A UNIFYING SEMANTICS FOR MARKOV KERNELS AND LINEAR OPERATORS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Pedro Henrique Azevedo de Amorim

August 2023

A UNIFYING SEMANTICS FOR MARKOV KERNELS AND LINEAR OPERATORS

Pedro Henrique Azevedo de Amorim, Ph.D.

Cornell University 2023

There has been much work done in developing semantic structures for interpreting probabilistic programs. In particular, there have been many models based either on Markov kernels or linear operators, each with their own set of strengths and weaknesses.

Concurrently, mathematicians have been working on categorical semantics for probability theory with the goal of obtaining a more abstract understanding of the field. This has led to the definition of Markov categories, an abstraction of Markov kernels. However, a similar treatment to the linear operator approach to probability is currently eluded by existing methods.

This thesis sits at the intersection of probabilistic semantics and categorical probability theory. We propose a new categorical semantics and core calculus that extends Markov categories with linear operators, we justify its viability by showing how many useful categories used in probabilistic semantics are instances of our framework and, furthermore, we define a new model inspired by a functional-analytic treatment of measure theory. We conclude by showing how this formalism can be used to reason about a generalized notion of probabilistic independence via a substructural type system.

BIOGRAPHICAL SKETCH

Pedro was born in Campinas, Brazil. In 2011 he joined the Computer Engineering program at Campinas State University (UNICAMP). In 2013 he was accepted to the cycle ingenieur program at École polytechnique in France as part of a double degree program. In December 2017 he concluded the double degree program by graduating from Unicamp and in August 2018 he started a PhD program at Cornell University in Ithaca, New York.

Para Julia

ACKNOWLEDGEMENTS

A PhD dissertation is the culmination of many years of learning how to become a researcher. The path from consuming science to actually contributing to it is filled with many obstacles, both intellectual as well as emotional ones. Reaching the end of this path involves direct, and indirect, help from many people. I feel grateful for everyone who helped and encouraged me during my PhD.

I want to start by thanking my advisor Dexter Kozen. His support and encouraging advising style allowed me to freely explore research ideas I was interested in and his deep technical knowledge and creativity were constants sources of inspiration. I would also like to thank Justin Hsu, who has showed me the importance of clear and effective scientific writing.

I was also very fortunate to have collaborated with amazing scientists. Thanks to all my coauthors: Dexter Kozen, Justin Hsu, Andrew Hirsch, Ethan Cecchetti, Ross Tate, Owen Arden, Michael Roberts, Prakash Panangaden, Radu Mardare, Maxi Wuttke, Deepak Garg, Leon Witzman, Rachit Nigam and Adrian Sampson. I was also lucky to mentor Christopher Lam and Sophia Roshal while they were undergraduates at Cornell. I feel honored to have been a part of their early research development and they helped me appreciate the challenges of advising students.

I would also like to thank Cornell for fostering a warm and welcoming community and, in particular, the Computer Science department for being a specially lively environment. I would like to give special thanks to Rachit, Michael, Rolph, Ryan, Jialu, Priya, Christopher, Sophia, Andrew, Ann, Diego, Jenny, Natasha, the $(Nat \to Nat) \to (Nat \to Nat) \to (Nat \to Nat)$ group chat, my numerous office mates and the programming languages discussion group (PLDG)

for giving me so many fond memories over the past five years. I would like to thank my parents, Sebastião and Inês for always believing in my potential, my brothers Arthur — my first research mentor — and Bernardo. A heartfelt thanks goes to my grandpa Alberto. I wish he was here to read this thesis.

I would be remiss not to mention the impact that the Covid-19 pandemic had on my development as a researcher. During the first months of lockdown it was extremely hard to focus on research, so I am extremely thankful for reconnecting with Eduardo, Reilton, Sergio, Esteban, Gustavo and Francisco. Our weekly video calls and gaming sessions made the pandemic feel much less isolating.

Finally, I would like to thank my wife, Julia, whose continuous emotional support and companionship made all of this possible.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

What is a computer program? To the hardware it is simply a sequence of instructions telling it how to operate on a finite set of memory locations. To programmers, however, through human-made abstractions, we can distance ourselves from implementation details and focus on mathematical properties of programs. Indeed, when programming with linked lists in high-level programming languages such as OCaml and Haskell we are not thinking about them as memory locations containing a value and a pointer to the next element. Instead, we understand them through their algebraic properties given by operations such as map, filter, folds, etc.

One of the great insights in computer science is that there are different level of abstractions for the interpretation of programs: all the way from the bare metal execution up-to the platonic ideal of the mathematical meaning of the program.

Each of these levels have pros and cons. The most concrete interpretation, for example, can give insights into how you should implement a particular programming language given a specific computer architecture, but this lack of abstraction makes it hard to reason about programs. On the other extreme, purely abstract methods, usually through denotational semantics, can obfuscate how to most efficiently implement a language given the constraints of the real-world, but can provide crystal-clear reasoning principles for programs. Indeed, denotational methods advocate that syntax is merely a tool for representing mathematical objects and structures.

Different semantics for different needs. Resting at a lower level of abstraction are operational methods which give meaning to programs by viewing

them as transition systems and, in many cases, have the syntax play a larger role in the interpretation of programs. Since the beginning of more principled approaches to programming language there is a tension between syntactic/operational methods and denotational ones. One of the most notable examples is the untyped λ-calculus. Its syntactic properties such as confluence, weak-normalization, etc, have been studied for a long time — long before it found applications in computer science — though researchers were not sure if giving a denotational interpretation of the untyped λ-calculus would even be possible.

This question was finally settled by Scott [83] in the early 70s who used complete lattices and Scott-continuous functions as the semantic foundation of the untyped λ-calculus. He later refined his model by weakening complete lattices to complete partial orders (CPOs) which, in turn, gave rise to the field of domain theory. During the coming decades much work has been done in using domain theory as a general theory of programming language semantics [3, 1].

A few years later Plotkin [76] created "structural operational semantics" as a unifying and general formalism for defining operational semantics — To this day it is the way operational semantics are still defined. In subsequent work he stated and proved theorems that relate a particular operational semantics with a particular denotational semantics for the PCF language [75]. These theorems are now known as adequacy and full-abstraction.

These theorems provided an easy-to-replicate methodology for judging the validity of a denotational semantics with respect to an operational semantics. Unfortunately there were many useful features in programming languages that were out of reach from denotaitonal methods, local state being a notorious example. In part due to these difficulties, Wright and Felleisen [92] have proposed

a purely syntactic way of proving type soundness, a property stating that "well-typed programs cannot 'go wrong"', where "go wrong" means, roughly, that a program will encounter a type error, such as using an integer as a function, in the course of its execution. Their thesis was that type soundness can be established by progress and preservation theorems. At a high-level, progress means that programs are either fully executed, i.e. they are a value, or they may continue computing; preservation means that well-typedness is stable under computation steps.

This paper has been extremely influential and is partially responsible for the "denotational semantics winter" of the 90's and 00's. Type soundness theorems have found other applications, such as reasoning about information flow control [77] or concurrent memory writing [69]. Though these techniques have been widely used, they have serious methodological flaws. Soundness results based on operational semantics are extremely brittle when it comes to changes of application domain and language features. There are few theorems that can be reused in future developments and, more importantly, these techniques constrains language designers to reasoning about the metatheory of their language in terms of mostly progress and preservation properties. Furthermore, proofs based on operational semantics can get long to the point where no single person can keep track of it all. Indeed, it is not uncommon for the whole formalism of a paper to span over 50 pages of minute, never-seen before, delicate, inductive arguments.

A Denotational Renaissance For many kinds of programming languages, the drawbacks mentioned above, either by inertia or by other reasons, did not motivate researchers to look for other kinds of solutions. However, for languages that make use of probability, denotational semantics is the most natural way of

interpreting programs: reasoning about probabilistic programs is equivalent to reasoning about probability theory, not syntactic properties of programs.

Ever since the origins of the study of the denotational semantics of programming languages, probabilistic semantics has played a central role in the development of the field. Pioneering works [56, 80] have paved the way to the more sophisticated probabilistic semantics we have nowadays. Many of these semantic insights have been used to reason about cryptographic protocols and, after the meteoric rise of machine learning, computational statistics. These applications show how seamless using these semantics can be: once we have interpreted a language denotationally, we can reason about programs exactly how we would reason about their corresponding mathematical objects, benefiting from potentially decades-worth of useful lemmas and theorems.

Furthermore, as semantics of languages get more complicated, purely operational techniques get unwieldy and not expressive enough, which motivated the development of mixed operational/denotational techniques such as step-indexed logical relations [32, 17], one of the technical foundations of the Iris separation logic [53].

Though there is much that we can do with existing semantics, there are still important foundational questions that must be addressed. This is specially true for probabilistic semantics, where simple features such as higher-order functions in the presence of continuous distributions have only recently been denotationaly addressed [46, 37, 26]. The goal of this thesis is to contribute to our foundational understanding of denotational semantics for probabilistic reasoning by providing a formal connection between two important families of probabilistic denotational semantics.

Markov Kernels and Linear Operators

There are two classes of models of probabilistic programming — in its broad sense — that have found numerous applications: models based on linear logic and models based on Markov kernels. Since each kind of semantics has peculiarities that make them more or less adequate to give semantics to expressive programming languages, it is an important theoretical question to understand how these classes of models are related.

Linear Logic for Probabilistic Semantics The models of linear logic that have been used to give semantics to probabilistic languages are usually based on categories of vector spaces where programs are denoted by linear operators. We highlight two of them:

- Ehrhard et al. [37], Ehrhard [36], Danos and Ehrhard [28] have defined models of linear logic with probabilistic primitives and have used the translation of intuitionistic logic into linear logic $A \to B = !A \multimap B$, where $!A$ is the exponential modality, to give semantics to a stochastic λ-calculus.
- Dahlqvist and Kozen [26] have defined an imperative, higher-order, linear probabilistic language and added a type constructor ! to accommodate non-linear programs.

The main advantage of models based on linear logic is that programs are denoted by linear operators between spaces of distributions, a formalism that has been extensively used to reason about stochastic processes, as illustrated by Dahlqvist and Kozen who have used results from ergodic theory to reason about a Gibbs sampling algorithm written in their language, and by Clerc et al. who have shown how Bayesian inference can be given semantics using adjoint of linear operators [23].

Unfortunately, these insights are hard to realize in practice, since languages based on linear logic enforce that variables must be used exactly once, making it hard to use it as a programming language. The usual way linear logic deals with this limitation is through the ! modality which allows variables to be reused.

The problem with the exponential modality, when it comes to probabilistic programming, is that they are usually difficult to construct, do not have any clear interpretation in terms of probability, making the linear operator formalism not applicable anymore and, more operationally, through its connections with call-by-name (CBN) semantics [60], makes it mathematically hard to reuse sampled values.

Ehrhard et al. have found a way around this problem by introducing a call-by-value (CBV) let operator that allows samples to be reused [37, 90]. In the discrete case this operator is elegantly defined by a categorical argument which is unknown to scale to the continuous case, which they deal with by making use of an ad-hoc construction that is unclear if it can be generalized to other models of linear logic. Therefore, our current understanding of models of linear logic does not provide a uniform way of reusing samples.

The difference between CBV and CBN can be illustrated by the program $\mathsf{let}\ x = \mathsf{coin}\ \mathsf{in}\ x + x$, where coin is a primitive that outputs 0 or 1 with equal probability. In the CBN semantics each use of x corresponds to a new sample from coin, whereas in the CBV semantics the coin is only sampled once.

A subtler problem of probabilistic models based on linear logic is that they are ill-equipped to program with joint distributions. For instance, the language proposed by Ehrhard et. al can be easily extended with product types which, under their semantics, would make the type $\mathbb{R} \times \mathbb{R}$ be interpreted as $\mathcal{M}\mathbb{R} \times \mathcal{M}\mathbb{R}$, where $\mathcal{M}\mathbb{R}$ is the set of distributions over $\mathbb{R}$. This is isomorphic to the set of

independent distributions over $\mathbb{R}^2$. Dahlqvist and Kozen deal with this issue by adding primitive types $\mathbb{R}^n$ to their language which are interpreted as the set of joint distributions over $\mathbb{R}^n$. However, since they are not defined using the type constructors provided by the semantic domain, programs of type $\mathbb{R}^n$ can only be manipulated by primitives defined outside the language.

Markov Kernel Semantics Markov kernels are a generalization of transition matrices, i.e. functions that map states to probability distributions over them. They are appealing from a programming languages perspective because their programming model is usually captured by monads and Kleisli arrows, a common abstraction in programming languages semantics, and have been extensively used to reason about probabilistic programs [30, 82, 10]. By being related to monadic programming they differ from their linear operator counterpart by being able to naturally capture a call-by-value semantics which, as we argued above, is the most natural one for probabilistic programming.

Unfortunately, even though these semantics can be generalized to continuous distributions, they are notoriously brittle when it comes to higher-order programming. Only recently, with the introduction of quasi Borel spaces [46] and its probability monad, it is possible to give a kernel-centric semantics to higher-order probabilistic programming with continuous distributions.

However, due to quasi Borel spaces being a different foundation to probability theory, it is unclear which theorems and theories can be generalized to higher-order. For instance, martingale theory has been used in computer science to reason about termination of probabilistic programs [21, 62, 48]. In order to generalize these ideas to higher-order functions it would be necessary to define a quasi Borel version of martingales and prove appropriate versions of the main theorems from martingale theory, a non-trivial task.

Combining Both Kinds of Semantics Though both styles of semantics provide insights into how to interpret probabilistic programming languages (PPL), it is still too early to claim that we have a "correct" semantics which subsumes all of the existing ones. Both approaches mentioned above have their advantages and drawbacks.

In this thesis we shed some light into their formal connection by showing that it is possible to use both styles of semantics to interpret a linear calculus that has higher-order functions, looser linearity restrictions, a uniform way of dealing with sample reuse and better syntax for programming joint distributions while still being close to their kernel and linear operator counterparts. Interestingly, we identify the joint distribution problem described above to be a consequence of linear logic requiring the non-linear product to be Cartesian. In order to tackle this problem we build on categorical semantics of linear logic and on recent work on CD/Markov categories, a suitable categorical generalization of Markov kernels defined using semicartesian products.

We bridge the gap between these semantics by noting that the regular resource interpretation of linear logic, i.e. $A \multimap B$ being equivalent to "by using one copy of A I get one copy of B" is too restrictive an interpretation for probabilistic programming. Instead, we should think of usage as being equivalent to sampling. Therefore the linear arrow $A \multimap B$ should be thought of as "by sampling from A once I get B", which is the computational interpretation of Markov kernels.

We make this interpretation concrete through a multilanguage approach: we have one language that programs Markov kernels, a second language that programs linear operators and add syntax that transports programs from the former to the latter. To justify the viability of our categorical framework we show

how existing probabilistic semantics are models to our language, we show how, under mild conditions, this semantics can be generalized to commutative effects and we show that this core calculus can be adapted to define type theories for reasoning about conditional independence in probabilistic programs.

Summary of contributions We separate the main contributions of this thesis into three chapters. Before we present them we start with a background section that serves as a mostly self-contained reference to the math used throughout this thesis. Following it, the thesis is organized as follows:

- In Chapter 3 we present $\lambda^{\text{LL}}_{\text{MK}}$, a two-level calculus that can program with both a Markov kernel semantics as well as with a linear operator semantics. We define its categorical semantics and, in order to justify its viability, we show how important concrete models from the probabilistic semantics literature are actually instances of our abstract semantics. We also show how many of the ideas behind $\lambda^{\text{LL}}_{\text{MK}}$ are valid for programming with commutative effects.
- In Chapter 4 we define a new concrete model of linear logic for probabilistic reasoning $\mathbf{PBanLat_1}$ based on perfect Banach lattices, a commonly used abstraction in measure theory. This contribution is significant both for the theory of linear logic, since this model is the first continuous extension of probabilistic coherence spaces (PCS) that preserves the logical connectives up-to isomorphisms[1], as well as to the central theme of this thesis, since we show that $\mathbf{PBanLat_1}$ is a part of a $\lambda^{\text{LL}}_{\text{MK}}$ model. We also extend the core calculus with exponential types and term recursion.

[1] previous approaches either did not account for the classical structure of PCSs [36] or did not show that the connectives are preserved [86]

- In Chapter 5 we justify the applicability of $\lambda^{\mathrm{LL}}_{\mathrm{MK}}$ for reasoning about probabilistic programs by using it as the basis of a type system that can reason about probabilistic independence. Since our semantics is based on quite general categorical machinery, we show that our type system can also reason about an abstract notion of separation of resources, which is meaningful in other effectful contexts such as syntactic control of interference and distributed programming.

The results of this thesis were based on three papers [7, 6, 31].

CHAPTER 2

BACKGROUND

The results from this thesis rely quite heavily on probability theory and, more importantly, on category theory. The goal of this section is to present a mostly self-contained presentation on the mathematics used throughout this dissertation. If readers are familiar with the contents of this section they may skip it and consult it as they see fit.

2.1 Probability Theory

Definition 2.1.1. A (discrete) distribution over a set X is a function $\mu : X \to [0, 1]$ such that $\sum_{x \in X} \mu(x) = 1$.

Transition matrices are one of the simplest abstractions used to model stochastic processes. Given two countable sets A and B, the entry (a, b) of a transition matrix is the probability of ending up in state $b \in B$ whenever you start from the initial state $a \in A$ and every row adds up to 1.

Definition 2.1.2. The category **CountStoch** has countable sets as objects and transition matrices as morphisms. The identity morphism is the identity matrix and composition is given by matrix multiplication.

Though transition matrices are conceptually simple, they can only model discrete probabilistic processes and, in order to generalize them to continuous probability we must use measurable sets and Markov kernels.

Definition 2.1.3. A measurable set is a pair (A, Σ_A), where A is a set and $\Sigma_A \subseteq \mathcal{P}(A)$ is a σ-algebra, i.e. it contains the empty set and it is closed under complements and countable unions.

Definition 2.1.4. A function $f : (A, \Sigma_A) \to (B, \Sigma_B)$ is called measurable if for every $\mathcal{B} \in \Sigma_B$, $f^{-1}(\mathcal{B}) \in \Sigma_A$.

Definition 2.1.5. Let (A, Σ_A) be a measurable space. A probability distribution (A, Σ_A) is a function $\mu : \Sigma_A \to [0, 1]$ such that $\mu(\emptyset) = 0$, $\mu(A) = 1$ and $\mu(\uplus_{i\in\mathbb{N}} A_i) = \sum_{i\in\mathbb{N}} \mu(A_i)$.

Given two measurable sets (A, Σ_A) and (B, Σ_B) it is possible to define a σ-algebra over $A \times B$ generated by the sets $X \times Y$ which we denote by $\Sigma_A \otimes \Sigma_B$, where $X \in \Sigma_A$ and $Y \in \Sigma_B$. Furthermore, every pair of distributions μ_A and μ_B over A and B respectively, can be lifted to a distribution $\mu_A \otimes \mu_B$ over $A \times B$ such that $(\mu_A \otimes \mu_B)(X \times Y) = \mu_A(X)\mu_B(Y)$, for $X \in \Sigma_A$ and $Y \in \Sigma_B$.

Definition 2.1.6. Let (A, Σ_A) and (B, Σ_B) be two measurable spaces. A Markov kernel is a function $f : A \times \Sigma_B \to [0, 1]$ such that

- For every $a \in A$, $f(a, -)$ is a probability distribution.
- For every $\mathcal{B} \in \Sigma_B$, $f(-, \mathcal{B})$ is a measurable function.

Definition 2.1.7. The category **Kern** has measurable spaces as objects and Markov kernels as morphisms. The identity arrow is the function $id_A(a, \mathcal{A}) = 1$ if $a \in \mathcal{A}$ and 0 otherwise and Composition is given by $(f \circ g)(a, \mathcal{C}) = \int f(-, \mathcal{C}) d(g(a, -))$.

Joint distributions are distributions over meaurable spaces $A \times B$. Given a joint distribution μ over $A \times B$, its marginal distribution over S is defined as $\mu_A(\mathcal{A}) = \int_B \mathrm{d}\mu(\mathcal{A}, -)$ with and the second marginal μ_B being similarly defined.

Definition 2.1.8. A distribution μ over $A \times B$ is probabilistically *independent* if it is a product of its marginals μ_A and μ_B, i.e., $\mu(\mathcal{A} \times \mathcal{B}) = \mu_A(\mathcal{A})\mu_B(\mathcal{B})$, $\mathcal{A} \in \Sigma_A$ and $\mathcal{B} \in \Sigma_B$.

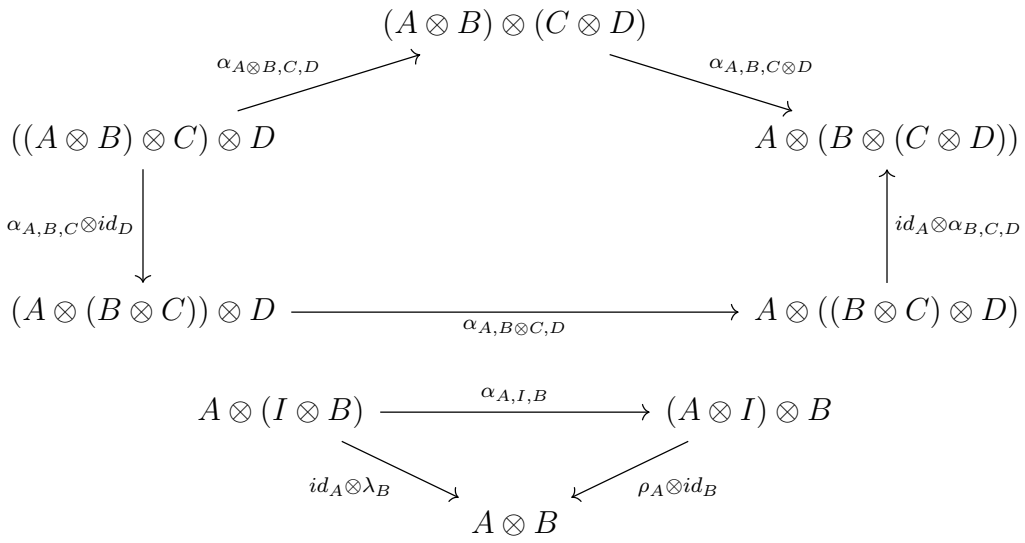


Figure 2.1: Coherence diagrams for monoidal categories

A probability monad can be defined for **Set**. Given a set X, let DX be the set of functions $\mu : X \to [0, 1]$ which are non-zero on finitely many values, and satisfy $\sum_{x \in supp(\mu)} \mu(x) = 1$ [41]. The unit of the monad is given by $\delta(a, b) = 1$ iff $a = b$ and 0 otherwise, while the bind is defined as $\mathsf{bind}(f)(\mu) = \sum_{x \in X} f(x)\mu(x)$.

2.2 Categories for Semantics

2.2.1 Symmetric Monoidal Categories and their Functors

Definition 2.2.1. A symmetric monoidal category is a triple $(\mathbf{C}, \otimes, I)$ equipped with natural ismorphisms $\alpha_{A,B,C} : A \otimes (B \otimes C) \to (A \otimes B) \otimes C$, $\lambda_A : I \otimes A \to A$ and $\rho_A : A \otimes I \to A$ such that the diagrams in Figure 2.1 comute.

What the monoidal axioms enables is reasoning about monoidal categories using *string* diagrams.

In order to simplify reasoning about morphisms in monoidal categories

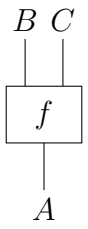
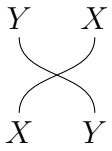


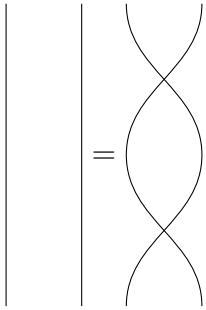
Figure 2.2: String diagram of a morphism $f : A \to B \otimes C$

without worrying about "up-to isomorphisms" concerns created by α, λ and ρ, category theorists have defined *string diagrams*. At a high-level, they are diagrams where morphisms are represented by boxes, wires are the inputs/outputs (objects) and parallel wires (resp. boxes) correspond to the tensor action on objects (resp. morphisms). For simplicity sake, wires corresponding to the unit object are omitted. They are read bottom-to-top and a depiction of a morphism $f : A \to B \otimes C$ is shown in Figure 2.2.

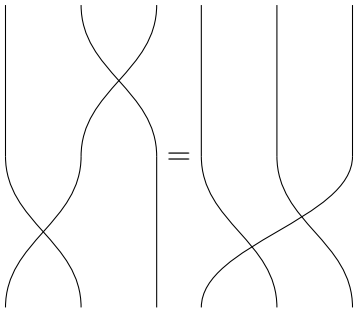
In computer science applications monoidal categories are usually symmetric, meaning that there is a natural isomorphism $b_{A,B} : A \otimes B \to B \otimes A$ satisfying a few axioms. This definition has a nice geometric interpretation in terms of string diagrams. The braid operation $b_{X,Y}$ is represented as the diagram



satisfying the equations



(2.1)



(2.2)

There are many important classes of functors between (symmetric) monoidal categories. In this thesis we will focus mainly on lax monoidal functors.

Definition 2.2.2. Let $(\mathbf{C}, \otimes_{\mathbf{C}}, 1_{\mathbf{C}})$ and $(\mathbf{D}, \otimes_{\mathbf{D}}, 1_{\mathbf{D}})$ be two monoidal categories. We say that a functor $F : \mathbf{C} \to \mathbf{D}$ is *lax monoidal* if there is a morphism $\epsilon : 1_{\mathbf{D}} \to F(1_{\mathbf{C}})$ and a natural transformation $\mu_{X,Y} : F(X) \otimes_{\mathbf{D}} F(Y) \to F(X \otimes_{\mathbf{C}} Y)$ making the diagrams in Figure 2.3 commute. If ϵ and $\mu_{X,Y}$ are isomorphisms we say that F is *strong* monoidal.

2.2.2 Linear Logic

Categorical Semantics

The categorical semantics of linear logic and its variants are extremely well-understood at this point. Even though the exponential initially posed some problems from the categorical point of view, they have been elegantly characterized using monoidal adjunctions. The semantics of the multiplicative fragment of linear logic is defined as a generalization of Cartesian closed categories, where the product structure only has to be symmetric monoidal.

Definition 2.2.3 ([63])**.** A category $\mathbf{C}$ is an LL model if it is symmetric monoidal closed (SMCC), i.e. the functors $- \otimes A$ have a right adjoint $A \multimap -$.

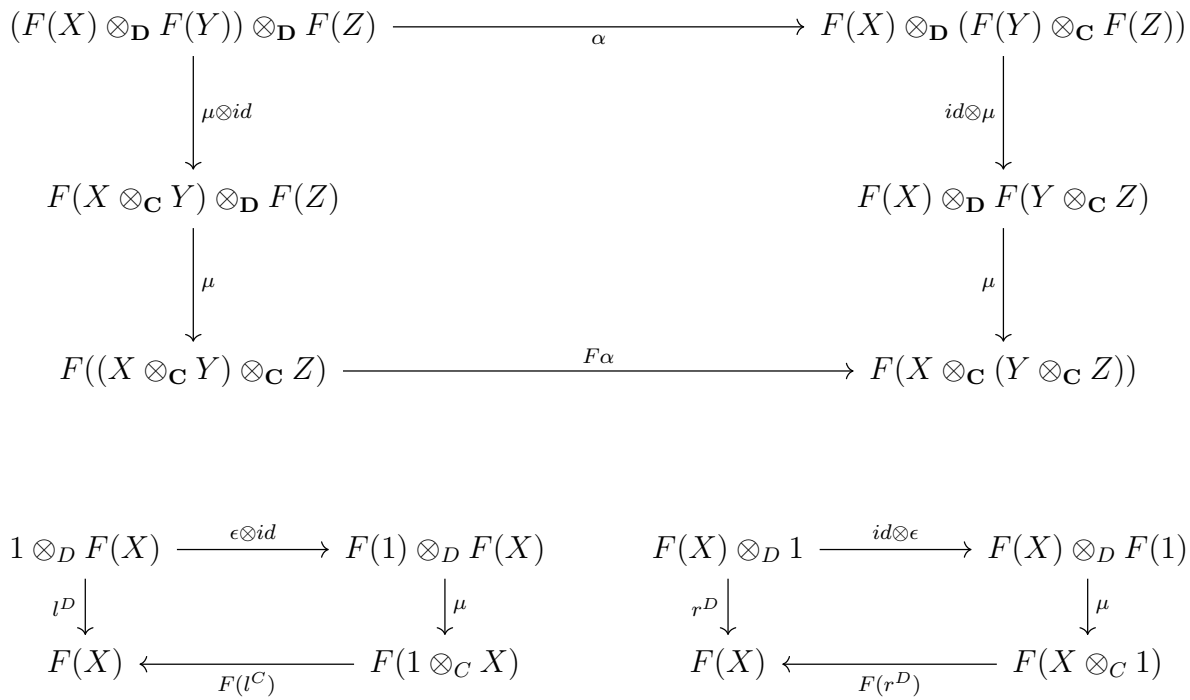


Figure 2.3: Lax monoidal diagrams

We denote the monoidal product as $\otimes$ and the space of (linear) maps between objects X and Y as $X \multimap Y$, $\mathsf{ev} : ((X \multimap Y) \otimes X) \to Y$ is the counit of the monoidal closed adjunction and $\mathsf{cur} : \mathbf{C}(X \otimes Y, Z) \to \mathbf{C}(X, Y \multimap Z)$ is the linear curryfication map. We use the triple $(\mathcal{C}, \otimes, \multimap)$ to denote such models.

In order to interpret the additive connectives we need to assume $\mathbf{C}$ to be (co)Cartesian. To be explicit, a category $\mathbf{C}$ is said to be Cartesian if for every pair of objects A and B there is an object $A \times B$ and morphisms $\pi_1 : A \times B \to A$, $\pi_2 : A \times B \to B$ such that for every pair of morphisms $f : C \to A$, $g : C \to B$ there is a unique morphism $\langle f, g \rangle : C \to A \times B$ such that $\langle f, g \rangle ; \pi_1 = f$ and $\langle f, g \rangle ; \pi_2 = g$. A category is said to be coCartesian if the opposite universal property above is satisfied. In linear logic notation, the Cartesian product models the connective $\&$ while the cocartesian product models the connective $\oplus$.

Definition 2.2.4. A category $\mathbf{C}$ is an MALL model if it is SMC and (co)Cartesian.

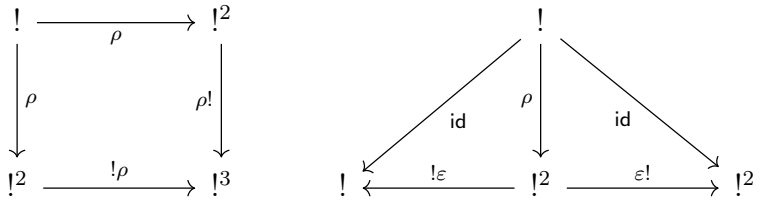


Figure 2.4: Comonad laws

The most complicated aspect of the model, from a categorical point of view, is the exponential. From a computational point of view, the exponential is a type constructor that allows variables to be erased or reused as many times as you want.

Definition 2.2.5. An *exponential* in a categorical model of linear logic is a comonad $(!, \varepsilon, \rho)$, where $\varepsilon_A : !A \multimap A$ and $\rho_A : !A \multimap !!A$, such that every object $!A$ carries a comonoid structure

$$(!A, \underline{d}_A : !A \to !A \otimes !A, \underline{e}_A : !A \to 1)$$

satisfying the Seely isomorphisms $!(A \,\&\, B) \cong !A \otimes !B$ and $!\top \cong 1$.

Definition 2.2.6. A model of intuitionistic linear logic is a symmetric monoidal closed category with (co)Cartesians structures and an exponential comonad.

Something elegant about linear logic is that intuitionistic logic can be embedded into it by using the following type translation

$$[\![A]\!] = A$$

$$[\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \otimes [\![\tau_2]\!]$$

$$[\![\tau_1 \to \tau_2]\!] = ![\![\tau_1]\!] \multimap [\![\tau_2]\!]$$

$$\tau := 1 \mid \tau \multimap \tau \mid \tau \otimes \tau$$

$$t, u := x \mid \mathsf{unit} \mid \lambda x.\, t \mid t\ u \mid t \otimes u \mid \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u$$

$$\Gamma := \cdot \mid x : \tau, \Gamma$$

Figure 2.5: Syntax LL

This is called Girard's translation and, categorically, can be formulated as the coKleisli category for the exponential comonad being Cartesian closed, meaning that it can interpret the simply-typed λ-calculus. A significant consequence of Cartesian closure is that the program equation $(\lambda x.\, t)\ u \equiv t\{u/x\}$ holds for any u, making it a call-by-name (CBN) semantics. This has semantic implications that we will further explore in Section 4.6.1.

In classical linear logic, linear negation is involutive, meaning that $A^{\perp\perp} = A$. Categorically, this is modeled by $*$-*autonomous categories*, which are symmetric monoidal closed categories $\mathbf{C}$ with a functor $(-)^* : \mathbf{C}^{\mathrm{op}} \to \mathbf{C}$ such that every object A is naturally isomorphic to A^{**} and for every three objects A, B, C, there is a natural bijection $\mathrm{Hom}(A \otimes B, C^*) \cong \mathrm{Hom}(A, (B \otimes C)^*)$. Equivalently, a $*$-autonomous category is a symmetric monoidal closed category $\mathbf{C}$ equipped with a *dualizing object* $\perp$ such that for every object A, the unit $\partial_A : A \to (A \multimap \perp) \multimap \perp$ is an isomorphism.

Linear λ-calculus

It is possible to organize models of linear logic in a category **Linear** which have models of linear logic as objects and functors preserving the logical connectives as morphisms, where the "model" here could be any of its fragments

$$\textsc{Axiom}\ \frac{}{x : \tau \vdash x : \tau} \qquad \textsc{Unit}\ \frac{}{\cdot \vdash \mathsf{unit} : 1}$$

$$\textsc{Abstraction}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \multimap \tau_2} \qquad \textsc{Application}\ \frac{\Gamma_1 \vdash t : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash u : \tau_1}{\Gamma_1, \Gamma_2 \vdash t\ u : \tau_2}$$

$$\textsc{Tensor}\ \frac{\Gamma_1 \vdash t : \tau_1 \qquad \Gamma_2 \vdash u : \tau_2}{\Gamma_1, \Gamma_2 \vdash t \otimes u : \tau_1 \otimes \tau_2} \qquad \textsc{LetTensor}\ \frac{\Gamma_1 \vdash t : \tau_1 \otimes \tau_2 \qquad \Gamma_2, x : \tau_1, y : \tau_2 \vdash u : \tau}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u : \tau}$$

Figure 2.6: Typing rules LL

$$\begin{aligned} (\lambda x.\, t)\ u &\equiv t\{u/x\} \\ (\lambda x.\, t\ x) &\equiv t \\ \mathsf{let}\ x_1 \otimes x_2 = t_1 \otimes t_2\ \mathsf{in}\ u &\equiv u\{t_1/x_1\}\{t_2/x_2\} \\ t &\equiv \mathsf{unit} \end{aligned}$$

Figure 2.7: Equational Theory: λ^2_{INI}

(multiplicative, additive, exponential, etc.). For the purpose of this section we will focus on the multiplicative fragment.

When it comes to programming languages we are usually interested in the syntactic model, which for the multiplicative fragment, it is simply the linear λ-calculus depicted in Figures 2.5 and 2.6. The equational theory is similar to the simply-typed case, and is depicted in Figure 2.7, with the basic congruence rules and typing hypothesis omitted.

Definition 2.2.7. The category $\mathbf{Syn}_{LL}$ has types as objects and programs $x : \tau \vdash t : \tau'$, modulo the equational theory, as morphisms.

Note that when modeling realistic programming languages we need to assume a set of primitives (a signature), which we can capture with a set Σ. For

every such set, the syntactic theory extended with typing rules for the primitives can again be made into a category. A well-known theorem is that the syntactic model is initial, see Crole [24] for a proof for the simply-typed case.

Lemma 2.2.8. *For every signature Σ the category $\mathbf{Syn}_{LL}^{\Sigma}$ is initial in $\mathbf{Linear}^{\Sigma}$.*

2.2.3 Monads and their Algebras

Monads. Following seminal work by Moggi [65], effectful computations can be given a semantics via monads. A *monad* over a category $\mathbf{C}$ is a triple (T, μ, η) such that $T : \mathbf{C} \to \mathbf{C}$ is a functor, $\mu_A : T^2A \to TA$ and $\eta_A : A \to TA$ are natural transformations such that $\mu_A \circ \mu_{TA} = \mu_A \circ T\mu_A$, $id_A = \mu_A \circ T\eta_A$ and $id_A = \mu_A \circ \eta_{TA}$.

Another useful, and equivalent, definition of monads requires a natural transformation η_A and a lifting operation $(-)^* : \mathbf{C}(A, TB) \to \mathbf{C}(TA, TB)$ such that objects from $\mathbf{C}$ and morphisms $A \to TB$ form a category, usually referred to as the *Kleisli category* $\mathbf{C}_T$. This category has the same objects as $\mathbf{C}$, and has $Hom_{\mathbf{C_T}}(A, B) = Hom_{\mathbf{C}}(A, TB)$. Kleisli categories are frequently used to give semantics to effectful programming languages.

Monad algebras. Given a monad T, a *T-algebra* is a pair $(A, f : TA \to A)$ such that $id_A = f \circ \eta_A$ and $f \circ \mu_A = f \circ Tf$. A *$T$-algebra morphism* $h : (A, f) \to (B, g)$ is a $\mathbf{C}$ morphism $h : A \to B$ such that $g \circ Th = h \circ f$. T-algebras and morphisms form a category $\mathbf{C}^T$, the *Eilenberg-Moore category*.

Monoidal Monads and Their Algebras

An important theorem from the categorical probability literature is that Markov categories are an abstraction of programming in the Kleisli category of monoidal affine monads, where affinity means that $T1 \cong 1$.

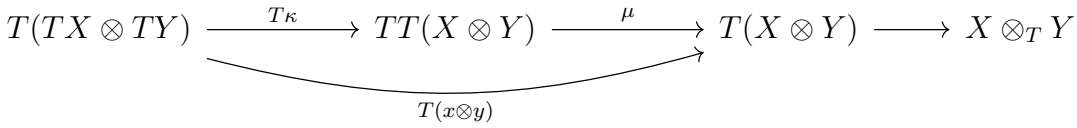


Figure 2.8: Symmetric Monoidal Structure in $\mathbf{C}^T$

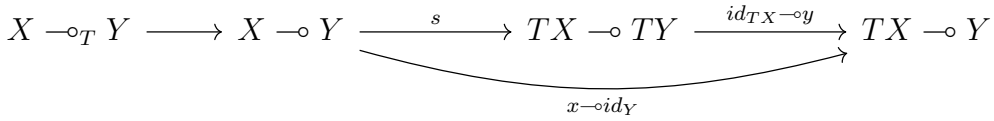


Figure 2.9: Closed Structure in $\mathbf{C}^T$

Theorem 2.2.9 ([41])**.** *Let* $(\mathbf{C}, \times, 1)$ *be a cartesian category and* $T : \mathbf{C} \to \mathbf{C}$ *a monoidal (affine) monad. The Kleisli category* $\mathbf{C}_T$ *is a Markov category.*

The monoidal product $\otimes_T$ of $\mathbf{C}_T$ is $\times$ with unit 1, the copy operation is given by $\Delta_X ; \eta_X : X \to T(X \times X)$ and the deletion operation is given by $T1 \cong 1$ and 1 being terminal.

Furthermore, under certain conditions, the Eilenberg-Moore category $\mathbf{C}^T$ for monoidal monads is symmetric monoidal closed. The monoidal unit is given by TI, the monoidal product is given by the coequalizer depicted in Figure 2.8 and the closed struture is given by the equalizer depicted in Figure 2.9.

Theorem 2.2.10. *Let* $\mathbf{C}$ *be a symmetric monoidal closed category with equalizers, reflexive co-equalizers and* $T : \mathbf{C} \to \mathbf{C}$ *a monoidal monad. The category* $\mathbf{C}^T$ *is also symmetric monoidal closed.*

Even though, in general, in order to define the monoidal product one requires a coequalizer, for our purposes we are only interested in products of the form $TA \otimes_T TB$ which, luckily, are easier to characterize, since the isomorphism $TX \otimes_T TY \cong T(X \otimes Y)$ holds [84].

In this case the lax monoidal transformations $\mu_{X,Y} : TX \otimes_T TY \to T(X \otimes Y)$ and $\epsilon : FI \to FI$ are given by the isomorphisms. Besides, by using the universal properties of coequalizers it is possible to show the equality $\tilde{\alpha}_{TX,TY,TZ} = \alpha_{X,Y,Z}$, where $\tilde{\alpha}$ is the associator for the monoidal product $\otimes_T$.

Theorem 2.2.11. *Let* $\mathbf{C}$ *be a symmetric monoidal category with reflexive co-equalizers and* $T : \mathbf{C} \to \mathbf{C}$ *a monoidal monad. The triple* (ι, μ, ϵ) *is a lax monoidal functor.*

Proof. The proof follows by unfolding the definitions. □

2.2.4 Enriched Category Theory

When first learning about categories, it is useful to think that for every objects A and B, there is a *set* $\mathrm{Hom}(A, B)$. Enriched category theory is the study of category-like structures where the hom sets have more structure.

Definition 2.2.12. Let $\mathcal{V}$ be a monoidal category. A $\mathcal{V}$-category $\mathbf{C}$ is a collection of objects such that for every pair A and B, there is an object $\mathbf{C}(A, B)$ in $\mathcal{V}$ such that there are morphisms in $\mathcal{V}$

- $id : I \to \mathbf{C}(A, A)$
- $\circ : \mathbf{C}(A, B) \otimes \mathbf{C}(B, C) \to \mathbf{C}(A, C)$

such that the usual category axioms hold.

Example 2.2.13. Locally small categories, i.e. categories where the morphisms between any two objects form a set is a $\mathbf{Set}$-category, where the monoidal product is the Cartesian product.

Example 2.2.14. An important example of enriched categories in the PL literature are $\mathbf{O}$-categories, which are categories where the hom-sets are CPOs and morphism composition is continuous on both arguments. It would be more precise to call them $\mathbf{CPO}$-categories.

Note that, in general, an enriched category might not be a category.

Example 2.2.15. Let $\overline{\mathbf{R}}^+$ be the poset category over the extended positive real numbers, i.e. it is category that has positive real numbers as objects and there is a unique arrow between r_1 and r_2 if, and only if, $r_1 \geq r_2$. This category is monoidal with its action on objects being addition and the unit 0. In this case a $\overline{\mathbf{R}}^+$-category consists of a collection of objects and to every pair of objects A and B we can associate a real number $d(A, B)$ such that $d(A, A) \leq 0$ and $d(A, B) + d(B, C) \leq d(A, C)$. This is basically a metric space with the exception that the distance between every point to itself might not be 0 and the distance might not be symmetric. In the category theory literature these enriched categories are called Lawvere metric spaces.

Example 2.2.16. If $\mathbf{C}$ is a monoidal closed category, then it is enriched over itself. The collection of objects is the same as $\mathbf{C}$'s and to every pair of objects A and B we associate the $\mathbf{C}$ object $A \multimap B$, where the identity and composition morphisms are given by the monoidal closed structure.

Something interesting about enriched categories is that many constructions and definitions from basic category theory can be generalized to the enriched setting.

Definition 2.2.17. A $\mathcal{V}$-symmetric monoidal category $\mathbf{C}$ is a collection of objects $ob(\mathbf{C})$, a collection of $\mathcal{V}$ objects $\mathbf{C}(A, B)$ for every $A, B \in ob(\mathbf{C})$, a distinct object $I_{\mathbf{C}}$ and a binary operation over objects $\otimes$ such that there are $\mathcal{V}$ morphisms $\mathbf{C}(A_1, B_1) \otimes \mathbf{C}(A_2, B_2) \to \mathbf{C}(A_1 \otimes A_2, B_1 \otimes B_2)$, $I \to \mathbf{C}(A \otimes B, B \otimes A)$, $I \to \mathbf{C}(A \otimes (B \otimes C), (A \otimes B) \otimes C)$ and $I \to \mathbf{C}(I_{\mathbf{C}} \otimes A, A)$ satisfying the axioms of symmetric monoidal categories.

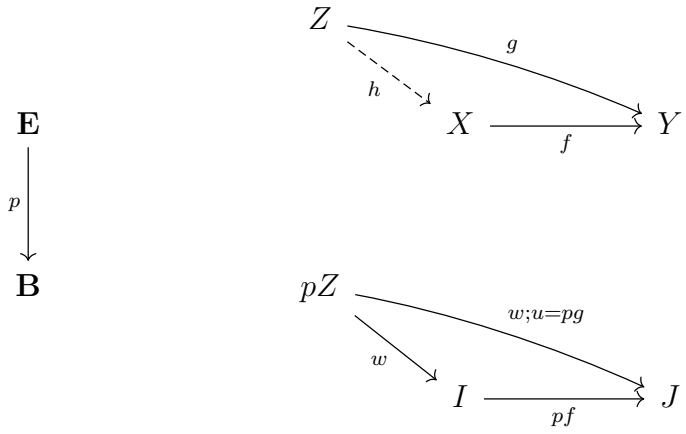


Figure 2.10: Cartesian morphism

2.2.5 Fibered Category Theory

Something quite common in type systems is that we are usually presenting programs "with respect to" its free variables. Categorically, this is represented using fibered categories, which are also referred to as relative categories.

Definition 2.2.18 ([51], Chapter 1)**.** Let $p : \mathbf{E} \to \mathbf{B}$ be a functor,

- $f : X \to Y$ a morphism in $\mathbf{E}$ and $u : I \to J$ a morphism in $\mathbf{B}$ such that $fp = u$. The morphism f is said to be Cartesian over u if for every $g : Z \to Y$ for which one has $pg = w; u$ for some $w : pZ \to I$ there is a unique $h : Z \to X$ such that $h; f = g$, as depicted in Figure 2.10.
- The functor p is said to be a fibration if for every $Y \in \mathbf{E}$ and $u : I \to pY$ there is a Cartesian morphism $f : X \to Y$ over it.

The way fibrations are usually used in computer science is when the category $\mathbf{B}$ corresponds to a category of contexts and $\mathbf{E}$ corresponds to a category of programs. Something useful about fibrations is the notion of fibered category.

Definition 2.2.19. Let $p : \mathbf{E} \to \mathbf{B}$ be a fibration and $I \in \mathbf{B}$. The fiber category $\mathbf{E}_I$ is the subcategory of $\mathbf{E}$ such that its objects are $p^{-1}(I)$ and the morphisms are

the morphisms in $\mathbf{E}$ that are mapped to id_I.

This shows that for every fibration there is a mapping $\mathbf{B} \to \mathbf{Cat}$, where $\mathbf{Cat}$ is the category of small categories and functors. In general, however this construction is not a functor, it is only a pseudofunctor. What is surprising is that given a pseudofunctor $F : \mathbf{B} \to \mathbf{Cat}$ it is possible to recover a fibration, this is called the Grothendieck construction.

Definition 2.2.20. Let $F : \mathbf{B} \to \mathbf{Cat}$ be a pseudofunctor, the Grothendieck construction $\Gamma(F)$ is the category with

- pairs (b, x), where $b \in \mathbf{B}$ and $x \in F(b)$ as objects
- morphisms $(b_1, x_1)to(b_2, x_2)$ are pairs (f, g), where $f : b_1 \to b_2$ and $g : F(f)(x_1) \to x_2$.

Theorem 2.2.21. *The forgetful functor $\Gamma(F) \to \mathbf{B}$ is a fibration.*

Example 2.2.22 ([51] §1.3)**.** The most useful example of a fibration for our purposes is the simple fibration. Let $\mathbf{C}$ be a Cartesian category and $s(\mathbf{C})$ is the category with pairs (I, X) as objects and morphisms are a pair (f, g), where $f : I \to J$ and $g : I \times X \to Y$. The functor $(I, X) \mapsto I$, $(f, g) \mapsto f$ is a fibration. This fibration can be used to give semantics to the simply-typed λ-calculus

2.3 CD and Markov Categories

The field of categorical probability was developed in order to get a more conceptual understanding of Markov kernels. One of its cornerstone definitions are those of CD and Markov categories which are categories where objects are abstract sample spaces, morphisms are abstract Markov kernels and every object

has "contraction" and "weakening" morphisms which correspond to duplicating and discarding a sample, respectively, without adding any new randomness.

More concretely, morphisms in CD categories are abstract unnormalized Markov kernels and morphisms in Markov categories are normalized Markov kernels.

Basic definitions Something quite nice about the theory of Markov categories is that much of the reasoning can be done exclusively in terms of string diagrams. We start by presenting the textual definition of Markov categories.

Definition 2.3.1 (CD and Markov category [41])**.** A CD category is a symmetric monoidal category $(\mathbf{C}, \otimes, 1)$ in which every object A comes equipped with a commutative comonoid structure, denoted by $\mathsf{copy}_X : X \to X \otimes X$ and $\mathsf{delete}_X : X \to 1$, where copy satisfies

$$\mathsf{copy}_{X\otimes Y} = (id_X \otimes b_{Y,X} \otimes id_Y) \circ (\mathsf{copy}_X \otimes \mathsf{copy}_Y),$$

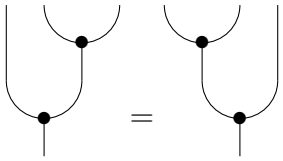
If delete_X is natural, then $\mathbf{C}$ is Markov.

where $b_{Y,X}$ is the natural isomorphism $Y \otimes X \cong X \otimes Y$. The deletion operation can be used to define projection morphisms $\pi_1 : A \otimes B \to A$ and $\pi_2 : A \otimes B \to B$, but it is not Cartesian because the equation $(\pi_1 \circ f, \pi_2 \circ f) = f$ does not hold in general which, intuitively, corresponds to the fact that joint distributions might be correlated.

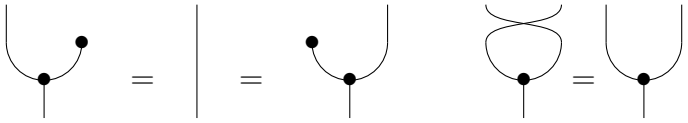
From a graphical point of view, the structural morphisms are represented as

(2.3)

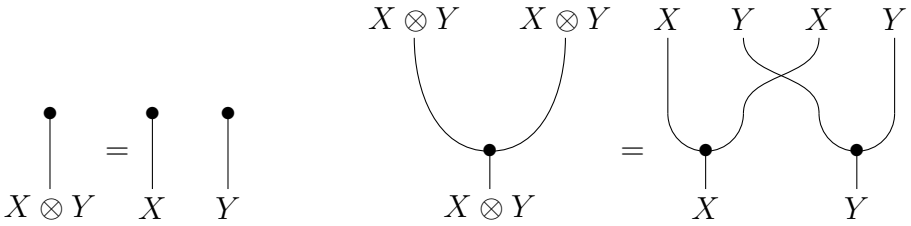
While the CD axioms are represented as



(2.4)

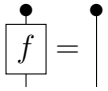


(2.5)



(2.6)

In case we also have naturality of the deletion operation, we must add the axiom



(2.7)

Something appealing about this formalism is that many of the common categories used to reason about probability are instances of Markov categories.

Theorem 2.3.2 ([41]). **CountStoch** *is a Markov category.*

The monoidal product is given by the Cartesian product and the monoidal unit is the singleton set. The copy_X morphism is the matrix $X \times X \times X \to [0, 1]$ which is 1 in the positions (x, x, x) and 0 elsewhere, and the delete_X morphism is the constant 1 matrix indexed by X.

If we consider the variant of **CountStoch** where morphisms must be sub-transition matrices, then it is a CD category but not Markov.

There is useful full subcategory of **CountStoch**, **FinStoch**, where the objects are finite sets.

Theorem 2.3.3 ([41]). **Kern** *is a Markov category.*

This category is the continuous generalization of **CountStoch** and the monoidal product is the Cartesian product with the *product* σ-algebra and the monoidal unit is the singleton set $\{*\}$. The copy_X morphism is the Markov kernel $\mathsf{copy}_X : X \times \Sigma_X \otimes \Sigma_X \to [0,1]$ such that $\mathsf{copy}_X(x, S \times T) = 1$ if $x \in S \cap T$ and 0 otherwise. Its delete morphism is simply the function that given any element in X, returns the function which is 1 on the measurable set $\{*\}$ and 0 on the empty measurable set.

As we can see, morphisms in Markov categories should be thought of operations that given an input, outputs a probability distribution and, when there are no inputs, i.e. morphisms $1 \to A$, they correspond to probability distributions over A. This is reminiscent of monadic programming, where the Kleisli morphisms $A \to TB$, where T is a monad, play an important role in the categorical semantics of effectful programming languages. This intuition can be formally justified by the following theorem:

Theorem 2.3.4. *If* $\mathbf{C}$ *is a Cartesian category and* $T : \mathbf{C} \to \mathbf{C}$ *a commutative monad then the Klesli category* $\mathbf{C}_T$ *is a CD category. Furthermore, if* $T1 \cong 1$*, i.e it is an affine monad, then* $\mathbf{C}_T$ *is a Markov category.*

This is a very powerful and useful result. Probability monads are usually either commutative or commutative and affine, meaning that when you prove a theorem about Markov categories you are automatically proving properties about the Kleisli categories of commutative affine monads. Furthermore, as we have explained in Section 2.2.3, these monads are deeply related to commutative effects, meaning that Markov categories can also be used to reason about them in great generality.

We are now ready to present how concepts from probability theory can be represented in Markov categories.

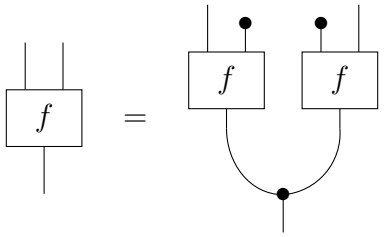
Definition 2.3.5. Given a distribution $\psi : 1 \to X \otimes Y$, its first marginal ψ_X is $\psi; (id_X \otimes \mathsf{delete}_Y)$. The second marginal is defined symmetrically.

With this, we can define independent distributions.

Definition 2.3.6. A state $\psi : 1 \to X \otimes Y$ is independent if we have the following equality $\psi = \psi_X \otimes \psi_Y$.

This definition can also be generalized to deal with conditional independence.

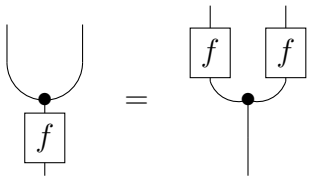
Definition 2.3.7. A morphism $f : X \to Y \otimes Z$ is said to be conditionally independent if the following equation holds:



(2.8)

It is easy to see that it is possible to recover the original definition of independence by choosing $X = 1$. Next, we present deterministic morphisms.

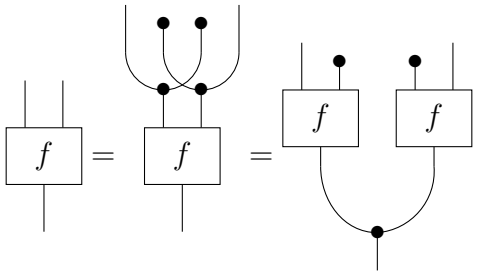
Definition 2.3.8. A morphism $f : X \to Y$ is deterministic if it commutes with copy, i.e.



(2.9)

Lemma 2.3.9. *Let $f : X \to Y \otimes Z$ be a deterministic morphism, then it is conditionally independent.*

Proof. The proof follows by simple string-diagrammatic reasoning:



(2.10)

□

Theorem 2.3.10 ([67])**.** *Let* $\mathbf{C}$ *be a Markov category and* $\mathbf{C}_{det}$ *its subcategory of deterministic morphisms.* $\mathbf{C}_{det}$ *is Cartesian and it is the largest Cartesian subcategory of* $\mathbf{C}$*.*

Deterministic morphisms interact nicely with conditionally independent morphisms.

Lemma 2.3.11. *If* $f : X \to Y \otimes Z$ *is a conditionally independent morphism and* $g : X' \to X$ *is a deterministic morphism then* $g; f$ *is also conditionally independent.*

This lemma will play an important role in designing a type system for reasoning about conditional independence in Chapter 5.

Definition 2.3.12. A Markov category is said to have conditional distributions if for every state $f : 1 \to X \otimes Y$ there is a morphism $f_{|X} : X \to Y$ such that $f = f; ((copy; (id \otimes f_{|X})) \otimes del)$

Example 2.3.13. The category **FinStoch** has conditional distributions by using Bayes' theorem.

Once again, we can generalize this definition to morphisms with arbitrary codomains.

$$\tau := 1 \mid X \mid \tau \times \tau$$

$$M, N := x \mid \mathsf{unit} \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid (M, N) \mid \pi_1 M \mid \pi_2 N \mid f(M)$$

$$\Gamma := \cdot \mid x : \tau, \Gamma$$

Figure 2.11: Syntax MK

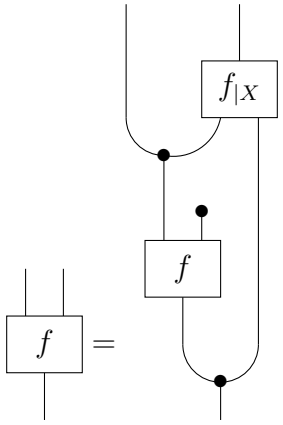
$$\textsc{Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Unit}\ \frac{}{\Gamma \vdash \mathsf{unit} : 1}$$

$$\textsc{Let}\ \frac{\Gamma \vdash t : \tau_1 \qquad \Gamma, x : \tau_1 \vdash u : \tau}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : \tau} \qquad \textsc{Primitive}\ \frac{\Gamma \vdash M : \tau_1 \qquad f \in \mathbf{M}(\tau_1, \tau_2)}{\Gamma \vdash f(M) : \tau_2}$$

$$\textsc{Pair}\ \frac{\Gamma \vdash t : \tau_1 \qquad \Gamma \vdash u : \tau_2}{\Gamma \vdash (t, u) : \tau_1 \times \tau_2} \qquad \textsc{Proj1}\ \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 t : \tau_1} \qquad \textsc{Proj2}\ \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2 t : \tau_2}$$

Figure 2.12: Typing rules MK

Definition 2.3.14. A Markov category is said to have conditional if for every morphism $f : X \to Y \otimes Z$ there is a morphism $f_{|Y} : X \otimes Y \to Z$ such that



(2.11)

Categories of Models and Syntactic Model As it is usually the case with categorical semantics, it is possible to organize them in their own categories. We

$$V_1, V_2 \coloneqq x \mid \mathsf{unit} \mid (V_1, V_2) \mid \pi_i V \mid \mathsf{let}\ x = V_1\ \mathsf{in}\ V_2$$

Figure 2.13: Value grammar for Markov categories

start by defining Markov functors.

Definition 2.3.15. Let $\mathbf{C}$ and $\mathbf{D}$ be Markov categories, a functor $F : \mathbf{C} \to \mathbf{D}$ is a Markov functor if it preserves the monoidal product and the natural transformations copy and delete.

Definition 2.3.16. The category **Markov** has Markov categories as objects and Markov functors as morphisms.

Something useful about Markov categories is that every Markov category has a simple internal language, i.e. a syntax and typing rules that can soundly write morphisms in an arbitrary Markov category, as depicted in Figures 2.11 and 2.12. Once again we are assuming the existence a signature Σ, which gives rise to the category $\mathbf{Markov}^{\Sigma}$, where the objects are Markov categories with a distinguished morphism for every operation in Σ and morphisms are functors that preserve the Markov structure and the signature.

We have presented the string diagrammatic way of representing morphisms in Markov categories. As it was noted by Stein [88], something useful about these diagrams is that they hide the annoying coherence morphisms existing in monoidal categories but keep the structural morphisms such as copy and deletion explicit. Programming languages also provide a way of representing morphisms in categories, except that then even the structural morphisms are kept implicit. We present the internal language of an arbitrary Markov category $\mathbf{M}$ in Figures 2.11 and 2.12.

Once again it is possible to define the syntactic category given a particular signature Σ. The objects are types while morphisms are programs $x : \tau \vdash M : \tau'$

$$
\begin{aligned}
\mathsf{let}\ x = t\ \mathsf{in}\ x &\equiv t \\
\mathsf{let}\ x_1 = x_2\ \mathsf{in}\ t &\equiv t\{x_2/x_1\} \\
\mathsf{let}\ y = (\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 &\equiv \mathsf{let}\ x = M_1\ \mathsf{in}\ (\mathsf{let}\ y = M_2\ \mathsf{in}\ M_3) \\
\\
\mathsf{let}\ x = M_1\ \mathsf{in}\ (\mathsf{let}\ y = M_2\ \mathsf{in}\ M_3) &\equiv \mathsf{let}\ y = M_2\ \mathsf{in}\ (\mathsf{let}\ x = M_1\ \mathsf{in}\ M_3) \\
\mathsf{let}\ x = V\ \mathsf{in}\ t &\equiv t\{V/x\} \\
\mathsf{let}\ x = u\ \mathsf{in}\ t &\equiv t\{u!x\} \\
\mathsf{let}\ x = t\ \mathsf{in}\ f(x) &\equiv f(t) \\
\\
\mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{let}\ y = u\ \mathsf{in}\ (x, y) &\equiv (t, u) \\
\pi_1(x_1, x_2) &\equiv x_1 \\
\pi_2(x_1, x_2) &\equiv x_2 \\
(\pi_1(x), \pi_2(x)) &\equiv x \\
\mathsf{unit} &\equiv x
\end{aligned}
$$

Figure 2.14: Equational Theory for Markov Categories

modulo the equational theory presented in Figure 2.14, where the congruence rules and typing hypothesis were omitted. We call this category $\mathbf{Syn}_{MK}^{\Sigma}$. Note that in order to define the equational theory we need the notion of values, as depicted in Figure 2.13, and we are using the notation $t\{u!x\}$ from Stein [88] for substitution when the variable x occurs at most once in t. The following lemma is straightforward but tedious to prove by induction on typing derivations.

Lemma 2.3.17. *For every signature Σ the category $\mathbf{Syn}_{MK}^{\Sigma}$ is the initial object.*

CHAPTER 3

A HIGHER-ORDER LANGUAGE FOR MARKOV KERNELS AND LINEAR OPERATORS

3.1 Introduction

This chapter is based on Azevedo de Amorim [7]

Probabilistic primitives have been a standard feature of programming languages since the 70s. At first, randomness was mostly used to program so called random algorithms, i.e. algorithms that require access to a source of randomness. Recently, however, with the rise of computational statistics and machine learning, randomness is also used to program statistical models and inference algorithms.

Programming languages researchers have seen this rise in interest as an opportunity to further study the interaction of probability and programming languages, establishing it as an active subfield within the PL community.

One of the main goals of this subfield is giving semantics to programming languages that are both expressive in the regular PL sense as well as in its abilities to program with randomness. One particular difficulty is that the mathematical machinery used for probability theory, i.e. measure theory, does not interact well with higher-order functions [4].

Our contributions are:

- We define a multi-language syntax that can program both Markov kernels as well as linear operators.(§3.2)
- We define its categorical semantics and prove certain interesting equations satisfied by it. (§3.3)

- We show that our semantics is already present in existing models for discrete and continuous probabilistic programming. (§3.4)
- We show how our semantics can be generalized to commutative effects. (§3.5)

3.2 Syntax

In this section we will design a syntax that reflects the fact that linearity corresponds to sampling, not variable usage. We achieve this by making use of a multi-language semantics that enables the programmer to transport programs defined in a Markov kernel-centric language (MK) to a linear, higher-order, language (LL).

Our thesis is that in the context of probabilistic programming, linear logic, through its connection with linear algebra, departs from many of its Computer Science applications of enforcing syntactic invariants and, instead, provides a natural mathematical formalism to express ideas from probability theory, as shown by Dahlqvist and Kozen [26].

Therefore, since many probabilistic programming constructs, such as Bayesian inference and Markov kernels, can be naturally interpreted in linear logic terms, we believe that our calculus allows the user to benefit from the insights linearity provides to PPL while unburdening them from worrying about syntactic restrictions by making it possible to also program using kernels.

We use standard notation from the literature: $\Gamma \vdash t : \tau$ means that the program t has type τ under context Γ, $t\{u/x\}$ means substitution of u for x in t and $t\{\overrightarrow{u}/\overrightarrow{x}\}$ is the simultaneous substitution of the term list $\overrightarrow{u}$ for a variable list $\overrightarrow{x}$ in t.

Both languages will be defined in this section and, for presentation's sake, we are going to use orange to represent MK programs and purple to represent LL programs.

3.2.1 A Markov Kernel Language

We need a language to program Markov kernels. Since we are aiming at generality, we are assuming the least amount of structure necessary to make the language non-degenerate. As such we will be working with the internal language of Markov categories, as presented in Figure 2.11 and Figure 2.12. Note that we are implicitly assuming a set of primitives for the functions f.

By construction, every Markov category can interpret this language, with types being interpreted as

$$[\![1]\!] = 1$$

$$[\![\tau_1 \times \tau_2]\!] = [\![\tau_1]\!] \times [\![\tau_2]\!]$$

and the contexts are interpreted using $\times$ over the interpretation of the types. However, as it stands, it is not very expressive, since it does not have any probabilistic primitives nor does it have any interesting types since $1 \times 1 \cong 1$.

When working with concrete models (c.f. Section 3.4) we can extend the language with more expressive types as well as with concrete probabilistic primitives. For instance, in the context of continuous probabilities we could add a $\mathbb{R}$ datatype and a $\cdot \vdash \mathsf{uniform} : \mathbb{R}$ uniform distribution primitive.

Note that even though this language does not have any explicit sampling operators, this is implicitly achieved by the let operator. For instance, the program $\mathsf{let}\ x = \mathsf{uniform}\ \mathsf{in}\ x + x$ samples from a uniform distribution, binds the result to the variable x and adds the sample to itself.

3.2.2 A Linear Language

Our second language is a linear simply-typed λ-calculus, with the usual typing rules shown in Figure 2.6, which can be interpreted in every symmetric monoidal closed category, with types interpreted by

$$[\![1]\!] = 1$$

$$[\![\tau_1 \otimes \tau_2]\!] = [\![\tau_1]\!] \otimes [\![\tau_2]\!]$$

$$[\![\tau_1 \multimap \tau_2]\!] = [\![\tau_1]\!] \multimap [\![\tau_2]\!]$$

and the contexts are interpreted using $\otimes$ over the interpretation of the types. Once again, we are aiming at generality instead of expressivity. In a concrete setting it would be fairly easy to extend the calculus with a datatype $\mathbb{N}$ for natural numbers and probabilistic primitives such as $\cdot \vdash \mathsf{coin} : \mathbb{N}$ that flips a fair coin.

The idea behind the particular linear logic models that we are interested in is that, by integration, Markov kernels can be seen as linear operators between vector spaces of probability distributions. As such, an LL program $x : \mathbb{N} \vdash_{LL} t : \mathbb{N}$ will be denoted by a linear function between distributions over the natural numbers. Therefore, from a programming point of view, variables are placeholders for probability distributions, i.e. computations, not values, and sampling occurs when variables are used.

3.2.3 Combining Languages

The main drawback of the linear calculus above is that the syntactic linearity restriction makes it hard to program with it, while the main drawback of the Markov language is that it does not have higher-order functions. In this section we will show how we can combine both language so that we get a calculus,

$$\tau ::= 1 \mid \tau \times \tau$$
$$\underline{\tau} ::= 1 \mid \mathcal{M}\tau \mid \underline{\tau} \multimap \underline{\tau} \mid \underline{\tau} \otimes \underline{\tau}$$

$$\begin{aligned} M, N ::= {} & x \mid \mathsf{unit} \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid f(M) \\ & \mid (M, N) \mid \pi_1 M \mid \pi_2 M \\ t, u ::= {} & x \mid unit \mid \lambda x.\, t \mid t\ u \mid t \otimes u \mid \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u \\ & \mid \mathsf{sample}\ t_i\ \mathsf{as}\ x_i\ \mathsf{in}\ M \end{aligned}$$

Figure 3.1: Syntax LL+MK

which we call $\lambda_{\text{MK}}^{\text{LL}}$, with looser linearity restrictions while still being higher-order.

As we will show in Section 3.4, when looking at concrete models for these languages we can see that the semantic interpretations of variables in both languages are completely different: in the MK language variables should be thought of as values, i.e. the values that were sampled from a distribution, whereas in the LL language, variables of ground type are distributions. In order to bridge these languages we must use the observation that Markov kernels — i.e. open MK terms — have a natural resource-aware interpretation of being "sample-once" stochastic processes and, by integration, can be seen as linear maps between measure spaces — i.e. open LL terms. The combined syntax for the language is depicted in Figure 3.1.

We now have a language design problem: we want to capture the fact that every open MK program is, semantically, also an open LL term. The naive typing rule is:

$$\frac{x_1 : \tau_1, \cdots, x_n : \tau_n \vdash_{MK} M : \tau}{x_1 : \mathcal{M}\tau_1, \cdots, x_n : \mathcal{M}\tau_n \vdash_{LL} \mathsf{MK}(M) : \mathcal{M}\tau}$$

The problem with this rule is that it breaks substitution: the variables in the

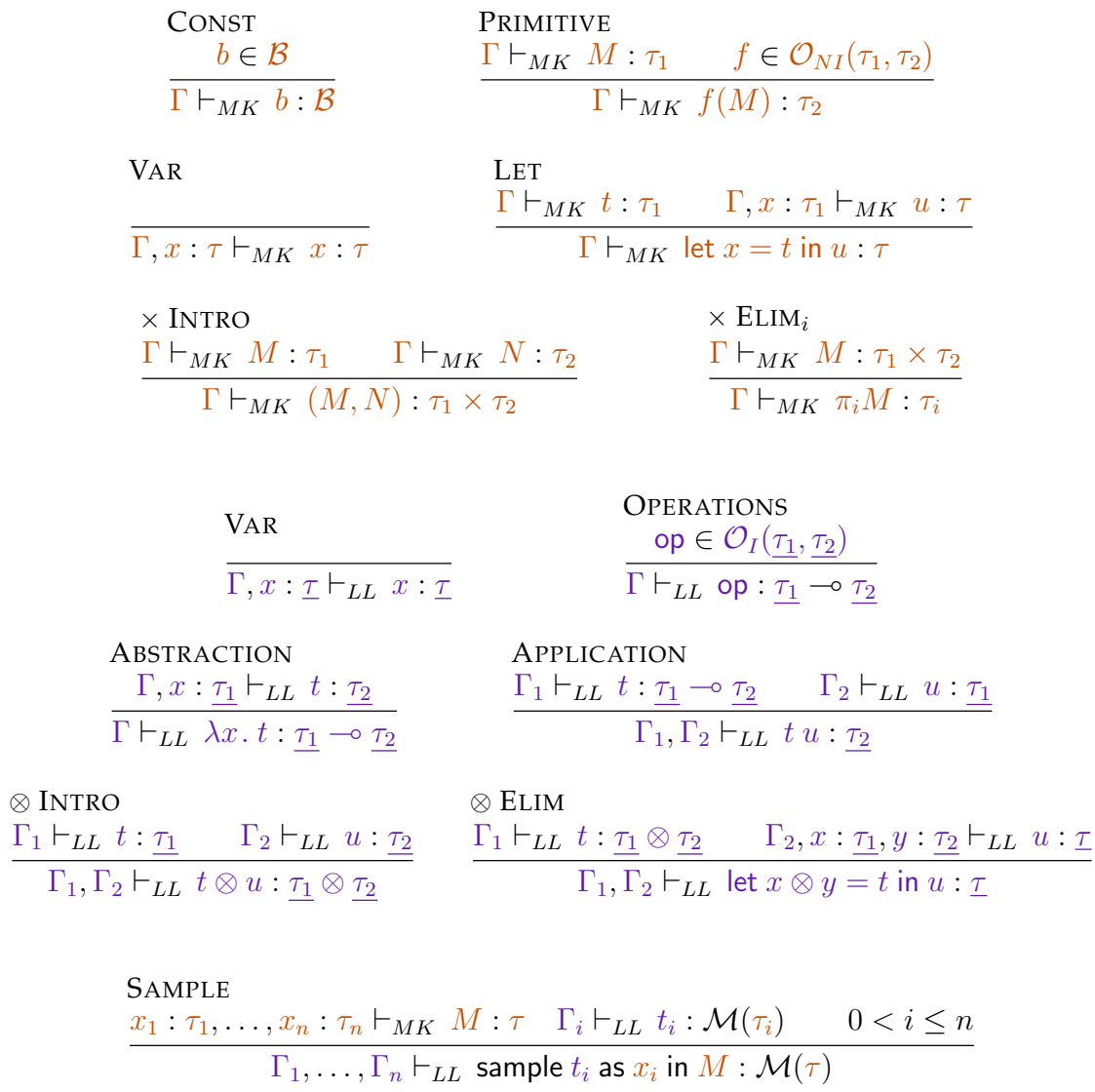


Figure 3.2: Typing Rules: $\lambda^{\text{LL}}_{\text{MK}}$

premise are MK variables whereas the ones in the conclusion are LL variables.

We solve this problem by making the syntax reflect a common idiom of PPLs: compute distributions (elements of $\mathcal{M}\tau$), sample from them and then use the results in a non-linear continuation. This is captured by the following syntax:

$$\mathsf{sample}\ t_1, \cdots, t_n\ \mathsf{as}\ x_1, \cdots, x_n\ \mathsf{in}\ M$$

Note that we are sampling from LL programs t_i (possibly an empty list), out-

putting the results to MK variables x_i and binding them to an MK program M. When clear from the context we simply use sample t_i as x_i in M. Its corresponding typing rule is:

$$\textsc{Sample}\quad \frac{x_1 : \tau_1 \cdots x_n : \tau_n \vdash_{MK} M : \tau \quad \Gamma_i \vdash_{LL} t_i : \mathcal{M}\tau_i \qquad 0 \leq i < n}{\Gamma_1, \cdots, \Gamma_n \vdash_{LL} \mathsf{sample}\ t_i\ \mathsf{as}\ x_i\ \mathsf{in}\ M : \mathcal{M}\tau}$$

As the typing rule suggests, its semantics should be some sort of composition. However, since we are composing programs that are interpreted in different categories, we must have a way of translating MK programs into LL programs — as we will see in Section 3.3 this translation will be functorial. The operational interpretation of this rule is that we have a set of distributions $\{t_i\}$ defined using the linear language — possibly using higher-order programs — we sample from them, bind the samples to the variables $\{x_i\}$ in the MK program M where there are no linearity restrictions. Note that the rule above looks very similar to a monadic composition, though they are semantically different (cf. Section 3.3).

With this new syntax we can finally program in accordance with our new resource interpretation of linear logic, allowing us to write the program

$$\mathsf{sample}\ \mathsf{coin}\ \mathsf{as}\ x\ \mathsf{in}\ (x = x),$$

which flips a coin once and tests the result for equality with itself, making it equivalent to true.

This combined calculus enjoys the expected substitution properties[1].

Theorem 3.2.1. *Let* $\Gamma, x : \underline{\tau_1} \vdash_{LL} t : \underline{\tau}$ *and* $\Delta \vdash_{LL} u : \underline{\tau_1}$ *be well-typed terms, then* $\Gamma, \Delta \vdash_{LL} t\{u/x\} : \underline{\tau}$

[1]To avoid visually polluting the proofs we will drop the color code in Theorem 3.2.1 and Theorem 3.3.4

Proof. The proof can be found in Appendix A.2. □

The following example illustrates how we can use the MK language to duplicate and discard linear variables.

Example 3.2.2. The program which samples from a distribution t and then returns a perfectly correlated pair is given by:

$$\cdot \vdash_{LL} \mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ (x, x) : \mathcal{M}(\tau \times \tau)$$

Similarly, the program that samples from a distribution t and does not use its sampled value is represented by the term

$$\cdot \vdash_{LL} \mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{unit} : \mathcal{M}1$$

Example 3.2.3. Suppose that we have a Markov kernel given by an open MK term $x : \mathbb{N} \vdash M : \mathbb{N}$. If we want to encapsulate it as a linear program of type $\mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N}$ we can write:

$$\cdot \vdash_{LL} \lambda meas.(\mathsf{sample}\ meas\ \mathsf{as}\ x\ \mathsf{in}\ M) : \mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N}$$

Example 3.2.4. As we explain in the introduction, Dahlqvist and Kozen must add many primitives to their language to work around their linearity restrictions. For instance, in order to write projection functions $\mathbb{R}^n \to \mathbb{R}^m$, $n > m$ they must add projection primitives to the language.

By having compositional type constructors that can represent joint distributions , i.e. $\mathcal{M}(\tau \times \tau)$, it is possible to write the program $\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ (\pi_1\ x, \pi_3\ x)$ of type $\mathcal{M}(\tau_1 \times \tau_3)$ which samples from a distribution over triples and returns only the first and third components by only using the syntax of products in MK.

Unfortunately there are some aspects of this language that still are restrictive. For instance, imagine that we want to write an LL program that receives two

"Markov kernels" $\mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N}$ and a distribution over $\mathbb{N}$ as inputs, samples from the input distribution, feeds the result to the Markov kernels, samples from them and adds the results. Its type would be

$$(\mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N}) \multimap (\mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N}) \multimap \mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N}$$

Even though the program only requires you to sample once from each distribution, it is still not possible to write it in the linear language.

We will show in Section 3.3 how the type constructor $\mathcal{M}$ actually corresponds to an applicative functor [61], and the limitation above is actually a particular case of a fundamental difference between programming with applicative functors compared to programming with monads.

Remark 3.2.5. We now have two languages that can interpret probabilistic primitives such as coin. However, every primitive M in the MK language can be easily transported to an LL program by using an empty list of LL programs: sample _ as _ in M. Therefore it makes sense to only add these primitives to the MK language.

3.3 Categorical Semantics

As it is the case with categorical interpretations of languages/logics, types and contexts are interpreted as objects in a category and every well-typed program/proof gives rise to a morphism.

In our case, MK types τ are interpreted as objects $[\![\tau]\!]$ in a Markov category $(\mathbf{M}, \times)$ and well-typed programs $\Gamma \vdash_{MK} M : \tau$ are interpreted as an $\mathbf{M}$ morphism $[\![\Gamma]\!] \to [\![\tau]\!]$. Similarly, LL types $\underline{\tau}$ are interpreted as objects $[\![\underline{\tau}]\!]$ in a model of linear logic $(\mathbf{C}, \otimes, \multimap)$ and well-typed programs $\Gamma \vdash_{LL} t : \underline{\tau}$ are interpreted as a $\mathbf{C}$ morphism $[\![\Gamma]\!] \to [\![\underline{\tau}]\!]$.

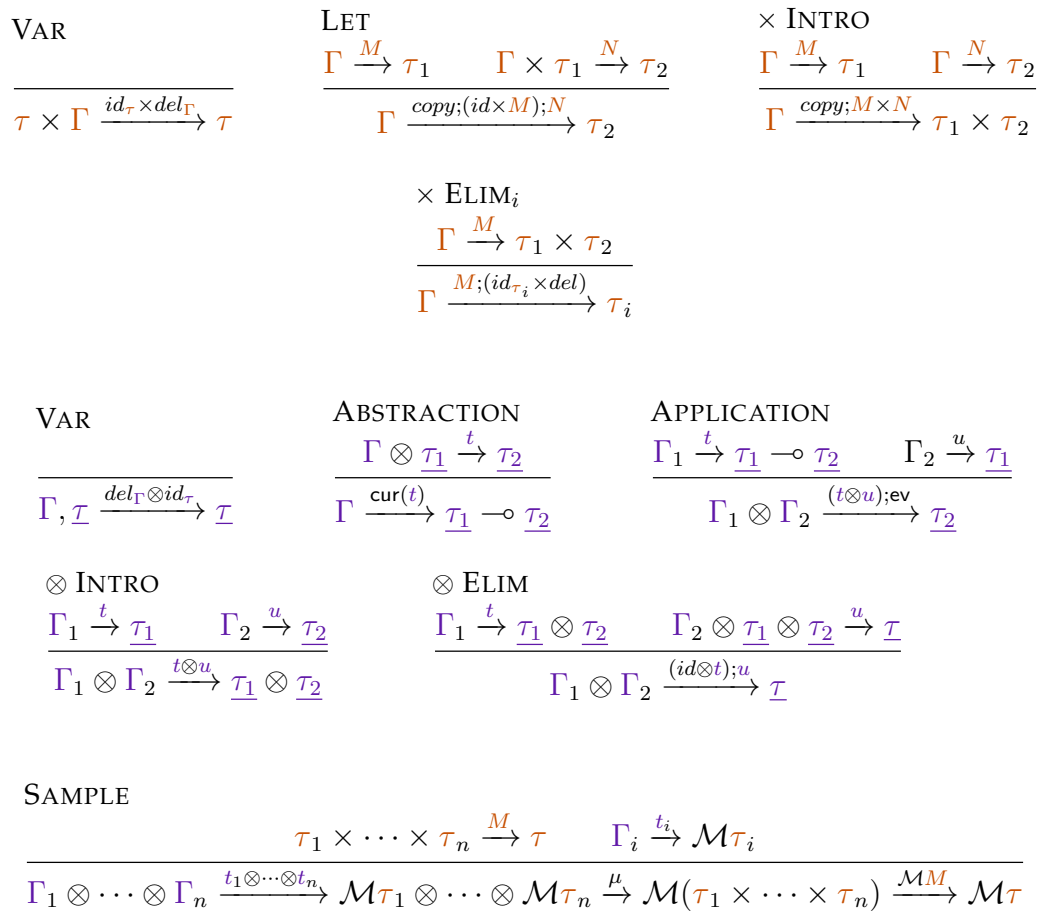


Figure 3.3: Categorical Semantics: λ^2_{INI}

To give semantics to the combined language is not as straightforward. The sample rule allows the programmer to run LL programs, bind the results to MK variables and use said variables in an MK continuation. The implication of this rule in our formalism is that our semantics should provide a way of translating MK programs into LL programs. In category theory this is usually achieved by a functor $\mathcal{M} : \mathbf{M} \to \mathbf{C}$.

However, we can easily see that functors are not enough to interpret the sample rule. Consider what happens when you apply $\mathcal{M}$ to an MK program

$x : \tau_1, y : \tau_2 \vdash_{MK} N : \tau$:

$$\mathcal{M}\,[\![N]\!] : \mathcal{M}(\tau_1 \otimes \tau_2) \to \mathcal{M}\tau$$

To precompose it with two LL programs outputting $\mathcal{M}\tau_1$ and $\mathcal{M}\tau_2$ we need a mediating morphism $\mu_{\tau_1,\tau_2} : \mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2 \to \mathcal{M}(\tau_1 \times \tau_2)$. Furthermore, if N has three or more free variables, there would be several ways of applying μ. Since from a programming standpoint it should not matter how the LL programs are associated, we require that μ_{τ_1,τ_2} makes the lax monoidality diagrams to commute. Therefore, assuming lax monoidality of μ we can interpret the sample rule:

SAMPLE

$$\frac{\tau_1 \times \cdots \times \tau_n \xrightarrow{N} \tau \qquad \Gamma_i \xrightarrow{t_i} \mathcal{M}\tau_i}{\Gamma \xrightarrow{t_1 \otimes \cdots \otimes t_n} \mathcal{M}\tau_1 \otimes \cdots \otimes \mathcal{M}\tau_n \xrightarrow{\mu} \mathcal{M}(\tau_1 \times \cdots \times \tau_n) \xrightarrow{\mathcal{M}N} \mathcal{M}\tau}$$

In case it only has one MK variable, the semantics is given by $[\![t]\!]\,;\mathcal{M}\,[\![N]\!]$ and in case it does not have any free variables the semantics is $\epsilon;\mathcal{M}\,[\![N]\!]$.

The equational theory of the LL languages is the well-known theory of the simply-typed λ-calculus and the MK equational theory has been described, in graphical notation, by Fritz [41]. Something which is not obvious is understanding how they interact at their boundary. This is where $\mathcal{M}$ being a functor becomes relevant, since from functoriality it follows the two program equivalences:

Theorem 3.3.1. *Let t, M and N be well-typed programs,*

$$[\![(\lambda y.\ \mathsf{sample}\ y\ \mathsf{as}\ z\ \mathsf{in}\ N)\ (\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ M)]\!] =$$

$$[\![\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{let}\ y = M\ \mathsf{in}\ N)]\!]$$

Proof.

$$[\![(\lambda y\,.\,\mathsf{sample}\ y\ \mathsf{as}\ z\ \mathsf{in}\ N)\ (\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ M)]\!] =$$

$$[\![t]\!]\,;\mathcal{M}\,[\![M]\!]\,;\mathcal{M}\,[\![N]\!] = [\![t]\!]\,;\mathcal{M}([\![M]\!]\,;[\![N]\!]) =$$

$$[\![\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{let}\ y = M\ \mathsf{in}\ N)]\!]$$

□

Theorem 3.3.2. *Let t be a well-typed program,*

$$[\![\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ x]\!] = [\![t]\!]$$

Proof. $[\![\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ x]\!] = [\![t]\!]\,;\mathcal{M}([\![x]\!]) = [\![t]\!]\,;\mathcal{M}(id) = [\![t]\!]\,;id = [\![t]\!]$ □

Furthermore, we also have a modularity property that can be easily proven:

Theorem 3.3.3. *Let t, M and N be well-typed programs. If $[\![M]\!] = [\![N]\!]$ then*

$$[\![\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ M]\!] = [\![\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ N]\!]$$

The expected compositionality of the semantics also holds:

Theorem 3.3.4. *Let $x_1 : \tau_1, \cdots, x_n : \tau_n \vdash t : \tau$ and $\Gamma_i \vdash t_i : \tau_i$ be well-typed terms.* $\left[\!\!\left[\Gamma_1, \cdots, \Gamma_n \vdash t\{\overrightarrow{t_i}/\overrightarrow{x_i}\} : \underline{\tau}\right]\!\!\right] = ([\![\Gamma_1 \vdash t_1 : \underline{\tau_1}]\!] \otimes \cdots \otimes [\![\Gamma_n \vdash t_n : \underline{\tau_n}]\!]); [\![\Gamma_1, \cdots, \Gamma_n]\!] \vdash t : \underline{\tau}$.

Proof. The proof can be found in Appendix A.1. □

From this theorem we can conclude:

Corollary 3.3.5. *The Subst rule*

$$\textsc{Subst}\quad \frac{\Gamma \vdash u_1 : \tau' \qquad \Gamma \vdash u_2 : \tau' \qquad \Gamma, x : \tau' \vdash t : \tau \qquad \Gamma \vdash u_1 \equiv u_2 : \tau'}{\Gamma \vdash t\{u_1/x\} \equiv t\{u_2/x\} : \tau}$$

is sound with respect to the categorical semantics.

Lax monoidal functors, under the name *applicative functors*, are widely used in programming languages research [61]. They are often used to define embedded domain-specific languages (eDSL) within a host language. This suggests that from a design perspective the Markov kernel language can be thought of as an eDSL inside a linear language.

We have just shown that $\mathcal{M}$ being lax monoidal is sufficient to give semantics to our combined language, but what would happen if it had even more structure? A property which is easy to satisfy is faithfulness, which is verified by both models in the next section. In this case the translation of the MK language into the LL language would be fully-abstract in the following sense:

Theorem 3.3.6. *Let* $x : \tau_1 \vdash M : \tau_2$ *and* $x : \tau_1 \vdash N : \tau_2$ *be two well-typed MK programs. If* $\mathcal{M}$ *is faithful then* $[\![\mathsf{sample}\ y\ \mathsf{as}\ x\ \mathsf{in}\ M]\!] = [\![\mathsf{sample}\ y\ \mathsf{as}\ x\ \mathsf{in}\ N]\!]$ *implies* $[\![M]\!] = [\![N]\!]$.

Proof. $[\![\mathsf{sample}\ y\ \mathsf{as}\ x\ \mathsf{in}\ M]\!] = [\![\mathsf{sample}\ y\ \mathsf{as}\ x\ \mathsf{in}\ N]\!] \implies id_{\mathcal{M}\tau_1}; \mathcal{M}\,[\![M]\!] = id_{\mathcal{M}\tau_1}; \mathcal{M}\,[\![N]\!] \implies [\![M]\!] = [\![N]\!]$. □

As we will see in Section 3.6, if the functor is full we can extend $\lambda_{\mathrm{MK}}^{\mathrm{LL}}$ even further by using ideas from enriched category theory.

3.4 Concrete Models

In this section we show how existing models for both discrete as well as continuous probabilities fit within our formalism.

3.4.1 Discrete Probability

For the sake of simplicity we will denote the monoidal product of $\mathbf{CountStoch}$ as $\times$.

The probabilistic coherence space model of linear logic has been extensively studied in the context of semantics of discrete probabilistic languages [28] and we go over it in more details in Section 4.2.

Theorem 3.4.1. *There is a lax monoidal functor* $\mathcal{M} : \mathbf{CountStoch} \to \mathbf{PCoh}$.

Proof. The functor is defined using the lemmas above. Functoriality holds due to the functor being the identity on arrows. The lax monoidal structure is given by $\epsilon = id_1$ and $\mu_{X,Y} = id_{X\times Y}$ □

Lemma 3.4.2. *If* $\mu \in \{x \otimes y \,|\, x \in \mathcal{M}(X), y \in \mathcal{M}(Y)\}^{\perp}$ *then for every* $x \in X$ *and* $y \in Y$, $\mu(x, y) \leq 1$.

Proof. If there were such indices such that $\mu(x_1, y_1) > 1$ then $\sum\sum \mu(x, y)(\delta_{x_1} \otimes \delta_{y_1})(x, y) > \mu(x_1, y_1)(\delta_{x_1} \otimes \delta_{y_1})(x_1, y_1) = \mu(x_1, y_1) > 1$, which is a contradiction. □

Lemma 3.4.3. *Let* X *and* Y *be two countable sets, then*

$$\mathcal{M}X \otimes \mathcal{M}Y = \left(X \times Y, \{\mu : X \times Y \to \mathbb{R}^+ \mid \sum_{x\in X}\sum_{y\in Y} \mu(x, y) \leq 1\}\right) =$$

$$\mathcal{M}(X \times Y).$$

Proof. By the lemma above it follows that if we have a joint probability distribution $\tilde{\mu}$ over $X \times Y$ and an element $\mu \in \{x \otimes y \,|\, x \in \mathcal{M}(X), y \in \mathcal{M}(Y)\}^{\perp}$ then $\sum\sum \mu(x, y)\tilde{\mu}(x, y) \leq \sum\sum \tilde{\mu}(x, y) \leq 1$. □

Theorem 3.4.4. *Both* ϵ *and* $\mu_{X,Y}$ *are isomorphisms.*

Proof. Since ϵ is the identity morphism, it is trivially an isomorphim. The morphisms $\mu_{X,Y}$ being an isomorphism is a direct consequence of the lemmas above. □

Theorem 3.4.5. *The functor $\mathcal{M}$ is full.*

Both results above can be directly used to enhance the syntax of the combined language. From Theorem 3.4.4 we can conclude that elements of type $\mathcal{M}(\tau_1 \times \tau_2)$, by projecting their marginal distributions, can be manipulated as if they had type $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$. Something to note is that when we do this marginalization process we lose potential correlations between the elements of the pair.

3.4.2 Continuous Probability

In order to accommodate continuous distributions we can use regularly ordered Banach spaces, whose detailed definition goes beyond the scope of this paper.

Theorem 3.4.6. *There is a lax monoidal functor $\mathcal{M} : \mathbf{Kern} \to \mathbf{RoBan}$.*

Proof. The functor acts on objects by sending a measurable space to the set of signed measures over it, which can be equipped with a **RoBan** structure. On morphisms it sends a Markov kernel f to the linear function $\mathcal{M}(f)(\mu) = \int f d\mu$.

The monoidal structure of **RoBan** satisfies the universal property of tensor products and, therefore, we can define the natural transformation $\mu_{X,Y} : \mathcal{M}(X) \otimes \mathcal{M}(Y) \to \mathcal{M}(X \times Y)$ as the function generated by the bilinear function $\mathcal{M}(X); \mathcal{M}(Y) \multimap \mathcal{M}(X \times Y)$ which maps a pair of distributions to its product measure. The map ϵ is, once again, equal to the identity function.

The commutativity of the lax monoidality diagrams follows from the universal property of the tensor product: it suffices to verify it for elements $\mu_A \otimes \mu_B \otimes \mu_C$. □

In **RoBan** the uniform distribution over the interval $[0, 1]$ is an element of $\mathcal{M}\mathbb{R}$, meaning that it can soundly interpret a $\cdot \vdash_{LL} \mathsf{uniform} : \mathcal{M}\mathbb{R}$ primitive.

Even though $\mathcal{M}$ looks very similar to the discrete case, it follows from a well-known theorem from functional analysis that the functor is *not* strong monoidal, meaning that there are joint probability distributions (elements of $\mathcal{M}(A \times B)$) that cannot be represented as an element of the tensor product $\mathcal{M}(A) \otimes \mathcal{M}(B)$ and, as such, programs of type $M(A \times B)$ must be manipulated in MK language, as shown in Example 3.2.4.

3.5 Beyond Probability

We have seen that this new resource interpretation is present in different models of linear logic models for probabilistic programming. In this section we show that this model can be generalized to commutative effects, i.e. effects where the program equation Commutativity below holds. Categorically, these effects are captured by monoidal monads[2]. Due to length issues, we will not fully detail the definition of monoidal monads, but we suggest the interested reader to read Seal [84].

COMMUTATIVITY

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash u : \tau}{\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ (\mathsf{let}\ x_2 = t_2\ \mathsf{in}\ u) \equiv \mathsf{let}\ x_2 = t_2\ \mathsf{in}\ (\mathsf{let}\ x_1 = t_1\ \mathsf{in}\ u) : \tau}$$

[2]Monoidal monads are equivalent to commutative monads, which is the nomenclature usually used in the context of programming languages semantics.

Definition 3.5.1 ([84]). Let $(\mathbf{C}, \otimes, I)$ be a monoidal category and (T, η, μ) a monad over it. The monad T is called monoidal if it comes equipped with a natural transformation $\kappa_{X,Y} : TX \otimes TY \to T(X \otimes Y)$ making certain diagrams commute.

For probability monads the transformation κ corresponds to forming the product probability distribution and, more generally, this can be thought of a program that runs both of its (effectful) inputs and pairs the outputs.

Every monad give rise to the interesting categories $\mathbf{C}_T$ and $\mathbf{C}^T$ which are, respectively, the Kleisli category and Eilenberg-Moore category. The objects of $\mathbf{C}_T$ are the same as $\mathbf{C}$ and morphisms between A and B are $\mathbf{C}$ morphisms $A \to TB$, with the identity morphism being equal to the unit η of the monad and composition is given by $f; g = f; Tg; \mu$.

The objects of the category $\mathbf{C}^T$ are pairs (X, x), where X is a $\mathbf{C}$ object and $x : TX \to X$ is a $\mathbf{C}$ morphism such that $\mu; x = Tx; x$ and $\eta; x = id_X$, and morphisms between objects (X, x) and (Y, y) are $\mathbf{C}$ morphisms $f : X \to Y$ such that $x; f = Tf; y$.

For every monad T there is a canonical inclusion functor $\iota : \mathbf{C}_T \to \mathbf{C}^T$ which maps X to (TX, μ) and $f : X \to Y$ to $Tf; \mu_Y$.

Theorem 3.5.2 ([19]). *The functor ι is full and faithful.*

As we explain in Appendix 2.2.3, assuming enough structure on the category $\mathbf{C}$ we can show that the triple $(\mathbf{C}_T, \mathbf{C}^T, \iota)$ is a model to the $\lambda^{\text{LL}}_{\text{MK}}$ language and we can bring our new resource interpretation of linear logic to other commmutative effects.

An illustrative example is the powerset monad $\mathcal{P} : \mathbf{Set} \to \mathbf{Set}$ which is monoidal and since $\mathbf{Set}$ has the necessary structure, the triple $(\mathbf{C}_{\mathcal{P}}, \mathbf{C}^{\mathcal{P}}, \mathcal{P})$ is a

model to our language and can be used to give semantics to non-deterministic computation.

In the context of commutative effects other than randomness, the syntax sample t as x in M does not make as much sense, in which case we can use the syntax observe t_i as x_i in M instead. Once again, operationally, the programs t_i are fully executed, the values are bound to x_i in M which is then executed.

Furthermore, other effects have other relevant effectful operations and, therefore, we can assume that there is a set of operations in the MK language that are interpreted in the Kleisli category and can be transported to LL using observe, similar to how it was done in the probabilistic case.

For the non-deterministic case we can assume the existence of typing rules for non-deterministic choice and failure:

$$\textsc{Choice}\quad \frac{\Gamma \vdash_{MK} t_1 : \tau \qquad \Gamma \vdash_{MK} t_2 : \tau}{\Gamma \vdash_{MK} t_1 \oplus t_2 : \tau} \qquad \textsc{Null}\quad \frac{}{\Gamma \vdash_{MK} 0_\tau : \tau}$$

satisfying the expected equations and interpreted using set-theoretic union and the empty set, respectively.

A similar connection between linear logic and monoidal monads has been made by Benton and Wadler [14], where they want to relate Moggi's monadic λ-calculus with linear logic by showing that if a monad is monoidal and the category has equalizers and coequalizers, then the Eillenberg-Moore category is a model of linear logic.

3.6 An Extension Based on Enriched Category Theory

Though the core calculus described above provides some interesting insights into how to program with these objects, it is still somewhat limited. For in-

$$\tau \coloneqq A$$
$$\underline{\tau} \coloneqq 1 \mid \mathbf{C}(A, B) \mid \underline{\tau} \multimap \underline{\tau} \mid \underline{\tau} \otimes \underline{\tau}$$

$$M, N \coloneqq x \mid \mathsf{let}\ x = M\ \mathsf{in}\ N \mid f(M)$$
$$t, u \coloneqq x \mid unit \mid \lambda x.\, t \mid t\ u \mid t \otimes u \mid \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u$$
$$\mid \mathsf{box}(M) \mid t\ (M)$$

Figure 3.4: Enriched type grammar

stance, consider a primitive $x : \mathbb{R}, y : \mathbb{R} \vdash \mathsf{normal}(x, y) : \mathbb{R}$ that outputs a normal distribution with expected value x and variance y, and the program $\mathsf{let}\ x = \mathsf{uniform}\ \mathsf{in}\ \mathsf{normal}(x, x)$ which outputs normal distributions with equal variance and expected value, both of which are uniformly distributed. Without further assumptions on the functor $\mathcal{M}$ it is not possible to write the program above in our core calculus, even though it only samples from the uniform distribution once.

The goal of this section is to explore why that is the case and propose a new extension to this calculus that solves this problem.

3.6.1 A Simple Enriched Calculus

Let $\mathbf{L}$ be a symmetric monoidal closed category and $\mathbf{C}$ an $\mathbf{L}$-category. The intuition behind enriched category theory is that inside the language interpreted by $\mathbf{L}$ it has types $\mathbf{C}(\tau, \tau')$ that corresponds to programs in $\mathbf{C}$ of type τ' with one free variable of type τ. Furthermore, the enrichment gives ways of programming $\mathbf{C}$ programs inside $\mathbf{L}$. We model this by the type and term grammars presented in Figure 3.4.

One of the main novelties is that both languages are now interpreted as

$\mathbf{L}$ morphisms and there are two typing judgements $\Gamma \vdash t : \underline{\tau}$ and $\Gamma; \Delta \vdash M : \tau$ which are interpreted, respectively, as morphisms $\mathbf{L}(\llbracket \Gamma \rrbracket, \llbracket \underline{\tau} \rrbracket)$ and $\mathbf{L}(\llbracket \Gamma \rrbracket, \mathbf{C}(\llbracket \Delta \rrbracket, \llbracket \tau' \rrbracket))$. This calculus looks very similar to the non-enriched one, with the exceptions that now the MK language now has a linear context Γ in its typing judgement and the interface between both languages is given by the $\mathsf{box}(M)$ construct that has the typing rules:

$$\textsc{Var}\ \frac{}{\cdot, x : \tau \vdash x : \tau} \qquad \textsc{Box}\ \frac{\Gamma; \tau \vdash M : \tau'}{\Gamma \vdash \mathsf{box}(M) : \mathbf{C}(\tau, \tau')} \qquad \textsc{Unbox}\ \frac{\Gamma_1 \vdash t : \mathbf{C}(\tau, \tau') \qquad \Gamma_2; \Delta \vdash M : \tau}{\Gamma_1, \Gamma_2; \Delta \vdash t\ (M) : \tau'}$$

The semantics of the Var rule is the enriched morphism id, the rule Box is interpreted as $\llbracket \mathsf{box}(M) \rrbracket = \llbracket M \rrbracket$ and Unbox is interpreted as $\llbracket t\ (M) \rrbracket = (\llbracket M \rrbracket \otimes \llbracket t \rrbracket); \circ$, where the morphism $\circ$ is the enriched composition operation.

3.6.2 An Enriched Markov Category Calculus

Now we want to develop the theory of enriched Markov categories so that we can extend the enriched calculus in the previous section with Markov categories primitives. In this section we will assume that the functor $\mathcal{M}$ is full.

Definition 3.6.1. An $\mathbf{L}$-Markov category is an $\mathbf{L}$-category $\mathbf{C}$ with a binary operation on objects $\otimes$ and morphisms $\mathbf{C}(A_1, B_1) \otimes \mathbf{C}(A_2, B_2) \to \mathbf{C}(A_1 \otimes A_2, B_1 \otimes B_2)$, $1 \to \mathbf{C}(A, A \otimes A)$ and $1 \to \mathbf{C}(A, 1)$ such that they obey the Markov categories axioms.

Lemma 3.6.2. *If $(\mathbf{C}, \mathbf{L}, \mathcal{M})$ is a model then $\mathbf{C}$ is an $\mathbf{L}$-Markov category.*

Proof. The collection of objects is the same as $\mathbf{C}$ and the $\mathbf{L}$ object is $\mathcal{M}A \multimap \mathcal{M}B$. The necessary morphisms are defined using the fact that $\mathcal{M}$ is full. □

Under this particular semantics it is not necessary to add a type constructor $\mathbf{C}(\tau, \tau')$, since it can be represented as $\mathcal{M}\tau' \multimap \mathcal{M}\tau$. Furthermore, the box$(M)$ will be syntactic sugar of $\lambda\,\mu.\,\mathsf{sample}\ \mu\ \mathsf{as}\ x\ \mathsf{in}\ M$ and the Unbox rule can be represented using regular function application as follows:

$$\textsc{Unbox}\quad \frac{\Gamma_1 \vdash t : \mathcal{M}\tau \multimap \mathcal{M}\tau' \qquad \Gamma_2; \Delta \vdash M : \tau}{\Gamma_1, \Gamma_2; \Delta \vdash t\ (M) : \tau'}$$

Just for the sake of illustration, in this new setting, the unary sample rule becomes:

$$\frac{\Gamma_1 \vdash_{LL} t : \mathcal{M}\tau_1 \qquad \Gamma_2; \Delta \vdash_{LL} M : \tau_2}{\Gamma_1, \Gamma_2; \Delta \vdash_{LL} \mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ M : \mathcal{M}\tau_2}$$

As we mentioned above, there are many "sample-once" programs that cannot be represented in the non-enriched case. This extension increases the expressivity of our base calculus as the examples below illustrate:

Example 3.6.3. Consider the parametrized distribution $\cdot \vdash \mathsf{Uniform} : \mathcal{M}(\mathbb{R} \times \mathbb{R}) \multimap \mathcal{M}\mathbb{R}$ that given an input (r_1, r_2) it outputs the uniform distribution over the interval $[r_1, r_2]$. The Box-Muller transform is a sampling algorithm that given two samples from the uniform distribution over the unit interval returns two independent samples from the normal distribution. In the enriched calculus we can represent this closed program with type $\mathcal{M}\mathbb{R} \otimes \mathcal{M}\mathbb{R} \multimap \mathcal{M}(\mathbb{R} \times \mathbb{R})$ as

$$\begin{aligned}
&\cdot \vdash \lambda\, r_1\, r_2.\ \mathsf{sample}\ r_1, r_2\ \mathsf{as}\ x, y\ \mathsf{in}\\
&\qquad \mathsf{let}\ z_1 = \mathsf{Uniform}(x, y)\ \ \mathsf{in}\\
&\qquad \mathsf{let}\ z_2 = \mathsf{Uniform}(x, y)\ \ \mathsf{in}\\
&\qquad \left(\sqrt{-2 \ln z_1}\cos(2\pi z_2), \sqrt{-2 \ln z_1}\sin(2\pi z_2)\right)
\end{aligned}$$

In the non-enriched calculus the only way of writing the program above is by adding the primitive Uniform to the MK language.

Example 3.6.4. Now it is possible to write the problematic program from before where we want to add the output of two distinct kernels when given the same input:

$$f : \mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N}, g : \mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N} \vdash$$
$$\lambda\, \mu.\, \mathsf{sample}\ \mu\ \mathsf{as}\ x\ \mathsf{in}\ (f(x) + g(x)) : \mathcal{M}\mathbb{N} \multimap \mathcal{M}\mathbb{N}$$

The advantage of this calculus is, of course, that now the linear language becomes even more expressive, which makes the higher-order functions even more useful. For instance, the Box-Muller transform can be parametrized by a parametric distribution $\mathcal{M}(\mathbb{R} \times \mathbb{R}) \multimap \mathcal{M}\mathbb{R}$, something that is not possible if we were to reify it into the MK language.

Example 3.6.5 ([37])**.** In their probabilistic semantics, Ehrhard et al. have extended their probabilistic λ-calculus with the following typing rule

$$\frac{\Gamma \vdash t : \mathbb{R} \qquad \Gamma, x : \mathbb{R} \vdash u : \mathbb{R}}{\Gamma \vdash \mathsf{let}\ x = t\ \mathsf{in}\ u : \mathbb{R}}$$

which samples from t using a CBV semantics, as opposed to the CBN semantics of the simply-typed λ-calculus. This rule is subsumed by the Unbox rule. We note, however, that in order to recover the full expressive power of the let construct above, we need to add ! as a type constructor, which is exactly what we do in Chapter 4.

Example 3.6.6. As a last example, let us assume that the Markov category has conditionals. At the syntactic level this is equivalent to adding the following typing rule to the MK language.

$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash \mathsf{observe}(N)\ \mathsf{in}\ \mathsf{M} : \tau_2}$$

With this rule we can lift the conditioning to the LL language as follows:

$$\mathsf{inference} : (\mathcal{M}\tau \multimap \mathcal{M}\tau') \multimap \mathcal{M}\tau \multimap \mathcal{M}\tau' \multimap \mathcal{M}\tau$$

$$\mathsf{inference}\ f\ \mathsf{prior}\ \mathsf{evidence} = \mathsf{sample}\ (\mathsf{sample}\ \mathsf{prior}\ \mathsf{as}\ x\ \mathsf{in}\ (x, f(x))), \mathsf{evidence}\ \mathsf{as}\ x, y\ \mathsf{in}$$
$$\mathsf{observe}(y)\ \mathsf{in}\ x$$

I see the example above as a nice starting point for understanding how to do synthetic probability theory in the context of symmetric monoidal closed categories. Indeed, it seems to tie up quite nicely with the formalism for Bayesian conditioning studied by Clerc et al. [23].

3.7 Related Work

Semantics of Probabilistic Programming

Ehrhard et al. [37, 36] have defined a model of linear logic **CLin** which can be used to interpret a higher-order probabilistic programming language. They have used the call-by-name translation of intuitionistic logic into linear logic $A \to B = !A \multimap B$ to give semantics to their language. The authors extend their language with a call-by-value let syntax which makes it possible to reuse sampled values. In order to give semantics to this new language they introduce a new category $\mathbf{CLin_m}$ which can interpret this new operator, at the cost of complicating their model.

Because there is an analogous proof of Theorem 3.4.6 with the category **CLin** replacing **RoBan**, we can use their original, simpler, model to interpret our language, while not needing to use the linear logic exponential to interpret non-linear programs.

Dahlqvist and Kozen [26] have defined a category of partially ordered Banach spaces and shown that it is a model of intuitionistic linear logic. An important difference from their approach and the one mentioned above is that they embrace variable linearity as part of their syntax. As we argued in this paper, we believe that the syntactic restriction of linearity they have used is not adequate for the purposes of probabilistic programming. They deal with this limitation by adding primitives to their languages which, by using the results of Section 3.4, could be programmed using the MK language.

Quasi Borel spaces [46] are a conservative extension of **Meas** that are Cartesian closed and have a commutative probability monad. The drawback of this model is that it is still not as well understood as its measure-theoretic counterpart, and there are theorems from probability theory used to reason about programs that may not hold in the category of quasi Borel spaces **QBS**.

Recently, Geoffroy [42] has made progress in connecting linear logic and quasi Borel Spaces by showing that a certain subcategory of the Eillenberg-Moore category for the probability monad in **QBS** is a model of classical linear logic, which we see as an instance of our model where the MK language can have higher-order functions as well.

Call-by-Push-Value

The idea of having two distinct type systems that are connected by a functorial layer is reminiscent of Call-by-Push-Value (CBPV) [57], which has a type

system for values and a type system for computations that are connected by an adjunction. In recent work, Ehrhard and Tasson [90] use the Eilenberg-Moore adjunction of the linear logic exponential ! to give semantics to a calculus that can interpret lazy and eager probabilistic computation, allowing for the interpretation of an eager let operator which is operationally similar to our sample construct. However, the existence of the let operator depends on properties of the ! that are unknown to hold for continuous distributions, while our semantics can naturally deal with continuous distributions as we have shown in Section 3.4.

Furthermore, the exponential which lies at the center of their approach is, semantically, hard to work with and does not have any clear connections to probability theory, making it unlikely that their semantics can be seen as a bridge between the Markov and linear semantics, which is the case for the models presented in Section 3.4.

Goubault-Larrecq [45] has defined a CBPV domain semantics to a language that mixes probability and non-determinism, a long-standing challenge in the theory of programming languages. His focus is in understanding how to make probability interact with non-determinism in a sound way. He studies the full-abstraction of his semantics but does not deal with connections to linear logic.

CHAPTER 4

PERFECT BANACH LATTICES AND VECTOR SPACE MODELS OF LINEAR LOGIC

4.1 Introduction

This chapter is based on joint work with Leon Witzman and Dexter Kozen [6].

Computer Scientists who were introduced to linear logic as a typing discipline for enforcing fine-grained reasoning about variable usage, find it surprising when they learn that linear logic has deep connections to linear algebra. Indeed, many categories of vector spaces and linear transformations are models of linear logic.

The multiplicative connectives are interpreted as the familiar tensor product and space of linear transformations, the additive connectives are interpreted using directed sums, linear negation is interpreted as the dual vector space and the exponential are interpreted using Fock-like spaces. These spaces were originally introduced in quantum mechanics as a way of representing an unknown number of copies of a single particle. Note that this situation is analogous to the linear logic one, since in vector space models a space V corresponds to a single variable usage, so its exponential should be a space representing an unknown amount of uses of V. Formally, this construction is quite subtle and requires a careful analysis of the domain at hand. Morally, the exponential should be thought of as $!V = \bigoplus_{n=0}^{\infty} V^{\otimes n}$, where $V^{\otimes n}$ is V tensored with itself n times.

In recent years, these kinds of models have found numerous applications in probabilistic semantics. In particular, they provide a clean solution to the problem of giving semantics to higher-order probabilistic languages. The standard

way of giving semantics to such languages is by using Cartesian closed categories. Unfortunately, the standard measure-theoretic apparatus of probability and measure theory is notably not Cartesian closed, even when restricted to discrete spaces [4]. This limitation has motivated the construction of *quasi-Borel spaces* [46], a cartesian closed conservative extension of measurable spaces.

Linear logic, on the other hand, by using symmetric monoidal closed categories, provides a different perspective on higher-order languages and because of that, models of linear logic have been extensively used as semantic basis to probabilistic programming languages. The first one is the category **PCoh** of probabilistic coherence spaces. These spaces were defined as a generalization of coherence spaces, one of the first models of linear logic. The category **PCoh** is useful from a probabilistic semnatics point of view. For instance, through the Kleisli category of the exponential it can interpret the simply-typed λ-calculus, its vector space structure can be used to interpret (discrete) probabilistic primitives such as biased coins, its CPO structure can interpret arbitrary types and term recursion. One of its main drawbacks is that it cannot interpret continuous distributions.

This limitation of **PCoh** has led to the search of generalizations that can accommodate the features of the discrete case while also properly handling continuous distributions. The first attempt was made in [37], where the authors propose a category **CStab** of positive cones and stable Scott-continuous functions and conjecture that it is the Kleisli category to an exponential comonad.

In the same paper, they refine this category and define $\mathbf{CStab}_m$ of positive measurable cones and stable measurable functions with the goal of giving semantics to an eager let operator. Two other models were proposed.

Dahlqvist and Kozen have shown that the category **RoBan** of regular Ba-

nach spaces and regular linear operators is a model of intuitionistic linear logic and used it as a semantic basis to a higher-order probabilistic language. An appealing aspect of the **RoBan** model is that ordered Banach spaces are mathematically well-understood objects with a well-developed classical theory, thus providing a plethora of useful theorems to reason about programs. This is illustrated in [26] by using results from ergodic theory to prove the correctness of a Gibbs sampling algorithm implemented in a higher-order language. However, the programming model supported by the semantics is somewhat brittle, in that the soundness of the system depends on a tricky interaction between three different type grammars with several syntactic restrictions.

Slavnov has defined a category of coherent cones and shown that it is a model of classical linear logic.

Although these previous approaches are valuable contributions to our understanding of higher-order probabilistic programming through linear logic, missing up to now is a comprehensive model that embodies all desirable aspects:

- extends **PCoh** to admit continuous measures;
- is a model of classical linear logic, not just intuitionistic linear logic;
- has a simple and expressive programming model that can handle higher-order computation with recursion;
- is based on well-understood classical structures from measure theory and functional analysis.

In this chapter we propose such a model. Our model extends **PCoh** with continuous probabilities and satisfies all of the properties above. Our model is

based on complete normed vector lattices, called *Banach lattices*. To accommodate the second point, we work with spaces with an involutive linear negation, the so-called *perfect spaces*.

Compared to previous models, our model has simpler tensor product and exponential constructions, which we believe lead to a more perspicuous and theoretically satisfying generalization of $\mathbf{PCoh}$. For example, we invite a comparison with $\mathbf{CLin}_m$, where the constructions rely on categorical machinery which, though elegant, are nonconstructive, as opposed to our construction.

Most importantly, Banach lattices can be seen as an abstraction of ordinary measure spaces and are well-studied in functional analysis, with many results from measure theory holding for certain classes of Banach lattices. There is a vast literature on the subject; see [39] for a thorough introduction.

In order to justify the viability of our model, we show that it can be used to interpret the calculus introduced in the previous section, and we extend it with recursion and the nonlinear modality !.

This chapter is organized as follows: we start by reviewing some of the existing models, we go over some background definitions from the Riesz space literature. Then we define the category $\mathbf{PBanLat_1}$

Summary of chapter

- In §4.2 we provide a discussion on vector space models of linear logic and go over a couple of concrete examples.
- in §4.3 we provide some background definitions and results from the Riesz space literature.
- In §4.4, we define the category $\mathbf{PBanLat_1}$ of perfect Banach lattices and order-continuous positive linear operators with norm at most 1 and show

that it is a model of classical linear logic.

- In §4.5, we show that there is a full and faithful monoidal closed functor $\mathbf{PCoh} \to \mathbf{PBanLat_1}$. This is a more adequate extension than the model $\mathbf{CLin}_m$ proposed in [36], since it also accommodates the classical aspects of the linear structure of $\mathbf{PCoh}$.
- In §4.6, we show that $\mathbf{PBanLat_1}$ is isomorphic to a category of lattices of positive cones.
- In §4.7, we show that $\mathbf{PBanLat_1}$ is part of a λ^2_{INI} model.

Our work contributes both to the study of quantitative models of linear logic as well as to a deeper understanding of higher-order probability theory, shedding light on the importance of linear logic as a vehicle to interpret higher-order programs without Cartesian closure. Furthermore, in the tradition of [56, 26], by working with vector spaces that are deeply connected to measure theory (see e.g. [39] for a thorough presentation), powerful classical results can be brought to bear on the verification of probabilistic programs, though we leave this for future work.

4.2 Vector Space Models for Probabilistic Semantics

Throughout this thesis we will make use of a few different vector space models of linear logic. We reserve this section to go over the details of these models. Since we use $\mathbf{PCoh}$ in multiple parts of this thesis we will spend more time explaining their constructions.

4.2.1 Probabilistic Coherence Spaces

One of the first models of linear logic is **Coh** of coherence spaces and linear functions, these objects are sets $|X|$ equipped with functions $|X| \to 2$ — called a *web* — satisfying certain properties. This model was an important step in understanding denotational models of sequential computation and, more broadly, models of linear logic [16].

In an attempt to get a deeper understanding of such models, Girard [44] proposed a variant of coherence space where the web is real-valued instead of boolean-valued and coined the term probabilistic coherence space. Later, Danos and Ehrhard [28] have refined this idea and proposed the definition currently used.

Definition 4.2.1. A probabilistic coherence space (PCS) is a pair $(|X|, \mathcal{P}(X))$ where $|X|$ is a countable set and $\mathcal{P}(X) \subseteq |X| \to \mathbb{R}^+$ is a set, called the *web*, such that:

- $\forall a \in X\ \exists \varepsilon_a > 0\ \varepsilon_a \cdot \delta_a \in \mathcal{P}(X)$, where $\delta_a(a') = 1$ iff $a = a'$ and 0 otherwise, and we use the notation $\varepsilon_a = \varepsilon(a)$;
- $\forall a \in X\ \exists \lambda_a\ \forall x \in \mathcal{P}(X)\ x_a \leq \lambda_a$;
- $\mathcal{P}(X)^{\perp\perp} = \mathcal{P}(X)$, where $\mathcal{P}(X)^{\perp} = \{x \in X \to \mathbb{R}^+ \mid \forall v \in \mathcal{P}(X)\ \sum_{a \in X} x_a v_a \leq 1\}$.

We can define a category **PCoh** where objects are probabilistic coherence spaces and morphisms $X \multimap Y$ are matrices $f : |X| \times |Y| \to \mathbb{R}^+$ such that for every $v \in \mathcal{P}(X)$, $(f \cdot v) \in \mathcal{P}(Y)$, where $(f \cdot v)_b = \sum_{a \in |A|} f_{(a,b)} v_a$. The identity morphism is the identity matrix and composition is given by matrix multiplication.

The computational interpretation behind probabilistic coherence spaces is that an element of the web v corresponds to a probabilistic computation, an

element of the dual v' corresponds to a continuation and the interaction $\langle v, v' \rangle$ corresponds to the probability of the execution terminating.

Furthermore, something quite appealing about using this model of linear logic as a semantic basis is that the web equipped with the pointwise order is a CPO. This structure has been used to give semantics to recursion and recursive types.

For the remainder of this section we will present the linear logic structure of **PCoh**. We start by briefly describing the linear negation, where if $(|X|, \mathcal{P}(X))$ is a PCS, its negation is defined as $X^\perp = (|X|, \mathcal{P}(X)^\perp)$.

Multiplicative Structure

Definition 4.2.2. Let $(|X|, \mathcal{P}(X))$ and $(|Y|, \mathcal{P}(Y))$ be PCS, we define $X \otimes Y = (|X| \times |Y|, \{x \otimes y \,|\, x \in \mathcal{P}(X), y \in \mathcal{P}(Y)\}^{\perp\perp})$, where $(x \otimes y)(a, b) = x(a)y(b)$

The connection between PCS and (discrete) probability is given by the following theorem.

Lemma 4.2.3. *Let X be a countable set, the pair $(X, D_{\leq 1}(X))$ is a PCS.*

Proof. The first two points are obvious, as the Dirac measure is a subprobability measure and every subprobability measure is bounded above by the constant function $\mu_1(x) = 1$.

To prove the last point we use the — easy to prove — fact that $\mathcal{P}X \subseteq \mathcal{P}X^{\perp\perp}$. Therefore we must only prove the other direction. First, observe that, if $\mu \in D_{\leq 1}(X)$, then we have $\sum \mu(x)\mu_1(x) = \sum 1\mu(x) = \sum \mu(x) \leq 1$, $\mu_1 \in D_{\leq 1}(X)^\perp$.

Let $\tilde{\mu} \in D_{\leq 1}(X)^{\perp\perp}$. By definition, $\sum \tilde{\mu}(x) = \sum \tilde{\mu}(x)\mu_1(x) \leq 1$ and, therefore, the third point holds. □

This lemma can be used to give semantics to probabilistic primitives. For instance, a fair coin is interpreted as a function $\mathsf{coin} : \mathbb{N} \to [0,1]$ which is $.5$ at 0 and 1 and 0 elsewhere and is an element of $\mathcal{P}(\mathbb{N})$.

Lemma 4.2.4. *Let $X \to Y$ be a* $\mathbf{CountStoch}$ *morphism. It is also a* $\mathbf{PCoh}$ *morphism.*

Definition 4.2.5. Let $(|X|, \mathcal{P}(X))$ and $(|Y|, \mathcal{P}(Y))$ be PCS, $X \multimap Y = (|X| \times |Y|, \{f \mid \forall v \in \mathcal{P}(X), f \cdot v \in \mathcal{P}(Y)\})$.

Theorem 4.2.6 ([28])**.** *For every PCS X, $- \otimes X \dashv X \multimap -$.*

Additive Structure

Both additives connectives have the same underlying sets. It is their web that differentiate them. Let $(|X|, \mathcal{P}(X))_1$ and $(|X|, \mathcal{P}(X))_2$ be PCS, then $|X_1 \,\&\, X_2| = |X_1| + |X_2|$ and $\mathcal{P}(X_1 \,\&\, X_2) = \{v : |X_1| + |X_2| \to \mathbb{R}^+ \mid p_i(v) \in \mathcal{P}(X_i)\}$, where $p_i(v)_{x_i} = v_{x_i}$ — i.e. we want the elements such that their restrictions are in their respective webs.

The projection matrix π_i is defined as $\pi_i(a,b) = 1$ if $a \in |X_i|$ and $a = b$ and 0 otherwise.

Theorem 4.2.7. *The operation $\&$ together with the projection morphisms π_i is a Cartesian product.*

The disjunction can be defined through the linear logic equivalence $A \oplus B = (A^\perp \,\&\, B^\perp)^\perp$.

Exponential Structure

Let $(|X|, \mathcal{P}(X))$ be a probabilistic coherence space and $\mathcal{M}_{fin}(|X|)$ the set of finite multisets over $|X|$. If $v \in \mathcal{P}(X)$ we define $v^! : \mathcal{M}_{fin}(|X|) \to$

$\mathbb{R}^+$ as $v^!([(n_1, x_1), \ldots, (n_m, x_m)]) = \prod_{i=1}^m v(x_i)^{n_i}$ and given a multiset $\mu = [(n_1, x_1), \ldots, (n_m, x_m)]$, its size $|\mu|$ is $\sum_{i=1}^m n_i$.

Definition 4.2.8. Let $(|X|, \mathcal{P}(X))$ be a probabilistic coherence space, we define $!X = (\mathcal{M}_{fin}(|X|), \{v^! \mid v \in \mathcal{P}(X)\}^{\perp\perp})$

This construction is hard to parse at first. With the intuition that the underlying set of a PCS corresponds to the possible outputs of a program, multisets are used to capture the possibly different outputs a probabilistic program produces when reused a finite amount of times. Of course, when programming with higher-order functions, this analogy breaks.

Theorem 4.2.9 ([25]). *Let $f \in \mathbf{PCoh}(!X, Y)$, then f is equivalent to the power series*

$$\hat{f}(x) = \sup_{N \in \mathbb{N}} \sum_{b \in |Y|} \Big(\sum_{|\mu| \leq N} f_{\mu,b} \cdot x_\mu^! \Big) \cdot \delta_b,$$

Therefore, by the definition of $\mathbf{PCoh}$ morphism and the result above, every coKleisli morphism is actually a morphism $\mathcal{P}(X) \to \mathcal{P}(Y)$ that can be represented as a power series.

We conclude by noting that every PCS has a canonical vector space associated to it that was originally defined by Ehrhard and Danos [28]. Their construction did not preserve the linear logic connectives, in particular, it did not preserve linear negation. One of the main results of this chapter is showing that we can adapt their construction to our model and show that it is still functorial and, in addition, it preserves the multiplicative structure of the model as well.

4.2.2 Regularly Ordered Banach Space

Banach spaces are an important class of vector spaces that have found applications in many areas of mathematics, physics and computer science. They are

normed vector spaces such that the metric space generated by the norm is complete. Partially ordered Banach spaces are an important class of Banach spaces that have found applications in measure theory and are quite common. As it is the case with partially ordered spaces, there are quite a few different restrictions one can impose on the order. This is also the case for ordered Banach spaces. One such variant are regularly ordered Banach space.

A partially ordered Banach space is a Banach space $(V, \|\cdot\|)$ equipped with a partial order $\leq$ such that if $-y \leq x \leq y$ then $\|x\| \leq \|y\|$ and the set of positive elements is norm closed. Furthermore, if $\|x\| = \inf\{\|y\| \mid -y \leq x \leq y\}$ then it is regularly ordered Banach space. A linear function is called positive if it maps positive elements to positive elements. A function is called regular if it is the difference of two positive functions. It is possible to show that positive linear operators are norm-bounded and, therefore, norm-continuous.

Definition 4.2.10 ([26])**.** The category **RoBan** has regularly ordered Banach spaces as objects and regular linear functions as morphisms.

It has been known for a long time that **RoBan** is a symmetric monoidal closed category [64]. Dahlqvist and Kozen have shown that it is actually a model of intuitionistic linear logic.

4.2.3 Categories of Cones

In the context of probabilistic semantics, it is not necessary to care about nonpositive vectors and functions. With this in mind, authors have defined categories of normed cones, which are a generalization of positive cones of ordered normed vector spaces. More concretely, a normed cone C is a $\mathbb{R}^+$-semimodule with a norm $\|\cdot\| : C \to \mathbb{R}^+$.

Every cone can be equipped with the partial order $x \leq y$ if, and only if, there is a unique z such that $x + z = y$, meaning that it is possible to define a partial subtraction operation whenever $x \leq y$, calling $y - x$ the element such that $x + (y - x) = y$.

A function $f : C_1 \to C_2$ between cones is linear if it commutes with addition and scalar multiplication, it is monotonic if it preserves the order relation and it is Scott-continuous if for every directed set x_α with supremum x, $\sup_\alpha f(x_\alpha) = f(x)$. As it is the case with partially-ordered vector spaces, there are different classes of cones where the order and the norm have particular properties:

Definition 4.2.11. A cone C is said to be:

- Sequentially complete if every norm-bounded sequence has a least upper bound.
- Directed complete if every norm-bounded directed set has a least upper bound.
- A lattice cone if the poset structure is a lattice.

Each kind of cones organize themselves in different categories, which we denote by ω**CLin**, **CLin** and **LatCone**, respectively, where the morphisms are the appropriate kind of Scott-continuous with norm at most 1. For the purposes of programming languages semantics, cones should be complete so that they can interpret recursive programs, which is why we also introduce the category **CLatLin** of complete lattice cones and Scott continuous linear functions. Such completeness can be more abstractly stated as requiring the cone category to be enriched over **CPO**. These categories were one of the first generalizations of probabilistic coherence spaces that can also handle continuous probability, though they are not models of classical linear logic, only intuitionistic linear logic.

4.3 Riesz spaces

Now we will go over more carefully the technical definitions and constructions that will be used to define our novel model. This section contains a brief self-contained presentation of the vector lattice literature. We point the interested reader to the introductory texts [2, 93] for good presentations of much of the material presented in this section.

Although we are primarily interested in Banach lattices—normed vector lattices with a completeness property—we start by defining the objects in the general unnormed case.

Definition 4.3.1. Let $\mathbb{R}_+ = \{a \in \mathbb{R} \,|\, a \geq 0\}$. A *Riesz space* is a partially-ordered vector space $(V, \leq)$ over $\mathbb{R}$ such that

- if $x \leq y$, then $x + w \leq y + w$;
- if $x \leq y$, then $\alpha x \leq \alpha y$ for $\alpha \in \mathbb{R}_+$; and
- it is an upper semilattice with respect to $\leq$ with join operation $\vee$.

Definition 4.3.2. Let $\mathbb{R}_+ = \{a \in \mathbb{R} \,|\, a \geq 0\}$. A *Riesz space* is a partially-ordered vector space $(V, \leq)$ over $\mathbb{R}$ such that

- if $x \leq y$, then $x + w \leq y + w$;
- if $x \leq y$, then $\alpha x \leq \alpha y$ for $\alpha \in \mathbb{R}_+$; and
- it is an upper semilattice with respect to $\leq$ with join operation $\vee$.

It follows that the space is also a lattice with meet operation $x \wedge y = -(-x \vee -y)$.

Many standard vector spaces are Riesz spaces.

Example 4.3.3. The following are Riesz spaces:

- $\mathbb{R}^n$ with the pointwise ordering;
- the set of bounded sequences of real numbers with pointwise ordering;
- the set of signed measures on a measurable space;

- the set of bounded measurable functions on a measureable space.

Unlike the real numbers, there are elements that are neither negative nor positive, but a notable characteristic of Riesz spaces is that every element decomposes uniquely into its positive and negative parts.

Definition 4.3.4. For v an element of a Riesz space, define $v^+ = v \vee 0$, $v^- = (-v) \vee 0$ and $|v| = v \vee -v = v^+ + v^-$.

Then v^+ and v^- are the unique positive elements such that $v = v^+ - v^-$ and $v^+ \wedge v^- = 0$. Thus Riesz spaces are completely characterized by their positive elements. This often simplifies constructions, as one can often prove a property for the positive elements, then extend to the entire space using this decomposition.

Given a Riesz space V, let V^+ denote the set of positive elements of V. Using the decomposition property mentioned above, it follows that $V = V^+ - V^+$, where $-$ applied to sets denotes elementwise subtraction.

4.3.1 Order convergence

Every topology gives rise to a notion of convergence. For normed spaces, one usually studies convergence in the norm topology. However, ordered spaces also carry an *order topology*.

Definition 4.3.5. Let D be a directed set and V a Riesz space. A *net* $\{v_\alpha\}_{\alpha \in D}$ is a function $D \to V$. We say that the net is increasing (respectively, decreasing) and write $\{v_\alpha\}\uparrow$ (respectively, $\{v_\alpha\}\downarrow$) if $\alpha \leq_D \beta$ implies $v_\alpha \leq_V v_\beta$ (respectively, $v_\alpha \geq_V v_\beta$).

Definition 4.3.6. Given a decreasing net $\{x_\alpha\}$, we write $\{x_\alpha\} \downarrow 0$ if $\inf\{x_\alpha\} = 0$.

Definition 4.3.7 (Order convergence)**.** We say that a net $\{x_\alpha\}$ *converges in order* to x and write $x_\alpha \to x$ if there is a decreasing net $\{y_\alpha\} \downarrow 0$ such that for all α, $|x_\alpha - x| \leq y_\alpha$.

In general, this notion of convergence is neither weaker nor stronger than convergence in norm. However, when a net converges in both order and norm, it converges to the same value in both.

4.3.2 Riesz subspaces, solids, ideals and bands

In the theory of Riesz spaces, there are classes of subspaces that have many interesting properties that will be used in our constructions.

Definition 4.3.8. A subset S of a Riesz space is

- *solid* if $x \in S$ and $|y| \leq |x|$ implies $y \in S$,
- an *ideal* if it is a solid linear subspace,
- a *band* if it is an ideal and closed under existing suprema.

Definition 4.3.9. We say that a Riesz space V is *Archimedean* if for every $v \in V^+$, $\{v/n\}_{n\in\mathbb{N}} \downarrow 0$. Furthermore, if every bounded subset of V admits a supremum, then we say that V is *Dedekind complete*.

Theorem 4.3.10 ([2])**.** *Let V be a Riesz space. The following are equivalent:*

- *V is Dedekind complete;*
- *every positive ascending bounded net $0 \leq \{v_\tau\}\uparrow$ has a supremum;*
- *every positive descending net $0 \leq \{v_\tau\}\downarrow$ has an infimum.*

Proposition 4.3.11. *Every band in a Dedekind complete Riesz space is Dedekind complete.*

Definition 4.3.12. A Riesz subspace $A \subseteq V$ is said to be *order dense* if for every element $0 < v \in V$ there is an element $a \in A$ such that $0 < a \leq v$.

Theorem 4.3.13 ([2], Ch. 1)**.** *A Riesz subspace A is order dense in an Archimedean Riesz space V iff for every $v \in V^+$, $\{a \in A \,|\, 0 \leq a \leq v\} \uparrow v$.*

4.3.3 Order-continuous functions

As usual when studying vector spaces with extra structure, we care only about linear maps that interact nicely with the extra structure. In our case, the linear functions will have to respect the partial order.

We call a linear function $f : V \to W$ *positive* if it maps positive elements of V to positive elements of W; that is, it restricts to a function $V^+ \to W^+$. A linear function is *regular* if it can be written as the difference of two positive functions.

Definition 4.3.14. A linear function $T : V \to W$ is *order-continuous* if it is continuous in the order topology. Equivalently, T is *order-continuous* if $Tv_\alpha \to Tv$ whenever $\{v_\alpha\}$ is an increasing net with supremum v.

We can also characterize the positive order-continuous functions as those that preserve existing suprema and infima.

Order continuity interacts well with order density. Indeed, it is possible to show using Theorem 4.3.13 that if two order-continuous functions $f_1, f_2 : V \to W$ agree on an order dense set, and if V is Archimedian, then $f_1 = f_2$. This property will come in handy when constructing our model.

Theorem 4.3.15 ([2])**.** *If W is Dedekind complete, then the set of order-continuous linear functions $V \to W$ is a band in the space of regular functions, thus forms a Dedekind-complete Riesz space.*

Definition 4.3.16. A Riesz space is *separated* if for every distinct pair $v_1, v_2 \in V$, there exists an order-continuous linear functional $f : V \to \mathbb{R}$ such that $f(v_1) \neq f(v_2)$.

4.3.4 Normed Riesz spaces

Now we will introduce normed Riesz spaces. In the context of probabilistic semantics, the norm has an important role, as it can be used to distinguish between arbitrary measures and (sub)-probability distributions, the measures with norm at most 1.

Definition 4.3.17. Let V be a real vector space. A *norm* is a function $\|\cdot\| : V \to \mathbb{R}^+$ such that:

- $\|v\| = 0$ iff $v = 0$
- $\|\alpha v\| = |\alpha| \, \|v\|$
- $\|v + u\| \leq \|v\| + \|u\|$.

For Riesz spaces, we require the norm to satisfy the additional property

$$|v| \leq |u| \text{ implies } \|v\| \leq \|u\|.$$

If the Riesz space is also complete with respect to the norm, we call it a *Banach lattice*. In vector space models of linear logic, the norm is typically used to distinguish between the product $\&$ and the coproduct $\oplus$, as they both have the same underlying set, but distinct norms. However, in the context of probabilistic programming, the norm also has the extra role of allowing the interpretation of recursive programs.

Example 4.3.18. The set $\mathcal{M}(\mathbb{R})$ of signed measures over the Borel σ-algebra on $\mathbb{R}$ is a Riesz space (cf. §4.3.6). We can equip it with the *total variation* norm

$$\|\mu\| = \mu^+(\mathbb{R}) + \mu^-(\mathbb{R}).$$

Theorem 4.3.15 shows that by assuming the right amount of structure on the Riesz space, the set of order-continuous linear functions between Riesz spaces also has a lattice structure. It is not immediately clear whether this result generalizes to the normed case. Luckily, Dedekind completeness is once again enough.

Example 4.3.19. Let V and W be Riesz spaces with W Dedekind complete. The set of order-continuous linear functions $V \to W$ can be equipped with the *regular norm*

$$\|T\|_r = \sup_{\|x\|_V \leq 1} \||T|(x)\|_W$$

where $|T|$ is given by Theorem 4.3.15 and Definition 4.3.4.

Definition 4.3.20. Let V be a normed Riesz space. The *closed unit ball* of V is the set $\mathcal{B}(V) = \{v \in V \mid \|v\| \leq 1\}$.

In categories of normed vector spaces, the relevant morphisms are the *norm-continuous* linear functions.

Definition 4.3.21. A linear function f between normed Riesz spaces V and W is said to be *norm-continuous* (or *norm-bounded*) if $\sup_{v \in \mathcal{B}(V)} \|f(v)\|$ is finite.

These objects can be organized in a category **NRiesz**.

Definition 4.3.22. The category **NRiesz** has separated normed Riesz spaces as objects and order-continuous, norm-continuous, positive with norm at most 1 linear functions as morphisms.

It is important to note that the definition of morphisms is not redundant. Indeed, there are order-continuous morphisms that are not norm-continuous and vice-versa.

Banach lattices

Banach lattices are normed Riesz spaces that are also Banach spaces. In general, the space of all norm-continuous linear functions between Banach lattices is not a Banach lattice, making them unable to give semantics to linear implication.

Theorem 4.3.23 ([39])**.** *If V and W are Banach lattices, then the set of order-continuous linear functions between V and W is a Banach lattice.*

In light of the theorem above, it makes sense to focus on order-continuous linear functions. Furthermore, every order-continuous linear function between a Banach lattice and an arbitrary normed Riesz space is norm-continuous [2].

Definition 4.3.24. The category $\mathbf{BanLat_1}$ has separated Banach lattices as objects and order-continuous positive linear functions of norm at most one as morphisms.

A subtlety when working with a norm and a partial order is that there are two distinct notions of convergence in play that on the surface appear only tenuously related. However, a useful property has been identified in the literature that brings some harmony between the two.

Definition 4.3.25. A normed Riesz space is said to satisfy the *(sequential) weak Fatou property* if every norm-bounded monotone (sequence) net has a supremum.

In the context of program semantics, the sequential version of this property has been used before to interpret recursive programs [28, 37].

Theorem 4.3.26. *Let $f : V \to V$ be a positive order-continuous function (not necessarily linear) such that $f(\mathcal{B}(V)) \subseteq \mathcal{B}(V)$. If V satisfies the weak Fatou property, then f admits a fixpoint.*

Proof. It can be directly shown that the limit of the ω-chain $\{f^n(0)\}_{n\in\mathbb{N}}$ is a fixpoint of f. Note that when f is linear, the theorem is trivially true, since $f(0) = 0$. □

Theorem 4.3.27 ([2])**.** *Every band in a Banach lattice is a Banach lattice.*

4.3.5 Dualities

The category $\mathbf{BanLat_1}$ seems to be a good candidate in which to interpret intuitionistic linear logic. However, since the linear negation connective $(-)^\perp$ is usually interpreted as the linear dual $V \multimap \mathbb{R}$ in models of linear logic based on vector spaces over $\mathbb{R}$, $\mathbf{BanLat_1}$ would not be able to model *classical* linear logic, since there are examples of Banach lattices that are not isomorphic to their bidual, e.g. finitely supported real sequences.

A recurring challenge in models of linear logic is to make an involutive linear negation—typical of finite-dimensional spaces—coexist with $!V$, which requires infinite-dimensional spaces. Since we are interested in defining a model of classical linear logic, we should only work with Riesz spaces that are isomorphic to their bidual.

Definition 4.3.28. Let V^σ denote the space of order-continuous functionals $V \multimap \mathbb{R}$. A Riesz space V is said to be *perfect* if the map $\sigma_V = \lambda x f\,.\, f(x) : V \multimap V^{\sigma\sigma}$ is an isomorphism.

We will write σ for σ_V when V is clear from context.

Definition 4.3.29. The category $\mathbf{PBanLat_1}$ has perfect Banach lattices as objects and positive order-continuous linear functions of norm at most one as morphisms.

The following theorem sheds light on how such spaces are useful for program semantics and how our construction relates to existing work.

Theorem 4.3.30 ([59], v. XIII, Th. 41.4)**.** *Let V be a separated normed Riesz space. Then V is perfect and Banach iff V has the weak Fatou property.*

Although the definition of perfect spaces is simple, it is difficult to manipulate in practice. The following theorem perfectly characterizes such spaces and provides an alternative definition:

Theorem 4.3.31 ([2])**.** *A Riesz space V is perfect iff*

- *it is separated;*
- *whenever $0 \leq x_\alpha \uparrow$ and $\sup_\alpha\{f(x_\alpha)\} < \infty$ for all positive $f \in V^\sigma$, there exists $x \in V$ such that $0 \leq x_\alpha \uparrow x$.*

Corollary 4.3.32. *Bands of perfect Riesz spaces are also perfect.*

Lemma 4.3.33. *Every perfect Riesz space is Dedekind complete.*

Proof. The proof follows from the second condition of Theorem 4.3.31. □

Lemma 4.3.34 ([2])**.** *For every Riesz space V, V^σ is Dedekind complete.*

Theorem 4.3.35. *Every Riesz space of the form V^σ is perfect.*

Proof. To show the first point of Theorem 4.3.31, assume that $f_1 \neq f_2 \in V^\sigma$. Then there is $v \in V$ such that $f_1(v) \neq f_2(v)$. Using the fact that $\lambda f.f(v)$ is an element of $V^{\sigma\sigma}$, we can conclude that V^σ is separated. For the second

point, let us assume that $0 \leq \{f_\alpha\}\uparrow$ and that for all $F \in V^{\sigma\sigma}$, if $F \geq 0$, then $\sup_\alpha F(f_\alpha) < \infty$. From this hypothesis, it follows that for all $v \in V$, if $v \geq 0$, then $\sup_\alpha f_\alpha(v) = \sup_\alpha \sigma(x)(f_\alpha) < \infty$. This means that the function $f(x) = \sup_\alpha f_\alpha(x)$ is well-defined, linear, and order-continuous. By Lemma 4.3.34, V^σ is Dedekind complete and f bounds f_α. □

An interesting fact that is not obvious from the definitions is that the bidual of Riesz spaces can be seen as a sort of completion procedure.

Lemma 4.3.36 ([2])**.** *Let V be an Archimedean Riesz space. The set $\sigma(V)$ is an order-dense Riesz subspace of $V^{\sigma\sigma}$.*

It is possible to categorify the theorem above by showing that $(-)^{\sigma\sigma}$ is a functor from a category of Riesz spaces and order-continuous linear functions to a category of perfect Riesz spaces with the same morphisms. This functor is defined analogously to the continuation monad from the theory of programming languages. Furthermore, there is an obvious forgetful functor $U : U : \mathbf{PBanLat_1} \to \mathbf{BanLat_1}$.

Theorem 4.3.37. *The functor $(-)^{\sigma\sigma} : \mathbf{BanLat_1} \to \mathbf{PBanLat_1}$ is left adjoint to the forgetful functor U.*

Proof. We observe that if $f : V \multimap W$, then $\sigma^{-1} \circ f^{\sigma\sigma} : V^{\sigma\sigma} \multimap W$. In the other direction, if we have a function $f : V^{\sigma\sigma} \multimap W$, we can consider its restriction $f \restriction V : V \multimap W$. To show that these operations are inverses, we use Theorem 4.3.13 and Theorem 4.3.36, which allow us to show that if two order-continuous functions agree on $\sigma(V)$, then they agree everywhere. □

Lemma 4.3.38. *If V is a separated Riesz space, then the function $\sigma : V \multimap V^{\sigma\sigma}$ is injective.*

The theorem and lemma above are useful because they imply that if V is a separated Riesz space and W is a perfect Riesz space, then every order-continuous linear function $f : V \multimap W$ extends uniquely to a function $V^{\sigma\sigma} \multimap W$.

There is a similar free construction between normed Riesz spaces and perfect Banach lattices.

Theorem 4.3.39. *The forgetful functor* $U : \mathbf{PBanLat_1} \to \mathbf{NRiesz}$ *has a right adjoint.*

Proof. The proof can be found in the appendix. □

Remark 4.3.40. It might possible to define a semantics for an expressive probabilistic programming language using only σ-perfect Riesz spaces, those that are isomorphic to their sequential bidual. If that is the case, it would be necessary to have a theorem analogous to Theorem 4.3.31.

The separability condition would need to be kept to guarantee injectivity, but the other conditions require further investigations, which we leave to future work.

4.3.6 Signed measures as Riesz spaces

Measures are usually defined as countably additive, nonnegative real-valued functions on a σ-algebra. *Signed measures* provide a slight generalization by dropping the requirement of nonnegativity.

Definition 4.3.41. Let (X, Σ) be a measurable space. A *signed measure* is a function $\mu : \Sigma \to \mathbb{R}$ such that $\mu(\emptyset) = 0$ and $\mu(\bigcup_{i\in\mathbb{N}} A_i) = \sum_{i\in\mathbb{N}} \mu(A_i)$ for disjoint sets $(A_i)_{i\in\mathbb{N}}$.

An important difference between ordinary measures and signed measures is that signed measures come equipped with a natural vector space structure. Indeed, it can be shown that signed measures are perfect Riesz spaces.

Theorem 4.3.42 ([39])**.** *Let (X, Σ) be a measurable space. The space $\mathcal{M}(X, \Sigma)$ of signed measures is a normed Riesz space.*

Proof. The vector space structure is defined pointwise with lattice structure defined by $\mu \vee \nu = (\mu - \nu)^+ + \nu$ using the Hahn-Jordan decomposition. □

The proof that these spaces are perfect is quite involved and can be found in [39]. The key insight is that the space of signed measures can be realized as the sequential order dual of a certain Riesz space. Since the Riesz space of sequential order-continuous linear functionals is a band in the set of regular functionals, we can conclude that it is perfect.

Theorem 4.3.43. *Let (X, Σ) be a measurable space. The space $\mathcal{M}(X, \Sigma)$ of signed measures with the total variation norm is a perfect Banach lattice.*

4.4 Models of linear logic

The categorical semantics of linear logic is very well understood; see Mellies [63] for an overview. In this section, we show that $\mathbf{PBanLat_1}$ is a model of classical linear logic.

4.4.1 Symmetric Monoidal Closed Structure

In order for $\mathbf{PBanLat_1}$ to interpret the multiplicative fragment of linear logic, i.e. give semantics to a linear λ-calculus with tensors, it must be a symmetric monoidal closed category. Concretely, it needs a *monoidal product* $\otimes$ such

that for every object A, the functor $A \otimes -$ has a right adjoint $A \multimap -$, known as *linear implication*.

For models based on vector spaces, the monoidal product is typically given by the *tensor product*. For such models, linear implication has a natural interpretation in terms of linear functions. Furthermore, since our spaces are perfect, we have an involutive *linear negation* $A^\perp$ defined as the space $A \multimap \mathbb{R}$, and the the equation $A \otimes B = (A \multimap B^\perp)^\perp$ holds, giving a model of *classical* linear logic. Thus the tensor product $\otimes$ can be defined in terms of linear implication $\multimap$ and negation $^\perp$ in such models.

Note that this circumvents one of the main complications with the model of [36], where the existence of a suitable monoidal product is established non-constructively using a categorical density argument.

Internal Homs

Since the category $\mathbf{PBanLat_1}$ has order-continuous linear functions with norm at most 1 as morphisms, it makes sense to define the internal hom object $V \multimap W$ as the space of order-continuous linear functions between perfect Banach lattices V and W. This definition is justified by the following theorem.

Lemma 4.4.1. *If V and W are perfect Riesz spaces, then the set of order continuous linear functions $V \multimap W$ is a perfect Riesz space.*

Proof. By Theorem 4.3.15, $V \multimap W$ is a Riesz space. Applying Theorem 4.3.31, we can also show that it is perfect. To show separability, let $f_1, f_2 : V \multimap W$ be distinct functions. Then there is a point $v \in V$ such that $f_1(v) \neq f_2(v)$. Since W is perfect, it is separated, therefore there exists $g : W \multimap \mathbb{R}$ such that $g(f_1(v)) \neq g(f_2(v))$. Then the order-continuous function $\lambda f\,.\,g(f(v))$ separates the points f_1 and f_2, therefore $V \multimap W$ is separated.

Now let $0 \leq \{f_\alpha\}\uparrow$ be an increasing net such that $\sup_\alpha F(f_\alpha) < \infty$ for all positive $F : (V \multimap W) \multimap \mathbb{R}$. We can define an f such that $f_\alpha \uparrow f$ pointwise. Let $v \in V^+$ and let $F : W \multimap \mathbb{R}$ be a positive functional. Consider the functional $\lambda f . F(f(v)) : (V \multimap W) \multimap \mathbb{R}$. By hypothesis, $\sup_\alpha(F(f_\alpha(v))) < \infty$, and since W is perfect and $\{f_\alpha(v)\}$ is a positive net in W, there exists $f(v) \in W$ such that $f_\alpha(v) \uparrow f(v)$. This defines f on elements of V^+, and for arbitrary $v \in V$ we take $f(v) = f(v^+) - f(v^-)$. Then $\sup_\alpha f_\alpha = f$. □

From Theorem 4.3.23 and the theorem above, it follows that if V and W are perfect Banach lattices, then so is $V \multimap W$. By using standard techniques from the literature on vector models of linear logic, we have

Theorem 4.4.2. *The operation* $\multimap : \mathbf{PBanLat_1}^{op} \times \mathbf{PBanLat_1} \to \mathbf{PBanLat_1}$ *is functorial.*

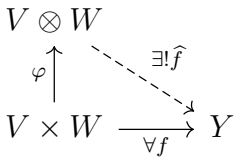
Monoidal structure

As mentioned above, the monoidal structure on vector space models of linear logic is usually defined as a tensor product, and monoidal closure is obtained from the universal property of tensor products. The usual recipe for defining tensor products is to use a free construction modulo the tensor product equations. When working with infinite-dimensional spaces, a completion procedure may be required as well.

Indeed, this is the approach taken in [40], in which a tensor product is defined for perfect Riesz spaces via a more traditional construction using the completion of the algebraic tensor product. It is also shown in [40] that $V \otimes W \cong (V \multimap W^\perp)^\perp$, meaning that their construction is isomorphic to ours.

In contrast, our construction starts with the definition $V \otimes W \triangleq (V \multimap W^\sigma)^\sigma$, as required by the laws of linear logic. We then show that it satisfies the expected

universal property of tensor products:



(4.1)

where φ and f are bilinear functions. We show this using the fact that the internal hom can be used to classify bilinear functions using $V \multimap (W \multimap Y)$, then showing that this space is isomorphic to $V \otimes W \multimap Y$.

Lemma 4.4.3. $V \otimes W \multimap Y \cong V \multimap W \multimap Y$.

Proof. Recall that if V and W are perfect Riesz spaces, then $V \multimap W \cong W^\sigma \multimap V^\sigma$. Then

$$\begin{aligned} V \otimes W \multimap Y &= (V \multimap W^\sigma)^\sigma \multimap Y \\ &\cong Y^\sigma \multimap (V \multimap W^\sigma) \cong V \multimap Y^\sigma \multimap W^\sigma \\ &\cong V \multimap W \multimap Y. \end{aligned}$$

□

Theorem 4.4.4. *$V \otimes W$, defined as $(V \multimap W^\sigma)^\sigma$, satisfies the universal property of tensor products* (4.1).

Proof. Observe that the set of (norm bounded) bilinear order-continuous functions $V \times W \to Y$ is (isometrically, in the normed case) isomorphic to $V \multimap W \multimap Y$. Restating the diagram (4.1) in this language, we must show that $V \otimes W \multimap Y \cong V \multimap W \multimap Y$. This is exactly Lemma 4.4.3. □

Using the universal property (4.1) and the (easy to prove) facts that $V \otimes (W \otimes Y) \cong (V \otimes W) \otimes Y$ and $V \otimes W \cong W \otimes V$, we can conclude:

Theorem 4.4.5. $\mathbf{PBanLat_1}$ *is a symmetric monoidal closed category.*

It is difficult in general to give an intuitive characterization of the elements of a tensor product. This is also the case with our construction. Nevertheless, in the context of measures, we can give some intuition for the elements of $\mathcal{M}(A) \otimes \mathcal{M}(B)$. Let μ_A and μ_B be probability distributions on measurable spaces A and B, respectively. The product distribution $\mu_A \otimes \mu_B$ is the joint probability distribution on $A \times B$ with marginals μ_A and μ_B obtained by sampling μ_A and μ_B independently. This is an element of $\mathcal{M}(A) \otimes \mathcal{M}(B)$, but there are also other joint distributions in $\mathcal{M}(A) \otimes \mathcal{M}(B)$ that do not represent independent samples. For example, let $A = B = \{0, 1\}$ and consider the joint distribution $\frac{1}{2}(\delta_0 \otimes \delta_0 + \delta_1 \otimes \delta_1)$. Sampling this distribution returns $(0, 0)$ or $(1, 1)$, each with probability $1/2$, so the two components are clearly not independent.

In general, not every joint distribution is an element of the tensor product, as explained in [26]. From a programming point of view, the universal property of tensor products says that the behavior of a program taking inputs of type $\mathcal{M}(A) \otimes \mathcal{M}(B)$ is fully characterized by its behavior on inputs that are independent distributions over A and B.

4.4.2 $*$-autonomous categories

Classical linear logic differs from its intuitionistic variant by requiring that linear negation be involutive, that is, $A^{\perp\perp} = A$ for every formula A. In our case, the dualizing object is $\mathbb{R}$, the unit is the linear function $\sigma_V : V \to V^{\sigma\sigma}$, and the isomorphism holds by assumption.

Theorem 4.4.6. $\mathbf{PBanLat_1}$ *is a* $*$*-autonomous category.*

4.4.3 Cartesian and co-Cartesian structure

Cartesian and co-Cartesian structure are useful in the formation of product and sum types. In models of linear logic, these are represented by linear conjunction $\&$ and disjunction $\oplus$, respectively. In $\mathbf{PBanLat_1}$, both operations have $V \times W$ as their underlying set with lattice operations defined componentwise. In the normed case, we can distinguish them by choosing different norms.

Definition 4.4.7. Let V and W be normed Riesz spaces. We define

- the product $V \,\&\, W = (V \times W, \|-\|_{\text{sum}})$, where $\|(v,w)\|_{\text{sum}} = \|v\| + \|w\|$.
- the coproduct $V \oplus W = (V \times W, \|-\|_{\max})$, where $\|(v,w)\|_{\max} = \max(\|v\|, \|w\|)$.

Since convergence for both is defined componentwise, by using Theorem 4.3.30 we can show that if V and W are perfect and Banach, then $V \,\&\, W$ and $V \oplus W$ are as well. The unit $\top$ for the product and 0 for the coproduct are both the trivial Riesz space $\{0\}$.

Theorem 4.4.8. $\mathbf{PBanLat_1}$ *is (co-)Cartesian.*

4.4.4 Exponentials

The trickiest part in defining models of linear logic is interpreting the exponential modality !, which can be used as a type constructor in a linear programming language that allows variables to be reused and discarded. In particular, this modality recovers the expressive power of the simply-typed λ-calculus and, in the context of probabilistic models of linear logic, it allows distributions to be resampled. Before we present its formal construction, we will give some intuition behind it.

In many concrete models, the construction of $!V$ captures the idea that its elements are those of the form $V^{\otimes n}$ for $n \geq 0$. This intuition also holds for vector space models, with the additional requirement that it must also accommodate a vector space structure.

Categorically, such a modality is captured by a monoidal comonad $!$ equipped with operations that model the structural rules of *contraction* and *weakening*.

In this section we are going to present two distinct, though similar, exponential construction, the second of which is inspired by [55].

Order-Continuous Exponential

Let V be a perfect Banach lattice and consider the positive cone $C(V)^+$ of bounded monotonic functions $f : \mathcal{B}(V)^+ \to \mathbb{R}$ such that for every ascending net $\{x_\alpha\}_\alpha \uparrow x$, $\lim_\alpha f(x_\alpha) = f(x)$. This set can be equipped with the pointwise partial order $f \leq g$ iff $\forall x\ f(x) \leq g(x)$ and be made into a vector space by taking the set $C(V) = C(V)^+ - C(V)^+$.

Lemma 4.4.9. *The space $C(V)$ is a Banach lattice.*

Proof. The complete proof can be found in the appendix (Theorem A.4.3). □

In order to define the exponential $!^o V$, we are interested in a particular subspace of $C(V)^\sigma$. For every $v \in \mathcal{B}(V)^+$, let $\delta_v \in C(V)^\sigma$ be the *Dirac delta distribution* defined as the functional $\delta_v(f) = f(v)$. We want $!V$ to be a perfect Banach lattice contained in $C(V)^\sigma$ which contains the linear span of the delta distributions as an order-dense subset. At a high level, our construction comprises two steps. The first step consists of extending the linear span of the Dirac distributions to a Riesz space containing this span as an order-dense subset.

Next, we take the *order closure* of this Riesz space in $C(V)^\sigma$ and obtain a perfect Banach lattice. By transitivity of order-density, the span of the delta distributions is once again dense in this space.

Free Riesz spaces

There has been much work done on free constructions for partially ordered vector spaces. We are interested in the construction of [91] showing that every partially ordered vector space can be extended to a Riesz space.

Theorem 4.4.10 ([91]). *Let A be a partially ordered vector space and V a Riesz space. There is a Riesz space $\tilde{A}$ and an order-continuous bipositive*[1] *inclusion function $\iota : A \to \tilde{A}$ such that the image of ι is order-dense in $\tilde{A}$.*

Lemma 4.4.11 ([18]). *If $v_1, \ldots, v_n \in \mathcal{B}(V)^+$ are distinct vectors, then the distributions $\delta_{v_1}, \ldots, \delta_{v_n}$ are linearly independent.*

Let $\Delta(V)$ be the Riesz space generated by the partially ordered vector space spanned by $\{\delta_v\}_{v \in \mathcal{B}(V)^+}$ according to Theorem 4.4.10.

Lemma 4.4.12. *The space $\Delta(V)$ is Archimedean.*

Proof. This is a direct consequence of the fact that $C(V)^\sigma$ is Archimedean and Corollary 3.10 from [91]. □

L-spaces and order closure

For the next part, we need to introduce some terminology. We say that a Banach lattice V is an *L-space* if for all positive vectors v_1, v_2, $\|v_1 + v_2\| = \|v_1\| + \|v_2\|$. These spaces are also called abstract Lebesgue spaces because they can be

[1] that is, preserves the poset structure exactly

seen as a generalization of L_1 spaces. In particular, for every measurable space, the space of signed measures over it is an L-space.

As previously mentioned, a tricky aspect of Banach lattices is that the order and norm topologies do not always give rise to the same continuous functionals. In L-spaces, this distinction disappears. This will be useful in showing that our construction gives both a Riesz and a Banach space.

Lemma 4.4.13. *The space $C(V)^\sigma$ is an L-space.*

Proof. Since $C(V)$ is a Banach lattice, so is $C(V)^\sigma$. Let $F \in C(V)^\sigma$ and positive. Then $\|F\| = F(\lambda x.1)$, as the function $\lambda x.1$ is the maximum element of the unit ball in $C(V)$ (see Theorem A.4.3). Therefore, for positive F_1, F_2, $\|F_1 + F_2\| = F_1(e) + F_2(e) = \|F_1\| + \|F_2\|$. □

We define the *exponential* $!^o V$ of a space V as the order-closure of $\Delta(V)$ in $C(V)^\sigma$.

Theorem 4.4.14. *The space $!^o V$ is a perfect Banach lattice with the the span of $\{\delta_v\}_{v \in \mathcal{B}(V)^+}$ as an order-dense subset.*

Proof. The proof can be found in the appendix. □

Lemma 4.4.15. *Let V and W be perfect Banach lattices. Every order-continuous function $f : \Delta(V) \multimap W$ extends uniquely to an order-continuous function $\tilde{f} : !^o V \multimap W$.*

Proof. Let $f : \Delta(V) \multimap W$ be an order-continuous linear function. To show existence, let $v \in !^o V$. If $v \in \Delta(V)$, then $\tilde{f}(v) \triangleq f(v)$. Otherwise, by order density and Lemma 4.4.12 we can assume that $v = \sup_{v' \leq v, v' \in \Delta(V)}$. In this case, we define

$$\tilde{f}(v) \triangleq \sup_{v' \leq v, v' \in \Delta(V)} f(v').$$

The expression above is well defined, because by assumption W is a perfect Banach lattice, and since f is norm-continuous, the net $f(v')$ is norm-bounded; then by Theorem 4.3.30 and the weak Fatou property we can conclude that the supremum exists. The uniqueness of the extension is a direct consequence of order-continuity and density. □

A more interesting consequence of Lemma 4.4.15 is that in order to define order-continuous functions $!^o V \multimap W$, it suffices to define order-continuous functions on the linear span of the Dirac distributions, which in turn are completely characterized by their action on the set $\{\delta_v\}_{v \in \mathcal{B}(V)^+}$.

Before defining the comonadic structure of $!^o$, we show that it defines a functor.

Theorem 4.4.16. *The mapping $V \mapsto !^o V$ is functorial.*

Proof. The action on morphisms $f : V \to W$ is given by the function $!^o V \multimap !^o W$ generated by $\delta_v \mapsto \delta_{f(v)}$, which is well defined by the hypothesis that f is order-continuous. The functor laws follow as a consequence of Lemma 4.4.15:

$$!^o(\mathsf{id}_V)(\delta_v) = \delta_v = \mathsf{id}_{!^o V}(\delta_v)$$
$$!^o(f \circ g)(\delta_v) = \delta_{f(g(v))} = (!^o f \circ !^o g)(\delta_v).$$ □

Now we can define a comonad structure over $!^o$, where the counit $\varepsilon_V : !^o V \multimap V$ is the function generated by $\delta_v \mapsto v$ and the comultiplication $\rho_V : !^o V \multimap !^o !^o V$ is generated by $\delta_v \mapsto \delta_{\delta_v}$. Their order-continuity is straightforward to show.

In order to show that these functions satisfy the comonadic laws depicted in Figure 2.4, we once again use Lemma 4.4.15 and check that the diagrams commute at points δ_v.

Theorem 4.4.17. *The triple $(!^o, \varepsilon_V, \rho_V)$ is a comonad in $\mathbf{PBanLat_1}$.*

Proof. We need only show commutativity of the diagrams in Figure 2.4 for the points δ_v. This gives us the equations

$$\begin{aligned}
\rho_{!^o V}(\rho_V(\delta_v)) &= \rho_{!^o V}(\delta_{\delta_v}) = \delta_{\delta_{\delta_v}} \\
&= (!^o \rho_V)(\delta_{\delta_v}) = (!^o \rho_V)(\rho_V(\delta_v)) \\
(!^o \varepsilon_{!^o V})(\rho_V(\delta_v)) &= (!^o \varepsilon_{!^o V})(\delta_{\delta_v}) = \delta_v \\
(\varepsilon_{!^{o2} V})(\rho_V(\delta_v)) &= (\varepsilon_{!^{o2} V})(\delta_{\delta_v}) = \delta_v.
\end{aligned}$$

□

Finally, we must show that the Seely isomorphisms $!^o(V \,\&\, W) \cong !^o V \otimes !^o W$ and $!^o \top \cong 1$ hold. The former is given by the pair of functions generated by $\delta_{(v,w)} \mapsto \delta_v \otimes \delta_w$ and $\delta_v \otimes \delta_w \mapsto \delta_{(v,w)}$, and the latter is given by the functions generated by $\delta_0 \mapsto 1$ and $1 \mapsto \delta_0$. Both of these isomorphisms can be checked by a simple direct calculation.

Theorem 4.4.18. *The category* $\mathbf{PBanLat_1}$ *is a model of classical linear logic.*

Power Series Exponential

Definition 4.4.19. A Scott-continuous n-homogeneous function between perfect Banach lattices V and W is a Scott-continuous, norm-bounded function $f : V^+ + \to W^+$ such that for every positive scalar λ, $f(\lambda x) = \lambda^n f(x)$ and such that there exists an n-linear Scott-continuous function f_n such that $f(x) = f_n(x, \ldots, x)$.

In this section, we are implicitly assuming that n-homogeneous functions are always Scott-continuous and norm-bounded. Consider functions $f : \mathcal{B}(V)^+ \to \mathcal{B}(W)^+$ that can be written as an infinite sum of n-homogeneous functions:

$$f(x) = \sum_{n=0}^{\infty} f_n(x)$$

In this case, since we are assuming that the sequence $\sum_{n=0}^{N} f_n(x)$ is positive, monotonically increasing in N and norm-bounded by 1, by the weak-Fatou property, the infinite series converges in order.

These functions should be thought of as the positive unit ball of a larger space. Therefore, we define the space $S(V,W)^+$ as the set of functions $f : \mathcal{B}(V)^+ \to W^+$ that can be written as an infinite sum of n-homogeneous functions: $f(x) = \sum_{n=0}^{\infty} f_n(x)$ such that $\sup_x f(x)$ is finite. We can equip this space with the componentwise order, addition, multiplication by scalar, lattice structure and define the normed Riesz space $S(V,W) = S(V,W)^+ - S(V,W)^+$, where the norm is given by $\|f\| = \sup_x f(x)$ which is finite because $f(x) = (f^+ - f^-)(x) \leq (f^+ + f^-)(x)$ and both f^+ and f^- have finite norm.

Lemma 4.4.20. *If V and W are perfect Banach lattices, then the normed Riesz space generated by the n-homogeneous positive order-continuous functions is a perfect Banach lattice, for every $n \in \mathbb{N}$.*

Proof. The Riesz space structure is defined pointwise and we can show that this space has the weak Fatou property, making it perfect and Banach as well. Let (f_n^α) be an ascending net of n-homogeneous functions. We can define $(\sup_\alpha(f_n^\alpha))(x) = \sup_\alpha((f_n^\alpha)(x))$, which is well-defined because W has the weak Fatou property, f_n^α are monotonic and norm-bounded. From the fact that scalar multiplication is order-continuous we can conclude that $\sup_\alpha(f_n^\alpha)$ is also n-homogeneous. □

Lemma 4.4.21. *If V and W are perfect Banach lattices, then the space $S(V,W)$ is a perfect Banach lattice as well.*

Proof. We prove this by showing that $S(V,W)$ has the weak Fatou property which, by Theorem 4.3.30 is equivalent to perfection and Banach. Let $(f^\alpha)_\alpha$ be

an ascending net of positive, norm-bounded elements in $S(V, W)$. By assumption, W has the weak Fatou property and the nets $(f_n^\alpha)_\alpha$ are an ascending net of positive and norm-bounded n-linear functions, which by the lemma above is also a perfect Banach lattices for every n, meaning that it has a supremum. Therefore, we define $\sup f^\alpha = (\sup f_0^\alpha, \sup_\alpha f_1^\alpha, \dots)$. The expression above is an element of $S(V, W)$ because $\sum_n (\sup_\alpha f_n^\alpha(x, \dots, x)) = \sup_\alpha \sum_n f_n^\alpha(x, \dots, x)$. The identity above follows from the nets $(f_n^\alpha)_\alpha$ being monotonic in α. □

Lemma 4.4.22. *The composition of two positive power series $f : \mathcal{B}(V)^+ \to \mathcal{B}(U)^+$ and $g : \mathcal{B}(U)^+ \to \mathcal{B}(W)^+$ is a power series.*

Proof sketch. We sketch the proof here, which is largely based on the proof provided by Kerjean and Tasson [55]. First, since both operations map the positive unit cone into the positive cone, their composition is well-defined. Next, observe that the composition of an n-homogenous function with an m-homogenous function is an $n + m$ homogenous function. Finally, using these two facts, the decomposition of n-homogenous functions as its n-linear counterpart and the weak Fatou property, we can conclude. □

The theorem above implies that there is a category $\mathbf{PBanSeries}$ of perfect Banach lattices as objects and positive power series with norm at most 1 as morphisms.

A monoidal adjunction There is a simple forgetful functor $\mathbf{PBanLat_1} \to \mathbf{PBanSeries}$ that is identity on objects and maps linear functions f to the power series $(0, f, 0, \dots)$. In order to go the other direction we use a construction similar to the one of the first exponential. To define the exponential $!^s V$, we are interested in a particular subspace of $S(V, \mathbb{R})^\sigma$. For every $v \in \mathcal{B}(V)^+$, we define the Dirac delta distributions $\delta_v \in S(V, \mathbb{R})^\sigma$ as $\delta_v(f) = f(v)$. Let $\Delta(V)$ be the

free normed Riesz space generated by the span of the Dirac distributions. By applying Theorem 4.3.39, we define $!^s V = (\Delta(V))'^{\sigma}$.

Lemma 4.4.23. *For every perfect Banach lattice and $n \in \mathbb{N}$, there is an n-homogeneous map $\theta_n : V \to !^s V$ such that $\theta_n(v)(f_0, f_1, \dots) = f_n(v)$.*

Proof. By construction, $!^s V$ is an ideal in $S(V, \mathbb{R})^{\sigma}$ and $\delta_v \in !^s V$. Furthermore, for every $v \in \mathcal{B}(V)^+$ the operator $\theta_n(v)$ is an element of $(S(V, R))^{\sigma}$ and $\theta_n(v) \leq \delta(v)$, meaning that $\theta_n(v) \in !^s V$ and the function θ_n restricts to an n-homogeneous map $V \to !^s V$. □

Lemma 4.4.24. *The function $\delta : \mathcal{B}(V)^+ \to !^s V$ is a power series with norm 1.*

Proof. By the lemma above we can define $\delta = \sum_n \theta_n$. By construction, for every $v \in \mathcal{B}(V)^+$, $\delta(v)(f) = \sum_n \theta_n(v)(f) = \sum_n f_n(v) = f(v)$. □

Theorem 4.4.25. *There is a functor $!^s : \mathbf{PBanSeries} \to \mathbf{PBanLat_1}$*

Proof. It acts on objects by mapping a perfect Banach lattice V to $!^s V$ and maps a power series $f : V \to W$ to the linear function generated by $\hat{f}(\delta_v) = \delta_{f(v)}$. Note that this functor is analogous to the order-continuous exponential. The functor laws follow by a similar reasoning to Theorem 4.4.16. □

Lemma 4.4.26. *For every perfect Banach lattices V and W there is a natural isomorphism $\mathbf{PBanSeries}(V, W) \cong \mathbf{PBanLat_1}(!^s V, W)$*

Proof. Let $f \in \mathbf{PBanSeries}(V, W)$, we define $\hat{f}(d) = \sigma_W^{-1}(\lambda F : W^{\sigma} \,.\, d(F \circ f))$. The function above is well-defined because d is an order-continuous functional over power series and $F \circ f$ is the composition of a linear function with a power series and, therefore, it is also a power series. Conversely, if $f \in \mathbf{PBanLat_1}(!^s V, W)$ then we define the composite power series $f \circ \delta$. By Lemmas 4.4.22 and 4.4.24

we can conclude that $f \circ \delta$ is indeed a power series. To show that this is indeed an isomorphism we use the fact that $\{\delta_v\}$ is order-dense in $!^s V$. □

Lemma 4.4.27. *For every perfect Banach lattices V and W there is a natural isomorphism $!(V \,\&\, W) \cong !^s V \otimes !^s W$*

Proof. The proof is analogous to the order-continuous exponential case. □

The results above allows us to conclude:

Theorem 4.4.28. *The 4-tuple $(\mathbf{PBanLat_1}, \mathbf{PBanSeries}, !^s, U)$ is an LNL model.*

4.4.5 Fixed Points

In order to give semantics to recursive probabilistic programs we need to be able to compute fixed points of morphisms $!^s V \multimap V$ with norm ≤ 1. Since our objects are perfect Banach spaces, they satisfy the weak Fatou property, which in turn can be used to define fixed points of Scott continuous endofunctions. After all, every positive Scott-continuous function $f : \mathcal{B}(V) \to \mathcal{B}(V)$ induces an ascending chain $\{f^n(0)\}_{n\in\mathbb{N}}$ which is norm bounded by assumption and therefore admits a supremum.

Definition 4.4.29. A function $f : \mathcal{B}(V)^+ \to \mathcal{B}(W)^+$ is called *analytic* if there is a positive function $g : !^s V \multimap W$ with norm ≤ 1 such that $f(v) = g(\delta_v)$.

Due to order density, every function $g : !^s V \multimap W$ with norm ≤ 1 may only be associated to one analytic function. Therefore, the definition above suggests that if the function $\delta : \mathcal{B}(V) \to \mathcal{B}(!^s V)$ is Scott continuous then every analytic function admits a fixed point:

Theorem 4.4.30. *The function $\delta : \mathcal{B}(V)^+ \to \mathcal{B}(!^s V)^+$ is Scott continuous.*

Proof. Let $\{v_\alpha\}_\alpha \subset \mathcal{B}(V)^+$ be an ascending chain of positive elements which, by the weak Fatou property converges to an element in $\mathcal{B}(V)^+$. We want to show that $\delta(\sup_\alpha v_\alpha) = \sup_\alpha \delta(v_\alpha)$. We have the following equations

$$\delta(\sup_\alpha v_\alpha)(f) = f(\sup_\alpha v_\alpha) = \sup_\alpha f(v_\alpha) = \sup_\alpha \delta(v_\alpha)(f)$$

The second equation holds because f is order-continuous. □

Therefore, every analytic function g is also order-continuous, since they can be seen as the composition of δ with g, meaning that if it is monotonic we can compute its fixed point by using Theorem 4.3.26. Next, consider the program $\lambda F\, f\, .\, f\ (F\ f)\ :\ !^s(!^s(!^s V \multimap\ !^s V) \multimap\ V)) \multimap\ !^s(!^s V \multimap\ V) \multimap\ V$. It is possible to show that it is positive and that it has norm 1 therefore, it has a fixed point, which we call Fix. By construction, we have the equation $(\mathsf{Fix}\, F)(f) = f((\mathsf{Fix}\, F)(f))$, making $(\mathsf{Fix}\, F)(f)$ the fixed point of f. Furthermore, by construction, $\mathsf{Fix}\, F$ has type $!^s(!^s V \multimap V) \multimap V$, meaning that it can be interpreted in $\mathbf{PBanLat_1}$.

4.4.6 Filtered colimits in $\mathbf{PBanLat_1}$

Besides the basic linear logic structure, for the purposes of giving semantics to programming languages, there are other categorical properties that are useful to have. In particular, inductive types are interpreted as certain filtered colimits. We start with the following lemma

Lemma 4.4.31. *Let $V_{n n\in\mathbb{N}}$ be a sequence of perfect Banach lattices such that $V_n \subseteq V_{n+1}$ for every n, the union $(\bigcup_n V_n)'^\sigma$ is a perfect Banach lattice*

Proof. It is straightforward to show that $\bigcup_n V_n$ is a normed Riesz space. However, in general, this is space is neither Banach nor perfect. We fix this by applying Theorem 4.3.39, and obtaining the space $(\bigcup_n V_n)'^\sigma$. □

For the purposes of inductive types, we only care about diagrams where the arrows are injections.

Theorem 4.4.32. *The category* $\mathbf{PBanLat_1}$ *has filtered colimits when the morphisms are injections.*

Proof. Since the morphims are assumed to be injections, we can assume that $V_n \subseteq V_{n+1}$, for every n. We will prove that $((\bigcup_n V_n)'^{\sigma}, \iota_n)$ is a colimiting cocone, where ι_n is the injection $V_n \hookrightarrow (\bigcup_n V_n)'^{\sigma}$. Let $(W, f_n : V_n \to W)$ be a cocone. We define the universal morphism $\mathsf{fold}_{f_n} : (\bigcup_i V_i)'^{\sigma} \to W$ by applying Theorem 4.3.39 to the unique order-continuous linear positive function $f : \bigcup_n V_n \to W$ such that $f(x) = f_n(x)$, when $x \in V_n$. This function is well-defined by injectivity of the morphisms and by assumption that $(W, f_n : V_n \to W)$ is a cocone. More abstractly, this theorem follows from the fact that the forgetful functor is full and faithful, meaning that it reflects colimits. □

In future work we will study if we can use the theorem above to use $\mathbf{PBanLat_1}$ as a semantic basis to a probabilistic language with recursive types.

4.5 Probabilistic coherence spaces and Banach lattices

Probabilistic coherence spaces (PCS) [28] are a model of linear logic with many connections to vector space models of linear logic. Recently it has been shown by Ehrhard [36] that its intuitionistic fragment can be fully and faithfully embedded in a category of positive cones. In this section, we show that Banach lattices can handle not only the intuitionistic fragment of PCS, but the classical one as well. We make use of the vector space construction presented in the original paper [28].

The linear-algebraic intuition behind Definition 4.2.1 is that the web of every PCS corresponds to the positive unit ball of a partially-ordered vector space. This idea is used by Ehrhard and Danos [28] to define a functor that maps every PCS to a Banach space. It is possible to show that this vector space can be equipped with a Riesz space structure, where the order is defined pointwise.

Definition 4.5.1. Given a PCS $(|X|, \mathcal{P}X)$, we define $BX = \{u \in \mathbb{R}^{|X|} \,|\, |u| \in \mathcal{P}X\}$ and $eX = \bigcup_{\lambda>0} \lambda BX$. The pair $(eX, u \mapsto \sup_{u' \in \mathcal{P}X^\perp} \langle |u|, u' \rangle)$ is the normed Riesz space associated with the PCS $(|X|, \mathcal{P}X)$.

It is shown in [28] that eX is a Banach space. Furthermore, the lattice structure can be defined pointwise, making eX a Banach lattice. Later in this section we will show that e can be made into a functor.

4.5.1 PCoh and duality

Ehrhard and Danos have shown that the partial order plays an important role in understanding how to generalize **PCoh**. In their attempt to obtain an intrinsic representation for probabilistic coherence spaces, the authors use exclusively the norm and cannot prove the equation $e(X^\perp) = e(X)^\perp$, where $e(X)^\perp$ is the norm dual. That said, we can show that this functor preserves order-duality. Note that the proof uses the fact that 2^X is a directed set.

Theorem 4.5.2. *For every probabilistic coherence space X, there is a natural isomorphism $e(X^\perp) \cong e(X)^\sigma$.*

Proof. If $u \in e(X^\perp)$, consider the element $f_u = \lambda x. \langle u^+, x \rangle - \langle u^-, x \rangle$. It is possible to show that the function $\lambda x. \langle u, x \rangle$ is positive and Scott-continuous, therefore order-continuous for every $u \in \mathcal{P}(X)$. Using this result, it is not hard to show that $f_u \in e(X)^\sigma$.

Conversely, consider an element $f \in e(X)^\sigma$. Without loss of generality, we can assume that f is positive. We want to associate to f an element in $e(X^\perp)$. As is shown by [28], we can alternatively characterize the space $e(X)$ as

$$\{u \in \mathbb{R}^{|X|} \mid \exists \lambda > 0 \, \forall u' \in \mathcal{P}(X^\perp) \, \langle |u|, u' \rangle \leq \lambda\}.$$

Consider the function $f_\delta = \lambda x. f(\delta_x)$. Let us show that $f_\delta \in e(X^\perp)$. To do this, we show that for every $u \in \mathcal{P}(X)$, $\langle |f'|, u \rangle$ is uniformly bounded. Let $(u_\alpha)_{\alpha \in \mathbb{P}_{\mathrm{fin}}(X)}$ be the ascending net $u_{\alpha,a} = u_a$ if $a \in \alpha$ and 0 otherwise. By expanding the definition, we get the equality

$$\langle |f_\delta|, u_\alpha \rangle = \sum_{a \in |X|} |f(\delta_a)| u_{\alpha,a} =$$

$$\sum_{a \in |X|} |f(\delta_a u_{\alpha,a})| = \sum_{a \in |X|} f(\delta_a u_{\alpha,a}).$$

We get the last equality from f being a positive function. Since every u_α has finite support, the expression above is well defined.

$$\sum_{a \in |X|} f(\delta_a u_{\alpha,a}) = f\left(\sum_{a \in |X|} \delta_a u_{\alpha,a}\right) = f(u_\alpha)$$

Since f is order-continuous and monotone and $\{u_\alpha\}$ is an increasing net, we can conclude that $\langle |f_\delta|, u \rangle \leq f(u)$, therefore for every $u \in \mathcal{P}(X)$, $\langle |f_\delta|, u \rangle \leq \|f\|$ and $f_\delta \in e(X^\perp)$. If f is not positive, we decompose it as the difference of two positive maps $f = f^+ - f^-$ and define $f_\delta = f^+_\delta - f^-_\delta$.

A direct calculation shows that this is indeed an isomorphism. □

Corollary 4.5.3. *For every PCS $(X, \mathcal{P}(X))$ the vector space eX is a perfect Banach lattice.*

Since convergence for PCS is defined component-wise, it is possible to use a similar proof technique to show

Theorem 4.5.4. *The operation e is monoidal closed and functorial.*

Proof. The functoriality of e has been proven in [28]. □

Another important theorem which is direct to show is.

Theorem 4.5.5. *The functor $e : \mathbf{PCoh} \to \mathbf{PBanLat_1}$ is full and faithful.*

We still do not know whether this functor also preserves the exponential. That being said, there is a natural transformation $!^s e \to e!$ which maps δ_v to $v^!$. We conjecture that this transformation is a natural isomorphism.

4.6 Categories of Cones

As we have seen throughout this paper, every Banach lattice gives rise to a positive cone. Furthermore, since every $\mathbf{PBanLat_1}$ morphism $f : V \to W$ is positive and has norm at most 1, it restricts to a linear function $\mathcal{B}(V)^+ \to \mathcal{B}(W)^+$. Categories of positive cones have been studied in the context of models of linear linear logic. Therefore, it is natural to ask how our model relates to cone-based models of linear logic. We start by stating a few definitions from [25, 36], which assume that the cones are separated.

Using this notation, it seems appropriate to imagine that there should be a functor $\mathbf{PBanLat_1} \to \mathbf{CLatLin}$, where $\mathbf{CLatLin}$ is the category of directed complete cone lattices and Scott continuous linear functions. It is unclear, however, if there is a mapping on morphisms. Luckily, the lemma below guarantees that the mapping is well-defined.

Lemma 4.6.1. *Let V and W be two perfect Banach lattices and $f : V \to W$ a linear, positive and normed at most 1 function. The function f is order-continuous if, and only*

if, $\sup_{x \in A} f(x) = f(v)$ *whenever* $A \subseteq V^+$ *is a non-empty upwards-directed set with supremum* v*.*

Proof. This result is a direct consequence of the weak Fatou property. □

Since the mapping on morphisms is basically the identity, the functorial laws hold, which allows us to conclude that there is a functor $\mathbf{PBanLat_1} \to \mathbf{CLatLin}$.

Next, we would like to map every positive cone to a vector space. Let C be a positive cone and define

$$C - C = \{(c_1, c_2) \mid c_1, c_2 \in C\} / \sim,$$

Where $\sim$ is the binary relation $(c_1, c_2) \sim (c_3, c_4)$ iff $c_1 + c_4 = c_2 + c_3$. Intuitively, $C - C$ corresponds to the vector space of formal differences $c_1 - c_2$ of elements in C. The equivalence relation is used to capture the fact that, for instance, $(3, 2)$ and $(4, 3)$ should represent the same real number, since $3 - 2 = 1 = 4 - 3$.

Theorem 4.6.2. *Let* C *be a directed complete cone lattice.* $C - C$ *is a perfect Banach lattice.*

Proof. The proof can be found in the appendix. □

By linearity, Scott-continuous functions $f : C \to D$ with norm at most 1 extend to order- continuous functions $f : (C - C) \to (D - D)$ with norm at most 1 and we can prove that there is a functor $\mathbf{CLatLin} \to \mathbf{PBanLat_1}$.

Theorem 4.6.3. *The categories* $\mathbf{PBanLat_1}$ *and* $\mathbf{CLatLin}$ *are isomorphic.*

Proof. The functors are defined above and they are isomorphisms as a direct consequence of Lemma A.6.1. □

4.6.1 Measurability Tests

One of the main drawbacks behind models of effectful programming languages based on linear logic is that Girard's translation of intuitionistic logic into linear logic is call-by-name. In the context of probabilistic languages this means that once a distribution is sampled — i.e. a variable is used — there is no way of reusing the sample. This is problematic because in practice reusing samples is necessary. Ehrhard et al work around this issue by defining a CBV let operator that allows samples to be reused. Semantically, the interpration of this operator is subtle and, unfortunately, the category $\mathbf{CStab}$ cannot interpret this operation.

At a high-level, given a distribution $\mu : \mathcal{M}\mathbb{R}$ and a $\mathbf{CStab}$ morphism $f : \mathcal{M}\mathbb{R} \to \mathcal{M}\mathbb{R}$, the semantics of $(\mathsf{let}\ x = \mu\ \mathsf{in}\ f)$ is $\int (f \circ \delta)\, \mathrm{d}\mu$, where $\delta : R \to \mathcal{M}\mathbb{R}$ is the Dirac delta distribution. Therefore, it follows that the function $f \circ \delta$ has to be measurable and, due to measurability interacting poorly with general order-continuity, this operation is only ω-order continuous in its argument f. In order to guarantee the second requirement, the morphisms in $\mathbf{CStab}$ are only ω-order continuous. In order to guarantee the first invariant, the authors refine $\mathbf{CStab}$ into $\mathbf{CStab}_m$ using a semantic logical relations technique.

The take-away message is that we could, in principle, enhance $\mathbf{PBanLat_1}$ using a similar construction and guarantee that $f \circ \delta$ is measurable, but it would not solve the order-continuity problem, since $\mathbf{PBanLat_1}$ has general directed order-continuity as morphisms.

That being said, by using the two-level language, we still have a mechanism of reusing samples. In order to fully adapt their semantic machinery to the Banach lattice context we would need to work with ω-perfect Banach lattices, i.e. Banach lattices that are isomorphic to their sequential bidual. As far as we

know, the theory of ω-perfect Banach lattices is not as well developed as their directed counterpart, so we leave this for future work.

4.7 A Probabilistic Recursive Calculus

Variables	x, y, z		
Reals	r	$\in$	$\mathbb{R}$
MK Expressions	M	$::=$	$\ldots \mid \mathsf{uniform}$
LL Expressions	t, u	$::=$	$\ldots \mid !t \mid \mathsf{let}\ !x = t\ \mathsf{in}\ u \mid \mathsf{fix}\, t$
Types MK	τ	$::=$	$\ldots$
Types LL	$\underline{\tau}$	$::=$	$\ldots \mid !\tau$

Figure 4.1: Terms and Types

Now we can use $\mathbf{PBanLat_1}$ to give semantics to $\lambda^{\text{LL}}_{\text{MK}}$. Given the structure of $\mathbf{PBanLat_1}$ we are interested in extending $\lambda^{\text{LL}}_{\text{MK}}$ with recursion and exponentials. To reiterate, each language has its own typing relation, $\Gamma \vdash_{MK} M : \tau$ for the first level language and $\Delta; \Gamma \vdash_{LL} t : \overline{\tau}$ for the linear language, where Γ is the context for the linear variables and Δ is the context for the non-linear variables. We present the extended typing rules in Figure 4.2.

In order to define an $\lambda^{\text{LL}}_{\text{MK}}$ model we also need a category that can give semantics to the MK language. We use the category $\mathbf{Kern}$ of measurable sets as objects and Markov kernels as morphisms, ie. functions $f : A \times \Sigma_B \to [0, 1]$ such that for every a, $f(a, -)$ is a probability distribution and for every Y, $f(-, Y)$ is a measurable function. The type constructor $\mathcal{M}$ is given by a lax monoidal functor between the MK and LL languages.

Theorem 4.7.1. *There is a lax monoidal functor* $\mathcal{M} : \mathbf{Kern} \to \mathbf{PBanLat_1}$.

Proof. There is a standard functor $\mathcal{M}$ that maps measurable sets to the vector space of signed measures and Markov kernels $f : A \to MB$ to the linear func-

$$\textsc{Fix}\ \frac{\Delta;\Gamma \vdash_{LL} t : !\tau \multimap \tau}{\Delta;\Gamma \vdash_{LL} \mathsf{fix}\, t : \tau} \qquad \textsc{!-Intro}\ \frac{\Delta;\_ \vdash_{LL} t : \tau}{\Delta;\_ \vdash_{LL} !t : !\tau}$$

$$\textsc{!-Elim}\ \frac{\Delta;\Gamma_1 \vdash_{LL} t : !\overline{\tau_1} \qquad \Delta, x : \overline{\tau_1};\Gamma_2 \vdash_{LL} u : \overline{\tau_2}}{\Delta;\Gamma_1,\Gamma_2 \vdash_{LL} \mathsf{let}\ !x = t\ \mathsf{in}\ u : \overline{\tau_2}}$$

Figure 4.2: New typing rules

tion $\mathcal{M}f(\mu) = \int f d\mu$. The proof of linearity is standard but order-continuity requires a few words. Let $\{\mu_\alpha\} \downarrow 0$ be a descending arrow, $\mathcal{M}f(\mu_\alpha) = \int f d\mu_\alpha \leq \int 1 d\mu_\alpha = \mu_\alpha(A)$ which, as μ_α goes to zero, so does $\mu_\alpha(A)$, making $\tilde{f}$ order-continuous. The functorial laws also follows from standard proofs from the literature.

To show that $\mathcal{M}$ is lax monoidal we need to define a natural transformations $\mu_{X,Y} : \mathcal{M}(X) \otimes \mathcal{M}(Y) \to \mathcal{M}(X \times Y)$ which is easily defined by the universal property of the tensor product and a morphism $\varepsilon : \mathbb{R} \multimap \mathcal{M}(1)$ which maps a real number r to the measure $r\delta_{\{*\}}$, where $*$ is the only member of the singleton set 1. Showing that the necessary diagrams commute follows from the universal property of the tensor product. □

We can now define the denotational semantics using the categories **Kern** and $\mathbf{PBanLat_1}$ using the standard construction. We use $[\![\cdot]\!]_{MK}$ to denote well-typed programs in the first language and $[\![\cdot]\!]_{LL}$ to denote well-typed programs in the linear language; the denotations for the new constructs are depicted in Figure 4.3.

$$[\![\mathsf{fix}\, t]\!]_{LL}\,(\delta,\gamma) = \mathsf{Fix}\,([\![t]\!]_{LL}\,(\delta,\gamma)) \qquad\qquad [\![!t]\!]_{LL}\,(\delta) = (!\,[\![t]\!]_{LL})(\rho(\delta))$$

$$[\![\mathsf{let}\ !x = t\ \mathsf{in}\ u]\!]\,(\delta,\gamma_1,\gamma_2) = [\![u]\!]\,(\delta,[\![t]\!]\,(\delta,\gamma_1),\gamma_2)$$

Figure 4.3: Denotational semantics

4.8 Related work

There have been a number of semantics of linear logic based on vector space-like objects. Two important families of such semantics are the ones based on probabilistic coherence spaces and the ones based on Banach spaces. As we will explain below, we see our model as a nice synthesis of these two approaches.

Positive Cone Semantics of Linear Logic

[28] shows that probabilistic coherence spaces are a model of classical linear logic and can interpret a call-by-name probabilistic functional programming language. [89] extends their call-by-name λ-calculus with a call-by-value let construct which can be easily interpreted in their original model.

To overcome the limitation that $\mathbf{PCoh}$ cannot represent continuous distributions, Ehrhard et al. define a cartesian closed category $\mathbf{CStab_m}$ [37], which uses normed $\mathbb{R}^+$-semimodule—which are in correspondence with positive cones of partially ordered vector spaces—to interpret a probabilistic variant of PCF with continuous distributions. In a follow-up paper, Ehrhard [36] has defined a category $\mathbf{CLin_m}$ of sequentially complete positive cones with measurability paths and linear Scott continuous maps in which $\mathbf{PCoh}$ embeds fully and faithfully.

A similar approach was taken in [86], where it is defined a category $\mathbf{CCones}$ of so-called coherent cones and linear contractive functions and shown that it is a model of classical linear logic. These cones come equipped with a different

notion of completeness which is stronger than sequentially complete but weaker than ours.

From a mathematical point of view, the objects of both $\mathbf{CCones}$ and $\mathbf{CStab_m}$ are not as well-understood as Banach lattices, making them not ideal semantic frameworks to reason about probabilistic programs. Besides, our model provides a clear mathematical justification for having Fatou-like properties in the semantics: it is forced upon it by Theorem 4.3.30 instead of being there for denotational reasons, as is the case of $\mathbf{CStab_m}$, or in enabling the exponential construction, as is the case of $\mathbf{CCones}$, showing a kind of canonicity of our model.

Vector Space Semantics of Linear Logic

Dahlqvist and Kozen [26] have defined a category of partially ordered Banach spaces $\mathbf{RoBan}$, showed that it is a model of intuitionistic linear logic and have used it to interpret a higher-order imperative probabilistic language with while loops and soft-conditioning.

Their model also uses a mathematically well-understood class of vector spaces. That being said, by using a more general class of vector spaces than we do, their model has less structure than ours. A practical consequence of this lack of structure is that in order to guarantee the soundness of their semantics, they define six type grammars that are used for different program constructs. As an example, in order to interpret conditionals and while loops the context may only have Dedekind complete types.

Furthermore, their semantics does not have an equivalent to the weak Fatou property and cannot interpret our fixed point operator. In their language, recursion can only be expressed with while-loops.

Another relevant vector space model is the one based on complex coherent

Banach spaces [43]. These vector spaces are not partially ordered, making them unsuitable for interpreting recursive programs, they have an involutive linear negation and the exponential was defined in a time when the categorical semantics of the modality was not fully developed and, as a consequence, morphisms $!A \multimap B$ do not compose, making it not a model of linear logic as they are currently defined.

Neither $\mathbf{RoBan}$ nor $\mathbf{CStab_m}$ are models of classical linear logic.

4.9 Conclusion and Future Work

In this chapter we have shown that $\mathbf{PBanLat_1}$ is a model of classical linear logic that conservatively extends $\mathbf{PCoh}$ and can give semantics to a recursive probabilistic calculus. Our model differs from existing extensions of $\mathbf{PCoh}$ that only extends its intuitionistic fragment, meaning that they do not have an involutive negation. We believe that our model is a good fit for formal verification purposes because Riesz spaces have decades of research and have been extensively used in the formalization of stochastic processes.

This work creates some exciting directions for future work. Ehrhard has shown that the category of Kothe spaces is a model of differential linear logic [34]. The model looks very similar to $\mathbf{PCoh}$, with the exception that they do not require a norm. It is a natural question to ask if by working with perfect unnormed Riesz spaces instead of perfect Banach lattices one gets a model of differential linear logic, providing a preliminary answer to the challenge of giving semantics to probabilistic differentiable languages.

CHAPTER 5

A TYPE SYSTEM FOR INDEPENDENCE

5.1 Introduction

This chapter is based on joint work with Justin Hsu [31].

A central challenge in the theory of programming languages is to come up with sound and expressive reasoning principles for effectful programs. In contrast with pure programs, where different programs can only affect each other at clearly defined interfaces (e.g., the input or output from a functional call), the interaction between effectful programs can be subtle and difficult to reason about. To simplify formal analysis, it is highly useful to know when different effectful computations are *separate*, i.e., they do not interfere with each other. For instance, in the presence of effects such as memory allocation or probability, it is useful to know when pointers do not refer to the same location, or when random quantities must be independent.

Prior Work: Reasoning About Resource Separation

While separated *effects* have received relatively little attention in the literature, there is a long line of work on reasoning about separation of *resources* [73, 78]. The concept of resource is ubiquitous in Computer Science and usually manifests itself when effectful programs interact with the external world. For example, when programming with memory allocation, the heap is a kind of resource; when programming with probabilistic sampling, randomness can be seen as a resource.

In some cases, it is useful to ensure that computations access resources separately. When programming with pointers, different pointers that *alias* refer

to the same address, making it difficult to reason about updates to the heap; requiring that programs do not alias can make formal verification more modular and compositional. In the example of probabilistic effects, separation of resources corresponds to probabilistic independence, while general joint distributions can share resources. Just like for other notions of separation, independence can simplify reasoning about programs. For instance, if two parts of a program produce independent distributions, their joint distribution will only depend on their individual probabilities—there are no unexpected probabilistic interaction between the two parts. Independence can also be an interesting property to verify; for instance, in cryptographic protocols, basic security properties can be stated in terms of independence [11]. Prior work has developed program logics that can about independence in the context of a first-order, imperative language [11]. Unfortunately, it is unclear how to capture independence in higher-order languages.

Our Work

We aim to develop a higher-order language that can reason about shared and separated effects in a variety of contexts. The closest work in this area is the bunched calculus [70], the Curry-Howard correspondent of the logic of Bunched Implications [72]. While O'Hearn [70] gives a presheaf model for the language and develops a concrete model for reasoning about memory-manipulating programs, other concrete models are harder to come by. Indeed, there are no known models for the bunched calculus that can accommodate probability, or other common monadic effects besides state.

Throughout this work we will use probabilistic effects as our guiding example. We start by using a resource interpretation of probabilistic samples to estab-

lish independence: if two computations use disjoint resources (i.e., probabilistic samples), then they produce independent random quantities. Our perspective yields two linear, higher-order languages that can reason about probabilistic independence. Both languages have a product type constructor $\otimes$ that enforces independence, in the sense that closed programs of type $\mathbb{N} \otimes \mathbb{N}$ should be denoted by independent distributions.

Our first language λ_{INI} is an linear λ-calculus with two product types: the $\otimes$ type constructor enforces that the components of the pair do not share any resources, while the $\times$ type constructor allows the components to share resources. Intuitively, $\otimes$ captures pairs of independent values, while $\times$ captures pairs of general, possibly-dependent values. We give a denotational semantics to λ_{INI} and prove its soundness theorem: the product $\otimes$ ensures probabilistic independence.

While conceptually clean, λ_{INI} has limited expressivity. For instance, extending it with sum types breaks the soundness property. In order to mitigate these issues, we define a richer, two-level language λ^2_{INI}, where the two product types of λ_{INI} are restricted to different layers. Intuitively, one layer allows computations that share randomness, while the other layer prevents computations from sharing randomness. To enable the layers to interact, the independent language has a modality that allows to soundly import programs written in the shared language. This design is inspired by recent work by Azevedo de Amorim [7], who proposed a two-level language to combine the sampling and linear operator semantics of probabilistic programming languages. We show that λ^2_{INI} supports two different kinds of sum types: a "shared" sum in the sharing layer, and a "separated" sum in the independent layer. We give a denotational semantics for the λ^2_{INI}, prove soundness, and give translations of two fragments of λ_{INI}

into λ^2_{INI}.

Categorical Semantics and Concrete Models

In order to show the generality of λ^2_{INI} and how it connects to other classes of effects, we propose a categorical semantics for λ^2_{INI} and prove a general soundness theorem of our type system. Then, we present concrete models of our language inspired by a variety of existing effectful programming languages.

- **Linear logic.** Models of linear logic have been used to give semantics to probabilistic languages [28, 37, 6]. We show that pairing these models with the category of Markov kernels yields models for λ^2_{INI}. Our soundness theorem guarantees probabilistic independence; as far as we know, our method is the first to ensure independence in these models.
- **Distributed programming.** Next, we develop a relational model of λ^2_{INI} for distributed programming. In this model, programs describe the implementation and communication patterns of multiple agents. Our soundness theorem shows that global programs of type $\tau_1 \otimes \tau_2$ can be compiled into two local programs that execute independently. This property is reminiscent of projection properties in choreographic languages [66].
- **Name generation.** Programming languages with name generation include a primitive that generates a fresh identifier. In some contexts, it is important to control when and how many times a name is generated; for instance, reusing a *nonce* value ("number once") in cryptographic applications may make a protocol vulnerable to replay attacks. We define a model of λ^2_{INI} based on name generation. Our soundness theorem states that the connective $\otimes$ enforces disjointness of the names used in each component.

- **Commutative effects.** We generalize the name generation and finite distribution models by noting that they are both example of monadic semantics of commutative effects. Under mild assumptions, every commutative monad gives rise to a model of λ^2_{INI}.
- **Bunched and separation logics.** A long line of work uses *bunched logics* to reason about separation of resources [72, 73]. We show that all models of affine bunched logics are also models of λ^2_{INI}, but not vice-versa. To illustrate, we revisit O'Hearn's SCI+, a bunched type system for programming with memory allocation [70]. We define a model of λ^2_{INI} based on SCI+, and give a sound translation of λ^2_{INI} into SCI+.

The diversity of models suggests that λ^2_{INI} is a suitable framework to reason about separation and sharing in effectful higher-order programs.

Outline.

- First, we define a linear, higher-order probabilistic λ-calculus called λ_{INI}, with types that can capture probabilistic independence and dependence. We give a denotational semantics of our language and prove that $\otimes$ captures probabilistic independence (§5.2).
- Next, we define a two-level, higher-order probabilistic λ-calculus called λ^2_{INI}. This language combines an independent fragment and a sharing fragment with two distinct sum types: an independent sum, and a sharing sum. We give a probabilistic semantics and prove that $\otimes$ captures probabilistic independence; we also embed two fragments of λ_{INI} into λ^2_{INI} (§5.3).
- Generalizing, we propose a categorical semantics for λ^2_{INI}. Our semantics is a weaker version of Benton's linear/non-linear (LNL) model for linear logic [15] and of the calculus proposed by Azevedo de Amorim [7] (§5.4.1).

- We present a range of models for λ^2_{INI}, described above. The soundness property of our type system ensures natural notions of independence in each of these models (§5.4.2).
- Finally, we prove a general soundness theorem: every program of type $\tau_1 \otimes \tau_2$ can be factored as two programs t_1 and t_2 of types τ_1 and τ_2, respectively. Our proof relies on a categorical gluing argument (§5.5).

We survey related work in (§5.7), and conclude in (§5.8).

5.2 A Linear Language for Independence

To motivate our language for separated and shared effects, we will focus on one effect: probabilistic sampling. We will build up two higher-order languages where types can ensure probabilistic independence, the natural notion of separation for probabilistic effects.

5.2.1 Independence Through Linearity

In many probabilistic programs, independent quantities are initially generated through sampling instructions. Then, a simple way to reason about independence of a pair of random expressions is to analyze which sources of randomness each component uses: if the two expressions use distinct sources of randomness, then they are independent; otherwise, they are possibly-dependent.

For instance, consider a simply typed first-order call-by-value language with a primitive $\vdash \mathsf{coin} : \mathcal{B}$ that flips a fair coin. The program

$$\mathsf{let}\ x = \mathsf{coin}\ \mathsf{in}\ \mathsf{let}\ y = \mathsf{coin}\ \mathsf{in}\ (x, y)$$

flips two fair coins and pairs the results. This program will produce a probabilistically independent distribution, since x and y are distinct sources of randomness. On the other hand, the program

$$\mathsf{let}\ x = \mathsf{coin}\ \mathsf{in}\ (x, x)$$

does not produce an independent distribution: the two components are always equal, and hence perfectly correlated. These principles are a natural fit for substructural type systems, which control when variables can be shared. To investigate this idea, we develop a language λ_{INI} with an affine type system that can reason about probabilistic independence.

5.2.2 Introducing the Language λ_{INI}

Syntax. Figure 5.13 presents the syntax of types and terms. Along with base types ($\mathcal{B}$), there are two product types: we view $\times$ as the shared, or possibly-dependent product, while $\otimes$ is the independent product. The language is higher-order, with a linear arrow type. The corresponding term syntax is fairly standard. We have variables, numeric constants, and primitive distributions (coin). The two kinds of products can be created from two kinds of pairs, and eliminated using projection and let-binding, respectively. Finally, we have the usual λ-abstraction and application. Our examples will use the standard syntactic sugar $\mathsf{let}\ x = t\ \mathsf{in}\ u \triangleq (\lambda x.\, u)\ t$.

Type system. Figure 5.2 shows the typing rules for λ_{INI}; the rules are standard from linear logic. The variable rule VAR is *affine*: variables in the context may not be used, and variables cannot be freely duplicated. For the sharing product $\times$, the introduction rule $\times$ INTRO shares the context across the premises: both

$$
\begin{array}{lllll}
\text{Variables} & x, y, z & & & \\
\text{Types} & \tau & ::= & \mathcal{B} \mid \tau \times \tau \mid \tau \otimes \tau \mid \tau \multimap \tau & \\
\text{Expressions} & t, u & ::= & x \mid b \in \mathcal{B} \mid \mathsf{coin} \mid (t, u) \mid \pi_i\, t & \\
 & & \mid & t \otimes u \mid \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u \mid \lambda x.\, t \mid t\ u & \\
\text{Contexts} & \Gamma & ::= & x_1 : \tau_1, \ldots, x_n : \tau_n &
\end{array}
$$

Figure 5.1: Types and Terms: λ_{INI}

components can use the same variables. Either component can be projected out of these pairs ($\times$ ELIM$_i$). For the independent product $\otimes$, in contrast, the introduction rule $\otimes$ INTRO requires both premises to use *disjoint* contexts. Thus, the components cannot share variables. Tensor pairs are eliminated by a let-pair construct that consumes both components ($\otimes$ ELIM). In substructural type systems, $\times$ is called an *additive* product, while $\otimes$ is called a *multiplicative* product. The abstraction and application rules are standard.

An additive arrow? Note that the application rule is multiplicative: the function cannot share variables with its argument. A natural question is whether the arrow should be additive: can we share variables between the function and its argument? Substructural type systems like bunched logic [72] include both a multiplicative and an additive arrow.

While we haven't defined the semantics of our language yet, we sketch an example showing that an additive arrow would make it difficult for $\otimes$ to capture probabilistic independence. If we allowed variables to be shared between the function and its argument, we would be able to type-check:

$$\cdot \vdash \mathsf{let}\ x = \mathsf{coin}\ \mathsf{in}\ (\lambda y.\, x \otimes y)\ x : \mathcal{B} \otimes \mathcal{B}$$

Under our semantics, which we will see next, this program is equivalent to $\mathsf{let}\ x = \mathsf{coin}\ \mathsf{in}\ x \otimes x$, which produces a pair of correlated values. Thus, we take a multiplicative arrow for our language.

$$\textsc{Const}\ \frac{}{\cdot \vdash b : \mathcal{B}} \qquad \textsc{Coin}\ \frac{}{\cdot \vdash \mathsf{coin} : \mathcal{B}} \qquad \textsc{Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \times\ \textsc{Intro}\ \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2}$$

$$\times\ \textsc{Elim}_i\ \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, t : \tau_i}$$

$$\otimes\ \textsc{Intro}\ \frac{\Gamma_1 \vdash t_1 : \tau \qquad \Gamma_2 \vdash t_2 : \tau_2}{\Gamma_1, \Gamma_2 \vdash t_1 \otimes t_2 : \tau_1 \otimes \tau_2} \qquad \otimes\ \textsc{Elim}\ \frac{\Gamma_1 \vdash t : \tau_1 \otimes \tau_2 \qquad \Gamma_2, x : \tau_1, y : \tau_2 \vdash u : \tau}{\Gamma_1, \Gamma_2 \vdash \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u : \tau}$$

$$\textsc{Abstraction}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \multimap \tau_2} \qquad \textsc{Application}\ \frac{\Gamma_1 \vdash t : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash u : \tau_1}{\Gamma_1, \Gamma_2 \vdash t\ u : \tau_2}$$

Figure 5.2: Typing Rules: λ_{INI}

5.2.3 Denotational Semantics

We can give a semantics to this language using the category Set and the finite probability monad D. From top to bottom, Figure 5.3 defines the semantics of types, contexts, and typing derivations producing well-typed terms. For types, we interpret both product types as products of sets. Arrow types are interpreted as the set of Kleisli arrows, i.e., maps $[\![\tau_1]\!] \to D\,[\![\tau_2]\!]$. Contexts are interpreted as products of sets.

We interpret well-typed terms as Kleisli arrows. We briefly walk through the term semantics, which is essentially the same as the Kleisli semantics proposed by Moggi [65]. Variables are interpreted using the unit of the monad, which maps a value v to the point mass distribution δ_v. Coins are interpreted as the fair convex combination of two point mass distributions over tt and ff.

The rest of the constructs involve sampling, which is semantically modeled by composition of Kleisli morphisms. We use monadic arrow notation to de-

$$
\begin{aligned}
[\![\mathcal{B}]\!] &= \mathcal{B} \\
[\![\tau \times \tau]\!] &= [\![\tau]\!] \times [\![\tau]\!] \\
[\![\tau \otimes \tau]\!] &= [\![\tau]\!] \times [\![\tau]\!] \\
[\![\tau_1 \multimap \tau_2]\!] &= [\![\tau_1]\!] \to D\,[\![\tau_2]\!]
\end{aligned}
$$

$$
[\![x_1 : \tau_1, \ldots, x_n : \tau_n]\!] = [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!]
$$

$$
[\![\Gamma \vdash t : \tau]\!] : [\![\Gamma]\!] \to D\,[\![\tau]\!]
$$

$$
\begin{aligned}
[\![x]\!]\,(\gamma, v_x) &= \mathsf{return}\ v_x \\
[\![b]\!]\,(*) &= \mathsf{return}\ b \\
[\![\mathsf{coin}]\!]\,(*) &= \frac{1}{2}(\delta_{\mathsf{tt}} + \delta_{\mathsf{ff}}) \\
[\![(t_1, t_2)]\!]\,(\gamma) &= x \leftarrow [\![t_1]\!]\,(\gamma); y \leftarrow [\![t_2]\!]\,(\gamma); \mathsf{return}\ (x, y) \\
[\![\pi_i\ t]\!]\,(\gamma) &= (x, y) \leftarrow [\![t]\!]\,(\gamma); \mathsf{return}\ x \\
[\![t_1 \otimes t_2]\!]\,(\gamma_1, \gamma_2) &= x \leftarrow [\![t_1]\!]\,(\gamma_1); y \leftarrow [\![t_2]\!]\,(\gamma_2); \mathsf{return}\ (x, y) \\
[\![\mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u]\!]\,(\gamma_1, \gamma_2) &= (x, y) \leftarrow [\![t]\!]\,(\gamma_1); [\![u]\!]\,(\gamma_2, x, y) \\
[\![\lambda x.\ t]\!]\,(\gamma) &= \mathsf{return}\ (\lambda x.\ [\![t]\!]\,(\gamma)) \\
[\![t\ u]\!]\,(\gamma_1, \gamma_2) &= f \leftarrow [\![t]\!]\,(\gamma_1); x \leftarrow [\![u]\!]\,(\gamma_2); f(x)
\end{aligned}
$$

Figure 5.3: Denotational Semantics: λ_{INI}

note Kleisli composition, i.e., $x \leftarrow f; g \triangleq g^* \circ f$. The two pair constructors have the same semantics: we sample from each component, and then pair the results. The projections for $\times$ computes the marginal of a joint distribution, while let-binding for $\otimes$ samples from the pair t and then uses the sample in the body u. Lambda abstractions are interpreted as point mass distributions, while applications are interpreted as sampling the function, sampling the argument, and then applying the first sample to the second one.

Example 5.2.1 (Correlated pairs)**.** It may seem as if there is no way of creating non-independent pairs, since the semantics for both kinds of pairs samples each component independently. However, consider the program $\mathsf{let}\ x = \mathsf{coin}\ \mathsf{in}\ (x, x)$.

By unfolding the definitions, its semantics is

$$x \leftarrow \frac{1}{2}(\delta_0 + \delta_1); y \leftarrow \delta_x; z \leftarrow \delta_x; \delta_{(y,z)} = x \leftarrow \frac{1}{2}(\delta_0 + \delta_1); \delta_{(x,x)} = \frac{1}{2}(\delta_{(0,0)} + \delta_{(1,1)}).$$

The resulting samples are perfectly correlated, not independent.

Example 5.2.2 (Independent pairs are correlated pairs)**.** Independent distributions are also possibly-dependent distributions. In λ_{INI}, this fact is reflected by the following program:

$$\cdot \vdash \lambda z.\, \mathsf{let}\ x \otimes y = z\ \mathsf{in}\ (x, y) : \tau_1 \otimes \tau_2 \multimap \tau_1 \times \tau_2.$$

If we unfold the semantics of this program, we see that this program does not modify the input.

5.2.4 Soundness

The type system of λ_{INI} guarantees that $\otimes$ enforces probabilistic independence. Concretely, if $\cdot \vdash t : \tau_1 \otimes \tau_2$ is well-typed, then $[\![t]\!]\,(*)$ is an independent probability distribution over $[\![\tau_1]\!] \times [\![\tau_2]\!]$. We show this soundness theorem by constructing a logical relation $\mathcal{R}_\tau \subseteq D([\![\tau]\!])$, defined as:

$$\begin{aligned}
\mathcal{R}_{\mathcal{B}} &= D(\mathcal{B}) \\
\mathcal{R}_{\tau_1 \otimes \tau_2} &= \{\mu_1 \otimes \mu_2 \in D([\![\tau_1]\!] \times [\![\tau_2]\!]) \mid \mu_i \in \mathcal{R}_{\tau_i}\} \\
\mathcal{R}_{\tau_1 \times \tau_2} &= \{\mu \in D([\![\tau_1]\!] \times [\![\tau_2]\!]) \mid \pi_i(\mu) \in \mathcal{R}_{\tau_i} \text{ for } i \in \{1,2\}\} \\
\mathcal{R}_{\tau_1 \multimap \tau_2} &= \{\mu \in D([\![\tau_1]\!] \to D([\![\tau_2]\!])) \mid \forall \mu' \in \mathcal{R}_{\tau_1}, x \leftarrow \mu'; f \leftarrow \mu; f(x) \in R_{\tau_2}\}.
\end{aligned}$$

Theorem 5.2.3. *If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$ and $\mu_i \in \mathcal{R}_{\tau_i}$ then*

$$(x_1 \leftarrow \mu_1; \cdots; x_n \leftarrow \mu_n; [\![t]\!]\,(x_1, \ldots, x_n)) \in \mathcal{R}_\tau.$$

Proof. Let the distribution above be ν. We write $\overline{x_i}$ as shorthand for $x_1, \ldots, x_n$, and $\overline{x_i \leftarrow \mu_i}$ as shorthand for $x_1 \leftarrow \mu_1; \cdots; x_n \leftarrow \mu_n$. We prove $\nu \in \mathcal{R}_\tau$ by induction on the derivation of $\Gamma \vdash t : \tau$.

CONST/COIN/VAR. Trivial. For instance, VAR: $\nu = \overline{x_i \leftarrow \mu_i}; \mathsf{return}\ x_i = \mu_i$ is in $\mathcal{R}_{\tau_i}$ by assumption.

$\times$ **INTRO.** We have $\nu = \overline{x_i \leftarrow \mu_i}; x \leftarrow [\![t_1]\!](\overline{x_i}); y \leftarrow [\![t_2]\!](\overline{x_i}); \mathsf{return}\ (x, y)$. It is straightforward to show that the first marginal of ν is $\overline{x_i \leftarrow \mu_i}; x \leftarrow [\![t_1]\!](\overline{x_i}); \mathsf{return}\ x$ which, by the induction hypothesis, in an element of $\mathcal{R}_{\tau_1}$; similarly, the second marginal of ν is an element of $\mathcal{R}_{\tau_2}$.

$\times$ **ELIM.** We have $\nu = \overline{x_i \leftarrow \mu_i}; (x, y) \leftarrow [\![t]\!](\overline{x_i}); \mathsf{return}\ x$. By the induction hypothesis, $[\![t]\!](x_i) \in \mathcal{R}_{\tau_1 \times \tau_2}$ and, by assumption, its marginals are elements of $\mathcal{R}_{\tau_1}$ and $\mathcal{R}_{\tau_2}$.

$\otimes$ **INTRO.** Let $\overline{\mu_i}$ be the sequence of distributions corresponding to Γ_1, and let $\overline{\eta_i}$ be the sequence of distributions corresponding to Γ_2. Since D is a commutative monad [19], we may apply associativity and commutativity to show:

$$\begin{aligned}
\nu &= x_i \leftarrow \overline{\mu_i}; y_i \leftarrow \overline{\eta_i}; x \leftarrow [\![t_1]\!](\overline{x_i}); y \leftarrow [\![t_2]\!](\overline{y_i}); \mathsf{return}\ (x, y) \\
&= \overline{x_i \leftarrow \mu_i}; x \leftarrow [\![t_1]\!](\overline{x_i}); \overline{y_i \leftarrow \eta_i}; y \leftarrow [\![t_2]\!](\overline{y_i}); \mathsf{return}\ (x, y) \\
&= (\overline{x_i \leftarrow \mu_i}; x \leftarrow [\![t_1]\!](\overline{x_i}); \mathsf{return}\ x) \otimes (\overline{y_i \leftarrow \eta_i}; y \leftarrow [\![t_2]\!](\overline{y_i}); \mathsf{return}\ y) = \nu_1 \otimes \nu_2.
\end{aligned}$$

Furthermore, by induction hypothesis, $\nu_i \in \mathcal{R}_{\tau_i}$ so $\nu = \nu_1 \otimes \nu_2 \in \mathcal{R}_{\tau_1 \otimes \tau_2}$ as desired.

$\otimes$ **ELIM.** Let $\overline{\mu_i}$ be the sequence of distributions corresponding to Γ_1, and let $\overline{\eta_i}$

be the sequence of distributions corresponding to Γ_2. We have:

$$\begin{aligned}
\nu &= \overline{x_i \leftarrow \mu_i}; \overline{y_i \leftarrow \eta_i}; (x, y) \leftarrow [\![t]\!]\,(\overline{x_i}); \\
&= \overline{x_i \leftarrow \mu_i}; (x, y) \leftarrow [\![t]\!]\,(\overline{x_i}); \overline{y_i \leftarrow \eta_i}; [\![u]\!]\,(\overline{y_i}, x, y) \\
&= (x, y) \leftarrow \nu_1 \otimes \nu_2; \overline{y_i \leftarrow \eta_i}; [\![u]\!]\,(\overline{y_i}, x, y) \\
&= \overline{y_i \leftarrow \eta_i}; x \leftarrow \nu_1; y \leftarrow \nu_2; [\![u]\!]\,(\overline{y_i}, x, y)
\end{aligned}$$

where the third equality is by the induction hypothesis from the first premise. By the induction hypothesis from the second premise, the final distribution is in $\mathcal{R}_\tau$, as desired.

ABSTRACTION. By unfolding the definitions, we need to show

$$x \leftarrow \mu; f \leftarrow (x_i \leftarrow \mu_i; \delta_{\lambda x.\ [\![t]\!](x_i)}); f(x) \in \mathcal{R}_{\tau_2},$$

for some $\mu \in \mathcal{R}_{\tau_1}$. This distribution is equal to $x_i \leftarrow \mu_i; x \leftarrow \mu; f \leftarrow \delta_{\lambda x.\ [\![t]\!](x_i)}; f(x)$, by associativity and commutativity. By the induction hypothesis and the fact that δ is the unit of the monad, we can conclude this case.

APPLICATION. This case follows directly from the induction hypotheses. □

Our soundness property for λ_{INI} follows immediately.

Corollary 5.2.4. *If $\cdot \vdash t : \tau_1 \otimes \tau_2$ then $[\![t]\!]\,(*)$ is an independent probability distribution over $[\![\tau_1]\!] \times [\![\tau_2]\!]$.*

5.3 A Two-Level Language for Independence

The affine type system of λ_{INI} can distinguish between independent and possibly dependent random quantities, but the language is not as expressive as we

would like. We first discuss these limitations, and then introduce a stratified, two-level language λ^2_{INI} that resolves these problems. Finally, we show how to embed two fragments of λ_{INI} into λ^2_{INI}.

5.3.1 Limitations of λ_{INI}: Sums and Let-Bindings

Adding sum types. Though there are base types like $\mathcal{B}$ in λ_{INI}, there are no conditionals. Extending λ_{INI} with sum types and case analysis immediately leads to problems. Consider the program:

$$\mathsf{if\ coin\ then\ tt} \otimes \mathsf{tt\ else\ ff} \otimes \mathsf{ff}$$

Operationally, this probabilistic program flips a fair coin and a pair with two copies of the result, $\mathsf{tt} \otimes \mathsf{tt}$ or $\mathsf{ff} \otimes \mathsf{ff}$. Since tt and ff are constants they do not share any variables, so both branches can be given type $\mathcal{B} \otimes \mathcal{B}$ and a standard case analysis rule would assign the whole program $\mathcal{B} \otimes \mathcal{B}$. However, this extension would break soundness (Theorem 5.2.3): the pair is not probabilistically independent because its components are always equal to each other.

This example illustrates that we should not allow case analysis to produce programs of type $\tau_1 \otimes \tau_2$. However, note that it is safe to allow case analysis to produce programs of type $\tau_1 \times \tau_2$ since this product does not assert independence. Thus, incorporating sum types into λ_{INI} while preserving soundness seems to require ad hoc restrictions on the elimination rule.

Reusing variables. Another restriction is that function application is multiplicative. The limitation can be seen when using let-bindings, which are syntactic sugar for application. In $\mathsf{let}\ x = t\ \mathsf{in}\ u$, the terms t and u *cannot* share any

variables. For instance, λ_{INI} does not allow the following program:

$$\begin{aligned}&\mathsf{let}\ x_1 = \mathsf{coin}\ \mathsf{in}\ \mathsf{let}\ x_2 = \mathsf{coin}\ \mathsf{in}\\ &\qquad \mathsf{let}\ y = f(x_1, x_2)\ \mathsf{in}\ \mathsf{let}\ z = g(x_1, x_2)\ \mathsf{in}\ (y, z)\end{aligned}$$

However, there are useful sampling algorithms (e.g., the Box-Muller transform [20]) that follow this shape. In order to write a well-typed version of this program in λ_{INI}, we could inline the definitions of y and z: the pair constructor $(-,-)$ is additive, so the two components can both use x_1 and x_2. However, it is awkward to require this change.

Similarly, given a term of type $\tau_1 \times \tau_2$, we can't directly project out both components at the same time. For instance, the program

$$\mathsf{let}\ x = \pi_1\, z\ \mathsf{in}\ \mathsf{let}\ y = \pi_2\, z\ \mathsf{in}\ f(x, y)$$

is not well-typed, since the outer let-binding shares the variable z with its body. These problems would be solved if function application in λ_{INI} was additive; however, as we saw in Section 5.2, allowing a function and an argument to share variables can also break soundness of λ_{INI}.

5.3.2 The Language λ^2_{INI}: Syntax, Typing Rules and Semantics

To address these limitations, we introduce a stratified language. We are guided by a simple observation about products, sums, and distributions, which might be of more general interest. In λ_{INI}, the product types correspond to two distinct ways of composing distributions with products: the sharing product $\tau_1 \times \tau_2$ corresponds to *distributions of products*, $M(\tau_1 \times \tau_2)$, while the separating product $\tau_1 \otimes \tau_2$ corresponds to *products of distributions*, $M\tau_1 \times M\tau_2$.

Similarly, there are two ways of combining distributions and sums: *distributions of sums*, $M(\tau_1 + \tau_2)$, and *sums of distributions*, $M\tau_1 + M\tau_2$. We think of the

first combination as a *sharing sum*, since the distribution can place mass on both components of the sum. In contrast, the second combination is a *separating sum*, since the distribution either places all mass on τ_1 or all mass on τ_2.

Finally, there are interesting interactions between sharing and separating, sums and products. For instance, the problematic sum example we saw above performs case analysis on coin—a sharing sum, because it has some probability of returning true and some probability of returning false—but produces a separating product $\mathcal{B} \otimes \mathcal{B}$. If we instead perform case analysis on a *separating* sum, then the program either always takes the first branch or always takes the second branch, and now there is no problem with producing a separating product.

These observations lead us to design a two-level language, where one layer includes the sharing connectives and the other layer includes the separating connectives. We call this language λ^2_{INI}, where INI stands for *independent/non-independent*.

Syntax. The program and type syntax of λ^2_{INI}, summarized in Figure 5.4, is stratified into two layers: a non-independent (NI) layer, and an independent (I) layer. We will color-code them: the NI -language will be orange , while the I -language will be purple .

The NI layer has base, product ($\times$), and sum types ($+$). The language is mostly standard: we have variables, constants, basic distributions (coin), and a set $\mathcal{O}(\tau_1, \tau_2)$ of primitive operations from τ_1 to τ_2, along with the usual pairing and projection constructs for products, and injection and case analysis constructs for sums. The NI layer does not have arrows, but it does allow let-binding.

The I -layer is quite similar to λ_{INI}: it has its own product ($\otimes$) and sum ($\oplus$) types, and a linear arrow type ($\multimap$). The type $\mathcal{M}(\tau)$ brings a type from the NI

-layer into the I -layer. The language is also fairly standard, with constructs for introducing and eliminating products and sums, and functions and applications. The last construct ($\mathsf{sample}\ \overline{t}\ \mathsf{as}\ \overline{x}\ \mathsf{in}\ M$) is from [7]: it allows the two layers to interact. Here, $\overline{t}$ and $\overline{x}$ are two (possibly empty) lists of the same length.

Intuitively, the NI -language allows sharing while the I -language disallows sharing. Each language has its own sum type, a sharing and separated sum, respectively, each of which interacts nicely with its own product type. The $\mathcal{M}$ modality can be thought of as an abstraction barrier between both languages that enables the manipulation of shared programs in a separating program while not allowing its sharing to be inspected, except when producing another boxed term.

Variables	x, y, z		
NI -types	τ	$::=$	$\mathcal{B} \mid \tau \times \tau \mid \tau + \tau$
I -types	$\underline{\tau}$	$::=$	$\underline{\tau} \otimes \underline{\tau} \mid \underline{\tau} \oplus \underline{\tau} \mid \underline{\tau} \multimap \underline{\tau} \mid \mathcal{M}(\tau)$
NI -expressions	M, N	$::=$	$x \mid b \in \mathcal{B} \mid \mathsf{coin} \mid f \in \mathcal{O}(\tau_1, \tau_2) \mid (M, N) \mid \pi_i\, M$
		$\mid$	$\mathsf{in_i\ t} \mid \mathsf{case}\, t\, \mathsf{of}\, (\lvert \mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2)$
		$\mid$	$\mathsf{let}\ x = M\ \mathsf{in}\ N$
I -expressions	t, u	$::=$	$x \mid t \otimes u \mid \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u \mid \mathsf{op} \mid \mathsf{in_i\ t}$
		$\mid$	$\mathsf{case}\, t\, \mathsf{of}\, (\lvert \mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) \mid \lambda x.\, t \mid t\ u$
		$\mid$	$\mathsf{sample}\ \overline{t}\ \mathsf{as}\ \overline{x}\ \mathsf{in}\ M$
NI -contexts	Γ	$::=$	$x_1 : \tau_1, \ldots, x_n : \tau_n$
I -contexts	Γ	$::=$	$x_1 : \underline{\tau}_1, \ldots, x_n : \underline{\tau}_n$

Figure 5.4: Types and Terms: λ^2_{INI}

We also present the equational theory to this language in Figure 5.5. Note that the equational theory is basically the union of the equations for Markov categories (minus the naturality of deletion rule) and symmetric monoidal closed categories. The equations related to the sample operator seem a bit daunting at first but they are merely stating that $\mathcal{M}$ is a functor - the first two equations - and that it satisfies the lax monoidal diagrams - the last three equations.

$$
\begin{array}{rcl}
\mathsf{case}\,(\mathsf{in}_1 M)\,\mathsf{of}\,(|\mathsf{in}_1 x \Rightarrow N_1 \mid \mathsf{in}_2 x \Rightarrow N_2) & \equiv & N_1\{M/x\} \\
\mathsf{case}\,(\mathsf{in}_2 M)\,\mathsf{of}\,(|\mathsf{in}_1 x \Rightarrow N_1 \mid \mathsf{in}_2 x \Rightarrow N_2) & \equiv & N_2\{M/x\} \\
\mathsf{case}\,x\,\mathsf{of}\,(|\mathsf{in}_1 x \Rightarrow M \mid \mathsf{in}_2 x \Rightarrow M) & \equiv & M \\
\\
\mathsf{let}\ x = t\ \mathsf{in}\ x & \equiv & t \\
\mathsf{let}\ x = x\ \mathsf{in}\ t & \equiv & t \\
\mathsf{let}\ y = (\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 & \equiv & \mathsf{let}\ x = M_1\ \mathsf{in}\ (\mathsf{let}\ y = M_2\ \mathsf{in}\ M_3) \\
\\
\pi_1(x_1, x_2) & \equiv & x_1 \\
\pi_2(x_1, x_2) & \equiv & x_2 \\
(\pi_1(x), \pi_2(x)) & \equiv & x \\
\\
\mathsf{let}\ x = M_1\ \mathsf{in}\ (\mathsf{let}\ y = M_2\ \mathsf{in}\ M_3) & \equiv & \mathsf{let}\ y = M_2\ \mathsf{in}\ (\mathsf{let}\ x = M_1\ \mathsf{in}\ M_3) \\
\\
(\lambda x.\, t)\, u & \equiv & t\{u/x\} \\
(\lambda x.\, t\, x) & \equiv & t \\
\mathsf{let}\ x_1 \otimes x_2 = t_1 \otimes t_2\ \mathsf{in}\ u & \equiv & u\{t_1/x_1\}\{t_2/x_2\} \\
\\
\mathsf{case}\,(\mathsf{in}_1 t)\,\mathsf{of}\,(|\mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) & \equiv & u_1\{t/x\} \\
\mathsf{case}\,(\mathsf{in}_2 t)\,\mathsf{of}\,(|\mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) & \equiv & u_2\{t/x\} \\
\\
\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ x & \equiv & t \\
\mathsf{sample}\ (\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ M)\ \mathsf{as}\ y\ \mathsf{in}\ N & \equiv & \mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{let}\ y = M\ \mathsf{in}\ N) \\
\\
\mathsf{sample}\ t, (\mathsf{sample}\ \_\ \mathsf{as}\ \_\ \mathsf{in}\ ())\ \mathsf{as}\ x, y\ \mathsf{in}\ x & \equiv & t \\
\mathsf{sample}\ (\mathsf{sample}\ \_\ \mathsf{as}\ \_\ \mathsf{in}\ ()), t\ \mathsf{as}\ x, y\ \mathsf{in}\ y & \equiv & t \\
\mathsf{sample}\ t_1, (\mathsf{sample}\ t_2, t_3\ \mathsf{as}\ x_2, x_3\ \mathsf{in}\ (x_2, x_3))\ \mathsf{as}\ x_1, y\ \mathsf{in}\ (x_1, \pi_1\, y, \pi_2\, y) & \equiv & \\
\mathsf{sample}\ (\mathsf{sample}\ t_1, t_2\ \mathsf{as}\ x_1, x_2\ \mathsf{in}\ (x_1, x_2)), t_3\ \mathsf{as}\ y, x_3\ \mathsf{in}\ (\pi_1\, y, \pi_2\, y, x_3) & &
\end{array}
$$

Figure 5.5: Equational Theory: λ^2_{INI}

Typing rules. The typing rules of λ^2_{INI} are presented in Figure 5.6. We have two typing judgments for the two layers; we use subscripts on the turnstiles to

indicate the layer. We start with the first group of typing rules, for the sharing (NI) layer. These typing rules are entirely standard for a first-order language with products and sums. Note that all rules allow the context to be shared between different premises. In particular, the let-binding rule is *additive* instead of multiplicative as in λ_{INI}: a let-binding is allowed to share variables with its body.

The second group of typing rules assigns types to the independent (I) layer. These rules are the standard rules for multiplicative additive linear logic (MALL), and are almost identical to the typing rules for λ_{INI}. Just like before, the rules treat variables affinely, and do not allow sharing variables between different premises. The rules for the sum $\tau_1 \oplus \tau_2$ are new. Again, the elimination (CASE) rule does not allow sharing variables between the guard and the body.

The final rule, SAMPLE, gives the interaction rule between the two languages. The first premise is from the sharing (NI) language, where the program M can have free variables $x_1, \ldots, x_n$. The rest of the premises are from the independent (I) language, where linear programs t_i have boxed type $\mathcal{M}\tau_i$. The conclusion of the rule combines programs t_i with M, producing an I -program of boxed type. Intuitively, this rule allows a program in the sharing language to be imported into the linear language. Operationally, sample t as x in M constructs a distribution t using the independent language, samples from it and binds the sample to x in the shared program M, and finally boxes the result into the linear language.

Probabilistic Semantics To keep the presentation concrete, in this section we will work with a concrete semantics motivated by probabilistic independence, where programs are probabilistic programs with discrete sampling. In the next section, we will present the general categorical semantics of λ^2_{INI} and consider

other models.

The probabilistic semantics for λ^2_{INI} is defined in Figure 5.7. For the NI -layer, we use the same semantics of λ_{INI}, i.e., well-typed programs are interpreted as Kleisli arrows for the finite distribution monad D. The Kleisli category $\mathbf{Set}_D$ has sets as objects, so we may simply define the semantics of each type to be a set. It is also known that $\mathbf{Set}_D$ has products and coproducts, which can be used to interpret well-typed programs in NI .

For the I-language, we use the category of algebras for the finite distribution monad D and plain maps, $\widetilde{\mathbf{Set}^D}$. Concretely, its objects are pairs (A, f), where f is an M-algebra, and a morphism $(A, f) \to (B, g)$ is a function $A \to B$. Given two objects (A, f) and (B, g) we can define a product algebra over the set $A \times B$. Furthermore, it is also possible to equip the set-theoretic disjoint union $A + B$ and exponential $A \Rightarrow B$ with algebra structures, making it a model of higher-order programming with case analysis [85]. We only need to explicitly define the algebraic structure when interpreting the type constructor $\mathcal{M}$, which is interpreted as the free D-algebra with the multiplication for the monad as the algebraic structure.

Now that we have defined the probabilistic semantics of the λ^2_{INI}, we can prove its soundness theorem: just like in λ_{INI}, the type constructor $\otimes$ enforces probabilistic independence.

Theorem 5.3.1. *If* $\cdot \vdash_I t : \mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ *then* $[\![t]\!]$ *is an independent distribution.*

Proof. The semantics of $\cdot \vdash_I t : \mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ is a set-theoretic function $[\![t]\!] : 1 \to D\,[\![\tau_1]\!] \times D\,[\![\tau_2]\!]$, which is isomorphic to an independent distribution. □

$$\oplus\ \textsc{Intro}_i\quad \frac{\Gamma \vdash_{NI} M : \tau_i}{\Gamma \vdash_{NI} \mathsf{in}_i\, M : \tau_1 + \tau_2}$$

$$\oplus\ \textsc{Elim}\quad \frac{\Gamma \vdash_{NI} M : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash_{NI} N_1 : \tau \qquad \Gamma, x : \tau_2 \vdash_{NI} N_2 : \tau}{\Gamma \vdash_{NI} \mathsf{case}\, M\, \mathsf{of}\, (\mid \mathsf{in}_1\, x \Rightarrow N_1 \mid \mathsf{in}_2\, y \Rightarrow N_2) : \tau}$$

$$\oplus\ \textsc{Intro}_i\quad \frac{\Gamma \vdash_{I} t : \underline{\tau_i}}{\Gamma \vdash_{I} \mathsf{in}_i\, t : \underline{\tau_1} \oplus \underline{\tau_2}}$$

$$\oplus\ \textsc{Elim}\quad \frac{\Gamma_1 \vdash_{I} t : \underline{\tau_1} \oplus \underline{\tau_2} \qquad \Gamma_2, x : \underline{\tau_1} \vdash_{I} u_1 : \underline{\tau} \qquad \Gamma_2, y : \underline{\tau_2} \vdash_{I} u_2 : \underline{\tau}}{\Gamma_1, \Gamma_2 \vdash_{I} \mathsf{case}\, t\, \mathsf{of}\, (\mid \mathsf{in}_1\, x \Rightarrow u_1 \mid \mathsf{in}_2\, y \Rightarrow u_2) : \underline{\tau}}$$

Figure 5.6: Typing Rules: λ^2_{INI}

5.3.3 Revisiting Sums and Let-Binding

Let us revisit the problematic if-then-else program at the beginning of the section. The type system of λ^2_{INI} makes it impossible to produce an independent pair by pattern matching on values:

$$\mathsf{dist} : \mathcal{M}(1+1) \nvdash_I \mathsf{if}\ \mathsf{dist}\ \mathsf{then}\ (\mathsf{tt} \otimes \mathsf{tt})\ \mathsf{else}\ (\mathsf{ff} \otimes \mathsf{ff}) : \mathcal{MB} \otimes \mathcal{MB}$$

where if-statements are simply elimination of sum types over booleans. However, we can write a well-typed version of this program if we use the sharing product:

$$\mathsf{dist} : \mathcal{M}(1+1) \vdash_I \mathsf{sample}\ \mathsf{dist}\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{if}\ x\ \mathsf{then}\ (\mathsf{tt}, \mathsf{tt})\ \mathsf{else}\ (\mathsf{ff}, \mathsf{ff})) : \mathcal{M}(\mathcal{B} \times \mathcal{B})$$

The design of λ^2_{INI} also removes the limitations on let-bindings we discussed before, since the sharing layer has an *additive* let-binding. In particular, it is also

$$
\begin{array}{rlrl}
(\!|\mathcal{B}|\!) &= \mathcal{B} & [\![\mathcal{M}\tau]\!] &= (D\,[\![\tau]\!], \mu_{[\![\tau]\!]}) \\
(\!|\tau \times \tau|\!) &= (\!|\tau|\!) \times (\!|\tau|\!) & [\![\underline{\tau} \otimes \underline{\tau}]\!] &= [\![\underline{\tau}]\!] \times [\![\underline{\tau}]\!] \\
(\!|\tau + \tau|\!) &= (\!|\tau|\!) + (\!|\tau|\!) & [\![\underline{\tau} \oplus \underline{\tau}]\!] &= [\![\underline{\tau}]\!] + [\![\underline{\tau}]\!] \\
 & & [\![\underline{\tau} \multimap \underline{\tau}]\!] &= [\![\underline{\tau}]\!] \to [\![\underline{\tau}]\!] \\
(\!|x_1 : \tau_1, \ldots, x_n : \tau_n|\!) &= (\!|\tau_1|\!) \times \cdots \times (\!|\tau_n|\!) & [\![x_1 : \underline{\tau}_1, \ldots, x_n : \underline{\tau}_n]\!] &= [\![\underline{\tau}_1]\!] \times \cdots \times [\![\underline{\tau}_n]\!] \\
(\!|\Gamma \vdash M : \tau|\!) &\in \mathbf{Set}_D((\!|\Gamma|\!), (\!|\tau|\!)) & [\![\Gamma \vdash t : \underline{\tau}]\!] &\in \widetilde{\mathbf{Set}^D}([\![\Gamma]\!], [\![\underline{\tau}]\!])
\end{array}
$$

$$
\begin{aligned}
[\![x]\!]\,(\gamma, v_x) &= v_x \\
[\![t \otimes u]\!]\,(\gamma_1, \gamma_2) &= [\![t]\!]\,(\gamma_1) \times [\![u]\!]\,(\gamma_2) \\
[\![\mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u]\!]\,(\gamma_1, \gamma_2) &= [\![u]\!]\,(\gamma_2, [\![t]\!]\,(\gamma_1)) \\
[\![\lambda x.\, t]\!]\,(\gamma)(x) &= [\![t]\!]\,(\gamma)(x) \\
[\![t\ u]\!]\,(\gamma_1, \gamma_2) &= [\![t]\!]\,(\gamma_1, [\![u]\!]\,(\gamma_2) \\
[\![\mathsf{in}_i t]\!]\,(\gamma) &= in_i([\![t]\!]\,(\gamma)) \\
[\![\mathsf{case}\, t\, \mathsf{of}\, (|\mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2)]\!]\,(\gamma_1, \gamma_2) &= \begin{cases} [\![u_1]\!]\,(\gamma_2, v), & [\![t]\!]\,(\gamma_1) = in_1(v) \\ [\![u_2]\!]\,(\gamma_2, v), & [\![t]\!]\,(\gamma_1) = in_2(v) \end{cases} \\
[\![\mathsf{sample}\ t_i\ \mathsf{as}\ x_i\ \mathsf{in}\ N]\!] &= \mu \circ D(\!|N|\!) \circ ([\![t_1]\!] \times \cdots \times [\![t_n]\!])
\end{aligned}
$$

Figure 5.7: Concrete Semantics: λ^2_{INI}

possible to express the problematic let-binding program we saw before:

$$
\begin{aligned}
&\cdot \vdash_I \mathsf{sample\ coin, coin\ as}\ x_1, x_2\ \mathsf{in} \\
&\qquad \mathsf{let}\ y = f(x_1, x_2)\ \mathsf{in\ let}\ z = g(x_1, x_2)\ \mathsf{in}\ M : \mathcal{M}(\tau)
\end{aligned}
$$

We can also project both components out of pairs in the sharing layer:

$$
\cdot \vdash_{NI} \mathsf{let}\ x = \pi_1\, M_1\ \mathsf{in\ let}\ y = \pi_2\, M_2\ \mathsf{in}\ M : \tau
$$

5.3.4 Embedding from λ_{INI} to λ^2_{INI}

Now that we have seen both λ_{INI} and λ^2_{INI}, a natural question is how these languages are related. We first show how to embed the fragment of λ_{INI} with-

out arrow types into λ^2_{INI}. The idea is that the semantics of λ_{INI} is given by a Kleisli category, so there is a translation into the NI -layer of λ^2_{INI}. The types are translated as follows:

$$\mathcal{T}(\mathcal{B}) \triangleq \mathcal{B} \qquad \mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1 \otimes \tau_2) \triangleq \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2)$$

At the term-level, the translation is the identity function.

Theorem 5.3.2. *If* $\Gamma \vdash M : \tau$ *in* λ_{INI} *then* $\mathcal{T}(\Gamma) \vdash_{NI} \mathcal{T}(M) : \mathcal{T}(\tau)$ *in* λ^2_{INI}.

Furthermore, this translation preserves equations between programs and is fully abstract.

Theorem 5.3.3. *Let* $\Gamma \vdash t_1 : \tau$ *and* $\Gamma \vdash t_2 : \tau$ *in* λ_{INI} *then* $[\![t_1]\!] = [\![t_2]\!]$ *if, and only if,* $[\![\mathcal{T}(t_1)]\!] = [\![\mathcal{T}(t_2)]\!]$.

Proof. The proof follows from the fact that the translation is a faithful functor. □

It is also possible to translate the multiplicative $(\otimes, \multimap)$ fragment of λ_{INI} into the I -layer of λ^2_{INI}, by translating the types as follows:

$$\mathcal{T}'(\mathcal{B}) \triangleq \mathcal{M}\mathcal{B} \qquad \mathcal{T}'(\tau_1 \otimes \tau_2) \triangleq \mathcal{T}'(\tau_1) \otimes \mathcal{T}'(\tau_2) \qquad \mathcal{T}'(\tau_1 \multimap \tau_2) \triangleq \mathcal{T}'(\tau_1) \multimap \mathcal{T}'(\tau_2)$$

Once again, the term translation is the identity function.

Theorem 5.3.4. *If* $\Gamma \vdash t : \tau$ *in* λ_{INI} *then* $\mathcal{T}'(\Gamma) \vdash_{I} \mathcal{T}'(t) : \mathcal{T}'(\tau)$ *in* λ^2_{INI}.

Proof. The proof follows by induction on the typing derivation $\Gamma \vdash t : \tau$. □

This translation is functorial and faithful, and therefore is sound and fully abstract with respect with the denotational semantics of λ_{INI} and λ^2_{INI}.

Remark 5.3.5. It is not possible to translate the whole λ_{INI} into λ^2_{INI}. Since only one of the languages of λ^2_{INI} has arrow types and there is no way of moving from I into NI , the translation would need to map λ_{INI} programs into I programs, which can only write probabilistically independent programs, making it impossible to translate the $\times$ type constructor. By adding an additive function type to the NI -layer of λ^2_{INI}, it would be possible to extend the first translation so that it encompasses the whole language; however, many of the concrete models that we will consider in the next section do not support an additive function type in the NI -layer.

5.4 Categorical Semantics and Concrete Models

In this section, we present the general, categorical semantics of λ^2_{INI}, by abstracting the probabilistic semantics we saw in the previous section. Then, we present a variety of concrete models for λ^2_{INI}, based on existing semantics for effectful languages. Our soundness theorem ensures natural notions of separation across these models.

5.4.1 Categorical Semantics of λ^2_{INI}

Suppose we have two effectful languages, $\mathcal{L}_1$ and $\mathcal{L}_2$. The first one has a product type $\times$ which allows for the sharing of resources, while the second one has the disjoint product type $\otimes$. Furthermore, we assume that $\mathcal{L}_2$ has a unary type constructor $\mathcal{M}$ linking both languages. The intuition behind this decision is that an element of type $\mathcal{M}\tau$ is a computation which might share resources. From a language design perspective, the constructor $\mathcal{M}$ serves to encapsulate a possibly dependent computation in an independent environment.

The first question is to understand is how the connectives $\times$ and $\otimes$ should be interpreted categorically. For $\times$, we need a comonoidal structure to duplicate and erase computation. This kind of structure is captured by *CD categories*, which are monoidal categories where every object A comes equipped with a commutative comonoid structure $A \to A \otimes A$ and $A \to I$ making certain diagrams commute [22]. For $\otimes$, we want to restrict copying—the separating layer of our language has an affine type system—so $\otimes$ should be a monoidal product with discard maps.

Finally, to model the type constructor $\mathcal{M}$, the typical categorical idea is that it should be some kind of functor from $\mathcal{L}_1$ to $\mathcal{L}_2$. Let us look at some of the intuitions provided by the type system. The type $\mathcal{M}(\tau_1 \times \tau_2)$ is for computations that may share resources and output both τ_1 and τ_2. Meanwhile, the type $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ is for computations that output τ_1 and τ_2 while using separate resources. This reading suggest that there should not be maps from $\mathcal{M}(\tau_1 \times \tau_2)$ to $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$, since there is no way of separating resources once they have been shared, but there should be maps from $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ to $\mathcal{M}(\tau_1 \times \tau_2)$, since separation is a specific example of sharing.

Categorically, the existence of these maps is captured by applicative functors, also known as lax monoidal functors, which are functors $F : (\mathbf{C}, \otimes_C, I_C) \to (\mathbf{D}, \otimes_D, I_D)$ between monoidal categories, equipped with morphisms $\mu_{A,B} : F(A) \otimes_D F(B) \to F(A \otimes_C B)$ and $\epsilon : I_D \to FI_C$ making certain diagrams commute [19].

Thus, we are led to our categorical model for λ^2_{INI}.

Definition 5.4.1. A λ^2_{INI} model is a triple $(\mathbf{C}, \mathbf{M}, \mathcal{M})$ where $\mathbf{C}$ is a symmetric monoidal closed category with weak coproducts, i.e. sum types satisfying only the existence part of the universal property, not the uniqueness, and with mor-

phisms $del_A : A \to I_C$, natural in A; $\mathbf{M}$ is a distributive CD category with coproducts, i.e., the canonical morphism$(A \otimes_M B) +_M (A \otimes_M C) \to A \otimes_M (B +_M C)$ is an isomorphism; and $\mathcal{M} : \mathbf{M} \to \mathbf{C}$ is lax monoidal.

While we need to assume distributivity in $\mathbf{M}$, distributivity in $\mathbf{C}$ holds automatically.

Lemma 5.4.2. *In every symmetric monoidal closed category with weak coproducts, the following isomorphism holds:* $A \otimes (B + C) \cong (A \otimes B) + (A \otimes C)$.

Proof. By assumption, the functor $A \otimes (-)$ is a left adjoint and, therefore, by Lemma 3.5 in [54], preserves weak coproducts and we can conclude. □

The denotational semantics is given in Figure 5.8 and most of the equational theory is presented in Figure 5.5. The lax monoidal equations for $\mathcal{M}$ are long and not very informative, so we leave them to the Appendix A.1.

Soundness. In categorical models, the soundness theorem of λ^2_{INI} can be stated as follows:

Theorem 5.4.3 (Soundness)**.** *Let* $\cdot \vdash_I t : \tau_1 \otimes \tau_2$ *then* $[\![t]\!] = f \otimes g$*, where* f *and* g *are morphisms* $I \to [\![\tau_1]\!]$ *and* $I \to [\![\tau_2]\!]$*, respectively.*

From a proof-theoretic perspective, the soundness theorem states that for every proof of type $\cdot \vdash \tau_1 \otimes \tau_2$, we can assume that the last rule is the introduction rule for $\otimes$. From a semantic perspective, the soundness theorem means that for every closed term $\cdot \vdash t : \tau_1 \otimes \tau_2$, the semantics $[\![t]\!]$ can be factored as two morphisms f_1 and f_2 such that $[\![t]\!] = f_1 \otimes f_2$.

Establishing soundness requires additional categorical machinery, so we defer the proof to Section 5.5. Here, we will exhibit a range of concrete models for λ^2_{INI}.

$$+\ \textsc{Intro}_i \quad \frac{\Gamma \xrightarrow{M} \tau_1}{\Gamma \xrightarrow{M;in_i} \tau_1 + \tau_2}$$

$$+\ \textsc{Elim} \quad \frac{\Gamma_1 \xrightarrow{N} \tau_1 + \tau_2 \qquad \Gamma_2 \times \tau_1 \xrightarrow{M_1} \tau \qquad \Gamma_2 \times \tau_2 \xrightarrow{M_2} \tau}{\Gamma_1, \Gamma_2 \xrightarrow{N \times id_{\Gamma_2}} (\tau_1 + \tau_2) \times \Gamma_2 \cong (\tau_1 \times \Gamma_2) + (\tau_2 \times \Gamma_2) \xrightarrow{[M_1, M_2]} \tau}$$

$$\oplus\ \textsc{Intro}_i \quad \frac{\Gamma \xrightarrow{t} \underline{\tau_i}}{\Gamma \xrightarrow{t;in_i} \underline{\tau_1} + \underline{\tau_2}} \qquad \oplus\ \textsc{Elim} \quad \frac{\Gamma_1 \xrightarrow{u} \underline{\tau_1} + \underline{\tau_2} \qquad \underline{\tau_1} \otimes \Gamma_2 \xrightarrow{t_1} \underline{\tau} \qquad \underline{\tau_2} \otimes \Gamma_2 \xrightarrow{t_2} \underline{\tau}}{\Gamma_1, \Gamma_2 \xrightarrow{u \otimes id_{\Gamma_2}} (\underline{\tau_1} + \underline{\tau_2}) \otimes \Gamma_2 \cong (\underline{\tau_1} \otimes \Gamma_2) + (\underline{\tau_2} \otimes \Gamma_2) \xrightarrow{[t_1, t_2]} \underline{\tau}}$$

Figure 5.8: Categorical Semantics for sum types: λ^2_{INI}

5.4.2 Concrete models

To warm up, we present some basic probabilistic models λ^2_{INI}. While prior work has also investigated similar models [7], we adapt these models to λ^2_{INI} and explain how our soundness theorem ensures independence.

Discrete Probability

Our first concrete model is a different semantics for discrete probability. For the sharing category, we take the category **CountStoch** with countable sets as objects, and transition matrices as morphisms, i.e. functions $f : A \times B \to [0, 1]$ such that for every $a \in A$, $f(a, -)$ is a (discrete) probability distribution [41].

For the independent category, we take the probabilistic coherence space model of linear logic, a well-studied semantics for discrete probabilistic languages [28].

As we have shown in Chapter 3, the following theorem holds:

Theorem 5.4.4. *There is a lax monoidal functor* $\mathcal{M} : \mathbf{CountStoch} \to \mathbf{PCoh}$.

Summing up, we have a model of λ^2_{INI} based on probabilistic coherence spaces.

Theorem 5.4.5. *The triple* $(\mathbf{PCoh}, \mathbf{CountStoch}, \mathcal{M})$ *is a* λ^2_{INI} *model.*

In **PCoh** it is possible to show that $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2 \subseteq \mathcal{M}(\tau_1 \times \tau_2)$ meaning that well typed programs of type $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ are denoted by joint distributions over $\tau_1 \times \tau_2$. Furthermore, by taking a closer look at Definition 4.2.2 we see that $\mu_A \otimes \mu_B$ corresponds exactly to the product distribution of μ_A and μ_B, so our soundness theorem implies that closed programs of type $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ are denoted by independent probability distributions.

Continuous Probability

Next, we consider models for continuous probability. For the sharing layer, the generalization of **CountStoch** to continuous probabilities is **BorelStoch**, which has standard Borel spaces as objects and Markov kernels as morphisms [41]. For the separating layer, we want a model of linear logic that can interpret continuous randomness. We use the model $\mathbf{PBanLat_1}$ defined in the previous section.

As we have explained in Chapter 4, for every measurable space (X, Σ_X) the space of signed measures over it is a perfect Banach space, meaning that it can, for instance, interpret continuous probability distributions over the real line. Furthermore, the map assigning (X, Σ_X) to its space of signed measures is functorial and lax monoidal, as we have shown in Theorem 4.7.1.

Theorem 5.4.6. *The triple* $(\mathbf{PBanLat_1}, \mathbf{BorelStoch}, \mathcal{M})$ *is a* λ^2_{INI} *model.*

This model can be seen as the continuous generalization of the previous model, since there are full and faithful embeddings $\mathbf{CountStoch} \hookrightarrow \mathbf{BorelStoch}$ and $\mathbf{PCoh} \hookrightarrow \mathbf{PBanLat}_1$ [6]. In this model, our soundness theorem once again ensures probabilistic independence, i.e. programs of type $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ are denoted by independent distributions.

Non-Determinism and Communication

Next, we show that the relational model of linear logic gives rise to a λ^2_{INI} model, with applications with distributed programming.

Semantics Our starting point is the category $\mathbf{Rel}$ of sets and binary relations, one of the most well-known models for linear logic. By pairing this category with the Kleisli category $\mathbf{Set}_{\mathcal{P}}$, for the powerset monad $\mathcal{P}$ we immediately obtain a model for λ^2_{INI}.

Theorem 5.4.7. *The triple* $(\mathbf{Rel}, \mathbf{Set}_{\mathcal{P}}, id)$ *is a* λ^2_{INI} *model.*

Proof. Binary relations over sets A and B are represented either as subsets $R \subseteq A \times B$ or, equivalently, as functions $A \to \mathcal{P}(B)$. From this observation it is possible to show that the identity functor is an isomorphism and it easily follows from this that id is lax monoidal. Since $\mathbf{Rel}$ is a model of linear logic, it has coproducts and, by isomorphism, so does $\mathbf{Set}_{\mathcal{P}}$. The natural transformation del in $\mathbf{Rel}$ is the delete operation from $\mathbf{Set}_{\mathcal{P}}$. □

Application to Distributed Programming While this model arises from linear logic, we show that it leads to a suitable language for distributed programming. We assume a two-tier approach to programming with communication: the NI language is used for writing local programs, while the I language is

used to orchestrate the communication between local code. Programs of type $\mathcal{M}\underline{\tau}$ correspond to local computations that can be manipulated by the communication language. Programs in the I language are interpreted as maps of the form $A \to \mathcal{P}(B)$; we view these maps as allowing *non-deterministic* or *lossy* communication.

To align the syntax with this interpretation, we tweak the syntax sample t_i as x_i in M to send t_i as x_i in M which sends the values computed by the local programs t_i, binds them to x_i and continues as the local program M. To see how how distributed programs can be written in this language, we consider a simple distributed voting protocol between two parties. We suppose that there is a leader that receives two messages containing the votes and if they are the same, the election is decided and the leader announces the winner. If the votes disagree, the leader outputs a tagged unit value saying that there has been a draw. In λ^2_{INI}, the leader can be implemented as:

$$\mathsf{leader} : \mathcal{M}\mathbb{N} \otimes \mathcal{M}\mathbb{N} \multimap \mathcal{M}(\mathbb{N} \oplus 1)$$

$$\mathsf{leader} = \lambda\, x_1\, x_2.\, \mathsf{send}\, x_1, x_2\, \mathsf{as}\, n_1, n_2\, \mathsf{in}\, \mathsf{if}\, n_1 = n_2\, \mathsf{then}\, (\mathsf{in}_1\, n_1)\, \mathsf{else}\, (\mathsf{in}_2\, ())$$

Given a program $\mathsf{votes} : \mathcal{M}\mathbb{N} \otimes \mathcal{M}\mathbb{N}$ that computes what each agent will vote, the full distributed program can be represented as the application $\mathsf{leader}\ \mathsf{votes}$.

Soundness theorem In this model, our soundness result ensures that if we have a closed program of type $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$, then it can be factored as two local programs that can be run locally, and do not require any extra communication other than the send instructions. To understand why this guarantee is non-trivial, consider the problematic program from Section 5.3:

$$\mathsf{message} : \mathcal{M}(1+1) \not\vdash_I \mathsf{if}\, \mathsf{message}\, \mathsf{then}\, (\mathsf{tt} \otimes \mathsf{tt})\, \mathsf{else}\, (\mathsf{ff} \otimes \mathsf{ff}) : \mathcal{M}\mathcal{B} \otimes \mathcal{M}\mathcal{B}$$

Under our interpretation, the if-statement is conditioning on the contents of the program variable message and producing two local computations that have the same outputs. There are two potential sources of implicit communication in this program. First, the contents of message are non-deterministic, so the local computations must communicate in order to agree on what value to return. Second, by conditioning on the same value, the message must be sent to both local computations. These indirect communications have already been addressed in the choreography literature, as illustrated by Hirsch and Garg [47], where their language allows pattern matching on local computation but the chosen branch must be broadcast to programs that depend on it, which is not problematic in a setting where communication is reliable.

To illustrate the soundness guarantee, we can revisit the distributed voting example. By the soundness theorem, the program votes is equal to $t_1 \otimes t_2$ for programs $t_1, t_2 : \mathcal{M}\mathbb{N}$. Thus, the only communication required are explicit sends.

Expressivity and Limitations Intuitively, closed programs in λ^2_{INI} of type $\mathcal{M}\tau$ are equivalent to send t_i as x_i in M, which we view as a local program M that starts by receiving n different messages, runs its body M with the received messages as bound variables, and makes its output available to be sent to a different local computation. Therefore, each local program may only have one block of receives at the beginning and one send at the end, limiting the allowed communication patterns.

These limitations have been addressed in other modal logic approaches to distributed programming by having a static set of agents and a modality annotated by elements of this set representing computations that are executed by a particular agent of the distributed system [47]. We conjecture that by extending

λ^2_{INI} with type constructors $\mathcal{M}_\ell \tau$, where ℓ is the name of an agent, it might be possible to represent more intricate communication patterns, but we leave this for future work.

Related Work Distributed programming is challenging and error-prone, and there is a long history of language design in this setting. Two notable examples are session types [49] and choreographic programming [66]. Session types adopts a linear typing discipline where type constructors model the desired protocol. On the other hand, choreographic programming adopts a monolithic approach: The entire system is written as a single program that can be compiled to "local computations", with the compiler adding the appropriate communication instructions.

Our model of λ^2_{INI} blends aspects of both approaches. It still has a substructural communication type system, but it also represents protocols using a single global program with a two-tier language that distinguishes between local and global computation. We leave a more thorough comparison between these languages for future work.

Commutative Effects

In this section we will present a large class of models based on commutative monads which are monads where, in a Kleisli semantics of effects, the program equation $(\mathsf{let}\ x = t\ \mathsf{in}\ \mathsf{let}\ y = u\ \mathsf{in}\ w) \equiv (\mathsf{let}\ y = u\ \mathsf{in}\ \mathsf{let}\ x = t\ \mathsf{in}\ w)$ holds.

The Kleisli category of commutative monads has many useful properties.

Theorem 5.4.8 (Fritz [41])**.** *Let* $\mathbf{C}$ *be a Cartesian category and* T *a commutative monad over it. The category* $\mathbf{C}_T$ *is a CD category.*

Lemma 5.4.9. *Let* $\mathbf{C}$ *be a distributive category and* T *a monad over it. Its Kleisli*

category $\mathbf{C}_T$ *has coproducts and is also distributive.*

Proof. It is straightforward to show that Kleisli categories inherit coproducts from the base category. Furthermore, by using the distributive structure of $\mathbf{C}$, applying T to it and using the functor laws, it follows that $\mathbf{C}_T$ is distributive. □

Another useful category of algebras is the category of algebras and plain maps $\widetilde{\mathbf{C}^T}$ which has T algebras as objects and $\widetilde{\mathbf{C}^T}((A, f), (B, g)) = \mathbf{C}(A, B)$.

Theorem 5.4.10 (Simpson [85])**.** *Let* $\mathbf{C}$ *be a Cartesian closed category and* T *a commutative monad over it. The category of* T*-algebras and plain maps is Cartesian closed, and* 1 *is a terminal object.*

Theorem 5.4.11 (Levy [57], Proposition 121)**.** *Let* $\mathbf{C}$ *be a distributive category and* T *a strong monad over it. The category of* T*-algebras and plain maps has weak coproducts.*

Proof sketch. Let (A, α_A) and (B, α_B) be two T-algebras, their weak coproduct is defined as $(T(A+B), \mu_{A+B})$. The injections are defined as $in_i; \eta_{A+B}$, for $i \in \{1, 2\}$ and the pattern-matching of morphisms $f : (A, \alpha_A) \to (C, \alpha_C)$ and $g : (B, \alpha_B) \to (C, \alpha_C)$ is $T([f, g]); \alpha_C$. □

Therefore, we choose the Kleisli category to interpret NI and the category of T-algebras and plain maps to interpret I. We only have to show that there is an applicative functor between them.

Theorem 5.4.12. *There exists an applicative functor* $\iota : \mathbf{C}_T \to \widetilde{\mathbf{C}^T}$.

Proof. The functor acts by sending objects A to the free algebra (TA, μ_A) and morphisms $f : A \to TB$ to f^*. Now, for the lax monoidal structure, consider the natural transformation $\mu \circ T\tau \circ \sigma : TA \times TB \to T(A \times B)$ and $\eta_1 : 1 \to T1$, where τ and σ are the strengths of T. It is possible to show that this corresponds

to an applicative functor by using the fact that T is commutative and that the comonoid structure $A \to 1$ is natural. □

Theorem 5.4.13. *The triple* $(\widetilde{\mathbf{C}^T}, \mathbf{C}_T, \iota)$ *is a* λ^2_{INI} *model.*

Name generation Simple concrete examples of commutative effects are probability and non-determinism, which we saw before. A more interesting example is the name generation monad used to give semantics to the ν-calculus, a language that has a primitive for generating "fresh" symbols [87]. This is a useful abstraction, for instance, in cryptography, where a new symbol might be a secret that you might not want to share with adversaries.

A concrete semantics to the ν-calculus was presented by Stark [87] where the base category is the functor category $[\mathbf{Inj}, \mathbf{Set}]$, with $\mathbf{Inj}$ being the category of finite sets and injective functions. In this case the (commutative) name generation monad acts on functors as

$$T(A)(s) = \{(s', a') \mid s' \in \mathbf{Inj}, a' \in A(s + s')\}/\sim$$

where $(s_1, a_1) \sim (s_2, a_2)$ if, and only if, for some s_0 there are injective functions $f_1 : s_1 \to s_0$ and $f_2 : s_2 \to s_0$ such that $A(id_s + f_1)a_1 = A(id_s + f_2)a_2$. The intuition is that $T(A)$ is a computation that, given a finite set s of names used, produces the newly generated names s', and a value a'. By Theorem 5.4.13 the triple $(\widetilde{[\mathbf{Inj}, \mathbf{Set}]^T}, [\mathbf{Inj}, \mathbf{Set}]_T, \iota)$ is a λ^2_{INI} model.

Syntactically, we can extend the type grammar of the NI language with a type Name for names, and the NI language with an operation $\cdot \vdash \mathsf{fresh} : \mathsf{Name}$ for name generation. Our soundness theorem says that for a program of type $\mathcal{M}\tau \otimes \mathcal{M}\tau$, the names used to compute the first component are *disjoint* from the ones used to compute the second component.

It is also possible to define a variant to this algebra model using the Eilenberg-Moore category since this category is known to be symmetric monoidal closed under a few minor hypothesis [7].

Remark 5.4.14 (Call-by-Value and Call-by-Name Semantics of Effects)**.** Categories of algebras and plain maps were used as a denotational foundation for call-by-name programming languages while Kleisli categories can be used to interpret call-by-value languages [85]. Thus, the I language can be seen as a CBN interpretation of effects, while NI can be seen as a CBV interpretation of effects. The operational interpretation of $\mathsf{sample}\ \bar{t}\ \mathsf{as}\ \overline{x}\ \mathsf{in}\ M$ is to force the execution of CBN computations $\bar{t}$, bind the results to $\overline{x}$, and run them eagerly in the program M.

Affine Bunched Typing

The logic of bunched implications (BI) [72] is a substructural logic, developed for reasoning about sharing and separation of resources like pointers to a heap memory [73], or permissions to enter some critical section in concurrent code [71]. The proof theory of BI gives rise to functional languages with bunched type systems, where contexts are trees (so-called *bunches*) rather than lists [70].

It is natural to wonder how BI is related to $\lambda^{2}_{\mathrm{INI}}$. Semantically, bunched calculi are interpreted using a *doubly closed category* (DCC), a single category that has both a Cartesian closed and a (usually distinct) monoidal closed structure. In order to understand how these systems are related, let us consider the affine variant of the bunched calculus, i.e., when the monoidal unit is a terminal object in the semantic category, meaning that there is a discard operation $A \otimes B \to A$. Given an affine BI model $\mathbf{C}$, there is a morphism $A \otimes B \to A \times B$ given by the

universal property of products applied to the discard morphisms $A \otimes B \to A$ and $A \otimes B \to B$. Furthermore, by assumption $I \cong 1$, where 1 is the unit for the Cartesian product and I is the unit for the monoidal product. Finally, such a structure makes the lax monoidality diagrams commute, making the identity functor $id : (\mathbf{C}, \times, 1) \to (\mathbf{C}, \otimes, I)$ a lax monoidal functor between the two monoidal structures over $\mathbf{C}$. Thus:

Theorem 5.4.15. *For every model of affine BI* $\mathbf{C}$ *the triple* $(\mathbf{C}, \mathbf{C}, id)$ *is a model of* λ^2_{INI}.

Remark 5.4.16. From a more abstract point of view, by initiality of the syntactic model of λ^2_{INI} (Theorem 5.5.4) and the theorem above, there is a translation from λ^2_{INI} to the bunched calculus. Thus, affine bunched calculi can be seen as a degenerate version of our language, where the two layers are collapsed into one.

Syntactic Control of Interference To illustrate a useful model of the affine bunched calculus, let us consider O'Hearn's bunched language SCI+ [70]. This language allows allocating memory and reasoning about aliasing, building on Reynolds' Syntactic Control of Interference (SCI), a linear type system. In the denotational semantics of SCI+, types are objects in the functor category $\mathbf{Set}^{\mathcal{P}(Loc)}$, where $\mathcal{P}(Loc)$ is the poset category of subsets of Loc, an infinite set of names (i.e., memory addresses). Intuitively, a presheaf maps a subset of locations to the set of computations that use those locations. It is well-known that this category is a model of affine BI: The Cartesian closed structure is given by the usual construction on presheaves, while the monoidal closed structure is given by a different product on presheaves, called the Day convolution [19].

By Theorem 5.4.15 the triple $(\mathbf{Set}^{\mathcal{P}(Loc)}, \mathbf{Set}^{\mathcal{P}(Loc)}, id)$ is a λ^2_{INI} model and, therefore, satisfies its soundness property. To understand what it means in this

$$
\begin{array}{lrcl}
\text{types} & \tau & ::= & \mathsf{cell} \mid \mathsf{exp} \mid \mathsf{comm} \mid \tau \to \tau \mid \tau \multimap \tau \mid \tau \times \tau \\
\text{contexts} & \Gamma & ::= & \cdot \mid x : \tau \mid \Gamma; \Gamma \mid \Gamma, \Gamma
\end{array}
$$

Figure 5.9: Types and Terms: SCI+

context, we look at how the model is defined. Given presheaves A and B over $\mathcal{P}(Loc)$, the monoidal product $A \otimes B$ is defined as

$$
\begin{aligned}
(A \otimes B)(X) &\triangleq \{(a, b) \in A(X) \times B(X) \mid support(a) \cap support(b) = \emptyset\} \\
(A \otimes B)(f) &\triangleq (Afa, Bfb)
\end{aligned}
$$

The $support$ function acts on sets and has a slightly technical definition that models which resources in Loc were used to produce the set—the interested reader should consult the original paper [70]. At a high level, the disjointness of the support captures the fact that the memory locations used to produce a are disjoint from the memory locations used to produce b. Therefore, our soundness theorem guarantees that the components of closed programs of type $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ do not share any memory locations.

At the syntactic level, the SCI+ calculus shares some similarities with λ_{INI}, such as having two distinct product types, but it also has many differences. For instance it has two context concatenation operations, making it possible to accommodate two different kinds of arrow types, shown in Figure 5.9. Additionally, it features ground types exp, cell and comm for expressions, memory cells and commands, respectively, and primitive operations to manipulate them.

For our purposes, we are mainly interested in the SCI+ operations presented in Figure 5.10. The first two rules are for composing commands either sequentially or in parallel, respectively. The following two rules are the ones related to memory manipulation, where the first one allocates a new memory location and the second one assigns a value to a location. The final two are the two ap-

$$\frac{\Gamma \vdash M : \mathsf{comm} \qquad \Gamma \vdash N : \mathsf{comm}}{\Gamma \vdash M; N : \mathsf{comm}} \qquad \frac{\Gamma_1 \vdash M : \mathsf{comm} \qquad \Gamma_2 \vdash N : \mathsf{comm}}{\Gamma_1, \Gamma_2 \vdash M || N : \mathsf{comm}}$$

$$\frac{\Gamma, x : \mathsf{cell} \vdash M : \mathsf{comm}}{\Gamma \vdash \mathsf{new}\, x.M : \mathsf{comm}} \qquad \frac{\Gamma \vdash M : \mathsf{cell} \qquad \Gamma \vdash N : \mathsf{exp}}{\Gamma \vdash M := N : \mathsf{comm}}$$

$$\frac{\Gamma \vdash M : \tau_1 \to \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\ N : \tau_2} \qquad \frac{\Gamma_1 \vdash M : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash N : \tau_1}{\Gamma_1, \Gamma_2 \vdash M\ N : \tau_2}$$

Figure 5.10: Typing Rules: SCI+ (selected)

plications: the first allows the context to be shared, while the second does not.

A notorious difficulty of running stateful programs in parallel is that there might be concurrent writes to the same memory location. This is avoided in SCI+ by using the separating concatenation of contexts, guaranteeing that no such conflict of writes can occur. When programs are sequentially composed, no such issues come up and the context may be shared. When a new memory cell is allocated using the $\mathsf{new}\, x.M$ syntax, a new variable is bound to the context representing the new location which is disjoint from the existing ones, hence the separating context extension.

SCI+ in $\lambda^2_{\mathbf{INI}}$ As we have explained, a direct consequence of Theorem 5.4.15 is that there is a translation of λ^2_{INI} into the BI calculus. However, it is not a direct consequence that the cell and command operations can be given similar typing rules and semantics to their original formulation. By slightly modifying λ^2_{INI} we can accommodate them as we show in Figure 5.11. Sequential composition is done in the NI language while parallel composition is done at the I language. The cell assignment rule is added to the NI language, since there is no reason to require that a cell's address and its value are computed using separate locations. For cell allocation, the original rule requires the new cell to be disjoint from the

$$\textsc{Sequential}\quad \frac{\Gamma \vdash_{NI} M : \mathsf{comm} \qquad \Gamma \vdash_{NI} N : \mathsf{comm}}{\Gamma \vdash_{NI} M; N : \mathsf{comm}}$$

$$\textsc{Parallel}\quad \frac{\Gamma_1 \vdash_{I} t : \mathcal{M}\mathsf{comm} \qquad \Gamma \vdash_{I} u : \mathcal{M}\mathsf{comm}}{\Gamma_1, \Gamma_2 \vdash_{I} t \,||\, u : \mathcal{M}\mathsf{comm}}$$

$$\textsc{New}\quad \frac{\Gamma, x : \mathcal{M}\mathsf{cell} \vdash_{I} t : \mathcal{M}\mathsf{comm}}{\Gamma \vdash_{I} \mathsf{new}\, x.t : \mathcal{M}\mathsf{comm}} \qquad \textsc{Assign}\quad \frac{\Gamma \vdash_{NI} M : \mathsf{cell} \qquad \Gamma \vdash_{NI} N : \mathsf{exp}}{\Gamma \vdash_{NI} M := N : \mathsf{comm}}$$

Figure 5.11: Typing Rules: λ^2_{INI} extended with SCI primitives

existing ones, making it natural to use the I language.

Example 5.4.17 (O'Hearn [70])**.** Consider the λ^2_{INI} program $(\lambda x\, y.\ x := 1; y := 2)\ z\ z$. There are two possible types for the λ-abstraction. The type $\mathcal{M}\mathsf{cell} \multimap \mathcal{M}\mathsf{cell} \multimap \mathcal{M}\mathsf{comm}$ requires that the input locations x and y must be disjoint, while the type $\mathcal{M}(\mathsf{cell} \times \mathsf{cell}) \multimap \mathcal{M}\mathsf{comm}$ allows x and y to be shared. The former makes the application ill-typed, since the arguments to the abstraction are the same, while the latter is well-typed. Note, however, that it is only well-typed because the assignments are sequentially composed. If they were composed in parallel the program would be ill-typed, just like in SCI+, since parallel composition requires disjoint memory locations.

A more expressive λ^2_{INI} SCI+ supports more fine-grained sharing/disjointness policies that interleave the $\times$ and $\otimes$ type constructors—these programs are difficult to express in λ^2_{INI}. For instance, it is not possible to represent the type $\mathcal{M}(A \otimes B) \times \mathcal{M}(C \otimes D)$ in our language. This limitation is because there is only one modality mapping the NI language into the I language, and no modality going the other way. This limitation can also be seen in the following simple

program, which cannot be expressed in λ^2_{INI}: $x := 1; (y := 2) \,||\, (z := 3)$. The program is ill-typed because only NI programs can be sequentially composed and only I programs can be composed in parallel. In the concrete model, however, the lax monoidal functor is the identity functor, allowing us to add the clause $\tau := \tau \mid \cdots$ to the NI type grammar and making the following typing rule sound:

$$\frac{\Gamma \vdash_{I} t : \tau}{\Gamma \vdash_{NI} t : \tau}$$

which makes it possible to type check the troublesome program above.

5.5 Soundness Theorem

So far we have seen two proofs of soundness. For λ_{INI}, we proved soundness using logical relations (Theorem 5.2.3). For λ^2_{INI} with a probabilistic semantics, we used an observation about algebras for the distribution monad (Theorem 5.3.1). This proof is slick, but the strategy does not generalize to other models of λ^2_{INI}.

Thus, to prove our general soundness theorem for λ^2_{INI}, we will return to logical relations. The statement of our soundness theorem is as follows.

Theorem 5.5.1. *If* $\cdot \vdash_{I} t : \mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ *then* $[\![t]\!]$ *can be factored as two morphisms* $[\![t]\!] = f_1 \otimes f_2$, *where* $f_1 : I \to \mathcal{M}\,[\![\tau_1]\!]$ *and* $f_2 : I \to \mathcal{M}\,[\![\tau_2]\!]$.

Logical relations are frequently used to prove metatheoretical properties of type theories and programming languages. However, they are usually used in concrete settings, i.e., for a concrete model where we can define the logical relation explicitly. In our case, however, this approach is not enough, since we are working with an abstract categorical semantics of λ^2_{INI}. Thus, we will leverage

the categorical treatment of logical relations, called *Artin gluing*, a construction originally used in topos theory [52, 50].

5.5.1 Category of Models

A model for λ^2_{INI} is given by a CD category $\mathbf{M}$ with coproducts, a SMCC $\mathbf{C}$ with weak coproducts and a lax monoidal functor $\mathcal{M} : \mathbf{M} \to \mathbf{C}$. A morphism between two models $(\mathbf{M}_1, \mathbf{C}_1, \mathcal{M}_1)$ and $(\mathbf{M}_2, \mathbf{C}_2, \mathcal{M}_2)$ is a pair of functors $(F : \mathbf{M}_1 \to \mathbf{M}_2, G : \mathbf{C}_1 \to \mathbf{C}_2)$ that preserves the logical connectives. By defining morphism composition component-wise and the pair $(id_{\mathbf{C}}, id_{\mathbf{M}})$ as the identity morphism, this structure constitutes a category which we call $\mathbf{Mod}$.

In categorical treatments of type theories it is important to show that the equational theory is a sound approximation of the categorical semantics. In the case of λ^2_{INI}, since the language does not use any fancy type theoretic constructions, the soundness property is straightforward to prove by induction of the typing derivations.

Theorem 5.5.2. *Let* $(\mathbf{C}, \mathbf{M}, \mathcal{M})$ *be a* λ^2_{INI} *model. If* $\Gamma \vdash_{NI} M \equiv N : \tau$ *then* $[\![M]\!] = [\![N]\!]$ *and if* $\Gamma \vdash_{I} t \equiv u : \tau$ *then* $[\![t]\!] = [\![u]\!]$.

The main subtlety is that we have to be a bit more precise in the presentation of the equational theory for the I language. Note that the sample construct can sample simultaneously from any number of distributions, while applicative functors only provide a binary sampling operator. Formally, this is resolved by restricting sample to two arguments and add the following rules to the equational theory.

Note that even though the rule looks intimidating, it is basically the lax monoidal commutativity diagram in syntax form, which says that the sample

operation is associative and, as a consequence, there is a unique way of defining the n-ary operation sample $t_1, \ldots t_n$ as $x_1, \ldots, x_n$ in M, for $n \geq 2$.

An important λ^2_{INI} model is the syntactic object $\mathbf{Syn}$, which is a triple $(\mathbf{Syn}_{lin}, \mathbf{Syn}_{CD}, \mathcal{M})$, where $\mathbf{Syn}_{CD}$ is the syntactic category of CD categories with coproducts while $\mathbf{Syn}_{lin}$ is the syntactic category of symmetric monoidal closed categories with weak coproducts and an applicative modality and $\mathcal{M}$ is the type constructor for the modality. Concretely each of these categories have types as objects and morphisms are programs with one free variables modulo the equational theories presented in Figure 5.5. It follows by a simple inspection that $\mathbf{Syn}$ is a λ^2_{INI} model.

Theorem 5.5.3. $\mathbf{Syn}$ *is a* λ^2_{INI} *model.*

Theorem 5.5.4. $\mathbf{Syn}$ *is the initial object of* $\mathbf{Mod}$.

Proof. Let $(\mathbf{C}, \mathbf{M}, \mathcal{M})$ be a model. It is possible to construct a morphism $[\![\cdot]\!] : \mathbf{Syn} \to (\mathbf{C}, \mathbf{M}, \mathcal{M})$ by defining two functors $[\![\cdot]\!]_1 : \mathbf{Syn}_{lin} \to \mathbf{C}$ and $[\![\cdot]\!]_2 : \mathbf{Syn}_{CD} \to \mathbf{M}$. Since $\mathbf{Syn}_{lin}$ and $\mathbf{Syn}_{CD}$ are freely generated, the action of the functors on objects is characterized by a simple induction on the types. The action on morphisms is defined by induction on the typing derivation using Figure 5.8.

The proof that this function is well-defined follows from Theorem 5.5.2. Uniqueness follows by assuming the existence of two semantics and showing, by induction on the typing derivation, that they are equal. □

5.5.2 Glued category

We construct the logical relations category by using a comma category. Formally, a comma category along functors $F : \mathbf{C_1} \to \mathbf{D}$ and $G : \mathbf{C_2} \to \mathbf{D}$ has

triples (A, X, h) as objects, where A is an $\mathbf{C_1}$ object, X is an $\mathbf{C_2}$ objects and $h : FA \to GX$, and its morphisms $(A, X, h) \to (A', X', h')$ are pairs $f : A \to A'$ and $g : X \to X'$ making certain diagrams commute. In Computer Science applications of gluing, it is usually assumed that F is the identity functor and $\mathbf{D} = \mathbf{Set}$. Furthermore, to simplify matters, sometimes it is also assumed that we work with full subcategories of the glued category, for instance we can assume that we only want objects such that $A \to GB$ is an injection, effectively representing a subset of GB.

Therefore, in the setting we are interested in a glued category along a functor $G : \mathbf{C} \to \mathbf{Set}$ has pairs $(A, X \subseteq G(A))$ as objects and its morphisms $(A, X) \to (B, Y)$ is a $\mathbf{C}$ morphism $f : A \to B$ such that $G(f)(X) \subseteq Y$. Note that this condition can be seen as a more abstract way of phrasing the usual logical relations interpretation of arrow types: mapping related things to related things. At an intuitive level we want to use the functor G to map types to predicates satisfied by its inhabitants.

Now, we are ready to define the glued category and show that it constitutes a model for the language. Given a triple $(\mathbf{M}, \mathbf{C}, \mathcal{M})$ we define the triple $(\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}})$, where the objects of $\mathbf{Gl}(\mathbf{C})$ are pairs $(A \in \mathbf{C}, X \subseteq \mathbf{C}(I, A))$ and the morphisms are $\mathbf{C}$ morphisms that preserve X, i.e. we are gluing $\mathbf{C}$ along the global sections functor $\mathbf{C}(I, -)$. The functor $\mathcal{M} : \mathbf{M} \to \mathbf{C}$ is lifted to a functor $\widetilde{\mathcal{M}} : \mathbf{C} \to \mathbf{Gl}(\mathbf{C})$ by mapping objects X to $(\mathcal{M}\, X, \mathbf{C}(I, \mathcal{M}\, X))$ and by mapping morphisms f to $\mathcal{M}\, f$.[1] Now we have to show that the triple is indeed a model of our language.

Something that simplifies our proofs is that morphisms in $\mathbf{Gl}(\mathbf{C})$ are simply morphisms in $\mathbf{C}$ with extra structure and composition is kept the same. There-

[1] Note that its predicate set is every $\mathbf{C}$ morphism $I \to \mathcal{M}\, X$, similar to how ground types are interpreted in usual logical relations proofs.

fore, once we establish that a $\mathbf{C}$ morphism is also a $\mathbf{Gl}(\mathbf{C})$ morphism all we have to do in order to show that a certain $\mathbf{Gl}(\mathbf{C})$ diagram commutes is to show that the respective $\mathbf{C}$ diagram commutes.

Theorem 5.5.5 ([50]). $\mathbf{Gl}(\mathbf{C})$ *is a SMCC with weak coproducts and with a natural transformation* $del_A : A \to I$.

Proof. Let (A, X) and (B, Y) be $\mathbf{Gl}(\mathbf{C})$ objects, we define $(A, X) \otimes (B, Y) = (A \otimes B, \{f : I \to A \otimes B \mid f = f_A \otimes f_B, f_A \in X, f_B \in Y\})$. The monoidal unit is given by $(I, \mathbf{C}(I, I))$ and the natural transformation del is the same one as the one in $\mathbf{C}$, which is a morphism in $\mathbf{Gl}(\mathbf{C})$ because $X_I = \mathbf{C}(I, I)$.

Let (A, X) and (B, Y) be $\mathbf{Gl}(\mathbf{C})$ objects, we define $(A, X) \multimap (B, Y) = (A \multimap B, \{f : I \to (A \multimap B) \mid \forall f_A \in X_A, \epsilon_B \circ (f_A \otimes f) \in X_B\}$, where $\epsilon_B : (A \multimap B) \otimes A \to B$ is the counit of the monoidal closed adjunction.

To show $A \otimes (-) \dashv A \multimap (-)$ we can use the (co)unit characterization of adjunctions, which corresponds to the existence of two natural transformations $\epsilon_B : A \otimes (A \multimap B) \to B$ and $\eta_B : B \to A \multimap (A \otimes B)$ such that $1_{A \otimes -} = \epsilon(A \otimes -) \circ (A \otimes -)\eta$ and $1_{A \multimap -} = (A \multimap -)\epsilon \circ \eta(A \multimap -)$, where 1_F is the identity natural transformation between F and itself. By choosing these natural transformations to be the same as in $\mathbf{C}$, since the adjoint equations hold for them by definition, all we have to do is show that they are also $\mathbf{Gl}(\mathbf{C})$ morphisms, which follows by unfolding the definitions.

Finally, we can show that $\mathbf{Gl}(\mathbf{C})$ has weak coproducts. Let (A_1, X_1) and (A_2, X_2) be $\mathbf{Gl}(\mathbf{C})$ objects, we define $(A_1, X_1) \oplus (A_2, X_2) = (A_1 \oplus A_2, \{\mathsf{in_i}\, f_i \mid f_i \in X_i\})$. To show that it satisfies the weak universal property of sum types. Let $f_1 : A_1 \to B$ and $f_2 : A_2 \to B$ be $\mathbf{Gl}(\mathbf{C})$ morphisms. Consider the $\mathbf{C}$ morphism $[f_1, f_2]$. We want to show that this morphism is also a $\mathbf{Gl}(\mathbf{C})$ morphism. Consider $g \in X_{A_1 \oplus A_2}$ which, by assumption, $g = \mathsf{in}_1 g_1$ or $g = \mathsf{in}_2 g_2$. By case analysis

and the facts $f_i \circ g_i \in Y$ and $[f_1, f_2] \circ \mathsf{in}_i g_i = f_i \circ g_i$ we can conclude that $[f_1, f_2]$ is indeed a $\mathbf{Gl}(\mathbf{C})$ morphism. □

These constructions are known in the categorical logic literature [50], but since it is simple enough we think that it is helpful to also present it here. Since every construction so far uses the same objects as the ones in $\mathbf{C}$, it is possible to show that the forgetful functor $U : \mathbf{Gl}(\mathbf{C}) \to \mathbf{C}$ preserves every type constructor and is a $\mathbf{Mod}$ morphism. Next, we have to show that $\widetilde{\mathcal{M}}$ is lax monoidal which follows from the fact that μ and ϵ preserve the plot sets, by a simple unfolding of the definitions. We can now easily conclude that the lax monoidality diagrams commute, since composition is the same and $\mathcal{M}$ is lax monoidal.

Thus, the glued category is a model.

Theorem 5.5.6. *The triple* $(\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}})$ *is a* $\mathbf{Mod}$ *object.*

There is a forgetful map from the glued model to the original model.

Theorem 5.5.7. *There is a* $\mathbf{Mod}$ *morphism* $U : (\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}}) \to (\mathbf{M}, \mathbf{C}, \mathcal{M})$.

Finally, by initiality of $\mathbf{Syn}$, we can prove

Theorem 5.5.8. *There is a* $\mathbf{Mod}$ *morphism* $(\!|\cdot|\!) : \mathbf{Syn} \to (\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}})$.

With this map in hand, we may now construct a functor $U \circ (\!|\cdot|\!) : \mathbf{Syn} \to (\mathbf{M}, \mathbf{C}, \mathcal{M})$ which, by initiality of $\mathbf{Syn}$, is equal to the functor $[\![\cdot]\!]$, as illustrated by Figure 5.12.

5.5.3 General Soundness Theorem

Theorem 5.5.9. *If* $\cdot \vdash_I t : \underline{\tau}$, *then* $[\![t]\!] \in X_{\underline{\tau}}$.

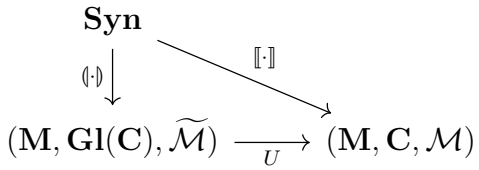


Figure 5.12: The essence of the soundness proof

Proof. We know that $[\![\cdot]\!] = U \circ (\!|\cdot|\!)$ and that $(\!|t|\!)$ is a $\mathbf{Gl}(\mathbf{C})$ morphism. As such we have that $[\![t]\!] = (\!|t|\!) = (\!|t|\!) \circ id_I \in X_{\underline{\tau}}$, since, by definition, $id_I \in X_I$. □

Theorem 5.6.1 follows immediately, as a corollary.

Corollary 5.5.10. *If* $\cdot \vdash_I t : \mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ *then* $[\![t]\!]$ *can be factored as two morphisms* $[\![t]\!] = f_1 \otimes f_2$*, where* $f_1 : I \to \mathcal{M}\,[\![\tau_1]\!]$ *and* $f_2 : I \to \mathcal{M}\,[\![\tau_2]\!]$*.*

Proof. By Theorem 5.5.9, if $\cdot \vdash_I t : \mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$, then $[\![t]\!] \in X_{\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2}$ which, by unfolding the definitions, means that there exists $f_1 : I \to \mathcal{M}\,[\![\tau_1]\!]$ and $f_2 : I \to \mathcal{M}\,[\![\tau_2]\!]$ such that $[\![t]\!] = f_1 \otimes f_2$. □

5.6 Conditional Independence

Now we propose a similar type system that can also reason about conditional independence. In this section, though we will also use a two-level language, we will simplify the presentation by dropping higher-order functions from the language, even though they are conceptually simple to define. As a starting point, we will design a type system that works for an arbitrary Markov category and then we will define its general categorical semantics.

The main idea is to focus on Definition 2.3.7 which provides a general way of defining conditional independence in the context of Markov categories. The intuition behind conditional independence is that the output distribution is independent modulo the sharing that may happen with the components you are

$$
\begin{array}{lllll}
\text{Variables} & x, y, z & & & \\
\text{Types} & \tau & ::= & \mathcal{B} \mid \tau \otimes \tau & \\
\text{Expressions} & t, u & ::= & x \mid b \in \mathcal{B} \mid \mathsf{coin} \mid (M, N) \mid \pi_i\, N & \\
 & & \mid & \mathsf{let}\ x = M\ \mathsf{in}\ N \mid \mathsf{let}_{\mathsf{det}}\ x = M\ \mathsf{in}\ N \mid f(M) & \\
\text{Contexts} & \Gamma; \Delta & ::= & (x_1 : \tau_1, \ldots, x_n : \tau_n); (y_1 : \tau'_1, \ldots, y_m : \tau'_m) &
\end{array}
$$

Figure 5.13: Types and Terms

$$
\textsc{Var}\ \frac{}{x : \tau, \Gamma; \Delta \vdash x : \tau} \qquad \textsc{VarDet}\ \frac{}{\Gamma; x : \tau, \Delta \vdash x : \tau} \qquad \textsc{Coin}\ \frac{}{\Gamma; \Delta \vdash \mathsf{coin} : \mathcal{B}}
$$

$$
\frac{\Gamma_1; \Delta \vdash M : \tau_1 \qquad \Gamma_2; \Delta \vdash N : \tau_2}{\Gamma_1, \Gamma_2; \Delta \vdash (M, N) : \tau_1 \otimes \tau_2} \qquad \frac{\Gamma; \Delta \vdash M : \tau_1 \otimes \tau_2}{\Gamma; \Delta \vdash \pi_i\, M : \tau_i}
$$

$$
\frac{\Gamma_1; \Delta \vdash M : \tau_1 \qquad \Gamma_2, x : \tau_1; \Delta \vdash N : \tau}{\Gamma_1, \Gamma_2; \Delta \vdash \mathsf{let}\ x = M\ \mathsf{in}\ N : \tau}
$$

$$
\frac{\Delta \vdash_{\mathsf{det}} M : \tau_1 \qquad \Gamma; x : \tau_1, \Delta \vdash N : \tau}{\Gamma; \Delta \vdash \mathsf{let}_{\mathsf{det}}\ x = M\ \mathsf{in}\ N : \tau} \qquad \textsc{Const}\ \frac{f \in \mathbf{C}(\tau_1, \tau_2) \qquad \Gamma; \Delta \vdash M : \tau_1}{\Gamma; \Delta \vdash f(M) : \tau_2}
$$

Figure 5.14: Typing rules for CI

conditionally independent on. This suggests that, differently from λ^2_{INI} there can be sharing of resources, but only reserved to a few variables. The other important observation is that, Lemma 2.3.11, conditional independence is preserved by precomposition with deterministic morphisms.

5.6.1 A Type System

Let us assume that there is a base type $\mathcal{B}$ of Booleans. We will use a dual context type system where the first context Γ will be for variables that cannot be shared while the context Δ will be for variables may be shared by can only be precomposed by deterministic programs.

The typing rules for $\vdash_{\mathsf{det}}$ are exactly the same except for the Const rule, where f is restrited to deterministic morphisms, and Coin, which does not exist in $\mathbf{C}_{det}$.

We want this type system to capture conditional independence in the following sense:

Theorem 5.6.1. *Let $\cdot; \Delta \vdash M : \tau_1 \otimes \tau_2$ be a well-typed program, then $[\![M]\!]$ has the CI property.*

This proof does not work by a simple induction but, once again, it can be proved by a logical relations argument. We present the more general proof in Theorem 5.6.17.

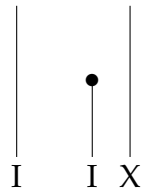
A Fibrational Semantics

Before we present the most general semantics, we will illustrate how fibrations provide some nice insights into the problem. In Cartesian categories $\mathbf{B}$ it is possible to define the *simple fibration*, as explained in Section 2.2.5, which are functors $p : s(\mathbf{B}) \to \mathbf{B}$, where $s(\mathbf{B})$ is the category which has pairs (I, X) of $\mathbf{B}$ as objects and morphisms $(I, X) \to (J, Y)$ are a pair of morphisms (u, f) of $\mathbf{B}$, where $U : I \to J$ and $f : I \times X \to Y$. The functor p maps (I, X) to I and (u, f) to u. This construction is fairly general but, unfortunately, it does not scale to symmetric monoidal categories: the assumption that $\mathbf{C}$ is Cartesian is necessary when defining the category $s(\mathbf{B})$.

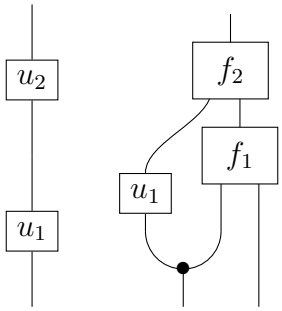
Something particular about Markov categories, however, is that, by Theorem 2.3.10, they have well-behaved Cartesian subcategories which can be used to define variants of the simple fibration. We define $s_{det}(\mathbf{C})$ which has the same objects as before, except that morphisms are pairs (u, f) where $u : I \to J$ has to be deterministic and $f : I \otimes X \to Y$ can be any morphism.

Theorem 5.6.2. $s_{det}(\mathbf{C})$ *is a category.*

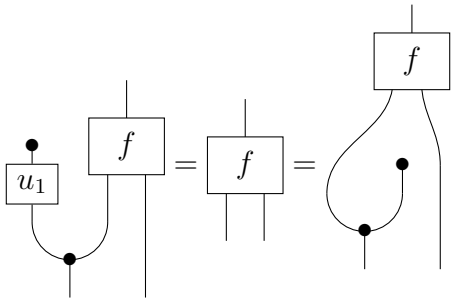
Proof. The identity morphism is given by



Composition is defined as



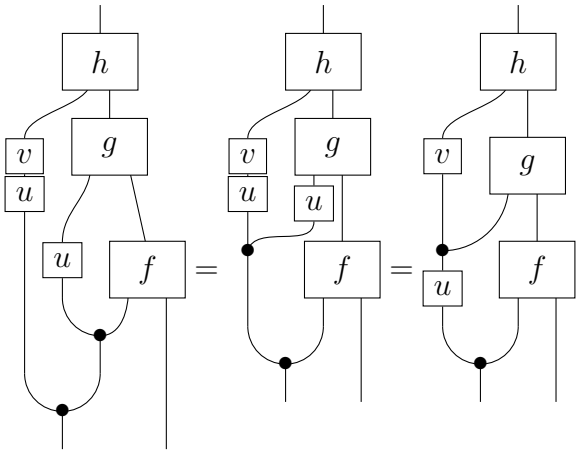
The proof that this is indeed a category follows by string diagram chasing. Note that since composition for the deterministic morphisms $I \to J$ are given by regular composition of morphisms in a category, they obviously satisfy the category axioms. We now show that the morphisms $I \otimes X \to Y$ do follow the category axioms. The identity axioms follows from



(5.1)

The associativity of composition is given by the following string diagram ma-

nipulation



(5.2)

□

The intuition behind $s_{det}(\mathbf{C})$ is that the objects I correspond to the deterministic contexts of figure 5.14, hence the restriction of it only being allowed to be composed by deterministic morphisms, while the objects X are the contexts Γ.

In order to show that $s_{det}(\mathbf{C})$ is indeed the total category of a fibration, we construct it via the Grothendieck construction.

Theorem 5.6.3. *There is a functor* $F : \mathbf{C}_{det}^{op} \to \mathbf{Cat}$.

Proof. Let I be an object in $\mathbf{C}$ and we map it to the category which has the same objects as $\mathbf{C}$ with morphisms $I \otimes X \to Y$ in $\mathbf{C}$. The action on morphisms $u : I \to J$ is given by mapping a morphism $f : J \otimes X \to Y$ into $(u \otimes 1_X); f : I \otimes X \to Y$. The proof that this construction is indeed a functor between the fiber categories follows by theorem 5.6.2. The proof that the mapping F is functorial follows by unfolding definitions. □

By applying the Grothendieck construction to F we obtain a fibration such that the total category is $s_{det}(\mathbf{C})$. The proof follows by a direct calculation similar to the simple fibration case. We use this observation as a starting point for

defining a more general semantics and proving the conditional independence soundness theorem.

5.6.2 An Abstract Categorical Semantics

Remember that the definition of subcategory is not very categorical. It is more natural to do things in terms of faithful functors between categories. But first, consider the generalization of Theorem 5.6.3.

Theorem 5.6.4. *Let* $\mathbf{D}$ *be a Cartesian category,* $\mathbf{C}$ *a symmetric monoidal category and* $F : \mathbf{D} \to \mathbf{C}$ *a monoidal functor. There is a functor* $F : \mathbf{D}^{op} \to \mathbf{Cat}$.

Proof. Analogous to the proof of theorem 5.6.3. □

Once again, by using the Grothendieck construction, we can turn the functor above into a fibration $s_{det}(\mathbf{C}) \to \mathbf{D}$, which we call *deterministic* simple fibration. In this section we will do things a bit differently from λ^2_{INI}. We start by studying the ideal categorical semantics so that later we can define an appropriate type system for it.

Definition 5.6.5. A CI_1 model is given by a triple $(\mathbf{C}, \mathbf{M}, F)$ where $\mathbf{C}$ is a Cartesian category, $\mathbf{M}$ is a symmetric monoidal category and $F : \mathbf{C} \to \mathbf{M}$ is a monoidal functor.

It is immediate to see that this definition is a generalization of the semantics presented in the section above, where for a given Markov category $\mathbf{C}$, $\mathbf{C} = \mathbf{C}_{det}$, $\mathbf{M} = \mathbf{C}$ and $F = id$. The same fibrational definitions above should directly generalize to this new setting with a few minor updates, such as we restricting ourselves to morphisms $FI \otimes X \to Y$.

Lemma 5.6.6. *Let* $(\mathbf{C}_1, \mathbf{M}_1, F_1)$ *and* $(\mathbf{C}_2, \mathbf{M}_2, F_2)$ *be* CI_1 *models and* $F : \mathbf{C}_1 \to \mathbf{C}_2$ *and* $G : \mathbf{M}_1 \to \mathbf{M}_2$ *be monoidal functors that make the obvious diagram commute. It generates a functor of fibrations between their simple deterministic fibrations.*

Proof. The functor F is the base part of the fibration morphism. We define $G'(I, X) = (FI, GX)$ and it maps the morphism $f : I \to J$ to Ff and $g : F_1 I \otimes X \to Y$ to $Gg : G(F_1 I \otimes X) \cong FF_2 I \otimes GX \to GY$. □

Definition 5.6.7. A morphism between CI_1 models $(\mathbf{C}_1, \mathbf{M}_1, F_1)$ and $(\mathbf{C}_2, \mathbf{M}_2, F_2)$ is a pair of monoidal functors F and G that makes the obvious diagram commute and such that the restriction of the canonical morphisms of fibrations preserve the monoidal structure.

We can now organize our category of models as follows

Definition 5.6.8. The category $\mathbf{Mod}_{CI}$ has CI_1 models as objects and CI_1 morphisms as morphisms.

Syntactic model

In order to make the logical relations proof go through we must define the syntactic model for CI_1 models and prove that it is initial. Let us start by defining the syntax. It seems reasonable that given the fact that a CI_1 model is basically a $\lambda^{\text{LL}}_{\text{MK}}$ model, that their initial models should also be similar. Indeed, we define their syntax to be the same with the exception that we add the following typing rule

$$\frac{\Gamma \vdash_{LL} t : F(\tau_1 \times \tau_2)}{\Gamma \vdash_{LL} \mathsf{marginal}(t) : F\tau_1 \otimes F\tau_2} \text{ MARGINAL}$$

Semantically, this is simply applying the inverse of the lax monoidal morphism. At the level of the equational theory we must change a few things. Since

the category $\mathbf{C}$ must be Cartesian we must change the equations accordingly in order to accommodate the universal property of products. Next, since the lax structure is an isomorphism, we must add equations capturing that. In the end their equational theories are the same with the exception that we extend it with the equations

$$(\pi_1\, M, \pi_2\, M) \quad \equiv \quad M$$

$$\begin{aligned} t \quad &\equiv \quad \mathsf{marginal}(\mathsf{let}\ x \otimes y = t\ \mathsf{in\ sample}\ x, y\ \mathsf{as}\ x', y'\ \mathsf{in}\ (x', y')) \\ t \quad &\equiv \quad \mathsf{let}\ x \otimes y = \mathsf{marginal}(t)\ \mathsf{in\ sample}\ x, y\ \mathsf{as}\ x', y'\ \mathsf{in}\ (x', y') \end{aligned}$$

Which are just the η-law for the Cartesian product and the isomorphism equations, respectively. The proof of initiality follows from familiar, albeit thedious, methods.

Theorem 5.6.9. *The triple* $\mathbf{Syn} = (\mathbf{Syn}_{Cartesian}, \mathbf{Syn}_{Monoidal}, F)$ *is a* CI_1 *model.*

Proof. We have shown in Theorem 5.5.4 that the lax monoidal syntax is a λ^2_{INI} model. From the equations we have added we can show that the monoidal structure of $\mathbf{Syn}_{Cartesian}$ is Cartesian and that the lax monoidal structure is strong. □

Though this proof is a bit indirect, the fiber categories can be quite easily characterized. An object in $\mathbf{C}_I$ is a type τ and a morphism $\mathbf{C}(\tau, \tau')$ is a well-typed program $x : \tau; y : FI \vdash M : \tau'$ modulo the equational theory of the language.

Theorem 5.6.10. $\mathbf{Syn}$ *is the initial object of* $\mathbf{Mod}$.

Glued category

We will prove theorem 5.6.1 by a categorical logical relations argument. Let $s_{det}(\mathbf{C}) \xrightarrow{p} \mathbf{C}_{det}$ be a deterministic fibration. Consider the category $\mathbf{Gl}(\mathbf{C})$ which is the category $\mathbf{C}$ glued along the functor $\mathbf{C}(F1, -)$.

Lemma 5.6.11 ([50]). *If $\mathbf{C}$ is SMC then category $\mathbf{Gl}(\mathbf{C})$ is SMC as well.*

Lemma 5.6.12. *The monoidal functor $F : \mathbf{C}_{det} \to \mathbf{C}$ lifts to a monoidal functor $\tilde{F} : \mathbf{C}_{det} \to \mathbf{Gl}(C)$*

Proof. Its action on objects is $\tilde{F}(I) = (FI, \{Ff \mid f \in \mathbf{C}_{det}(1, I)\}$ and its actions on morphisms is the same as F. The proof that Ff is a $\mathbf{Gl}$ morphism follows from the functor laws. The proof that $\tilde{F}$ is monoidal follows from $\mathbf{C}_{det}$ being a Cartesian category and from the naturality of the monoidal structure of F. □

Lemma 5.6.13. *For every object I in $\mathbf{C}_{det}$ the fiber category $s_{det}\mathbf{Gl}(\mathbf{C})_{FI}$ is symmetric monoidal.*

Proof. Let (A, X_A) and (B, Y_B) be two $\mathbf{Gl}$ objects, we define $(A, X_A) \otimes (B, Y_B) = (A \otimes B, \{\mathsf{copy}; (f \otimes g) \mid f \in X_A, g \in Y_B\})$. It inherits its action on morphisms and unit from $\mathbf{C}$. In order to show that the bifunctorial action lifts to the glued setting we make use of the associativity of the copy operation. □

Note that a morphism $s_{det}\mathbf{Gl}(\mathbf{C})_I(1, X \otimes Y)$ corresponds to a $\mathbf{C}$ morphism $f : FI \to X \otimes Y$ such that there are two morphisms f_1 and f_2 such that $f = \mathsf{copy}; (f_1 \otimes f_2)$, i.e. they are morphisms that have the CI property.

Theorem 5.6.14. *If $(\mathbf{C}_{det}, \mathbf{C}, F)$ is a CI_1 model then the triple $(\mathbf{C}_{det}, \mathbf{Gl}(\mathbf{C}), \tilde{F})$ is a CI_1 model.*

Lemma 5.6.15. *The forgetful functor $\mathbf{Gl}(\mathbf{C}) \to \mathbf{C}$ lifts to a CI_1 morphism $(\mathbf{C}_{det}, \mathbf{Gl}(\mathbf{C}), \tilde{F}) \to (\mathbf{C}_{det}, \mathbf{C}, F)$.*

Lemma 5.6.16. *The forgetful functor* $(\mathbf{C}_{det}, \mathbf{Gl}(\mathbf{C}), \tilde{F}) \to (\mathbf{C}_{det}, \mathbf{C}, F)$*, when lifted to a functor of simple deterministic fibrations preserve the monoidal product on the fiber categories.*

Theorem 5.6.17. *If* $\cdot; \Delta \vdash M : \tau_1 \otimes \tau_2$ *then* $[\![M]\!] \in s_{det}(\mathbf{C})_{[\![\Delta]\!]}(1, [\![\tau_1]\!] \otimes [\![\tau_2]\!])$ *has the CI property.*

Proof. Let $[\![M]\!]_{\mathbf{Gl}(\mathbf{C})}$ be semantics of M under the glued category. By the same argument as the unconditional independence case, we can conclude that $[\![M]\!]_{\mathbf{Gl}(\mathbf{C})}$ can be factored as $f \otimes_{[\![\Delta]\!]} g$ which, when unfolding the definitions of the fibered monoidal product results in $\mathsf{copy}_{[\![\Delta]\!]}; (f \otimes g)$. □

5.7 Related Work

Linear logics and probabilistic programs. A recent line of work uses linear logic as a powerful framework to provide semantics for probabilistic programming languages. Notably, Ehrhard et al. [38] show that a probabilistic version of the coherence-space semantics for linear logic is fully abstract for probabilistic PCF with discrete choice, and Ehrhard et al. [37] provide a denotational semantics inspired by linear logic for a higher-order probabilistic language with continuous random sampling; probabilistic versions of call-by-push-value have also been developed [90]. Linear type systems have also been developed for probabilistic properties, like almost sure termination [27] and differential privacy [79, 5].

As we have mentioned, our categorical model for λ^2_{INI} is inspired by models of linear logic based on monoidal adjunctions, most notably Benton's LNL [15]. From a programming languages perspective, these models decompose the linear λ-calculus with exponentials in two languages with distinct product types

each: one is a Cartesian product and the other is symmetric monoidal. The adjunction manifests itself in adding functorial type constructor in each language, similar to our $\mathcal{M}$ modality. These two-level languages are very similar to λ^2_{INI}, and indeed it is possible to show that every LNL model is a λ^2_{INI} model. At the same time, the class of models for λ^2_{INI} is much broader than LNL—none of the models presented in Section 5.4.2 are LNL models.

Higher-order programs and effects. There is a very large body of work on higher-order programs effects, which we cannot hope to summarize here. The semantics of λ_{INI} is an instance of Moggi's Kleisli semantics, from his seminal work on monadic effects [65]; the difference is that our one-level language uses a linear type system to enforce probabilistic independence.

Another well-known work in this area is Call-by-Push-Value (CBPV) [57]. It is a two-level metalanguage for effects which subsumes both call-by-value and call-by-name semantics. Each level has a modality that takes from one level to the other one. There is a resemblance to λ^2_{INI}, but the precise relationship is unclear—none of our concrete models are CBPV models.

Our two-level language λ^2_{INI} can also be seen as an application of a novel resource interpretation of linear logic developed by Azevedo de Amorim [7], which uses an applicative modality to guarantee that the linearity restriction is only valid for computations, not values. Our focus is on separation and effects: we show how different sum types for effectful computations can be naturally accommodated in this framework, we consider a more general class of categorical models, and we prove a soundness theorem ensuring separation for effectful computations.

Bunched type systems. Our focus on sharing and separation is similar to the motivation of another substructural logic, called the logic of bunched implicates (BI) [72]. Like our system, BI features two conjunctions modeling separation of resources, and sharing of resources. Like in λ_{INI}, these conjunctions in BI belong to the same language. Unlike our work, BI also features two implications, one for each conjunction. The leading application of BI is in separations logic for concurrent and heap-manipulating programs [73, 71], where pre- and post-conditions are drawn from BI.

Most applications of BI use a truth-functional, Kripke-style semantics [78]. By considering the proof-theoretic models of BI, O'Hearn [70] developed a bunched type system for a higher-order language. Its categorical semantics is given by a *doubly closed category*: a Cartesian closed category with a separate symmetric monoidal closed structure. While O'Hearn [70] showed different models of this language for reasoning about sharing and separation in heaps, few other concrete models are known. It is not clear how to incorporate effects into the bunched type system; in contrast, our models can reason about a wide class of monadic effects.

There are natural connections to both of our languages. Our language λ_{INI} resembles O'Hearn's system, with two differences. First, λ_{INI} only has a multiplicative arrow, not an additive arrow—as we described in Section 5.2, it is not clear how to support an additive arrow in λ_{INI} without breaking our primary soundness property. Second, contexts in λ_{INI} are flat lists, not tree-shaped bunches; it would be interesting to use bunched contexts to represent more complex dependency relations.

Our stratified language λ^2_{INI} is also similar to O'Hearn's system. Though our categorical model only has a single multiplicative arrow, in the I-layer, many—

but not all—of our concrete models also support an additive arrow, in the NI -layer. Furthermore, by assuming a single category, instead of two categories as in our approach, in BI it is possible to layer the connectives $\times$ and $\otimes$ to create intricate dependency structures. In contrast our two-layer language only allows to create dependencies of the form $\mathcal{M}(\tau \times \cdots \times \tau) \otimes \cdots \otimes \mathcal{M}(\tau \times \cdots \times \tau)$. At the same time, it is not clear how the two sum types in λ^2_{INI} would function in a bunched type system.

Probabilistic independence in higher-order languages. There are a few probabilistic functional languages with type systems that model probabilistic independence. Probably the most sophisticated example is due to Darais et al. [29], who propose a type system combining linearity, information-flow control, and probability regions for a probabilistic functional language. Darais et al. [29] show how to use their system to implement and verify security properties for implementations of oblivious RAM (ORAM). Our work aims to be a core calculus capturing independence, with a clean categorical model.

Lobo Vesga et al. [58] present a probabilistic functional language embedded in Haskell, aiming to verify accuracy properties of programs from differential privacy. Their system uses a taint-based analysis to establish independence, which is required to soundly apply concentration bounds, like the Chernoff bound. Unlike our work, Lobo Vesga et al. [58] do not formalize their independence property in a core calculus.

Probabilistic separation logics. A recent line of work develops separation logics for first-order, imperative probabilistic programs, using formulas from the logic of bunched implications to represent pre- and post-conditions. Systems can reason about probabilistic independence [11], but also refinements like con-

ditional independence [8], and negative association [9]. These systems leverage different Kripke-style models for the logical assertions; it is unclear how these ideas can be adapted to a type system or a higher-order language. There are also quantitative probabilistic separation logics [12, 13].

5.8 Conclusion and Future Directions

We have presented two linear, higher-order languages with types that can capture probabilistic independence, and other notions of separation in effectful programs. We see several natural directions for further investigation.

Other variants of independence. In some sense, probabilistic independence is a trivial version of dependence: it captures the case where there is no dependence whatsoever between two random quantities. Researchers in statistics and AI have considered other notions that model more refined dependency relations, such as conditional independence, positive association, and negative dependence (e.g., [33]). Some of these notions have been extended to other models besides probability; for instance, Pearl and Paz [74] develop a theory of *graphoids* to axiomatize properties of conditional independence. It would be interesting to see whether any of these notions can be captured in a type system.

Bunched type systems for independence. Our work bears many similarity to work on bunched logics; most notably, bunched logics feature an additive and a multiplicative conjunction. While bunched logics have found strong applications in Hoare-style logics, the only bunched type system we are aware of is due to O'Hearn [70]. This language features a single layer with two product types and also two function types, and the typing contexts are tree-shaped bunches,

rather than flat lists. Developing a probabilistic model for a language with a richer context structure would be an interesting avenue for future work.

Non-commutative effects. Our concrete models encompass many kinds of monadic effects, but we only support effects modeled by commutative monads. Many common effects are modeled by non-commutative monads, e.g., the global state monad. It may be possible to extend our language to handle non-commutative effects, but we would likely need to generalize our model and consider non-commutative logics.

Towards a general theory of separation for effects. We have seen how in the presence of effects, constructs like sums and products come in two flavors, which we have interpreted as sharing and separate. Notions of sharing and separation have long been studied in programming languages and logic, notably leading to separation logics. We believe that there should be a broader theory of separation (and sharing) for effectful programs, which still remains to be developed.

CHAPTER 6

CONCLUSION

This thesis has presented a new formalism for programming with probabilities in a way that unifies the Markov kernel and linear operator semantics of probabilistic programming. We see it as a starting point for a major extension to synthetic methods in probability theory.

It is important to understand what are the ramifications of this new paradigm. We already see some exciting directions to further pursue in the near future.

Linear Logic Semantics for Probability and Differentiability A central theme of this thesis is that many probabilistic semantics are based on "convex" models of linear logic. Besides, Ehrhard [35] has shown that linear logic has an extension where non-linear morphisms $!A \multimap B$ have a semantic notion of derivative. Ehrhard has synthesized this idea into differential linear logic, which is a variant of linear logic extended with a syntactic differentiation operations.

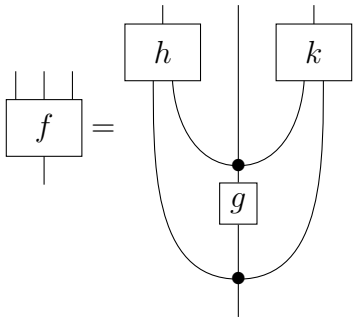
An exciting question is whether these models of "probabilistic" linear logic are actually an instance of a more general axiomatization. It seems that morphisms having a notion of convex sum play a unifying role in these models. They are also $\mathbf{CPO}$, which makes it possible to interpret term recursion.

Once this axiomatization is properly defined, would it possible to make a formal connection between "probabilistic" linear logic (PrLL) and differential linear logic (DiLL)? In existing models it seems like these categories come in pairs: $\mathbf{PCoh}$ is a model of PrLL while $\mathbf{Kothe}$ is a model of DiLL, the category $\mathbf{Coh}$ of coherence spaces should be a model of PrLL while the category $\mathbf{Finiteness}$ of finiteness spaces is a model of DiLL and, it seems that there is a

variant of $\mathbf{PBanLat_1}$, where we drop the Banach requirement, that is a model of DiLL. I conjecture that better understanding these models will provide some important insights into the semantics of linear logic abd probabilistic differentiable programming languages.

Reasoning About Conditional Independence We have seen that as it stands, it is possible to extend $\lambda_{\text{MK}}^{\text{LL}}$ to reason about a simplified version of conditional independence. In the Markov category literature, Fritz [41] has proposed a more general definition that encompasses other notions of conditional independence.

Definition 6.0.1. A morphism $f : X \to Y \otimes W \otimes Z$ is said to have the (extended) CI property if there are morphisms g, h and k such that



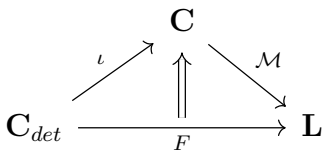
(6.1)

This new definition has some interesting composition principles and, in particular, interacts nicely with conditionals

Theorem 6.0.2 (Fritz [41])**.** *If $f : X \to Y \otimes (W_1 \otimes W_2) \otimes Z$ has the CI property and the Markov category has conditionals, then the morphism $f_{|W_1} : X \otimes W_1 \to Y \otimes W_2 \otimes Z$ has the CI property as well.*

It would be interesting to extend our type system so that it can also handle this more general form of CI. From a design point of view it seems like that typing judgment will require an "output" context for the variables that can be conditioned on. Finally, note that in order to reason about CI in $\lambda_{\text{MK}}^{\text{LL}}$ it is necessary

to add a new monoidal modality F that can copy contexts. From a probabilistic point of view this modality corresponds, roughly, to point-mass distributions. An important observation from probability theory is that atomless probability distributions, i.e. distributions that map every singleton set to 0, can be arbitrarily approximated by convex combination of point-mass distributions. We capture this by the following diagram:



I conjecture that it should be a left Kan extension, which is an abstraction of the approximation property described above.

Synthetic Reasoning for Linear Operators for Probability Markov categories have proved themselves extremely useful for giving general proofs and definitions for probability. It is natural to wonder if the same synthetic methods can be adapted to the $\lambda^{\mathrm{LL}}_{\mathrm{MK}}$ setting.

As an speculative example, the well-known Radon-Nikodym theorem is an extremely useful theorem in measure theory that has found applications in statistical inference. A well-known theorem from the Riesz space literature is the Freudenthal's Spectral Theorem, which can be seen as a generalization of the Radon-Nikodym theorem. Is it possible to adapt a variant of this theorem to the abstract categorical setting?

Recent work shows that it is possible to prove theorems from ergodic theory using the Markov category formalism [68], showing that this is not a far-fetched idea.

APPENDIX A

PROOFS

A.1 Theorem 3.2.1

Let $\Gamma, x : \underline{\tau_1} \vdash t : \underline{\tau}$ and $\Delta \vdash u : \underline{\tau_1}$ be well-typed terms, then $\Gamma, \Delta \vdash t\{u/x\} : \underline{\tau}$

Proof. The proof follows by structural induction on the typing derivation $\Gamma, x : \underline{\tau_1} \vdash t : \underline{\tau}$:

- Axiom: Since $t = x$ then $t\{u/x\} = u$ and $\underline{\tau_1} = \underline{\tau}$.
- Abstraction: By hypothesis, $\Gamma, x : \underline{\tau_1}, y : \underline{\tau_2} \vdash t : \underline{\tau_3}$. Since we can assume wlog that $x \neq y$ and that $y \notin \Delta$, $\lambda y.\, t\{u/x\} = \lambda y.\, t\{u/x\}$. Therefore we can show that $\Gamma, \Delta \vdash \lambda y.\, t\{u/x\} : \tau_2 \multimap \tau_3$ by applying the rule Abstraction and by the induction hypothesis.
- Application: $t_1\, t_2\{u/x\} = t_1\{u/x\}\, t_2\{u/x\}$. Since the language LL is linear, only one of t_1 or t_2 will have x as a free variable. By symmetry we can assume that t_1 has x as a free variable and we can prove $\Gamma, \Delta \vdash t_1\{u/x\}\, t_2 : \underline{\tau}$ by applying the rule Application and by the induction hypothesis.
- Sample: It is easy to prove that $(\mathsf{sample}\ t\ \mathsf{as}\ y\ \mathsf{in}\ M)\{u/x\} = \mathsf{sample}\ (t\{u/x\})\ \mathsf{as}\ y\ \mathsf{in}\ M$

□

A.2 Theorem 3.3.4

Let $x_1 : \tau_1, \cdots, x_n : \tau_n \vdash t : \tau$ and $\Gamma_i \vdash t_i : \tau_i$ be well-typed terms. $\left[\!\left[\Gamma_1, \cdots, \Gamma_n \vdash t\{\overrightarrow{t_i}/\overrightarrow{x_i}\} : \underline{\tau}\right]\!\right] = (\left[\!\left[\Gamma_1 \vdash t_1 : \underline{\tau_1}\right]\!\right] \otimes \cdots \otimes \left[\!\left[\Gamma_n \vdash t_n : \underline{\tau_n}\right]\!\right]); \left[\!\left[\Gamma_1, \cdots, \Gamma_n\right]\!\right] \vdash t : \underline{\tau}$.

Proof. The proof follows by induction on the typing derivation of t.

- Axiom: Since $t = x$ then $t\{t_0/x\} = t_0$ and $[\![t\{t_0/x\}]\!] = [\![t_0]\!] = [\![t_0]\!]\,;id = [\![t_0]\!]\,;[\![x]\!]$.
- Unit: Since $t = x$ then $t\{t_0/x\} = t_0$ and $[\![t\{t_0/x\}]\!] = [\![t_0]\!] = [\![t_0]\!]\,;id = [\![t_0]\!]\,;[\![x]\!]$.
- Tensor: We know that $t = t_1 \otimes t_2$. Furthermore, from linearity we know that each free variable appears either in t_1 or in t_2. Without loss of generality we can assume that $(t_1 \otimes t_2)\{u_1, \cdots, u_n/x_1, \cdots, x_n\} = (t_1\{u_1, \cdots, u_k/x_1, \cdots, x_k\}) \otimes (t_2\{u_{k+1}, , \cdots, u_n/x_{k+1}, \cdots, x_n\})$. We can conclude this case from the induction hypothesis and functoriality of $\otimes$.
- LetTensor: This case follows from the functoriality of $\otimes$ and the induction hypothesis.
- Abstraction: This case follows from unfolding the definitions, using the induction hypothesis and by naturality of cur.
- Application: Analogous to the Tensor case
- Sample: This case is analogous to the Tensor case.

□

A.3 Theorem 4.3.39

The proof follows from well-known results.

Lemma A.3.1 ([2]). *If V be a normed Riesz space then its norm dual V' is a Banach lattice.*

Lemma A.3.2 ([2]). *The band generated by V in V'' is V'^{σ}. Furthermore, V is order-dense in its generated band.*

Theorem A.3.3. *The forgetful functor $U : \mathbf{PBanLat_1} \to \mathbf{NRiesz}$ has a right adjoint.*

Proof. With the theorems above, we can prove this theorem analogously to theorem 4.3.37 □

A.4 Lemma 4.4.9

Let $C(V)$ be the space of order-continuous bounded functions $f : \mathcal{B}(V)^+ \to \mathbb{R}$ equipped with the pointwise partial order $f \leq g$ iff $\forall x\ f(x) \leq g(x)$ and the sup norm $\|f\| = \sup_{x \in \mathcal{B}(V)^+} |f(x)|$.

Lemma A.4.1. *The space $C(V)$ is a Riesz space.*

Proof. Let $f, g \in C(V)$ and observe that $f \vee g = \lambda x.\, \max(f(x), g(x))$ is bounded and order-continuous and since we are assuming the pointwise order, it can be shown to be the least upper bound of f and g. □

The *order topology* on a Riesz space X is defined as follows: $A \subseteq X$ is *closed* if it contains the limit of any convergent net included in A; that is, $x \in A$ whenever $\{x_\alpha\}_\alpha \subseteq A$ and $x_\alpha \to x$.

Let $f : X \to Y$ be a map between Riesz spaces, not necessarily linear. We say that f is *order-continuous* if for all convergent nets $x_\alpha \to x$ in X, $f(x_\alpha) \to f(x)$ in Y. We say that f is *topologically continuous* if f^{-1} preserves closed sets; that is, if A is closed in Y, then $f^{-1}(A)$ is closed in X.

Lemma A.4.2. *Let $f : X \to Y$ be any map between Riesz spaces, not necessarily linear. If f is order-continuous, then it is topologically continuous.*

Proof. Suppose f is order-continuous. Let A be closed in Y and let $x_\alpha \to x$ be a convergent net. If $x_\alpha \in f^{-1}(A)$ for all α, then $f(x_\alpha) \in A$ for all α, and since f is order-continuous, $f(x_\alpha) \to f(x)$. Since A is closed, $f(x) \in A$, thus

$x \in f^{-1}(A)$, therefore $f^{-1}(A)$ is closed. As A was arbitrary, f^{-1} is topologically continuous. □

An *M-space* is a Banach lattice whose norm satisfies $\|x \vee y\| = \|x\| \vee \|y\|$ for positive x, y. If the unit ball of an M-space contains a largest element, it is known as a *unit* of the space.

Theorem A.4.3. *The space $C(V)$ is an M-space.*

Proof. Let $C_\tau(V)$ be the vector space of bounded (topologically) continuous functions $f : \mathcal{B}(V)^+ \to \mathbb{R}$. In [81, Example 2, p. 103], it is shown that for any topological space X, the vector space of all bounded, real-valued continuous functions on X with the pointwise order is an M-space with unit $\lambda x.1$. In such spaces, the closed unit ball is the order interval $[-\lambda x.1, \lambda x.1]$, so

$$\|f\| \leq 1 \;\Leftrightarrow\; f \in [-\lambda x.1, \lambda x.1] \;\Leftrightarrow\; \sup_{x \in X} |f(x)| \leq 1,$$

therefore $\|f\| = \sup_{x \in X} |f(x)|$. Taking $X = \mathcal{B}(V)^+$, we have that $C_\tau(V)$ is an M-space with norm $\|f\| = \sup_{x \in \mathcal{B}(V)^+} |f(x)|$. By Lemma A.4.2, the space $C(V)$ is a subspace of $C_\tau(V)$, and $C(V)$ is closed in both order and norm, therefore is also an M-space. □

A.5 Theorem 4.4.14

L-spaces are extremelly well-behaved and so are its subspaces. Our proof strategy relies on the fact that L-spaces are always perfect and in showing that $!V$ is an L-space in its own right.

Lemma A.5.1 ([39])**.** *Every L-space is perfect.*

Therefore, by showing that $!V$ is an L-space it follows that it is a perfect Banach lattice. A canonical way of obtaining a Banach space out of a normed vector space is by taking its norm-closure. In general, however, since our spaces are also Riesz, it might be the case that the norm-closure is not a Riesz space. Luckily, for L-spaces, order-closed spaces are also norm-closed making them Banach.

Lemma A.5.2 ([39]). *Every order-closed subspace of an L-space is also norm-closed.*

Corollary A.5.3. *Every order-closed Riesz subspace of an L-space is an L-space.*

Finally, we can conclude our proof by observing that since $!V$ is defined as the order-closure of a Riesz space, by the corollary above, it is also an L-space, which by Lemma A.5.1 implies that it is a perfect Banach lattice; the span of the deltas is order-dense by transitivity of order-density.

A.6 Theorem 4.6.2

Let C be a directed complete lattice cone. In order to define functions over it we use the universal property of quotients: it suffices to define it over every pair (c_1, c_2) while guaranteeing that the function acts the same over every equivalence class.

For instance, the vector space structure can be simply defined componentwise. Let $(c_1, c_2), (c_3, c_4) \in C - C$ then we define

$$(c_1, c_2) + (c_3, c_4) = (c_1 + c_3, c_2 + c_4)$$

$$\alpha(c_1, c_2) = (\alpha c_1, \alpha c_2) \text{ for } \alpha \geq 0$$

$$\alpha(c_1, c_2) = (-\alpha c_2, -\alpha c_1) \text{ otherwise}$$

The lattice operations require a bit more ingenuity, and we first observe the equation $u \vee v = u + (v - u)^+$ which holds in every Riesz space, reducing the lowest upper bound operation to addition and the positive part. By doing some algebraic manipulations we get that if $(c_1, c_2), (c_3, c_4) \in C - C$ then we define $(c_1, c_2) \vee (c_3, c_4) = (c_1, c_2) - ((c_3, c_4) - (c_1, c_2))^+ = (c_1, c_2) + (c_3 + c_2 - (c_1 + c_4) \wedge (c_2 + c_3), 0) = (c_1 + c_3 + c_2 - (c_1 + c_4) \wedge (c_2 + c_3), c_2)$. The lattice equations such as commutativity and idempotency follow by unfolding the definitions and from C being a lattice.

Before defining a norm over $C - C$ we first need the following lemma

Lemma A.6.1. $(C - C)^+ \cong \{(c, 0) \,|\, c \in C\} \cong C$.

Proof. The mapping $\{(c, 0) \,|\, c \in C\} \to (C - C)^+$ is the injection through the equivalence class function and the mapping in the other direction can be constructed by observing that whenever $(c_1, c_2) \geq (0, 0)$ it can be shown that $c_1 \geq c_2$ and, therefore, $(c_1 - c_2, 0) = (c_1, c_2)$ and this decomposition is unique, since $(c, 0) = (d, 0)$ implies, by definition of $\sim$ that $c = d$. The second isomorphism is trivial. □

Given a norm over C it is possible to extend it to a norm over $C - C$. This follows from the property of normed Riesz spaces, where $\||v|\| = \|v\|$ which forces us to define $\|(c_1, c_2)\| = \||(c_1, c_2)|\|_C$. Note that since $|(c_1, c_2)|$ is a positive element of $C - C$, by the lemma above it can be mapped back to an element of C which, in turn, has a norm.

Therefore, we have shown that $C - C$ is a normed Riesz space. Since C has the directed completeness property it follows that $C - C$ has the weak Fatou property and, therefore, it is Banach and perfect.

BIBLIOGRAPHY

[1] Samson Abramsky and Achim Jung. Domain theory. 1994.

[2] Charalambos D Aliprantis and Owen Burkinshaw. *Positive operators*. Springer, 2006.

[3] Roberto M Amadio and Pierre-Louis Curien. *Domains and lambda-calculi*. Number 46. Cambridge University Press, 1998.

[4] Robert J Aumann. Borel structures for function spaces. *Illinois Journal of Mathematics*, 5(4):614–630, 1961.

[5] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. Probabilistic relational reasoning via metrics. In *ACM/IEEE Symposium on Logic in Computer Science (LICS), Vancouver, British Columbia*, pages 1–19. IEEE, 2019. doi: 10.1109/LICS.2019.8785715. URL https://doi.org/10.1109/LICS.2019.8785715.

[6] Pedro Azevedo de Amorim, Leon Witzman, and Dexter Kozen. Classical linear logic in perfect banach spaces. *Preprint*, 2022.

[7] Pedro H. Azevedo de Amorim. A higher-order language for markov kernels and linear operators. In *Foundations of Software Science and Computation Structures (FoSSaCS), Paris, France*, 2023.

[8] Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. A bunched logic for conditional independence. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE, 2021.

[9] Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. A separation logic for negative dependence. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–29, 2022.

[10] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *Principles of Programming Languages (POPL)*, 2014.

[11] Gilles Barthe, Justin Hsu, and Kevin Liao. A probabilistic separation logic. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–30, 2019.

[12] Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.*, 3 (POPL):34:1–34:29, 2019. doi: 10.1145/3290347. URL https://doi.org/10.1145/3290347.

[13] Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, and Thomas Noll. Foundations for entailment checking in quantitative separation logic. In Ilya Sergey, editor, *Programming Languages and Systems - 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 57–84. Springer, 2022. doi: 10.1007/978-3-030-99336-8_3. URL https://doi.org/10.1007/978-3-030-99336-8_3.

[14] Nick Benton and Philip Wadler. Linear logic, monads and the lambda calculus. In *Symposium on Logic in Computer Science (LICS)*, 1996.

[15] P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models (extended abstract). In Leszek Pacholski and Jerzy Tiuryn, editors, *International Workshop on Computer Science Logic (CSL), Kazimierz, Poland*,

volume 933 of *Lecture Notes in Computer Science*, pages 121–135. Springer, 1994. doi: 10.1007/BFb0022251. URL https://doi.org/10.1007/BFb0022251.

[16] Gérard Berry. Stable models of typed λ-calculi. In *Automata, Languages and Programming: Fifth Colloquium*, 1978.

[17] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, Kristian Støvring, Jacob Thamsborg, and Hongseok Yang. Step-indexed kripke models over recursive worlds. *ACM SIGPLAN Notices*, 46(1):119–132, 2011.

[18] Richard Blute, Thomas Ehrhard, and Christine Tasson. A convenient differential category. *Cahiers de topologie et géométrie différentielle catégoriques*, 53(3):211–232, 2012.

[19] Francis Borceux. *Handbook of Categorical Algebra: Volume 2, Categories and Structures*, volume 2. Cambridge University Press, 1994.

[20] G. E. P. Box and Mervin E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics*, 29(2):610 – 611, 1958. doi: 10.1214/aoms/1177706645. URL https://doi.org/10.1214/aoms/1177706645.

[21] Aleksandar Chakarov and Sriram Sankaranarayanan. Probabilistic program analysis with martingales. In *International Conference on Computer Aided Verification (CAV)*, 2013.

[22] Kenta Cho and Bart Jacobs. Disintegration and bayesian inversion via string diagrams. *Math. Struct. Comput. Sci.*, 29(7):938–971, 2019. doi: 10.1017/S0960129518000488. URL https://doi.org/10.1017/S0960129518000488.

[23] Florence Clerc, Vincent Danos, Fredrik Dahlqvist, and Ilias Garnier. Pointless learning. In *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, 2017.

[24] Roy L Crole. *Categories for types*. Cambridge University Press, 1993.

[25] Raphaëlle Crubillé. Probabilistic stable functions on discrete cones are power series. In *Logic in Computer Science (LICS)*, 2018.

[26] Fredrik Dahlqvist and Dexter Kozen. Semantics of higher-order probabilistic programs with conditioning. In *Principles of Programming Languages (POPL)*, 2019.

[27] Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. *ACM Trans. Program. Lang. Syst.*, 41(2):10:1–10:65, 2019. doi: 10.1145/3293605. URL https://doi.org/10.1145/3293605.

[28] Vincent Danos and Thomas Ehrhard. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Information and Computation*, 209(6):966–991, 2011.

[29] David Darais, Ian Sweet, Chang Liu, and Michael Hicks. A language for probabilistically oblivious computation. *Proceedings of the ACM on Programming Languages*, 4(POPL):1–31, 2019.

[30] Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. Probabilistic relational reasoning via metrics. In *Symposium on Logic in Computer Science (LICS)*, 2019.

[31] Pedro H Azevedo de Amorim and Justin Hsu. Separated and shared effects in higher-order languages. *arXiv preprint arXiv:2303.01616*, 2023.

[32] Derek Dreyer, Amal Ahmed, and Lars Birkedal. Logical step-indexed logical relations. *arXiv preprint arXiv:1103.0510*, 2011.

[33] Devdatt P. Dubhashi and Desh Ranjan. Balls and bins: A study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998.

[34] Thomas Ehrhard. On Köthe sequence spaces and linear logic. *Mathematical Structures in Computer Science*, 12(5):579–623, 2002.

[35] Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Mathematical Structures in Computer Science*, 28 (7):995–1060, 2018.

[36] Thomas Ehrhard. On the linear structure of cones. In *Logic in Computer Science (LICS)*, 2020.

[37] Thomas Ehrhard, Michele Pagani, and Christine Tasson. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. In *Principles of Programming Languages (POPL)*, 2017.

[38] Thomas Ehrhard, Michele Pagani, and Christine Tasson. Full abstraction for probabilistic PCF. *J. ACM*, 65(4):23:1–23:44, 2018. doi: 10.1145/3164540. URL https://doi.org/10.1145/3164540.

[39] David H Fremlin. *Measure theory*. Torres Fremlin, 2000.

[40] DH Fremlin. Abstract Köthe spaces IV. In *Mathematical Proceedings of the Cambridge Philosophical Society*, pages 45–52. Cambridge University Press, 1968.

[41] Tobias Fritz. A synthetic approach to markov kernels, conditional indepen-

dence and theorems on sufficient statistics. *Advances in Mathematics*, 370: 107239, 2020.

[42] Guillaume Geoffroy. Extensional denotational semantics of higher-order probabilistic programs, beyond the discrete case (unpublished). 2021.

[43] Jean-Yves Girard. Coherent banach spaces: a continuous denotational semantics. *Theoretical Computer Science*, 227(1-2):275–297, 1999.

[44] Jean-Yves Girard. Between logic and quantic: a tract. *Linear logic in computer science*, 316:346–381, 2004.

[45] Jean Goubault-Larrecq. A probabilistic and non-deterministic call-by-push-value language. In *Logic in Computer Science (LICS)*, 2019.

[46] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *Logic in Computer Science (LICS)*, 2017.

[47] Andrew K Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proceedings of the ACM on Programming Languages*, 6 (POPL):1–27, 2022.

[48] Mingzhang Huang, Hongfei Fu, Krishnendu Chatterjee, and Amir Kafshdar Goharshady. Modular verification for almost-sure termination of probabilistic programs. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2019.

[49] Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*,

49(1), apr 2016. ISSN 0360-0300. doi: 10.1145/2873052. URL https://doi.org/10.1145/2873052.

[50] Martin Hyland and Andrea Schalk. Glueing and orthogonality for models of linear logic. *Theoretical computer science*, 294(1-2):183–231, 2003.

[51] Bart Jacobs. *Categorical logic and type theory*. Elsevier, 1999.

[52] Peter T Johnstone, Stephen Lack, and Paweł Sobociński. Quasitoposes, quasiadhesive categories and artin glueing. In *International Conference on Algebra and Coalgebra in Computer Science*, pages 312–326. Springer, 2007.

[53] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[54] Paul C Kainen. Weak adjoint functors. *Mathematische Zeitschrift*, 122:1–9, 1971.

[55] Marie Kerjean and Christine Tasson. Mackey-complete spaces and power series–a topological model of differential linear logic. *Mathematical Structures in Computer Science*, 28(4):472–507, 2018.

[56] Dexter Kozen. Semantics of probabilistic programs. In *Symposium on Foundations of Computer Science (SFCS)*, 1979.

[57] Paul Blain Levy. *Call-by-push-value*. PhD thesis, 2001.

[58] Elisabet Lobo Vesga, Alejandro Russo, and Marco Gaboardi. A programming language for data privacy with accuracy estimations. *ACM Trans.*

Program. Lang. Syst., 43(2):6:1–6:42, 2021. doi: 10.1145/3452096. URL https://doi.org/10.1145/3452096.

[59] WAJ Luxemberg and AC Zaanen. Notes on Banach function spaces VI-XIII. *Proceedings of the Koninklijke Nederlandse Akademie van Wetenschappen, Series A*, 66:251–263, 1963.

[60] John Maraist, Martin Odersky, David N Turner, and Philip Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Theoretical Computer Science*, 1999.

[61] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of functional programming*, 18(1):1–13, 2008.

[62] Annabelle McIver, Carroll Morgan, Benjamin Lucien Kaminski, and Joost-Pieter Katoen. A new proof rule for almost-sure termination. *Proceedings of the ACM on Programming Languages*, (POPL), 2017.

[63] Paul-André Mellies. Categorical semantics of linear logic. *Panoramas et syntheses*, 27:15–215, 2009.

[64] Kyung Chan Min. An exponential law for regular ordered banach spaces. *Cahiers de topologie et géométrie différentielle catégoriques*, 1983.

[65] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93 (1):55–92, 1991. doi: 10.1016/0890-5401(91)90052-4. URL https://doi.org/10.1016/0890-5401(91)90052-4.

[66] Fabrizio Montesi. *Choreographic Programming*. PhD thesis, Denmark, 2014.

[67] Sean Moss and Paolo Perrone. Probability monads with submonads of

deterministic states. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 1–13, 2022.

[68] Sean Moss and Paolo Perrone. A category-theoretic proof of the ergodic decomposition theorem. *arXiv preprint arXiv:2207.07353*, 2022.

[69] Rachit Nigam, Sachille Atapattu, Samuel Thomas, Zhijing Li, Theodore Bauer, Yuwei Ye, Apurva Koti, Adrian Sampson, and Zhiru Zhang. Predictable accelerator design with time-sensitive affine types. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 393–407, 2020.

[70] Peter W. O'Hearn. On bunched typing. *J. Funct. Program.*, 13(4):747–796, 2003. doi: 10.1017/S0956796802004495. URL https://doi.org/10.1017/S0956796802004495.

[71] Peter W. O'Hearn. Separation logic and concurrent resource management. In Greg Morrisett and Mooly Sagiv, editors, *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, page 1. ACM, 2007. doi: 10.1145/1296907.1296908. URL https://doi.org/10.1145/1296907.1296908.

[72] Peter W. O'Hearn and David J. Pym. The logic of bunched implications. *Bull. Symb. Log.*, 5(2):215–244, 1999. doi: 10.2307/421090. URL https://doi.org/10.2307/421090.

[73] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Pro-*

ceedings, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001. doi: 10.1007/3-540-44802-0_1. URL https://doi.org/10.1007/3-540-44802-0_1.

[74] Judea Pearl and Azaria Paz. Graphoids: Graph-based logic for reasoning about relevance relations or when would x tell you more about y if you already know z? In Benedict du Boulay, David C. Hogg, and Luc Steels, editors, *European Conference on Artificial Intelligence (ECAI), Brighton, UK*, pages 357–363. North-Holland, 1986.

[75] Gordon D. Plotkin. Lcf considered as a programming language. *Theoretical computer science*, 5(3):223–255, 1977.

[76] Gordon D Plotkin. *A structural approach to operational semantics*. Aarhus university, 1981.

[77] François Pottier and Vincent Simonet. Information flow inference for ml. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 319–330, 2002.

[78] David J. Pym, Peter W. O'Hearn, and Hongseok Yang. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.*, 315(1):257–305, 2004. doi: 10.1016/j.tcs.2003.11.020. URL https://doi.org/10.1016/j.tcs.2003.11.020.

[79] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In Paul Hudak and Stephanie Weirich, editors, *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, pages 157–168. ACM, 2010. doi: 10.

1145/1863543.1863568. URL https://doi.org/10.1145/1863543.1863568.

[80] Nasser Saheb-Djahromi. Probabilistic lcf. In *Mathematical Foundations of Computer Science 1978: Proceedings, 7th Symposium Zakopane, Poland, September 4–8, 1978*, 1978.

[81] H. H. Schaefer. *Banach Lattices and Positive Operators*. Springer, 1970.

[82] A Scibior, O Kammar, M Vakar, S Staton, H Yang, Y Cai, K Ostermann, SK Moss, C Heunen, and Z Ghahramani. Denotational validation of higher-order bayesian inference. *Proceedings of the ACM on Programming Languages*, 2018.

[83] Dana Scott. *Outline of a mathematical theory of computation*. Oxford University Computing Laboratory, Programming Research Group Oxford, 1970.

[84] Gavin J Seal. Tensors, monads and actions. *arXiv preprint arXiv:1205.0101*, 2012.

[85] Alex K Simpson. Recursive types in kleisli categories. *Unpublished manuscript, University of Edinburgh*, 1992.

[86] Sergey Slavnov. Linear logic in normed cones: probabilistic coherence spaces and beyond. *Mathematical Structures in Computer Science*, 31(5):495–534, 2021.

[87] Ian Stark. Categorical models for local names. *Lisp and Symbolic Computation*, 9(1):77–107, 1996.

[88] Dario Stein. Structural foundations for probabilistic programming languages. *University of Oxford*, 2021.

[89] Christine Tasson and Thomas Ehrhard. Probabilistic call by push value. *Logical Methods in Computer Science*, 15, 2019.

[90] Christine Tasson and Thomas Ehrhard. Probabilistic call by push value. *Logical Methods in Computer Science*, 2019.

[91] Martinus Bernardus Jozephus Gerhardus van Haandel. *Completions in Riesz space theory*. PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[92] Andrew K Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[93] Adriaan C Zaanen. *Introduction to operator theory in Riesz spaces*. Springer, 2012.