Separated and Shared Effects in Higher-Order Languages

ANONYMOUS AUTHOR(S)

Effectful programs interact in ways that go beyond simple input-output, making compositional reasoning challenging. Existing work has shown that when such programs are "separate", i.e., when programs do not interfere with each other, it can be easier to reason about them. While reasoning about separated resources has been well-studied, there has been little work on reasoning about separated effects, especially for functional, higher-order programming languages.

We propose two higher-order languages that can reason about sharing and separation for commutative effects. Our first language λ_{INI} has a bunched type system and probabilistic semantics, where the two product types capture independent and possibly-dependent distributions. Our second language λ^2_{INI} is a two-level, stratified language, inspired by Benton's linear-non-linear (LNL) calculus. We motivate this language with a probabilistic model, but we also provide a general categorical semantics and exhibit a range of concrete models beyond probabilistic programming. We prove soundness theorems for all of our languages; our general soundness theorem for our categorical models of λ^2_{INI} uses a categorical gluing construction.

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*.

Additional Key Words and Phrases: Probabilistic Programming, Denotational Semantics, Effects, Higher-Order Languages

ACM Reference Format:
Anonymous Author(s). 2018. Separated and Shared Effects in Higher-Order Languages. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 34 pages.

1 INTRODUCTION

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* [O'Hearn et al. 2001; Pym et al. 2004]. 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

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

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 [Barthe et al. 2019]. Prior work has developed program logics that can about independence in the context of a first-order, imperative language [Barthe et al. 2019]. 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 *commutative* effects in a variety of contexts. The closest work in this area is the bunched calculus [O'Hearn 2003], the Curry-Howard correspondent of the logic of Bunched Implications [O'Hearn and Pym 1999]. While O'Hearn [2003] 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 monadic effects.

Throughout this work we will use probabilistic effects as our guiding example. We start by using a resource interpretation of probabilistic samples to establish independence: if two computations use disjoint resources (i.e., probabilistic samples), then they produce independent random quantities. Our perspective yields two substructural, 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 a variation of the bunched calculus of O'Hearn [2003], i.e. it has two distinct product and arrow 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, and the soundness theorem for the probabilistic model is intricate and difficult to generalize to other effects. 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 [2023], 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 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 [Azevedo de Amorim and Kozen 2022; Danos and Ehrhard 2011; Ehrhard et al. 2017]. We show that pairing these models with categories 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 [Montesi 2014].
- **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 [O'Hearn and Pym 1999; O'Hearn et al. 2001]. 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 [O'Hearn 2003]. 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 higher-order programs with commutative effects.

Outline. After reviewing mathematical preliminaries (§2), we present our main contributions:

- First, we define a bunched, higher-order probabilistic λ-calculus called λ_{INI}, with types that can capture probabilistic independence and dependence. We give a denotational semantics to our language and prove that $\otimes$ captures probabilistic independence (§3).
- 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} (§4).
- 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 [Benton 1994] and of the calculus proposed by Azevedo de Amorim [2023] (§5.1).
- 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 (§5.2).
- 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 (§6).

We survey related work in (§7), and conclude in (§8).

2 BACKGROUND

2.1 Monads and their algebras

We will assume knowledge of basic concepts from category theory, including functors, products, coproducts, Cartesian closed categories, and symmetric monoidal closed categories (SMCC). The interested reader can consult Leinster [2014]; Mac Lane [2013] for good introductions to the subject.

Monads. Following seminal work by Moggi [1991], 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, $\mu_A : T^2 A \to TA$ 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.*

2.2 Probability Theory

We will use probabilistic programs and effects to illustrate our higher-order languages.

Definition 2.1. A distribution over a set X is a function $\mu : X \to [0, 1]$ such that $\sum_{x \in X} \mu(x) = 1$.

Joint distributions are distributions over sets $X \times Y$. Given a joint distribution μ over $X \times Y$, its marginal distribution over X is defined as $\mu_X(x) = \sum_{y \in Y} \mu(x, y)$ with and the second marginal μ_Y being similarly defined. Furthermore, given a distribution μ_1 over X and a distribution μ_2 over Y, we define $\mu_1 \otimes \mu_2(x, y) = \mu_1(x)\mu_2(y)$

Definition 2.2. A distribution μ over $X \times Y$ is probabilistically *independent* if it is a product of its marginals μ_X and μ_Y, i.e., $\mu(x, y) = \mu_X(x) \cdot \mu_Y(y)$, $x \in X$ and $y \in Y$.

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 $\sum_{x \in supp(\mu)} \mu(x) = 1$ [Fritz 2020]. The unit of the monad is given by $\delta(a, b) = 1$ iff $a = b$ and 0 otherwise, while the bind is defined as $\mathsf{bind}(f)(\mu) = \sum_{x \in X} f(x)\mu(x)$.

3 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.

3.1 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 $\vdash \mathsf{coin} : \mathbb{B}$ 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

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

Variables	x, y, z		
Context Shift Variables	r, s		
Types	τ	::=	$\mathbb{B} \mid \tau \times \tau \mid \tau \otimes \tau \mid \tau \multimap \tau \mid \tau \to \tau$
Expressions	t, u	::=	$x \mid b \in \mathbb{B} \mid \mathsf{coin} \mid (t, u) \mid \pi_i\, t \mid t \otimes u \mid \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u$
		$\mid$	$\lambda x.\, t \mid t\ u \mid \lambda_s x.\, t \mid t \mathbin{@} u \mid r[t] \mid \rho r.t$
Intuitionistic Contexts	Γ	::=	$\cdot_I \mid x : \tau \mid \Gamma, \Gamma \mid r[\Delta]$
Separated Context	Δ	::=	$\cdot_S \mid x : \tau \mid \Delta; \Delta \mid r[\Gamma]$

Fig. 1. Types and Terms: λ_{INI}

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 a bunched type system that can reason about probabilistic independence.

3.2 Introducing the Language λ_{INI}

The language λ_{INI} can be seen as an effectful version of the $\alpha\lambda$-calculus [O'Hearn 2003], a calculus based on the proof theory of the logic of bunched implications. BI was developed for reasoning about sharing and separation of resources like pointers to a heap memory [O'Hearn et al. 2001], or permissions to enter some critical section in concurrent code [O'Hearn 2007]. A distinct feature of the $\alpha\lambda$-calculus is that contexts are trees (so-called *bunches*) rather than lists [O'Hearn 2003].

Syntax. Figure 1 presents the syntax of types and terms. Along with base types ($\mathbb{B}$), 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 $\multimap$ and an intuitionistic one $\to$. 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-abstractions and applications, their main difference being that $\multimap$ cannot share the context while $\to$ can. Our examples will use the standard syntactic sugar $\mathsf{let}\ x = t\ \mathsf{in}\ u \triangleq (\lambda x.\, u)\ t$, where we use the linear expressions. The most unusual aspect of this calculus is the mutually recursive grammar for contexts, which was first developed by [Krishnaswami 2011] with the goal of making the structural rules in $\alpha\lambda$-calculus admissible. In order to recover the full expressivity of the $\alpha\lambda$-calculus you need the context modalities $r[\Gamma]$ and $r[\Delta]$, where r ranges over a set of symbols, and the introduction/elimination programs $r[t]$ and $\rho r.\, t$, respectively.

Type system. Figure 2 shows the typing rules for λ_{INI}; the rules are standard from bunched logic. There are two variable rules and both are *affine*: in separated contexts Δ variables may be dropped but not freely duplicated, while in shared contexts Γ variables may be dropped and duplicated. For the sharing product $\times$, the introduction rule $\times$ Intro shares the context across the premises: both 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 follow the same pattern as the products, where one is multiplicative ($\multimap$) and the other is additive ($\to$). Another key difference between them is that they extend each context differently. The multiplicative abstraction extends the (separated) context using the separated

$$\textsc{Const}\ \frac{}{\cdot \vdash b : \mathbb{B}} \qquad \textsc{Coin}\ \frac{}{\cdot \vdash \mathsf{coin} : \mathbb{B}} \qquad \textsc{Var}_I\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Var}_S\ \frac{}{\Delta; x : \tau \vdash x : \tau}$$

$$\times\ \textsc{Intro}\ \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : \tau_1 \times \tau_2} \qquad \times\ \textsc{Elim}_i\ \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i\, t : \tau_i}$$

$$\otimes\ \textsc{Intro}\ \frac{\Delta_1 \vdash t_1 : \tau \qquad \Delta_2 \vdash t_2 : \tau_2}{\Delta_1; \Delta_2 \vdash t_1 \otimes t_2 : \tau_1 \otimes \tau_2} \qquad \otimes\ \textsc{Elim}\ \frac{\Delta_1 \vdash t : \tau_1 \otimes \tau_2 \qquad \Delta_2, x : \tau_1, y : \tau_2 \vdash u : \tau}{\Delta_1; \Delta_2 \vdash \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u : \tau}$$

$$\textsc{Abstraction}\ \frac{\Delta; x : \tau_1 \vdash t : \tau_2}{\Delta \vdash \lambda x.\, t : \tau_1 \multimap \tau_2} \qquad \textsc{Application}\ \frac{\Delta_1 \vdash t : \tau_1 \multimap \tau_2 \qquad \Delta_2 \vdash u : \tau_1}{\Delta_1; \Delta_2 \vdash t\ u : \tau_2}$$

$$\textsc{Shared Abstraction}\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda_s x.\, t : \tau_1 \to \tau_2} \qquad \textsc{Shared Application}\ \frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash t\ @\ u : \tau_2}$$

$$\textsc{Shr. Region Elim}\ \frac{r[\Gamma] \vdash t : \tau}{\Gamma \vdash \rho r.t : \tau} \qquad \textsc{Sep. Region Elim}\ \frac{r[\Delta] \vdash t : \tau}{\Delta \vdash \rho r.t : \tau} \qquad \textsc{Shr. Region Intro}\ \frac{\Gamma \vdash t : \tau}{r[\Gamma] \vdash r[t] : \tau} \qquad \textsc{Sep. Region Intro}\ \frac{\Delta \vdash t : \tau}{\Gamma, r[\Delta] \vdash r[t] : \tau}$$

Fig. 2. Typing Rules: λ_{INI}

extension (;) while the additive abstraction extends the (shared) context with the shared extension (,). Note that there are two distinct empty contexts, $\cdot_I$ is the empty intuitionistic context while $\cdot_S$ is the empty separated context.

The most unusual rules are the context labeling ones. Their purpose is to guarantee that shared contexts can only be split when producing shared types, and similar to separated contexts. For example, note that the $\otimes$ introduction rule can only be applied when the context is separated, meaning that it cannot, for instance, be used to split the shared context $x : A, y : B$. These rules come in pairs and they provide a way of creating a new modal context with the introduction rules (Sep/Shr Region Intro) and opening a modal context with the elimination rule (Sep/Shr Region Elim).

3.3 Denotational Semantics

We can give a semantics to this language using the category **Set** and the finite probability monad D. From left to right and top to bottom, Figure 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.

Well-typed terms are interpreted as Kleisli arrows. We briefly walk through the term semantics, which is essentially the same as the Kleisli semantics proposed by Moggi [1991]. Variables are

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

$$
\begin{aligned}
[\![\cdot_I]\!]_I &= [\![\cdot_S]\!]_S = 1 \\
[\![x : \tau]\!]_I &= [\![x : \tau]\!]_S = [\![\tau]\!] \\
[\![\Gamma_1, \Gamma_2]\!]_I &= [\![\Gamma_1]\!]_I \times [\![\Gamma_2]\!]_I \\
[\![r[\Delta]]\!]_I &= [\![\Delta]\!]_S \\
[\![\Delta_1; \Delta_2]\!]_S &= [\![\Delta_1]\!]_S \times [\![\Delta_2]\!]_S \\
[\![r[\Gamma]]\!]_S &= [\![\Gamma]\!]_I
\end{aligned}
$$

$$
\begin{aligned}
[\![\Gamma \vdash t : \tau]\!] &: [\![\Gamma]\!]_I \to D\,[\![\tau]\!] \\
[\![\Delta \vdash t : \tau]\!] &: [\![\Delta]\!]_S \to D\,[\![\tau]\!]
\end{aligned}
$$

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

Fig. 3. Denotational Semantics: λ_{INI}

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 denote 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.

The modal context rules are, semantically, not interesting. Their purpose is to guarantee that shared and separated contexts are used and appended appropriately, which plays no role at the semantic level.

Example 3.1 (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 3.2 (Independent pairs are correlated pairs)**.** We now illustrate show to use the modal syntax by writing a program showing that independent distributions are also possibly-dependent distributions in λ_{INI}: $\cdot \vdash \lambda z.\, \mathsf{let}\ x \otimes y = z\ \mathsf{in}\ \rho r.\, (r[x], r[y]) : \tau_1 \otimes \tau_2 \multimap \tau_1 \times \tau_2$.

3.4 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:

$$\begin{aligned}
\mathcal{R}_{\mathbb{B}} &= D(\mathbb{B}) \\
\mathcal{R}_{\tau_1 \otimes \tau_2} &= \{\mu_1 \otimes \mu_2 \in D([\![\tau_1]\!] \times [\![\tau_2]\!]) \mid \mu_i \in \mathcal{R}_{\tau_i}\} \\
\mathcal{R}_{\tau_1 \times \tau_2} &= \{\mu \in D([\![\tau_1]\!] \times [\![\tau_2]\!]) \mid \pi_i(\mu) \in \mathcal{R}_{\tau_i} \text{ for } i \in \{1,2\}\} \\
\mathcal{R}_{\tau_1 \multimap \tau_2} &= \{\mu \in D([\![\tau_1]\!] \to D([\![\tau_2]\!])) \mid \forall \mu' \in \mathcal{R}_{\tau_1}, x \leftarrow \mu'; f \leftarrow \mu; f(x) \in R_{\tau_2}\} \\
\mathcal{R}_{\tau_1 \to \tau_2} &= \{\mu \in D([\![\tau_1]\!] \to D([\![\tau_2]\!])) \mid \forall \mu' \in D(\tau_1 \times (\tau_1 \to D(\tau_2))) \\
&\qquad \mu'_1 \in \mathcal{R}_{\tau_1} \wedge \mu'_2 = \mu \Rightarrow (x, h) \leftarrow \mu'; h(x) \in R_{\tau_2}\}.
\end{aligned}$$

Logical relations for contexts Γ and Δ can be defined as:

$$\begin{aligned}
\mathcal{R}_{\cdot} &= 1 & \mathcal{R}_{\cdot} &= 1 \\
\mathcal{R}_{x:\tau} &= \mathcal{R}_\tau & \mathcal{R}_{x:\tau} &= \mathcal{R}_\tau \\
\mathcal{R}_{\Gamma_1,\Gamma_2} &= \{\mu \in D([\![\Gamma_1]\!] \times [\![\Gamma_2]\!]) \mid \pi_i(\mu) \in \mathcal{R}_{\Gamma_i}\} & \mathcal{R}_{\Delta_1;\Delta_2} &= \{\mu_1 \otimes \mu_2 \in D([\![\Delta_1]\!] \times [\![\Delta_2]\!]) \mid \mu_i \in \mathcal{R}_{\Delta_i}\} \\
\mathcal{R}_{r[\Delta]} &= \mathcal{R}_\Delta & \mathcal{R}_{r[\Gamma]} &= \mathcal{R}_\Gamma
\end{aligned}$$

Theorem 3.3. *If $\Gamma \vdash t : \tau$ and $\mu \in \mathcal{R}_\Gamma$ then $(x \leftarrow \mu; [\![t]\!]\,(x)) \in \mathcal{R}_\tau$.*

Proof. The proof follows by induction on the derivation of $\Gamma \vdash t : \tau$. Most cases follow by simply using the induction hypothesis. The exception is the Shared Abstraction case. While the logical relations for the shared arrow uses joint distributions over the input space and the function space, the induction hypothesis is only valid for joint distributions over the extended context.

We solve this by using disintegration, which is a construction that given $\mu \in D(A \times B)$ and $\nu \in D(B)$, outputs a function $f : B \to D(A)$ such that $\mu = b \leftarrow \nu; a \leftarrow f(b); \text{return } (a, b)$. The full proof can be found in Appendix A.

□

Corollary 3.4. *If $\cdot \vdash t : \tau_1 \otimes \tau_2$ then $[\![t]\!]\,(*)$ is an independent probability distribution over $[\![\tau_1]\!] \times [\![\tau_2]\!]$.*

Note that even though the soundness property expressed by the corollary above only concerns closed programs of type $\tau_1 \otimes \tau_2$, the full soundness theorem is much more general than that. Indeed, the soundness theorem implies properties about the semantics of every program $\Gamma \vdash t : \tau$. For instance, if $\Gamma \vdash t : \mathbb{B}$, then $[\![t]\!]$ can be any Kleisli arrow. If, however, $\Gamma \vdash t : \mathbb{B} \otimes \mathbb{B}$, then $[\![t]\!]$ is a Kleisli arrow that maps any joint distribution over Γ in $\mathcal{R}_\Gamma$ to an independent distribution over $\mathbb{B} \times \mathbb{B}$.

Constants. An indirect consequence of this theorem is that it provides a blueprint of when it is sound to add a constant or base type to the language. Given a base type σ that has an interpretation in the Kleisli semantics, you can define $\mathcal{R}_\sigma = D([\![\sigma]\!])$. Furthermore, If you want to soundly add an operation $\Gamma \vdash \text{op} : \tau$ you must pick a semantics $[\![\text{op}]\!]$ such that for every distribution $\mu \in \mathcal{R}_\Gamma$, $\gamma \leftarrow \mu; [\![\text{op}]\!]\,(\gamma) \in \mathcal{R}_\tau$. In particular, it is sound to add any operation to the shared fragment of the language, i.e. the intuitionistic sublanguage of λ_{INI}, while one must be careful when adding operations to the substructural fragment of λ_{INI}, as to not break the logical relation invariant.

3.5 Shortcomings

We finish this section by noting that even though λ_{INI} is the first higher-order calculus that can reason about independence properties of programs, it still has a couple of shortcomings. While the intuitionistic fragment can be easily made complete with respect to the Kleisli semantics, if-statements and sum types are still problematic. Consider the simple program:

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

Operationally, this probabilistic program flips a fair coin and outputs a pair with two copies of the result, $\text{tt} \otimes \text{tt}$ or $\text{ff} \otimes \text{ff}$. Since tt and ff are constants they do not share any variables, so both branches can be given type $\mathbb{B} \otimes \mathbb{B}$ and a standard case analysis rule would assign the whole program $\mathbb{B} \otimes \mathbb{B}$. However, this extension would break soundness (theorem 3.3): the pair is not probabilistically independent because its components are always equal to each other.

The second problem with λ_{INI} is that the proof of Theorem 3.3 does not seem to scale beyond probabilistic effects, since the shared abstraction inductive case relies on disintegration. Furthermore, it is unclear how to scale this proof to accommodate even continuous probability distributions, where the existence of disintegration is much less straight-forward than in the discrete case [Dahlqvist et al. 2018].

4 A TWO-LEVEL LANGUAGE FOR INDEPENDENCE

The substructural 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, as explained in the previous section. In this section we 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}.

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

The stratified design of λ^2_{INI} is 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 $\mathbb{B} \otimes \mathbb{B}$. 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 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 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 [Azevedo de Amorim 2023]: 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	::=	$\mathbb{B} \mid \tau \times \tau \mid \tau + \tau$
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	::=	$x \mid b \in \mathbb{B} \mid (M, N) \mid \pi_i\, M \mid \mathsf{in_i}\ \mathsf{t}$
		$\mid$	$\mathsf{case}\ t\ \mathsf{of}\ (\mid \mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) \mid \mathsf{let}\ x = M\ \mathsf{in}\ N$
I-expressions	t, u	::=	$x \mid t \otimes u \mid \mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u \mid \mathsf{in_i}\ \mathsf{t}$
		$\mid$	$\mathsf{case}\ t\ \mathsf{of}\ (\mid \mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) \mid \lambda x.\, t \mid t\ u \mid \mathsf{sample}\ \overline{t}\ \mathsf{as}\ \overline{x}\ \mathsf{in}\ M$
NI-contexts	Γ	::=	$x_1 : \tau_1, \ldots, x_n : \tau_n$
I-contexts	Γ	::=	$x_1 : \underline{\tau}_1, \ldots, x_n : \underline{\tau}_n$

Fig. 4. Types and Terms: λ^2_{INI}

Typing rules. The typing rules of λ^2_{INI} are presented in Figure 5. 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, differently from λ_{INI}, which has both multiplicative and additive rules.

The second group of typing rules assigns types to the independent (I) layer. These rules are the standard rules for multiplicative linear logic , and are almost identical to the linear fragment of λ_{INI}. Unlike before, however, the rules treat variables linearly, 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, is 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, $\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ M$ 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 and we add a fair coin primitive $\cdot \vdash_{NI} \mathsf{coin} : \mathbb{B}$. 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 6. 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 **Set** 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 a D-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 [Simpson 1992]. 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. The Sample rule is interpreted using the joint probability operation $\otimes$ and the monad multiplication.

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 4.1. *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. □

4.2 Revisiting Sums

Let us revisit the problematic if-then-else program. The type system of λ^2_{INI} makes it impossible to produce an independent pair by pattern matching on values:

$$\text{dist} : \mathcal{M}(1+1) \nvdash_I \text{ if dist then } (\text{tt} \otimes \text{tt}) \text{ else } (\text{ff} \otimes \text{ff}) : \mathcal{M}\mathbb{B} \otimes \mathcal{M}\mathbb{B}$$

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:

$$\text{dist} : \mathcal{M}(1+1) \vdash_I \text{ sample dist as } x \text{ in } (\text{if } x \text{ then } (\text{tt}, \text{tt}) \text{ else } (\text{ff}, \text{ff})) : \mathcal{M}(\mathbb{B} \times \mathbb{B})$$

Constants. As it stands, λ^2_{INI} is not very expressive. Most languages based on core calculi usually guarantee a certain level of expressivity by adding base types and operations to the language. One of the basic examples are arithmetic expressions, as it is done for PCF. As such, in order to increase the expressivity of λ^2_{INI} we should add constants to the language.

Much like the λ_{INI} case, since we are interested in proving the soundness theorem, we should guarantee that the operations also validate it. For the semantics presented in Figure 6, adding new constants is straightforward from a semantic point of view, since $\otimes$ is denoted exactly by independent distributions, which means that any function between D-algebras can be soundly added to λ^2_{INI}. Furthermore, any D-algebra can be added as a new type of λ^2_{INI}.

Example: One-Time-Pad. We use this concrete semantics of λ^2_{INI} to extend it with a type constructor $\mathcal{M}_{\text{Unif}}(\tau)$ which is denoted by uniform distributions over τ, where τ is denoted by a finite set.

We can demonstrate this uniform constant through a simple program from cryptography. At a high level, the information-theoretic security of some cryptographic protocols can be formulated in terms of the interaction of uniform distributions and independence. One basic example is the one-time pad cryptographic scheme. This protocol receives as input a message, we can assume that it is a single bit m, samples a uniformly distributed bit k (key) and outputs the encrypted message $m \oplus k$, where $\oplus$ is the xor operation.

$$\frac{b \in \mathbb{B}}{\Gamma \vdash_{NI} b : \mathbb{B}}\ \textsc{Const} \qquad \frac{}{\Gamma, x : \tau \vdash_{NI} x : \tau}\ \textsc{Var} \qquad \frac{\Gamma \vdash_{NI} t : \tau_1 \qquad \Gamma, x : \tau_1 \vdash_{NI} u : \tau}{\Gamma \vdash_{NI} \mathsf{let}\ x = t\ \mathsf{in}\ u : \tau}\ \textsc{Let}$$

$$\frac{\Gamma \vdash_{NI} M : \tau_1 \qquad \Gamma \vdash_{NI} N : \tau_2}{\Gamma \vdash_{NI} (M, N) : \tau_1 \times \tau_2}\ \times\textsc{Intro} \qquad \frac{\Gamma \vdash_{NI} M : \tau_1 \times \tau_2}{\Gamma \vdash_{NI} \pi_i M : \tau_i}\ \times\textsc{Elim}_i$$

$$\frac{\Gamma \vdash_{NI} M : \tau_i}{\Gamma \vdash_{NI} \mathsf{in}_i\, M : \tau_1 + \tau_2}\ \oplus\textsc{Intro}_i \qquad \frac{\Gamma \vdash_{NI} M : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash_{NI} N_1 : \tau \qquad \Gamma, x : \tau_2 \vdash_{NI} N_2 : \tau}{\Gamma \vdash_{NI} \mathsf{case}\, M\ \mathsf{of}\ (\!|\, \mathsf{in}_1\, x \Rightarrow N_1 \mid \mathsf{in}_2\, y \Rightarrow N_2) : \tau}\ \oplus\textsc{Elim}$$

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

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

$$\frac{\Gamma \vdash_I t : \underline{\tau_i}}{\Gamma \vdash_I \mathsf{in}_i\, t : \underline{\tau_1} \oplus \underline{\tau_2}}\ \oplus\textsc{Intro}_i \qquad \frac{\Gamma_1 \vdash_I t : \underline{\tau_1} \oplus \underline{\tau_2} \qquad \Gamma_2, x : \underline{\tau_1} \vdash_I u_1 : \underline{\tau} \qquad \Gamma_2, y : \underline{\tau_2} \vdash_I u_2 : \underline{\tau}}{\Gamma_1, \Gamma_2 \vdash_I \mathsf{case}\, t\ \mathsf{of}\ (\!|\, \mathsf{in}_1\, x \Rightarrow u_1 \mid \mathsf{in}_2\, y \Rightarrow u_2) : \underline{\tau}}\ \oplus\textsc{Elim}$$

$$\frac{x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_{NI} M : \tau \qquad \Gamma_i \vdash_I t_i : \mathcal{M}(\tau_i) \qquad 0 < i \leq n}{\Gamma_1, \ldots, \Gamma_n \vdash_I \mathsf{sample}\ t_i\ \mathsf{as}\ x_i\ \mathsf{in}\ M : \mathcal{M}(\tau)}\ \textsc{Sample}$$

Fig. 5. Typing Rules: λ^2_{INI}

The security of this protocol rests on two ideas. First, the encryption scheme must output a uniformly distributed bit and it must be independent from its input. Without worrying about the security of the protocol, we can easily write it in λ^2_{INI} as $\mu : \mathcal{M}(2) \vdash_I \mathsf{sample}\ \mu\ \mathsf{as}\ x\ \mathsf{in}\ \mathsf{let}\ y = \mathsf{coin}\ \mathsf{in}\ x \oplus y : \mathcal{M}(2 \times 2)$.

Unfortunately, as it stands we cannot use λ^2_{INI}'s type system to prove that the protocol is secure. We rectify this by adding the operation $\cdot \vdash_I \mathsf{xor\_pair} : \mathcal{M}(2) \multimap \mathcal{M}(2) \otimes \mathcal{M}_{\mathsf{Unif}}(2)$ that corresponds to sampling from the input, xor-ing it with a fair coin and outputting the ciphered bit and the original bit. We can now write the protocol as the program $\mu : \mathcal{M}(2) \vdash_I : \mathsf{xor\_pair}\ \mu : \mathcal{M}(2) \otimes \mathcal{M}_{\mathsf{Unif}}(2)$, which has the right type.

4.3 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} without 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}.

$$
\begin{aligned}
(\!|\mathbb{B}|\!) &= \mathbb{B} & [\![\mathcal{M}\tau]\!] &= (D\,[\![\tau]\!]\,, \mu_{[\![\tau]\!]}) \\
(\!|\tau \times \tau|\!) &= (\!|\tau|\!) \times (\!|\tau|\!) & [\![\underline{\tau} \otimes \underline{\tau}]\!] &= [\![\underline{\tau}]\!] \times [\![\underline{\tau}]\!] \\
(\!|\tau + \tau|\!) &= (\!|\tau|\!) + (\!|\tau|\!) & [\![\underline{\tau} \oplus \underline{\tau}]\!] &= [\![\underline{\tau}]\!] + [\![\underline{\tau}]\!] \\
& & [\![\underline{\tau} \multimap \underline{\tau}]\!] &= [\![\underline{\tau}]\!] \to [\![\underline{\tau}]\!] \\
(\!|x_1 : \tau_1, \ldots, x_n : \tau_n|\!) &= (\!|\tau_1|\!) \times \cdots \times (\!|\tau_n|\!) & [\![x_1 : \underline{\tau}_1, \ldots, x_n : \underline{\tau}_n]\!] &= [\![\underline{\tau}_1]\!] \times \cdots \times [\![\underline{\tau}_n]\!] \\
(\!|\Gamma \vdash M : \tau|\!) &\in \mathbf{Set}_D((\!|\Gamma|\!), (\!|\tau|\!)) & [\![\Gamma \vdash t : \underline{\tau}]\!] &\in \widetilde{\mathbf{Set}^D}([\![\Gamma]\!]\,, [\![\underline{\tau}]\!])
\end{aligned}
$$

$$
\begin{aligned}
[\![x]\!]\,(\gamma, v_x) &= v_x \\
[\![t \otimes u]\!]\,(\gamma_1, \gamma_2) &= [\![t]\!]\,(\gamma_1) \times [\![u]\!]\,(\gamma_2) \\
[\![\mathsf{let}\ x \otimes y = t\ \mathsf{in}\ u]\!]\,(\gamma_1, \gamma_2) &= [\![u]\!]\,(\gamma_2, [\![t]\!]\,(\gamma_1)) \\
[\![\lambda x.\, t]\!]\,(\gamma)(x) &= [\![t]\!]\,(\gamma)(x) \\
[\![t\ u]\!]\,(\gamma_1, \gamma_2) &= [\![t]\!]\,(\gamma_1, [\![u]\!]\,(\gamma_2)) \\
[\![\mathsf{in}_i t]\!]\,(\gamma) &= in_i([\![t]\!]\,(\gamma)) \\
[\![\mathsf{case}\ t\ \mathsf{of}\ (|\mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2)]\!]\,(\gamma_1, \gamma_2) &= \begin{cases} [\![u_1]\!]\,(\gamma_2, v), & [\![t]\!]\,(\gamma_1) = in_1(v) \\ [\![u_2]\!]\,(\gamma_2, v), & [\![t]\!]\,(\gamma_1) = in_2(v) \end{cases} \\
[\![\mathsf{sample}\ t_i\ \mathsf{as}\ x_i\ \mathsf{in}\ N]\!] &= \mu \circ D(\!|N|\!) \circ ([\![t_1]\!] \otimes \cdots \otimes [\![t_n]\!])
\end{aligned}
$$

Fig. 6. Concrete Semantics: λ^2_{INI}

The types are translated as follows:

$$\mathcal{T}(\mathbb{B}) \triangleq \mathbb{B} \qquad \mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1 \otimes \tau_2) \triangleq \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2)$$

While contexts are interpreted as

$$\mathcal{T}(\cdot_I) = \mathcal{T}(\cdot_S) = \cdot \qquad \mathcal{T}(x : \tau_1) = \mathcal{T}(\tau_1) \qquad \mathcal{T}(\Gamma_1, \Gamma_2) = \mathcal{T}(\Gamma_1), \mathcal{T}(\Gamma_2)$$

$$\mathcal{T}(\Delta_1; \Delta_2) = \mathcal{T}(\Delta_1), \mathcal{T}(\Delta_2) \qquad \mathcal{T}(r[\Delta]) = \mathcal{T}(\Delta) \qquad \mathcal{T}(r[\Gamma]) = \mathcal{T}(\Gamma)$$

At the term-level, the translation is the identity function with the exception of the region operators, which are simply erased by the translation. We can prove by induction:

Theorem 4.2. *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 is sound and fully abstract:

Theorem 4.3. *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 by induction. □

It is also possible to translate the non-modal multiplicative ($\otimes$, $\multimap$) fragment of λ_{INI} into the I-layer of λ^2_{INI}, by translating the types as follows:

$$\mathcal{T}'(\mathbb{B}) \triangleq \mathcal{M}\mathbb{B} \qquad \mathcal{T}'(\tau_1 \otimes \tau_2) \triangleq \mathcal{T}'(\tau_1) \otimes \mathcal{T}'(\tau_2) \qquad \mathcal{T}'(\tau_1 \multimap \tau_2) \triangleq \mathcal{T}'(\tau_1) \multimap \mathcal{T}'(\tau_2)$$

The contexts are translated componentwise. Once again, the term translation is the identity function and the modalities are erased from terms and contexts.

Theorem 4.4. *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 $\Gamma \vdash t : \tau$. □

By direct inspection the translation is sound and fully abstract with respect with the denotational semantics of λ_{INI} and λ^2_{INI}.

Remark 4.5. 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 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.

5.1 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 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 [Cho and Jacobs 2019]. For $\otimes$, we want to restrict copying—the separating layer of our language has a linear type system—so $\otimes$ should be a monoidal product.

Finally, to model the type constructor $\mathcal{M}$, the usual 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 $\epsilon : I_D \to F(I_C)$ making certain diagrams commute [Borceux 1994]. Thus, we are led to our categorical model for λ^2_{INI}.

Definition 5.1. 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; $\mathbf{M}$ is a distributive CD category with coproducts, i.e., $A \otimes_M (B +_M C) \cong (A \otimes_M B) +_M (A \otimes_M C)$; and $\mathcal{M} : \mathbf{M} \to \mathbf{C}$ is lax monoidal.

We clarify the fact that weak coproducts are similar to regular coproducts except that the universal property only guarantees the existence of an arrow $A \oplus B \to C$ making the coproduct diagram commute, not uniqueness. Furthermore, contrary to **M**, distributivity in **C** holds automatically.

Lemma 5.2. *In every symmetric monoidal closed category with weak coproducts, the following isomorphism holds:* $A \otimes (B \oplus C) \cong (A \otimes B) \oplus (A \otimes C)$.

Proof. By assumption, the functor $A \otimes (-)$ is a left adjoint and, therefore, by Lemma 3.5 in [Kainen 1971], preserves weak coproducts and we can conclude. □

The denotational semantics is given in Figure 7 and most of the equational theory is presented in Figure 11, which can be found in Appendix B. Note that we omit the usual rules such as structural axioms and substitution.

Soundness. In categorical models, the soundness theorem of λ^2_{INI} can be stated as follows:

Theorem 5.3 (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 $\otimes$.

Establishing soundness requires additional categorical machinery, so we defer the proof to Section 6. We highlight the fact, however, that like the λ_{INI} case, we will prove a more general version of Theorem 5.3 which will imply properties for any well-typed λ^2_{INI} program and will also provide a list of requirements base types and operations must satisfy in order to be soundly added to the calculus. In the rest of the section, we will exhibit a range of concrete models for λ^2_{INI}.

5.2 Concrete models

To warm up, we present some basic probabilistic models λ^2_{INI}. While prior work has also investigated similar models [Azevedo de Amorim 2023], we adapt these models to λ^2_{INI} and explain how our soundness theorem ensures independence.

5.2.1 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 [Fritz 2020].

For the independent category, we take the probabilistic coherence space model of linear logic, a well-studied semantics for discrete probabilistic languages [Danos and Ehrhard 2011]. This model was originally used to explore the connections between probability theory and linear logic, and has recently been used to interpret recursive probabilistic programs and recursive types [Tasson and Ehrhard 2019]; it is also fully-abstract for probabilistic PCF [Ehrhard et al. 2018].

Definition 5.4 (Danos and Ehrhard [2011]). A *probabilistic coherence space (PCS)* is a pair $(|X|, \mathcal{P}(X))$ where $|X|$ is a countable set and $\mathcal{P}(X) \subseteq |X| \to \mathbb{R}^+$ satisfies:

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

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 $v \in \mathcal{P}(X)$, $f\,v \in \mathcal{P}(Y)$, where $(f\,v)_b = \sum_{a \in |X|} f_{(a,b)} v_a$. It is well-known that this category is a SMCC with coproducts; we will use the explicit definition of the monoidal product.

Var
$$\frac{}{\tau \times \Gamma \xrightarrow{id_\tau \times del_\Gamma} \tau}$$

Let
$$\frac{\Gamma \xrightarrow{M} \tau_1 \qquad \Gamma \times \tau_1 \xrightarrow{N} \tau_2}{\Gamma \xrightarrow{copy;(id \times M);N} \tau_2}$$

$\times$ Intro
$$\frac{\Gamma \xrightarrow{M} \tau_1 \qquad \Gamma \xrightarrow{N} \tau_2}{\Gamma \xrightarrow{copy;M \times N} \tau_1 \times \tau_2}$$

$\times$ Elim$_i$
$$\frac{\Gamma \xrightarrow{M} \tau_1 \times \tau_2}{\Gamma \xrightarrow{M;(id_{\tau_i} \times del)} \tau_i}$$

$+$ Intro$_i$
$$\frac{\Gamma \xrightarrow{M} \tau_1}{\Gamma \xrightarrow{M;in_i} \tau_1 + \tau_2}$$

$+$ Elim
$$\frac{\Gamma_1 \xrightarrow{N} \tau_1 + \tau_2 \qquad \Gamma_2 \times \tau_1 \xrightarrow{M_1} \tau \qquad \Gamma_2 \times \tau_2 \xrightarrow{M_2} \tau}{\Gamma_1, \Gamma_2 \xrightarrow{N \times id_{\Gamma_2}} (\tau_1 + \tau_2) \times \Gamma_2 \cong (\tau_1 \times \Gamma_2) + (\tau_2 \times \Gamma_2) \xrightarrow{[M_1, M_2]} \tau}$$

Var
$$\frac{}{\underline{\tau} \xrightarrow{id_\tau} \underline{\tau}}$$

Abstraction
$$\frac{\Gamma \otimes \underline{\tau_1} \xrightarrow{t} \underline{\tau_2}}{\Gamma \xrightarrow{\mathrm{cur}(t)} \underline{\tau_1} \multimap \underline{\tau_2}}$$

Application
$$\frac{\Gamma_1 \xrightarrow{t} \underline{\tau_1} \multimap \underline{\tau_2} \qquad \Gamma_2 \xrightarrow{u} \underline{\tau_1}}{\Gamma_1 \otimes \Gamma_2 \xrightarrow{(t \otimes u);\mathrm{ev}} \underline{\tau_2}}$$

$\otimes$ Intro
$$\frac{\Gamma_1 \xrightarrow{t} \underline{\tau_1} \qquad \Gamma_2 \xrightarrow{u} \underline{\tau_2}}{\Gamma_1 \otimes \Gamma_2 \xrightarrow{t \otimes u} \underline{\tau_1} \otimes \underline{\tau_2}}$$

$\otimes$ Elim
$$\frac{\Gamma_1 \xrightarrow{t} \underline{\tau_1} \otimes \underline{\tau_2} \qquad \Gamma_2 \otimes \underline{\tau_1} \otimes \underline{\tau_2} \xrightarrow{u} \underline{\tau}}{\Gamma_1 \otimes \Gamma_2 \xrightarrow{(id \otimes t);u} \underline{\tau}}$$

$\oplus$ Intro$_i$
$$\frac{\Gamma \xrightarrow{t} \underline{\tau_i}}{\Gamma \xrightarrow{t;in_i} \underline{\tau_1} + \underline{\tau_2}}$$

$\oplus$ Elim
$$\frac{\Gamma_1 \xrightarrow{u} \underline{\tau_1} + \underline{\tau_2} \qquad \underline{\tau_1} \otimes \Gamma_2 \xrightarrow{t_1} \underline{\tau} \qquad \underline{\tau_2} \otimes \Gamma_2 \xrightarrow{t_2} \underline{\tau}}{\Gamma_1, \Gamma_2 \xrightarrow{u \otimes id_{\Gamma_2}} (\underline{\tau_1} + \underline{\tau_2}) \otimes \Gamma_2 \cong (\underline{\tau_1} \otimes \Gamma_2) + (\underline{\tau_2} \otimes \Gamma_2) \xrightarrow{[t_1, t_2]} \underline{\tau}}$$

Sample
$$\frac{\tau_1 \times \cdots \times \tau_n \xrightarrow{M} \tau \qquad \Gamma_i \xrightarrow{t_i} \mathcal{M}\tau_i}{\Gamma_1 \otimes \cdots \otimes \Gamma_n \xrightarrow{t_1 \otimes \cdots \otimes t_n} \mathcal{M}\tau_1 \otimes \cdots \otimes \mathcal{M}\tau_n \xrightarrow{\mu} \mathcal{M}(\tau_1 \times \cdots \times \tau_n) \xrightarrow{\mathcal{M}M} \mathcal{M}\tau}$$

Fig. 7. Categorical Semantics: λ^2_{INI}

Definition 5.5. Let $(|X|, \mathcal{P}(X))$ and $(|Y|, \mathcal{P}(Y))$ be PCS, we define $X \otimes Y = (|X| \times |Y|, \{x \otimes y \mid x \in \mathcal{P}(X), y \in \mathcal{P}(Y)\}^{\perp\perp})$, where $(x \otimes y)(a, b) = x(a)y(b)$.

We can now define a functor $\mathcal{M} : \mathbf{CountStoch} \to \mathbf{PCoh}$.

Lemma 5.6 (see, e.g., Azevedo de Amorim [2023]). *Let X be a countable set, the pair $(X, \{\mu : X \to \mathbb{R}^+ \mid \sum_{x \in X} \mu(x) \leq 1\})$ is a PCS. Any* **CountStoch** *morphism $X \to Y$ is also a* **PCoh** *morphism.*

Lemma 5.7 (Azevedo de Amorim [2023]). *The functor $\mathcal{M} :$* **CountStoch** $\to$ **PCoh** *is lax monoidal.*

Summing up, we have a model of λ^2_{INI} based on probabilistic coherence spaces.

Theorem 5.8. *The triple* (**PCoh**, **CountStoch**, $\mathcal{M}$) *is a λ^2_{INI} model.*

Proof. **CountStoch** is well-known to be a CD category with coproducts [Fritz 2020], and **PCoh** is a symmetric monoidal closed category with coproducts because it is a model of linear logic [Danos and Ehrhard 2011]. Finally, lax monoidality of $\mathcal{M}$ is given by the previous lemma. □

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 5.5 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.

Something interesting about this model is that it allows encoding of one of the if-statements from Barthe et al. [2019], where they leverage the fact that independence is closed under if-statements of deterministic guards. In this model we can represent this deterministic if-statement as the program:

$$\mathsf{if}_{\mathsf{D}} : (\mathcal{M}1 \oplus \mathcal{M}1) \multimap \tau \multimap \tau \multimap \tau$$
$$\mathsf{if}_{\mathsf{D}}\ b\ t_1\ t_2 = \mathsf{if}\ b\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2$$

5.2.2 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 [Fritz 2020]; see Appendix C for details. For the separating layer, we want a model of linear logic that can interpret continuous randomness. We use a model based on perfect Banach lattices.

Definition 5.9 (Azevedo de Amorim and Kozen [2022])**.** The category $\mathbf{PBanLat_1}$ has perfect Banach lattices as objects and order-continuous linear functions with norm at most one as morphisms.

Intuitively, a perfect Banach lattice is a Banach space equipped with a lattice structure and an involutive linear negation. 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.

Theorem 5.10 (Azevedo de Amorim and Kozen [2022])**.** *There is a lax monoidal functor* $\mathcal{M}$: $\mathbf{BorelStoch} \to \mathbf{PBanLat_1}$.

Theorem 5.11. *The triple* $(\mathbf{PBanLat_1}, \mathbf{BorelStoch}, \mathcal{M})$ *is a* λ^2_{INI} *model.*

Proof. The category **BorelStoch** has a CD structure and has coproducts because it is isomorphic to the Kleisli category of a commutative monad over the category **Meas** [Fritz 2020]. The category $\mathbf{PBanLat_1}$ is a model of classical linear logic, making it a SMCC with coproducts [Azevedo de Amorim and Kozen 2022]. The lax monoidality of $\mathcal{M}$ follows from the previous theorem. □

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 $\mathbf{PCoh} \hookrightarrow \mathbf{PBanLat_1}$ [Azevedo de Amorim and Kozen 2022]. 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.

Something interesting about vector-space-based models of linear logic is that their monoidal unit, usually $\mathbb{R}$, is not a terminal object and form a model of affine linear logic, since there is always a linear transformation $V \multimap \mathbb{R}$ that maps everything to 0. From a programming point of view this has unexpected consequences, since for every well-typed program $\cdot \vdash t : \tau$, the program $\mathsf{let}\ x = * \ \mathsf{in}\ t$ is denotationally equal to the constant 0 function.

5.2.3 Non-Determinism and Communication. Next, we show that the relational model of linear logic gives rise to a λ^2_{INI} model, with applications to distributed programming.

Semantics. Our starting point is the category **Rel** of sets and binary relations, one of the most well-known models of 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 5.12. *The triple* (**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 **Rel** is a model of linear logic, it has coproducts and, by isomorphism, so does $\mathbf{Set}_{\mathcal{P}}$. □

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 leader votes. Note that if either of the messages drops, i.e. the input is the empty set, the whole protocol never terminates.

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 4:

$$\mathsf{message} : \mathcal{M}(1+1) \nvdash_I \ \mathsf{if}\ \mathsf{message}\ \mathsf{then}\ (\mathsf{tt} \otimes \mathsf{tt})\ \mathsf{else}\ (\mathsf{ff} \otimes \mathsf{ff}) : \mathcal{M}\mathbb{B} \otimes \mathcal{M}\mathbb{B}$$

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 [2022], 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 approaches to distributed programming by having a static A set of agents and a modality annotated by elements of A denoting computations that are executed by a particular agent of the distributed system [Hirsch and Garg 2022].

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 [Hüttel et al. 2016] and choreographic programming [Montesi 2014]. 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.

5.2.4 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 5.13 (Fritz [2020])**.** *Let* $\mathbf{C}$ *be a Cartesian category and* T *a commutative monad over it. The Kleisli category* $\mathbf{C}_T$ *is a CD category.*

Lemma 5.14. *Let* $\mathbf{C}$ *be a distributive category and* T *a monad over it. Its Kleisli category* $\mathbf{C}_T$ *has distributive coproducts.*

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 5.15 (Simpson [1992])**.** *Let* $\mathbf{C}$ *be a Cartesian closed category and* T *a strong monad over it. The category of* T*-algebras and plain maps is Cartesian closed, and* 1 *is a terminal object.*

Lemma 5.16. *Let* $\mathbf{C}$ *be a cocartesian category and* T *a monad over it. The category of* T*-algebras and plain maps has weak coproducts.*

Proof. Let (A, α) and (B, β) be two T-algebras. We define $(A, \alpha) \oplus (B, \beta) = (T(A + B), \mu_{A+B})$. Let us prove that this construction satisfies the weak universal property. We start by defining the injection morphism $in'_1 : (A, \alpha) \to (T(A + B), \mu)$, which is defined as $in_1; \eta_{A+B}$, where in_1 is the injection morphism in $\mathbf{C}$. Next, if $f_1 : (A, \alpha) \to (C, \gamma)$ and $f_2 : (B, \beta) \to (C, \gamma)$ are plain maps, their weak universal arrow is $T[f_1, f_2]; \gamma$, where $[f_1, f_2]$ is the cocartesian universal arrow in $\mathbf{C}$.

The weak universal property follows by $in_i; \eta_{A+B}; T[f_1, f_2]; \gamma = in_i; [f_1, f_2]; \eta_C; \gamma = f_i$ □

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 5.17. *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. Lax monoidality follows from T being commutative and the operation $del_A : A \to 1$ being natural. □

Theorem 5.18. *The triple $(\widetilde{\mathrm{C}^T}, \mathrm{C}_T, \iota)$ is a λ^2_{INI} model.*

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 [Azevedo de Amorim 2023].

Name generation. Simple concrete examples of commutative effects are probability and non-determinism, which we saw before. A less standard example is the name generation monad used to give semantics to the ν-calculus, a language that has a primitive for generating "fresh" symbols [Stark 1996]. 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 [1996] 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 5.18 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.

Example: Avoiding Replay Attacks. From a programming point of view, it is important to be able to enforce at the type-level when the set of names being used are disjoint, since failing to do so can create subtle security bugs. Consider the use case where fresh corresponds to a primitive that generates a new encryption key. A common security vulnerability is using the same key to encrypt distinct messages.

Consider a protocol that receives two distinct messages, generates two distinct encryption keys and outputs the two encrypted message. Furthermore, we will assume, as it is frequently the case in practice, that the key is much smaller than the message. For the sake of simplicity we will assume that messages are twice as long as keys and that there is a primitive $\mathsf{split} : \mathcal{M}(msg) \multimap \mathcal{M}(msg) \otimes \mathcal{M}(msg)$ that splits a message into its two key-sized blocks. In this setting we can write the program that receives as input two messages and outputs their encryption.

$$\begin{aligned}
&\cdot \vdash_I \lambda m_1\, m_2. \\
&\mathsf{let}\ f_1 \otimes f_2 = \mathsf{split}\ m_1\ \mathsf{in} \\
&\mathsf{let}\ f_1' \otimes f_2' = \mathsf{split}\ m_2\ \mathsf{in} \\
&\mathsf{sample}\ f_1, f_2, \mathsf{fresh}\ \mathsf{as}\ x_1, x_2, k\ \mathsf{in}\ M \otimes \mathsf{sample}\ f_1', f_2', \mathsf{fresh}\ \mathsf{as}\ x_1, x_2, k\ \mathsf{in}\ M \\
&: \mathcal{M}(msg) \otimes \mathcal{M}(msg) \multimap \mathcal{M}(msg) \otimes \mathcal{M}(msg),
\end{aligned}$$

where $M = (encrypt(x_1, k)) +\!\!+ (encrypt(x_2, k))$ and $+\!\!+$ is the list concatenation operation. Note that if we were to split m_1 twice, the program would no longer type check, effectively making certain bugs unrepresentable in the language.

Remark 5.19 (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 [Simpson 1992]. 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 sample $\overline{t}$ as $\overline{x}$ in M is to force the execution of CBN computations $\overline{t}$, bind the results to $\overline{x}$, and run them eagerly in the program M.

5.2.5 Affine Bunched Typing. It is natural to wonder how BI is related to λ^2_{INI}. We have seen that certain fragments of the BI inspired language λ_{INI} embeds in λ^2_{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 5.20. *For every cocartesian model of affine BI* $\mathbf{C}$ *the triple* $(\mathbf{C}, \mathbf{C}, id)$ *is a model of* λ^2_{INI}.

Remark 5.21. From a more abstract point of view, by initiality of the syntactic model of λ^2_{INI} (Theorem B.3) 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+ [O'Hearn 2003]. 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 [O'Hearn 2003].

By Theorem 5.20 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 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

$$(A \otimes B)(X) \triangleq \{(a, b) \in A(X) \times B(X) \mid support(a) \cap support(b) = \emptyset\}$$
$$(A \otimes B)(f) \triangleq (Afa, Bfb)$$

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 [O'Hearn 2003]. 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.

$$\begin{array}{llll} \text{types} & \tau & ::= & \mathsf{cell} \mid \mathsf{exp} \mid \mathsf{comm} \mid \tau \rightarrow \tau \mid \tau \multimap \tau \mid \tau \times \tau \\ \text{contexts} & \Gamma & ::= & \cdot \mid x : \tau \mid \Gamma ; \Gamma \mid \Gamma , \Gamma \end{array}$$

Fig. 8. Types and Terms: SCI+

$$\frac{\Gamma \vdash M : \mathsf{comm} \qquad \Gamma \vdash N : \mathsf{comm}}{\Gamma \vdash M ; N : \mathsf{comm}} \qquad \frac{\Gamma_1 \vdash M : \mathsf{comm} \qquad \Gamma_2 \vdash N : \mathsf{comm}}{\Gamma_1 , \Gamma_2 \vdash M || N : \mathsf{comm}}$$

$$\frac{\Gamma , x : \mathsf{cell} \vdash M : \mathsf{comm}}{\Gamma \vdash \mathsf{new}\, x . M : \mathsf{comm}} \qquad \frac{\Gamma \vdash M : \mathsf{cell} \qquad \Gamma \vdash N : \mathsf{exp}}{\Gamma \vdash M := N : \mathsf{comm}}$$

$$\frac{\Gamma \vdash M : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash N : \tau_1}{\Gamma \vdash M\ N : \tau_2} \qquad \frac{\Gamma_1 \vdash M : \tau_1 \multimap \tau_2 \qquad \Gamma_2 \vdash N : \tau_1}{\Gamma_1 , \Gamma_2 \vdash M\ N : \tau_2}$$

Fig. 9. Typing Rules: SCI+ (selected)

For our purposes, we are mainly interested in the SCI+ operations presented in Figure 9. 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 applications: 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 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 λ^2_{INI}. As we have explained, a direct consequence of Theorem 5.20 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 10. 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 existing ones, making it natural to use the I language.

Example 5.22 (O'Hearn [2003]). 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.

$$\frac{\textsc{Sequential} \quad \Gamma \vdash_{NI} M : \mathsf{comm} \qquad \Gamma \vdash_{NI} N : \mathsf{comm}}{\Gamma \vdash_{NI} M; N : \mathsf{comm}} \qquad \frac{\textsc{Parallel} \quad \Gamma_1 \vdash_I t : \mathcal{M}\mathsf{comm} \qquad \Gamma_2 \vdash_I u : \mathcal{M}\mathsf{comm}}{\Gamma_1, \Gamma_2 \vdash_I t \,||\, u : \mathcal{M}\mathsf{comm}}$$

$$\frac{\textsc{New} \quad \Gamma, x : \mathcal{M}\mathsf{cell} \vdash_I t : \mathcal{M}\mathsf{comm}}{\Gamma \vdash_I \mathsf{new}\, x.t : \mathcal{M}\mathsf{comm}} \qquad \frac{\textsc{Assign} \quad \Gamma \vdash_{NI} M : \mathsf{cell} \qquad \Gamma \vdash_{NI} N : \mathsf{exp}}{\Gamma \vdash_{NI} M := N : \mathsf{comm}}$$

Fig. 10. Typing Rules: λ^2_{INI} extended with SCI primitives

6 SOUNDNESS THEOREM

So far we have seen two proofs of soundness. For λ_{INI}, we proved soundness using logical relations (Theorem 3.3). For λ^2_{INI} with a probabilistic semantics, we used an observation about algebras for the distribution monad (Theorem 4.1). 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 6.1. *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 [Hyland and Schalk 2003; Johnstone et al. 2007].

A detailed description of this technique is beyond the scope of this paper. However, we highlight some of the essential aspects here. We have already introduced our class of models for λ^2_{INI}. Let $\Gamma \vdash_I t : \underline{\tau}$ be a well-typed program. For every concrete model $(\mathbf{C}, \mathbf{M}, \mathcal{M})$, we want to show that the interpretation $[\![t]\!]$ in this model must satisfy some properties in order to validate the soundness theorem. At a high level, there are three steps to the gluing argument:

(1) Define a category of models of λ^2_{INI}, and show that every interpretation $[\![\cdot]\!]$ can be encoded as a map from the *syntactic* model **Syn** to $(\mathbf{C}, \mathbf{M}, \mathcal{M})$; where the syntactic model has types as objects and typing derivations (modulo the equational theory of λ^2_{INI}) as morphisms. This property follows by showing that the syntactic model is initial.
(2) Define a triple $(\mathbf{Gl}(\mathbf{C}), \mathbf{M}, \widetilde{\mathcal{M}})$—where objects of the category $\mathbf{Gl}(\mathbf{C})$ are pairs $(A, X \subseteq \mathbf{C}(I, A))$, the subsets X are viewed as predicates on A, and morphisms preserve these predicates—and show that this structure is a model of λ^2_{INI}. We call this the *glued* model and there is an obvious forgetful model morphism $(\mathbf{Gl}(\mathbf{C}), \mathbf{M}, \widetilde{\mathcal{M}}) \to (\mathbf{C}, \mathbf{M}, \mathcal{M})$.
(3) Using initiality, define a map $(\!|\cdot|\!)$ from the syntactic model **Syn** to the glued model. The data of this map associates every I-type $\underline{\tau}$ in λ^2_{INI} to an object $(A_{\underline{\tau}}, X_{\underline{\tau}} \subseteq \mathbf{C}(I, A_{\underline{\tau}}))$; intuitively, $A_{\underline{\tau}} \in \mathbf{C}$ is the interpretation of $\underline{\tau}$ under $[\![\cdot]\!]$, and the subset $X_{\underline{\tau}}$ encodes the logical relation at type $\underline{\tau}$, so this map defines a logical relation. The functor $(\!|\cdot|\!)$ and its codomain encode the logical relations proof.

Finally, we can use $(\!|\cdot|\!)$ to map any global element in the syntactic category, i.e., well-typed term $\cdot \vdash_I t : \underline{\tau}$, to an element of $X_{\underline{\tau}}$. By initiality of **Syn**, $[\![t]\!]$ also is an element of $X_{\underline{\tau}}$, completing the proof by logical relations proof. We defer the details to Appendix B, where we also go over the details of how to soundly add base types and operations to λ^2_{INI}.

7 RELATED WORK

Linear logics and probabilistic programs. A recent line of work uses linear logic as a powerful framework to provide semantics to probabilistic programming languages. Notably, Ehrhard et al. [2018] 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. [2017] provide a denotational semantics inspired by linear logic for a higher-order probabilistic language with continuous random sampling. Linear type systems have also been developed for probabilistic properties, like almost sure termination [Dal Lago and Grellois 2019] and differential privacy [Azevedo de Amorim et al. 2019; Reed and Pierce 2010].

Our categorical model for λ^2_{INI} is inspired by models of linear logic based on monoidal adjunctions, most notably Benton's LNL [Benton 1994]. From a programming languages perspective, these models decompose the linear λ-calculus with exponentials in two languages with distinct product types each 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 5.2 are LNL models. Furthermore, the "shared" layer in LNL models is Cartesian closed, which is unsuitable for programming with effects, due to its call-by-name nature.

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 [Moggi 1991]; 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) [Levy 2001]. 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 [2023], 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) [O'Hearn and Pym 1999]. 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 λ^2_{INI}, BI also features two implications, one for each conjunction. The leading application of BI is in separations logic for concurrent and heap-manipulating programs [O'Hearn 2007; O'Hearn et al. 2001], where pre- and post-conditions are drawn from BI.

Though λ_{INI} also has a bunched type system, its semantics differs from the doubly closed categorical semantics of BI. It is still unclear how to characterize the categorical semantics of λ_{INI}, but we conjecture that it is equivalent to doubly strong monads over doubly closed categories.

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. [2019], who propose a type system combining linearity, information-flow control, and probability regions for a probabilistic functional language. Darais et al. [2019] 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. [2021] 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. [2021] 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 [Barthe et al. 2019], but also refinements like conditional independence [Bao et al. 2021; Li et al. 2023], and negative association [Bao et al. 2022]. 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 [Batz et al. 2022, 2019].

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., [Dubhashi and Ranjan 1998]). Some of these notions have been extended to other models besides probability; for instance, Pearl and Paz [1986] 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.

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.

REFERENCES

Arthur Azevedo de Amorim, Marco Gaboardi, Justin Hsu, and Shin-ya Katsumata. 2019. Probabilistic Relational Reasoning via Metrics. In *ACM/IEEE Symposium on Logic in Computer Science (LICS), Vancouver, British Columbia*. IEEE, 1–19. DOI: http://dx.doi.org/10.1109/LICS.2019.8785715

Pedro H. Azevedo de Amorim. 2023. A Higher-Order Language for Markov Kernels and Linear Operators. In *Foundations of Software Science and Computation Structures (FoSSaCS), Paris, France.*

Pedro H Azevedo de Amorim and Dexter Kozen. 2022. Classical Linear Logic in Perfect Banach Spaces. *Preprint* (2022).

Jialu Bao, Simon Docherty, Justin Hsu, and Alexandra Silva. 2021. A bunched logic for conditional independence. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS).* IEEE, 1–14.

Jialu Bao, Marco Gaboardi, Justin Hsu, and Joseph Tassarotti. 2022. A separation logic for negative dependence. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–29.

Gilles Barthe, Justin Hsu, and Kevin Liao. 2019. A Probabilistic Separation Logic. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–30.

Kevin Batz, Ira Fesefeldt, Marvin Jansen, Joost-Pieter Katoen, Florian Keßler, Christoph Matheja, and Thomas Noll. 2022. Foundations for Entailment Checking in Quantitative Separation Logic. In *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 (Lecture Notes in Computer Science)*, Ilya Sergey (Ed.), Vol. 13240. Springer, 57–84. DOI: http://dx.doi.org/10.1007/978-3-030-99336-8_3

Kevin Batz, Benjamin Lucien Kaminski, Joost-Pieter Katoen, Christoph Matheja, and Thomas Noll. 2019. Quantitative separation logic: a logic for reasoning about probabilistic pointer programs. *Proc. ACM Program. Lang.* 3, POPL (2019), 34:1–34:29. DOI: http://dx.doi.org/10.1145/3290347

P. N. Benton. 1994. A Mixed Linear and Non-Linear Logic: Proofs, Terms and Models (Extended Abstract). In *International Workshop on Computer Science Logic (CSL), Kazimierz, Poland (Lecture Notes in Computer Science)*, Leszek Pacholski and Jerzy Tiuryn (Eds.), Vol. 933. Springer, 121–135. DOI: http://dx.doi.org/10.1007/BFb0022251

Francis Borceux. 1994. *Handbook of Categorical Algebra: Volume 2, Categories and Structures.* Vol. 2. Cambridge University Press.

Kenta Cho and Bart Jacobs. 2019. Disintegration and Bayesian inversion via string diagrams. *Math. Struct. Comput. Sci.* 29, 7 (2019), 938–971. DOI: http://dx.doi.org/10.1017/S0960129518000488

Roy L Crole. 1993. *Categories for types.* Cambridge University Press.

Fredrik Dahlqvist, Alexandra Silva, Vincent Danos, and Ilias Garnier. 2018. Borel kernels and their approximation, categorically. *Electronic Notes in Theoretical Computer Science* (2018).

Ugo Dal Lago and Charles Grellois. 2019. Probabilistic Termination by Monadic Affine Sized Typing. *ACM Trans. Program. Lang. Syst.* 41, 2 (2019), 10:1–10:65. DOI: http://dx.doi.org/10.1145/3293605

Vincent Danos and Thomas Ehrhard. 2011. Probabilistic coherence spaces as a model of higher-order probabilistic computation. *Information and Computation* 209, 6 (2011), 966–991.

David Darais, Ian Sweet, Chang Liu, and Michael Hicks. 2019. A language for probabilistically oblivious computation. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–31.

Devdatt P. Dubhashi and Desh Ranjan. 1998. Balls and bins: A study in negative dependence. *Random Struct. Algorithms* 13, 2 (1998), 99–124.

Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2017. Measurable cones and stable, measurable functions: a model for probabilistic higher-order programming. In *Principles of Programming Languages (POPL).*

Thomas Ehrhard, Michele Pagani, and Christine Tasson. 2018. Full Abstraction for Probabilistic PCF. *J. ACM* 65, 4 (2018), 23:1–23:44. DOI: http://dx.doi.org/10.1145/3164540

Tobias Fritz. 2020. A synthetic approach to Markov kernels, conditional independence and theorems on sufficient statistics. *Advances in Mathematics* 370 (2020), 107239.

Andrew K Hirsch and Deepak Garg. 2022. Pirouette: higher-order typed functional choreographies. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–27.

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. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (apr 2016), 36 pages. DOI: http://dx.doi.org/10.1145/2873052

Martin Hyland and Andrea Schalk. 2003. Glueing and orthogonality for models of linear logic. *Theoretical computer science* 294, 1-2 (2003), 183–231.

Peter T Johnstone, Stephen Lack, and Paweł Sobociński. 2007. Quasitoposes, quasiadhesive categories and Artin glueing. In *International Conference on Algebra and Coalgebra in Computer Science.* Springer, 312–326.

Paul C Kainen. 1971. Weak adjoint functors. *Mathematische Zeitschrift* 122 (1971), 1–9.

Neel Krishnaswami. 2011. A new lambda calculus for bunched implications. (2011). https://semantic-domain.blogspot.com/2011/07/new-lambda-calculus-for-bunched.html [Online; accessed 2023-07-09].

Tom Leinster. 2014. *Basic category theory.* Vol. 143. Cambridge University Press.

Paul Blain Levy. 2001. *Call-by-push-value.* Ph.D. Dissertation.

John M Li, Amal Ahmed, and Steven Holtzen. 2023. Lilac: a Modal Separation Logic for Conditional Probability. *Programming Language Design and Implementation (PLDI)* (2023).

Elisabet Lobo Vesga, Alejandro Russo, and Marco Gaboardi. 2021. A Programming Language for Data Privacy with Accuracy Estimations. *ACM Trans. Program. Lang. Syst.* 43, 2 (2021), 6:1–6:42. DOI: http://dx.doi.org/10.1145/3452096

Saunders Mac Lane. 2013. *Categories for the working mathematician*. Vol. 5. Springer Science & Business Media.

Eugenio Moggi. 1991. Notions of Computation and Monads. *Inf. Comput.* 93, 1 (1991), 55–92. DOI: http://dx.doi.org/10.1016/0890-5401(91)90052-4

Fabrizio Montesi. 2014. *Choreographic Programming*. Ph.D. Dissertation. Denmark.

Peter W. O'Hearn. 2003. On bunched typing. *J. Funct. Program.* 13, 4 (2003), 747–796. DOI: http://dx.doi.org/10.1017/S0956796802004495

Peter W. O'Hearn. 2007. Separation logic and concurrent resource management. In *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, Greg Morrisett and Mooly Sagiv (Eds.). ACM, 1. DOI: http://dx.doi.org/10.1145/1296907.1296908

Peter W. O'Hearn and David J. Pym. 1999. The logic of bunched implications. *Bull. Symb. Log.* 5, 2 (1999), 215–244. DOI: http://dx.doi.org/10.2307/421090

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic, 15th International Workshop, CSL 2001. 10th Annual Conference of the EACSL, Paris, France, September 10-13, 2001, Proceedings (Lecture Notes in Computer Science)*, Laurent Fribourg (Ed.), Vol. 2142. Springer, 1–19. DOI: http://dx.doi.org/10.1007/3-540-44802-0_1

Judea Pearl and Azaria Paz. 1986. Graphoids: Graph-Based Logic for Reasoning about Relevance Relations or When would x tell you more about y if you already know z?. In *European Conference on Artificial Intelligence (ECAI), Brighton, UK*, Benedict du Boulay, David C. Hogg, and Luc Steels (Eds.). North-Holland, 357–363.

David J. Pym, Peter W. O'Hearn, and Hongseok Yang. 2004. Possible worlds and resources: the semantics of BI. *Theor. Comput. Sci.* 315, 1 (2004), 257–305. DOI: http://dx.doi.org/10.1016/j.tcs.2003.11.020

Jason Reed and Benjamin C. Pierce. 2010. Distance makes the types grow stronger: a calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Baltimore, Maryland*, Paul Hudak and Stephanie Weirich (Eds.). ACM, 157–168. DOI: http://dx.doi.org/10.1145/1863543.1863568

Alex K Simpson. 1992. Recursive types in Kleisli categories. *Unpublished manuscript, University of Edinburgh* (1992).

Ian Stark. 1996. Categorical models for local names. *Lisp and Symbolic Computation* 9, 1 (1996), 77–107.

Dario Maximilian Stein. 2021. *Structural foundations for probabilistic programming languages*. Ph.D. Dissertation. University of Oxford.

Christine Tasson and Thomas Ehrhard. 2019. Probabilistic call by push value. *Logical Methods in Computer Science* (2019).

A SOUNDNESS PROOF λ_{INI}

We remind the readers the logical relation for types:

$$\begin{aligned}
\mathcal{R}_{\mathbb{B}} &= D(\mathbb{B})\\
\mathcal{R}_{\tau_1 \otimes \tau_2} &= \{\mu_1 \otimes \mu_2 \in D(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket) \mid \mu_i \in \mathcal{R}_{\tau_i}\}\\
\mathcal{R}_{\tau_1 \times \tau_2} &= \{\mu \in D(\llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket) \mid \pi_i(\mu) \in \mathcal{R}_{\tau_i} \text{ for } i \in \{1,2\}\}\\
\mathcal{R}_{\tau_1 \multimap \tau_2} &= \{\mu \in D(\llbracket \tau_1 \rrbracket \to D(\llbracket \tau_2 \rrbracket)) \mid \forall \mu' \in \mathcal{R}_{\tau_1}, x \leftarrow \mu'; f \leftarrow \mu; f(x) \in R_{\tau_2}\}\\
\mathcal{R}_{\tau_1 \to \tau_2} &= \{\mu \in D(\llbracket \tau_1 \rrbracket \to D(\llbracket \tau_2 \rrbracket)) \mid \forall \mu' \in D(\tau_1 \times (\tau_1 \to D(\tau_2)))\\
&\qquad \mu'_1 \in \mathcal{R}_{\tau_1} \wedge \mu'_2 = \mu \Rightarrow (x,h) \leftarrow \mu'; h(x) \in R_{\tau_2}\}.
\end{aligned}$$

And for contexts:

$$\begin{aligned}
\mathcal{R}_{\cdot} &= 1 & \mathcal{R}_{\cdot} &= 1\\
\mathcal{R}_{x:\tau} &= \mathcal{R}_{\tau} & \mathcal{R}_{x:\tau} &= \mathcal{R}_{\tau}\\
\mathcal{R}_{\Gamma_1,\Gamma_2} &= \{\mu \in D(\llbracket \Gamma_1 \rrbracket \times \llbracket \Gamma_2 \rrbracket) \mid \pi_i(\mu) \in \mathcal{R}_{\Gamma_i}\} & \mathcal{R}_{\Delta_1;\Delta_2} &= \{\mu_1 \otimes \mu_2 \in D(\llbracket \Delta_1 \rrbracket \times \llbracket \Delta_2 \rrbracket) \mid \mu_i \in \mathcal{R}_{\Delta_i}\}\\
\mathcal{R}_{r[\Delta]} &= \mathcal{R}_{\Delta} & \mathcal{R}_{r[\Gamma]} &= \mathcal{R}_{\Gamma}
\end{aligned}$$

Theorem A.1. *If $\Gamma \vdash t : \tau$ and $\mu \in \mathcal{R}_\Gamma$ then $(x \leftarrow \mu; \llbracket t \rrbracket (x)) \in \mathcal{R}_\tau$.*

Proof. Let the distribution above be ν. We prove $\nu \in \mathcal{R}_\tau$ by induction on the derivation of $\Gamma \vdash t : \tau$. When the context is separated, we may assume that $x \leftarrow \mu$ is given by the list of the marginal distributions, in which case we will represent them as a list $\overline{\mu_i}$.

Const/Coin/Var. Trivial. For instance, Var_S: $\nu = \overline{x_i \leftarrow \mu_i}$; return $x_i = \mu_i$ is in $\mathcal{R}_{\tau_i}$ by assumption.

× Intro. We have $\nu = \gamma \leftarrow \mu; x \leftarrow [\![t_1]\!](\gamma); y \leftarrow [\![t_2]\!](\gamma);$ return (x, y). It is straightforward to show that the first marginal of ν is $\gamma \leftarrow \mu; x \leftarrow [\![t_1]\!](\gamma);$ 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 $\mathcal{R}_{\tau_2}$.

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

⊗ Intro. Let $\overline{\mu}$ be the distribution corresponding to Δ_1, and let $\overline{\eta}$ be the distribution corresponding to Δ_2. Since D is a commutative monad [Borceux 1994], we may apply associativity and commutativity to show:

$$\begin{aligned}
\nu &= x' \leftarrow \mu; y' \leftarrow \eta; x \leftarrow [\![t_1]\!](x'); y \leftarrow [\![t_2]\!](y'); \text{return } (x, y) \\
&= x' \leftarrow \mu; x \leftarrow [\![t_1]\!](x'); y' \leftarrow \eta; y \leftarrow [\![t_2]\!](y'); \text{return } (x, y) \\
&= (x' \leftarrow \mu; x \leftarrow [\![t_1]\!](x'); \text{return } x) \otimes (y' \leftarrow \eta; y \leftarrow [\![t_2]\!](y'); \text{return } y) = \nu_1 \otimes \nu_2.
\end{aligned}$$

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.

⊗ 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.

Shared Abstraction. By unfolding the definitions, we need to show that for every joint distribution μ' over $[\![\tau_1]\!]$ and $[\![\tau_1]\!] \to D([\![\tau_2]\!])$ such that its first marginal is an element of $\mathcal{R}\tau_1$ and its second marginal is equal to $\gamma \leftarrow \mu; [\![\lambda x. t]\!](\gamma)$ then $((x, f) \leftarrow \mu'; f(x)) \in \mathcal{R}\tau_2$. The full proof for this case is not as straightforward as the other ones. Here we will only present the case of when the function $\gamma \mapsto [\![\lambda x. t]\!](\gamma)$ is injective. By unfolding the definitions we obtain:

$$\begin{aligned}
&(x, f) \leftarrow \mu'; f(x) \\
&= \sum_{a,f} \mu'(a, f) f(a) \\
&= \sum_{a,\gamma} \mu'(a, \lambda x. [\![t]\!](\gamma, x)) [\![t]\!](\gamma, a)
\end{aligned}$$

The second equation is only true under the injectivity hypothesis. The induction hypothesis for $\Gamma, x : \tau_1 \vdash t : \tau_2$ says that for every joint distribution μ'' over Γ and τ_1 such that its marginals are elements of $\mathcal{R}_\Gamma$ and $\mathcal{R}_{\tau_1}$, respectively, $(\gamma, x) \leftarrow \mu''; [\![t]\!](\gamma, x) \in \mathcal{R}_{\tau_2}$. Consider the distribution $\mu''(\gamma, a) = \mu'(a, \lambda x. [\![t]\!](\gamma, x))$. We can easily show that $\mu''_2 = \mu'_1 \in \mathcal{R}_{\tau_1}$ and since $\mu'_2 = \gamma \leftarrow \mu; [\![\lambda x. t]\!](\gamma)$, $\mu \in \mathcal{R}_\Gamma$ and $\gamma \mapsto [\![\lambda x. t]\!](\gamma)$ is injective, $\mu'_2(\lambda x. [\![t]\!](\gamma, x)) = \mu(\gamma)$. By inspection, this case follows from the induction hypothesis by choosing the distribution μ''. The full case can be found below.

Shared Application. Let $\mu \in \mathcal{R}_\Gamma$, $\Gamma \vdash t : \tau_1 \to \tau_2$ and $\Gamma \vdash u : \tau_1$. We have to show that $(\gamma \leftarrow \mu; (f, x) \leftarrow ([\![t]\!](\gamma) \otimes [\![x]\!](\gamma)); f(x)) \in \mathcal{R}_{\tau_2}$. This follows by applying the induction hypothesis to $\Gamma \vdash u : \tau_1$ and $\Gamma \vdash t : \tau_1 \to \tau_2$, where the joint distribution over $[\![\tau_1]\!] \times ([\![\tau_1]\!] \to D[\![\tau_2]\!])$ is $(\gamma \leftarrow \mu; ([\![t]\!](\gamma) \otimes [\![x]\!](\gamma))$.

Context Modal Rules Follows directly from the induction hypothesis. □

In order for this proof to go through in the general case, we need a definition from probability theory.

Definition A.2. Let $f : A \to D(B)$ and $\mu \in D(A)$, we define $f_\mu^{-1} : B \to D(A)$

$$f_\mu^{-1}(b, a) = \begin{cases} \frac{\mu(a) f(a,b)}{\sum_{a'} \mu(a') f(a',b)} & \text{if } \sum_{a'} \mu(a') f(a', b) > 0 \\ \mu(a) & \text{otherwise} \end{cases}$$

The function above is basically a different presentation of Bayes' theorem. At a more conceptual level, this construction can be seen as a "weak" inverse of f in the following sense:

Lemma A.3. *Let $\mu : D(A)$ and $f : A \to D(B)$, then $(x \leftarrow \mu; y \leftarrow f(x); f_\mu^{-1}(y)) = \mu$.*

We can now present the full proof for the shared abstraction case. Assume that the soundness theorem holds for a program $\Gamma, x : \tau_1 \vdash t : \tau_2$ and that $\mu' \in D([\![\tau_1]\!] \times ([\![\tau_1]\!] \to D([\![\tau_2]\!])))$ satisfies $\mu'_1 \in \mathcal{R}_{\tau_1}$ and $\mu'_2 = \gamma \leftarrow \mu; [\![\lambda x. t]\!](\gamma)$, for some $\mu \in \mathcal{R}_\Gamma$. Furthermore, let us define $F(\gamma) = [\![\lambda x. t]\!](\gamma)$. In this case we can show

$$\begin{aligned} &(x, f) \leftarrow \mu'; f(x) \\ &= \sum_{a,f} \mu'(a, f) f(a) \\ &= \sum_{a,\gamma} \mu'(a, \lambda x. [\![t]\!](\gamma, x)) F_\mu^{-1}(\lambda x. [\![t]\!](x, \gamma), \gamma)\, [\![t]\!](a, \gamma) \end{aligned}$$

The second equation holds because for every γ, $\sum_f \frac{F(\gamma,f)}{\sum_f F(\gamma,f)} = 1$ and the only functions in the support of μ'_2 are of the form $[\![\lambda x. t]\!](\gamma)$, for some γ. Finally, by applying the induction hypothesis to $\Gamma, x : \tau_1 \vdash t : \tau_2$ with the joint distribution μ'' over the context equal to $((x, f) \leftarrow \mu'; \gamma \leftarrow F_\mu^{-1}(f); \text{return } (x, \gamma)$. we can show by a direct calculation that its first marginal is equal to $\mu'_1 \in \mathcal{R}_{\tau_1}$. In order to reason about its second marginal, consider the equalities.

$$\begin{aligned} \mu''_2 &= (x, f) \leftarrow \mu'; \gamma \leftarrow F_\mu^{-1}(f); \text{return } \gamma \\ &= (x, f) \leftarrow \mu'; F_\mu^{-1}(f) \\ &= \gamma \leftarrow \mu; f \leftarrow F(\gamma); F_\mu^{-1}(f) = \mu \end{aligned}$$

Furthermore, by unfolding the definitions, we can show that $(x, \gamma) \leftarrow \mu''_2; [\![t]\!](\gamma, x) = (x, f) \leftarrow \mu'; f(x)$, concluding this case.

B CATEGORICAL SOUNDNESS PROOF FOR λ^2_{INI}: DETAILS

B.1 Category of Models

A model for λ^2_{INI} is given by a CD category **M** with distributive coproducts, a SMCC **C** with weak coproducts and a lax monoidal functor $\mathcal{M} : \mathbf{M} \to \mathbf{C}$. A morphism between two models

$$
\begin{aligned}
\mathsf{case}\ (\mathsf{in}_1 M)\ \mathsf{of}\ (|\mathsf{in}_1 x \Rightarrow N_1 \mid \mathsf{in}_2 x \Rightarrow N_2) &\equiv N_1\{M/x\} \\
\mathsf{case}\ (\mathsf{in}_2 M)\ \mathsf{of}\ (|\mathsf{in}_1 x \Rightarrow N_1 \mid \mathsf{in}_2 x \Rightarrow N_2) &\equiv N_2\{M/x\} \\
\mathsf{case}\ N\ \mathsf{of}\ (|\mathsf{in}_1 x \Rightarrow M \mid \mathsf{in}_2 x \Rightarrow M) &\equiv M\{N/x\} \\[1em]
\mathsf{let}\ x = t\ \mathsf{in}\ x &\equiv t \\
\mathsf{let}\ x = x\ \mathsf{in}\ t &\equiv t \\
\mathsf{let}\ y = (\mathsf{let}\ x = M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ M_3 &\equiv \mathsf{let}\ x = M_1\ \mathsf{in}\ (\mathsf{let}\ y = M_2\ \mathsf{in}\ M_3) \\[1em]
(\lambda x.\, t)\ u &\equiv t\{u/x\} \\
(\lambda x.\, t\ x) &\equiv t \\
\mathsf{let}\ x_1 \otimes x_2 = t_1 \otimes t_2\ \mathsf{in}\ u &\equiv u\{t_1/x_1\}\{t_2/x_2\} \\[1em]
\mathsf{case}\ (\mathsf{in}_1 t)\ \mathsf{of}\ (|\mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) &\equiv u_1\{t/x\} \\
\mathsf{case}\ (\mathsf{in}_2 t)\ \mathsf{of}\ (|\mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) &\equiv u_2\{t/x\} \\[1em]
\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ x &\equiv t \\
\mathsf{sample}\ (\mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ M)\ \mathsf{as}\ y\ \mathsf{in}\ N &\equiv \mathsf{sample}\ t\ \mathsf{as}\ x\ \mathsf{in}\ (\mathsf{let}\ y = M\ \mathsf{in}\ N)
\end{aligned}
$$

Fig. 11. (Selected Rules) Equational Theory: λ^2_{INI}

$(\mathbf{M}_1, \mathbf{C}_1, \mathcal{M}_1)$ 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 up-to isomorphism. 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 **Mod**.

In categorical treatments of type theories it is important to show that the equational theory is a sound approximation of the categorical semantics. Most of the λ^2_{INI} equational theory is depicted in Figure 11. In the case of CD categories, there are some subtleties when defining their equational theory — more details can be found in Chapter 2 of [Stein 2021]. The equational theory of symmetric monoidal closed categories is very similar to the simply-typed case [Crole 1993]. Since the language does not use any fancy type theoretic constructions, the soundness property is straightforward to prove by induction on the typing derivations.

Theorem B.1. *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 lax monoidal functors only provide a binary sampling operator. Formally this is resolved by restricting sample to up to two arguments and adding the following rules to the equational theory:

$$\frac{\Gamma_i \vdash_I t_i : \mathcal{M}\tau_i \qquad i \in \{1,2,3\}}{\begin{array}{c}\Gamma_1, \Gamma_2, \Gamma_3 \vdash_I \texttt{sample } t_1, (\texttt{sample } t_2, t_3 \texttt{ as } x_2, x_3 \texttt{ in } (x_2, x_3)) \texttt{ as } x_1, y \texttt{ in } (x_1, \pi_1\, y, \pi_2\, y) \equiv \\ \texttt{sample } (\texttt{sample } t_1, t_2 \texttt{ as } x_1, x_2 \texttt{ in } (x_1, x_2)), t_3 \texttt{ as } y, x_3 \texttt{ in } (\pi_1\, y, \pi_2\, y, x_3) : \mathcal{M}(\tau_1 \times \tau_2 \times \tau_3)\end{array}}$$

$$\frac{\Gamma \vdash_I t : \mathcal{M}\tau}{\Gamma \vdash_I \texttt{sample } t, (\texttt{sample \_ as \_ in } ()) \texttt{ as } x, y \texttt{ in } x \equiv t : \mathcal{M}\tau}$$

$$\frac{\Gamma \vdash_I t : \mathcal{M}\tau}{\Gamma \vdash_I \texttt{sample } (\texttt{sample \_ as \_ in } ()), t \texttt{ as } x, y \texttt{ in } y \equiv t : \mathcal{M}\tau}$$

Note that even though the first 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 **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 11. In order for these to be considered categories each syntax must satisfy the substitution property, which has been proved in [Azevedo de Amorim 2023] for the sum-less version of λ^2_{INI}, which is not hard to extend to the version with sums. Finally, it follows by a simple inspection that **Syn** is a λ^2_{INI} model.

Lemma B.2. Syn *is a* λ^2_{INI} *model.*

Theorem B.3. Syn *is the initial object of* **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 7.

The proof that this function is well-defined follows from Theorem B.1. Uniqueness follows by assuming the existence of two semantics and showing, by induction on the typing derivation, that they are equal. □

B.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 $\widetilde{\mathcal{M}} : \mathbf{C} \to \mathbf{Gl}(\mathbf{C})$. 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. Therefore, 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 B.4. $\mathbf{Gl}(\mathbf{C})$ *is a SMCC and weak coproducts.*

Proof. Let (A, X) and (B, Y) be $\mathbf{Gl}(\mathbf{C})$ objects, we define $(A, X) \otimes (B, Y) = (A \otimes B, \{f : I \xrightarrow{\cong} I \otimes I \xrightarrow{f_A \otimes f_B} A \otimes B \mid f_A \in X, f_B \in Y\})$; the monoidal unit is given by $(I, \{id_I\})$.

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 $f_1 : (A_1, X_1) \to (B, Y)$ and $f_2 : (A_2, X_2) \to (B, Y)$ 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 [Hyland and Schalk 2003], but since they are 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 **Mod** morphism. Next, we have to lift $\mathcal{M}$ to the glued category. This follows from general category theoretic observations.

Definition B.5. If X is an $\mathbf{M}$ object then $\widetilde{\mathcal{M}}(X) = (\mathcal{M}(X), \{\varepsilon; \mathcal{M}f \mid f \in \mathbf{M}(1, X)\}$. Furthermore, if $f : X \to Y$ is an $\mathbf{M}$ morphism then $\widetilde{\mathcal{M}}(f) = \mathcal{M}(f)$.

Lemma B.6. *The operation* $\widetilde{\mathcal{M}} : \mathbf{M} \to \mathbf{Gl}(\mathbf{C})$ *is a lax monoidal functor.*

Proof. By assumption that $\mathcal{M}$ is a functor, it is mostly immediate that $\widetilde{M}$ is a functor, we only have to show that $\mathcal{M}f$ is a morphism in the glued category. Let $\varepsilon; \mathcal{M}g$ be a plot in the domain of $\mathcal{M}f$. In this case, $\varepsilon; \mathcal{M}g; \mathcal{M}f = \varepsilon; \mathcal{M}(g; f)$, which implies functoriality.

In order to prove lax monoidality, it suffices to prove that the operations $\varepsilon : I \to \mathcal{M}1$ and $\mu : \mathcal{M}X \otimes \mathcal{M}Y \to \mathcal{M}(X \times Y)$ can be lifted to the glued category, in which case lax monoidality follows by the assumption that $\mathcal{M}$ is lax monoidal. First, ε lifts to the glued category because

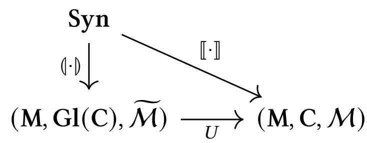

Fig. 12. The essence of the soundness proof

$id_I; \varepsilon = \varepsilon; \mathcal{M}(id_1)$. Next, showing that μ lifts as well is less straightforward: it follows from the naturality of μ, the naturality of $A \otimes I \cong A$ and the lax monoidal diagrams. □

Thus, the glued category is a λ^2_{INI} model.

Theorem B.7. *The triple* $(\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}})$ *is a* **Mod** *object.*

There is a forgetful map from the glued model to the original model.

Lemma B.8. *There is a* **Mod** *morphism* $U : (\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}}) \to (\mathbf{M}, \mathbf{C}, \mathcal{M})$.

Finally, by initiality of **Syn**, we can prove

Lemma B.9. *There is a* **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 **Syn**, is equal to the functor $[\![\cdot]\!]$, as illustrated by Figure 12.

B.3 General Soundness Theorem

Theorem B.10. *If* $\cdot \vdash_I t : \underline{\tau}$, *then* $[\![t]\!] \in X_{\underline{\tau}}$.

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 5.3 follows immediately, as a corollary.

Corollary B.11. *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 B.10, 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 $[\![t]\!] = f_1 \otimes f_2$. □

B.4 Adding Base Types and Constants

Suppose that we want to add a new base type to λ^2_{INI} and operations over it. If the type and operation are supposed to be added to the NI layer, then this addition is as simple as giving the type and operation a semantics in **M**.

If, however, we want to add a new type to the I layer then we must be careful, since besides it being necessary to give semantics in **C**, it becomes necessary showing that the semantics lifts to the glued category $\mathbf{Gl}(\mathbf{C})$. For instance, suppose that we want to add a type σ and an operation $\Gamma \vdash \text{op} : \sigma$. If there is an intended semantics $[\![\sigma]\!]$ and $[\![\text{op}]\!]$ in **C** we must define a predicate X_σ which could, for example, be equal to $\mathbf{C}(I, [\![\sigma]\!])$, and then we have to prove that for every $p \in X_\Gamma$, $p; [\![\text{op}]\!] \in X_\sigma$.

Something interesting about this approach is that the choice of X_σ is not unique. Consider, for instance, in the probabilistic case, a different way to define deterministic if-statements is by adding a constant $\mathcal{M}_{det}(2)$ which is interpreted as $(\mathcal{M}(2), \{\delta_0, \delta_1\})$ in the glued model. Now we can soundly add the constant $\text{if}_{det} : \mathcal{M}_{det}(2) \multimap \tau \multimap \tau \multimap \tau$.

C MEASURABLE SETS AND MARKOV KERNELS

A measurable space combines a set with a collection of subsets, describing the subsets that can be assigned a well-defined measure or probability.

Definition C.1. Given a set X, a *σ-algebra* $\Sigma_X \subseteq \mathcal{P}(X)$ is a set of subsets such that (i) $X \in \Sigma_X$, and (ii) Σ_X is closed complementation and countable union. A *measurable space* is a pair (X, Σ_X), where X is a set and Σ_X is a σ-algebra.

A *measurable function* between measurable spaces (X, Σ_X) and (Y, Σ_Y) is a function $f : X \to Y$ such that for every $A \in \Sigma_Y$, $f^{-1}(A) \in \Sigma_X$, where f^{-1} is the inverse image function. Measurable spaces and measurable functions form a category **Meas**.

Definition C.2. Standard Borel spaces (X, Σ_X) are spaces such that X can be equipped with a metric such that X is, as a metric space, complete and separable and Σ_X is the σ-algebra generated by the metric.

Example C.3. For every $n \in \mathbb{N}$, $\mathbb{R}^n$ with its standard σ-algebra is a standard Borel space.

Definition C.4. A *probability measure* is a function $\mu_X : \Sigma_X \to [0, 1]$ such that: (i) $\mu(\emptyset) = 0$, (ii) $\mu(X) = 1$, and $\mu(\biguplus A_i) = \sum_i \mu(A_i)$.

Definition C.5. A *Markov kernel* between measurable spaces (X, Σ_X) and (Y, Σ_Y) is a function $f : X \times \Sigma_Y \to [0, 1]$ such that:

- For every $x \in X$, $f(x, -)$ is a probability distribution.
- For every $B \in \Sigma_Y$, $f(-, B)$ is a measurable function.

Markov kernels $f : X \times \Sigma_Y \to [0, 1]$ and $g : Y \times \Sigma_Z \to [0, 1]$ can be composed with the following formula

$$(g \circ f)(x, C) = \int g(-, C) df(x, -)$$

The Dirac kernel $\delta(a, A) = 1$ if $a \in A$ and 0 otherwise is the unit for the composition defined above that this structure can be organized into a category **BorelStoch** with standard Borel spaces as objects and Markov kernels as morphisms.

Marginals and probabilistic independence. We will need some constructions on distributions and measures over products.

Definition C.6. Given a distribution μ over $X \times Y$, its *marginal* μ_X is the distribution over X defined by $\mu_X(A) = \int_Y d\mu(A, -)$. Intuitively, this is the distribution obtained by sampling a pair from μ and projecting to its first component. The other marginal μ_Y is defined similarly.

Definition C.7. A probability measure μ over $A \times B$ is probabilistically *independent* if it is a product of its marginals μ_A and μ_B, i.e., $\mu(X, Y) = \mu_A(X) \cdot \mu_B(Y)$, $X \in \Sigma_A$ and $Y \in \Sigma_B$.