* branch decomposition *is a unrooted binary tree* $\mathcal{B}$ *together with a bijection* η *from the leaves of* $\mathcal{B}$ *to the hyperedges of* G*.*

Let e *be an edge of* $\mathcal{B}$*. Then removing* e *from* $\mathcal{B}$ *partitions* $\mathcal{B}$ *into two components* $\mathcal{B}_1$*,* $\mathcal{B}_2$*. A vertex* v *of* G *is* at the interface of e *if there is a leaf* l_1 *of* $\mathcal{B}_1$ *and a leaf* l_2 *of* $\mathcal{B}_2$ *such that* v *is incident to both* $\eta(l_1)$ *and* $\eta(l_2)$*. The* order *of an edge* e *of* $\mathcal{B}$ *is the number of vertices in* G *at the interface of* e*.*

The width $\mathsf{width}(\mathcal{B})$ *of a branch decomposition* $\mathcal{B}$ *of* G *is the maximum order of any edge in* $\mathcal{B}$*. The* branch-width *of* G *is the minimum width of any branch decomposition of* G*.*

A branch decomposition of the dependency graph of a dot diagram can be visualised as hierarchical clustering of its assignments (“boxes”); see Figure 10. The next lemma turns a branch decomposition of the dependency graph into a hierarchical dot diagram with the desired properties.

▶ **Lemma 47.** *Let* `f` *be a* Σ*-diagram and let* $\mathcal{B}$ *be a branch decomposition of the dependency hypergraph of* `f`*. Then there exists a hierarchical* Σ*-diagram* `f`* *such that:*

1. $\mathsf{unfold}(\mathtt{f}^*) = \mathtt{f}$*.*
2. *Every dot diagram in* `f`* *has at most* $\mathsf{width}(\mathcal{B})$ *in- or outputs.*
3. *Every dot diagram in* `f`* *has at most two assignments.*
4. $|V(\mathtt{f}^*)| \leq |A_{\mathtt{f}}|$*.*

Moreover, such an `f`* *can be computed in* $O(|V(\mathcal{B})| \cdot \mathsf{width}(\mathcal{B}))$ *time.*

Once we have converted a dot diagram into hierarchical dot diagram in which only "small" dot diagrams appear, as provided by Lemma 47, we can apply Lemma 43 to obtain an algebraisation of small width. Hence, taking both lemmas together directly implies:

▶ **Theorem 48.** *Let* $\mathtt{f}$ *be a* Σ*-diagram and let* $\mathcal{B}$ *be a branch decomposition of the dependency hypergraph of* $\mathtt{f}$*. Then there exists an algebraisation* t *of* $\mathtt{f}$ *with* $\mathsf{width}(t) \leq 12\,\mathsf{width}(\mathcal{B})$ *and* $|t| = O(|A_{\mathtt{f}}| \cdot \mathsf{width}(\mathcal{B})^2)$*. Moreover, given the branch decomposition* $\mathcal{B}$*, such an algebraisation* t *can be computed in time* $O(|A_{\mathtt{f}}| \cdot \mathsf{width}(\mathcal{B})^2)$*.*

5.5 Proof of Theorem 32

Proof. We now show Theorem 32 by reducing it to Theorem 48. First, by a theorem of Korhonen [18, Theorem 1.1.], one can compute a tree decomposition $\mathcal{T}$ of width at most $2k+1$ of the primal graph $\mathtt{f}'$ of the given dot diagram $\mathtt{f}$, where k is the treewidth of $\mathtt{f}'$, in time $O(|A_{\mathtt{f}}| \cdot 2^{O(k)})$. Moreover, we may assume without loss of generality that $|V(\mathcal{T})|$ is linear in $|A_{\mathtt{f}}|$. We may assume further that $|A_{\mathtt{f}}| \geq 2$ (otherwise, we use the algebraisation algorithm of Corollary 39). Under these assumptions, one can convert tree decomposition $\mathcal{T}$ of $\mathtt{f}'$ into a branch decomposition $\mathcal{B}$ of the dependency hypergraph $\mathrm{Dep}(\mathtt{f})$ of width $\mathsf{width}(\mathcal{B}) \leq \mathsf{width}(\mathcal{T}) + 1$, in linear time with respect to $|A_{\mathtt{f}}|$. This construction is carried out in the proof of [29, (5.1)]. The notion of treewidth used there is equivalent to the treewidth of the primal graph; see also Figure 10. Using Theorem 48 we find an algebraisation t of width

$$\mathsf{width}(t) \leq 12 \cdot \mathsf{width}(\mathcal{B}) \leq 24k + 24 \in O(k)$$

and $|t| = O(|A_{\mathtt{f}}| \cdot k^2)$ in time $O(|A_{\mathtt{f}}| \cdot 2^{O(k)})$. ◀

5.6 Generalisation to the hierarchical case

Theorem 32 can now be easily generalised to hierarchical dot diagrams $\mathtt{f}$: when the primal graphs of each dot diagram appearing in $\mathtt{f}$ all have small treewidth, then an algebraisation of the unfolded dot diagram $\mathsf{unfold}(\mathtt{f})$ of small width and DAG-size can be computed efficiently. This is the subject of the following theorem, whose proof is analogous to the one of Lemma 43.

▶ **Theorem 49.** *Let* $\mathtt{f}$ *be a hierarchical dot diagram over a monoidal signature* Σ*. Moreover, let*

1. $k := \max_{v \in V(\mathtt{f})} \mathsf{tw}(\mathtt{f}'_v)$ *be the maximum treewidth of the primal graph of any* $\mathtt{f}_v$*,*
2. $L := 1 + \max_{v \in V(\mathtt{f})} |\mathrm{ch}(v)|$ *be one plus the maximum number of children of any vertex in the call graph of* $\mathtt{f}$*,*
3. $M := |V(\mathtt{f})|$*, and let*
4. $N := 1 + \max_{v \in V(\mathtt{f})} |A_{\mathtt{f}_v}|$ *be one plus the maximum number of assignments in any dot diagrams appearing in* $\mathtt{f}$*.*

Then an algebraisation of $\mathsf{unfold}(\mathtt{f})$ *of DAG-size in* $O(MNk^2)$ *and width in* $O(k)$ *can be computed in* $O(LMN \cdot 2^{O(k)})$ *time.*

6 Applications of algebraisation

With nice algebraisations in place, we now return to the question of computing semantics; this includes DPP inference. Of course, the precise complexity depends highly on the strict hypergraph category corresponding to the semantics. We first discuss general properties of semantics in matrix categories, and then apply it to Problem 1. Finally, we discuss other existing problems that can be viewed as the computation of dot diagram semantics.

6.1 Computing matrix semantics

To cover many of the applications that follow in a unified manner, we first consider the problem of computing the semantics of a hierarchical dot diagram in a matrix category over a semiring. The semantics of a hierarchical dot diagram are defined as the semantics of the simple dot diagram that results from unfolding it.

▶ **Definition 50.** *Let* $\mathtt{f}$ *be a hierarchical dot diagram over some monoidal signature* Σ*, and let* $\mathcal{I}\colon \Sigma \to C$ *be an interpretation of* Σ *in some hypergraph category* C*. The* semantics *of* $\mathtt{f}$ *under* $\mathcal{I}$ *are* $[\![\mathtt{f}]\!]_{\mathcal{I}} := [\![\mathsf{unfold}(\mathtt{f})]\!]_{\mathcal{I}}$.

When working over general semirings, a natural way to express the complexity of an algorithm is the number of additions and multiplications it takes. This can be made precise using the notion of an *arithmetic circuit* over a semiring: a representation of an arithmetic expression as a directed acyclic graph. The problem then becomes to *compile* a hierarchical dot diagram $\mathtt{f}$ to an equivalent arithmetic circuit. Theorem 52 shows that when the primal graph of each dot diagram appearing in $\mathtt{f}$ has bounded treewidth, this problem can be solved efficiently.

▶ **Definition 51.** *Let* R *be a semiring. An* arithmetic circuit *over* R *is a tuple* $C = (V(C), E(C), (\mathrm{op}_v)_{v\in V(C)})$ *such that:*

1. $(V(C), E(C))$ *is a directed acyclic graph.*
2. *For each vertex* $v \in V(C)$*, the* label op_v^C *of* v *is either an element of* R*, or one of the two symbols* $\{+, \cdot\}$.
3. *A vertex* $v \in V(C)$ *is a leaf in* $(V(C), E(C))$ *if and only if it is labelled by an element of* R*, i.e.* $\mathrm{op}_v^C \in R$.
4. *All non-leaf vertices have exactly two children.*

The semantics *of* C *at* $v \in V(C)$ *are given by*

$$[\![C]\!]_v := \begin{cases} \mathrm{op}_v^C & \text{if } v \text{ is a leaf,} \\ \sum_{u\in \mathrm{ch}(v)} [\![C]\!]_u & \text{if } \mathrm{op}_v^C = +, \\ \prod_{u\in \mathrm{ch}(v)} [\![C]\!]_u & \text{otherwise.} \end{cases}$$

Note that in contrast to other notions of *arithmetic circuit* in the literature [32], the arithmetic circuits we consider may have multiple outputs (i.e. roots). Moreover, as defined above, they only represent *closed* arithmetic expressions (i.e. expressions which not contain variables). The following theorem characterises a general algorithm for computing the semantics of dot diagrams in matrix hypergraph categories.

▶ **Theorem 52.** *Let* Σ *be a monoidal signature, let* R *be a semiring, and let* $\mathcal{I}\colon \Sigma \to \mathsf{Mat}_R$ *be an interpretation. Moreover, let* $\mathtt{f}$ *be a hierarchical dot diagram over* Σ*, and let* k*,* L*,* M *and* N *be defined as in Theorem 49. Finally, let* $m, n \in \mathbb{N}$ *be the dimensions of the matrix* $[\![\mathtt{f}]\!]_{\mathcal{I}} \in R^{m\times n}$.

Then there is an arithmetic circuit C *over* R *with* $O(2^{O(k)}MN)$ *vertices and a bijection* ν *from* $\{0, \dots, n-1\} \times \{0, \dots, m-1\}$ *to the set of roots of* C *such that*

$$([\![\mathtt{f}]\!]_{\mathcal{I}})_{ij} = [\![C]\!]_{\nu(i,j)}.$$

Moreover, this arithmetic circuit C *can be computed in* $O(2^{O(k)}LMN)$ *time.*

In particular, if the arithmetic operations in R *can be computed in constant time and the treewidth* k *is bounded, then the semantics of* $\mathtt{f}$ *can be computed in* $O(LMN)$ *time.*

6.2 Probabilistic inference

We now consider DPPs specifically.

▶ **Definition 53.** *Let* $\mathtt{f} \colon (\,) \to (\mathbb{B})$ *be a discrete probabilistic program with no inputs and a single output. Let* $\begin{pmatrix} q & p \end{pmatrix}^{\mathsf{T}} := [\![\mathtt{f}]\!]_{\mathcal{S}}$ *be the semantics of* $\mathtt{f}$ *under the substochastic interpretation. The* acceptance probability *of* $\mathtt{f}$ *is* $p_{\mathrm{acc}} := p + q$*. The* probability that $\mathtt{f}$ outputs true *is* $p_{\mathtt{f}} := p/p_{\mathrm{acc}}$ *if* $p_{\mathrm{acc}} > 0$ *and* 0 *otherwise.*

This leads to the following rephrasing of Problem 1:

▶ **Problem 54.** *Given a discrete probabilistic program* $\mathtt{f}$ *and a natural number* $d > 0$*, approximate* $p_{\mathtt{f}}$ *with an absolute error of at most* $2^{-(d+1)}$*.*

Next is our main result concerning inference in DPPs. Its proof is mainly an application of Theorem 52, together with standard arguments for bounding the error and complexity for matrix multiplications and Kronecker products of substochastic matrices.

▶ **Theorem 55.** *Let* $\mathtt{f} \colon (\,) \to (\mathbb{B})$ *be a discrete probabilistic program with no inputs and a single output, let* p_{acc} *be the acceptance probability of* $\mathtt{f}$*, and let* $d > 0$ *be an integer.*

Then the probability $p_{\mathtt{f}}$ *that* $\mathtt{f}$ *outputs* true *can be approximated with absolute error at most* $2^{-(d+1)}$ *in time*

$$O((LMN)^3 \cdot d^2 \cdot \log^3(p_{\mathrm{acc}}^{-1}) \cdot 2^{O(k)}),$$

where L*,* M*, and* N *are defined as in Theorem 49.*

In particular, if k *is bounded and the acceptance probability* p_{acc}^{-1} *is at most exponential in* LMN*, then Boolean probabilistic inference can be performed in polynomial time.*

A problem is said to be *fixed parameter tractable* if on every class of instances for which a given parameter is bounded, it can be solved in polynomial time. Hence, according to Theorem 55 Boolean probabilistic inference is fixed-parameter tractable for the parameter $(k, p_{\mathrm{acc}}^{-1})$. The dependency on p_{acc}^{-1} can be removed, if we assume that the call graph of the given probabilistic program has bounded depth, since in this case the acceptance probability is at most exponential in LMN. This also shows that Theorem 55 can be seen as a proper generalisation of the fixed-parameter tractability of Bayesian network inference.

▶ Remark 56. Even though Theorem 55 generalises the fixed-parameter tractability of Bayesian network inference, our inference algorithm does *not* reduce to the classical junction tree algorithm for Bayesian networks. One significant difference is that the junction tree algorithm operates directly on a tree decomposition of the moral graph (which is the same as the primal graph of a corresponding DPP). In contrast, our algorithm first converts this tree decomposition into a branch decomposition, making our algorithm is more similar to (but still distinct from) tensor network contraction algorithms.

6.3 Further applications

Being phrased in terms of dot diagrams rather than DPPs specifically, Theorem 32 can also be applied to derive a number of fixed parameter-tractability results in a unified manner.

Tensor networks and quantum circuit simulation

Tensor networks can be viewed as dot diagrams over Mat_R (viewed as a monoidal signature), and computing the semantics of such a dot diagram corresponds precisely to tensor network contraction [17, Theorem 5.1]. Hence, the known result that tensor network contraction

can be performed using polynomially many arithmetic operations for networks of bounded treewidth [22], e.g. for the purpose of quantum circuit simulation, can be viewed as special cases of Theorem 52.

Computing attack tree metrics

As was shown in [26], the problem of computing *attack tree metrics* [21] can naturally be formulated as the problem of computing the semantics of a string diagram, the given *attack tree*, in Mat_R for suitable semirings R. In particular, the metric associated to the tropical semiring $(\mathbb{N}, \min, +)$, the *min. cost metric*, quantifies the minimum cost for an attacker to compromise the system modelled by the attack tree. A direct application of Theorem 52 now gives that attack tree metrics can be computed in polynomial time on attack trees of bounded treewidth, as long as the size of any intermediate result of evaluating the corresponding arithmetic circuit is bounded. The latter condition applies, for instance, in the case of the min. cost metric. For this problem, no fixed-parameter tractable algorithms have previously been described in the literature.

Evaluating database queries

In the context of relational databases, it is known that conjunctive query evaluation is fixed-parameter tractable [4, 12]. One can view this as a consequence of Theorem 32, as follows. A conjunctive query (i.e. a select-project-join query) on a database can be viewed as a dot diagram over the hypergraph category FinRel of finite relations (viewed as a monoidal signature).[6] Here, FinRel can be defined as the matrix hypergraph category $\mathsf{Mat}_{\mathbb{B}}$, where $(\mathbb{B}, \vee, \wedge)$ is the Boolean semiring. In the context of databases, however, one should think of the morphisms of FinRel not as matrices with Boolean entries, but as tables that directly list the elements of a relation. The problem of evaluating a conjunctive query is then equivalent to computing the semantics of the corresponding dot diagram in FinRel. The primal graph of the dot diagram corresponds to the *query graph* of the conjunctive query. Therefore, it follows from Theorem 32 that a conjunctive query whose query graph has bounded treewidth can be evaluated in polynomial time in the size of the query and the largest table in the database. In contrast to the case of tensor networks and attack trees, this result does not use the compilation of string diagrams to arithmetic circuits, but directly uses the cartesian product and composition of relations in the evaluation step. This shows that the additional layer of abstraction provided by dot diagrams and hypergraph categories can be necessary for some applications.

7 Conclusion

Theorem 55 shows that despite the likely inherent intractability of inference for *general* discrete probabilistic programs, inference can nevertheless be fast on sufficiently structurally simple instances. Moreover, existing inference engines for discrete probabilistic programs do not satisfy our parametrised complexity bounds, and for deeply nested programs our algorithm yields exponential asymptotic improvements (see Figure 2a).

What remains open is whether, and to what extent, our method also yields benefits in practical implementations. This next step will also be an opportunity to investigate potential advantages of our approach that are difficult to analyse on the level of asymptotic complexity. One such potential advantage is that the algebraic representation of probabilistic programs

[6] This is explained in more detail in the appendix of the full version of this paper [27].

enables the use of traditional methods for term rewriting and program optimisation to simplify terms before evaluating them, making use of the sound and complete equational theory for discrete probabilistic programs from [28]. Another possible advantage is that our abstract formulation allows dense matrices to be easily replaced by more efficient representations of substochastic matrices, similar to decision diagrams used in quantum circuit simulation [37], without affecting the stated performance guarantees.

Finally, we have shown that our method, *string diagram algebraisation*, is applicable beyond probabilistic programming, enabling a unified view on a diverse variety of known fixed-parameter tractability results.

References

1 Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

2 Hans L Bodlaender. Discovering treewidth. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 1–16. Springer, 2005. doi:10.1007/978-3-540-30577-4_1.

3 Filippo Bonchi, Fabio Gadducci, Aleks Kissinger, Pawel Sobocinski, and Fabio Zanasi. String diagram rewrite theory I: Rewriting with Frobenius structure. *Journal of the ACM (JACM)*, 69(2):1–58, 2022. doi:10.1145/3502719.

4 Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000. doi:10.1016/S0304-3975(99)00220-0.

5 Adnan Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 819, 2011.

6 Adnan Darwiche and Pierre Marquis. A knowledge compilation map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002. doi:10.1613/JAIR.989.

7 Giovanni de Felice, Alexis Toumi, and Bob Coecke. DisCoPy: Monoidal categories in Python. In David I. Spivak and Jamie Vicary, editors, Proceedings of the 3rd Annual International *Applied Category Theory Conference 2020,* Cambridge, USA, 6-10th July 2020, volume 333 of *Electronic Proceedings in Theoretical Computer Science*, pages 183–197. Open Publishing Association, 2021. doi:10.4204/EPTCS.333.13.

8 Elena Di Lavore and Paweł Sobociński. Monoidal width. *Logical Methods in Computer Science*, 19, 2023.

9 Jeffrey M Dudek and Moshe Y Vardi. Parallel weighted model counting with tensor networks. *arXiv preprint*, 2020. arXiv:2006.15512.

10 Brendan Fong and David I Spivak. Hypergraph categories. *Journal of Pure and Applied Algebra*, 223(11):4746–4777, 2019.

11 Tobias Fritz and Wendong Liang. Free gs-monoidal categories and free markov categories. *Applied Categorical Structures*, 31(2):21, 2023. doi:10.1007/S10485-023-09717-0.

12 Martin Grohe, Thomas Schwentick, and Luc Segoufin. When is the evaluation of conjunctive queries tractable? In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 657–666, 2001. doi:10.1145/380752.380867.

13 Masahito Hasegawa, Martin Hofmann, and Gordon Plotkin. Finite dimensional vector spaces are complete for traced symmetric monoidal categories. In *Pillars of Computer Science: Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*, pages 367–385. Springer, 2008. doi:10.1007/978-3-540-78127-1_20.

14 Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, 2020. doi:10.1145/3428208.

15 Angelika Kimmig, Guy Van den Broeck, and Luc De Raedt. Algebraic model counting. *Journal of Applied Logic*, 22:46–62, 2017. doi:10.1016/J.JAL.2016.11.031.

16 Aleks Kissinger. *Pictures of Processes*. PhD thesis, University of Oxford, 2011.

17 Aleks Kissinger. Finite matrices are complete for (dagger-) hypergraph categories. *arXiv preprint*, 2014. arXiv:1406.5942.

18 Tuukka Korhonen. A single-exponential time 2-approximation algorithm for treewidth. *SIAM Journal on Computing*, pages FOCS21–174, 2023.

19 Steffen L Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society: Series B (Methodological)*, 50(2):157–194, 1988.

20 Milan Lopuhaä-Zwakenberg. Fault tree reliability analysis via squarefree polynomials: Mathematical and experimental analysis. *SN Computer Science*, 6(8):965, 2025. doi:10.1007/S42979-025-04450-Y.

21 Milan Lopuhaä-Zwakenberg, Carlos E Budde, and Mariëlle Stoelinga. Efficient and generic algorithms for quantitative attack tree analysis. *IEEE Transactions on Dependable and Secure Computing*, 20(5):4169–4187, 2022. doi:10.1109/TDSC.2022.3215752.

22 Igor L Markov and Yaoyun Shi. Simulating quantum computation by contracting tensor networks. *SIAM Journal on Computing*, 38(3):963–981, 2008. doi:10.1137/050644756.

23 Scott McLachlan, Kudakwashe Dube, Graham A Hitman, Norman E Fenton, and Evangelia Kyrimi. Bayesian networks in healthcare: Distribution by medical condition. *Artificial intelligence in medicine*, 107:101912, 2020. doi:10.1016/J.ARTMED.2020.101912.

24 Judea Pearl. Bayesian networks: A model of self-activated memory for evidential reasoning. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, volume 7, 1985.

25 Judea Pearl. *Probabilistic reasoning in intelligent systems: Networks of plausible inference*. Morgan Kaufmann, 1988.

26 Benedikt Peterseim and Milan Lopuhaä-Zwakenberg. A unified compositional view of attack tree metrics. *arXiv preprint*, 2025. doi:10.48550/arXiv.2511.14717.

27 Benedikt Peterseim and Milan Lopuhaä-Zwakenberg. Fixed-parameter tractable inference for discrete probabilistic programs, via string diagram algebraisation. *arXiv preprint*, 2026. arXiv:2604.25321.

28 Robin Piedeleu, Mateo Torres-Ruiz, Alexandra Silva, and Fabio Zanasi. A complete axiomatisation of equivalence for discrete probabilistic programming. In *European Symposium on Programming*, pages 202–229. Springer, 2025. doi:10.1007/978-3-031-91121-7_9.

29 Neil Robertson and Paul D Seymour. Graph minors. x. obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991. doi:10.1016/0095-8956(91)90061-N.

30 Celia Rubio-Madrigal and Jules Hedges. Rendering string diagrams recursively. *arXiv preprint*, 2024. doi:10.48550/arXiv.2404.02679.

31 Enno Ruijters and Mariëlle Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer science review*, 15:29–62, 2015. doi:10.1016/J.COSREV.2015.03.001.

32 Amir Shpilka, Amir Yehudayoff, et al. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3–4):207–388, 2010. doi:10.1561/0400000039.

33 Michael Stamatelatos, William Vesely, Joanne Dugan, Joseph Fragola, Joseph Minarick, and Jan Railsback. Fault tree handbook with aerospace applications, 2002.

34 Dario Stein and Sam Staton. Probabilistic programming with exact conditions. *Journal of the ACM*, 71(1):1–53, 2024. doi:10.1145/3632170.

35 Dario Maximilian Stein. *Structural foundations for probabilistic programming languages*. PhD thesis, University of Oxford, 2021.

36 Alexis Toumi, Richie Yeung, Boldizsár Poór, and Giovanni de Felice. DisCoPy: the hierarchy of graphical languages in Python. *arXiv preprint*, 2023. doi:10.48550/arXiv.2311.10608.

37 Robert Wille, Stefan Hillmich, and Lukas Burgholzer. Decision diagrams for quantum computing. In *Design automation of quantum computers*, pages 1–23. Springer, 2022.

38 Paul Wilson and Fabio Zanasi. Data-parallel algorithms for string diagrams. *arXiv preprint*, 2023. doi:10.48550/arXiv.2305.01041.

39 Peng Xie, Jason H Li, Xinming Ou, Peng Liu, and Renato Levy. Using Bayesian networks for cyber security analysis. In *2010 IEEE/IFIP international conference on dependable systems & networks (DSN)*, pages 211–220. IEEE, 2010. doi:10.1109/DSN.2010.5544924.

40 Fabio Zanasi. Rewriting in free hypergraph categories. *arXiv preprint*, 2017. arXiv:1712.09495.