Separation and Sharing in Higher-Order Languages with Effects

ANONYMOUS AUTHOR(S)

When reasoning about probabilistic programs, a fundamental property is *independence*. Recent works have developed separation logics capturing independence for imperative, first-order programs, but independence is more difficult to capture in functional, higher-order programs.

In this work, we take a more general perspective: we propose two higher-order languages where we can reason about sharing and separation in effects. Our first language λ_{INI} has a linear type system and probabilistic semantics, where the two product types capture independent and possibly-dependent pairs. Our second language λ^2_{INI} is a two-level, stratified language, inspired by Benton's linear-non-linear (LNL) caluclus. 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.

Additional Key Words and Phrases: Probabilistic Programming, Denotational Semantics

1 INTRODUCTION

Probabilistic semantics have been going through a renaissance in the last decade, mainly due to the rise in popularity of machine learning. There has been a lot of recent work exploring different semantics for probabilistic programming languages and understanding which properties they can naturally reason about. Another useful direction is exploring if there are type systems that are particularly well-suited for probabilistic programming.

Reasoning About Independence. In the context of probabilistic programming languages, independence is an assumption which is baked in their implementations: their sampling primitives usually assume that new samples are independent from previously sampled values. On the verification side of things, independence is used to simplify reasoning about programs: if it is known that two programs are independent, their joint distribution will only depend on their individual probabilities; there are no unexpected probabilistic interaction between them. Independence is also a property useful in the context of cryptographic protocols, where their security property might be stated as an independence property.

Probabilistic independence has a few properties that makes it approachable from a programming languages and formal methods perspective. For instance, in the context of probabilistic programs, probabilistic independence is preserved under local operations, e.g. if you have functions $f, g : \mathbb{N} \to \mathbb{N}$ and independent input $x, y : \mathbb{N}$ then $f(x)$ and $g(y)$ will be independent as well. Furthermore, there are combinators idiomatic in programming languages such as certain pairing operations that also preserve independence. This affinity has already been explored before by defining program logics that can about independence in the context of a first-order, imperative, language. Unfortunately, there are no higher-order languages for independence.

Our Work: Higher-Order Languages for Probabilistic Independence. An important idea behind independence is that by reasoning about the resources used by programs one can infer independence properties between them. In this work we define two substructural higher-order languages that can reason about probabilistic independence. They are, as far as we aware, the first languages that

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

do so. They both achieve this by adding a product type constructor ⊗ that enforces independence in the sense that closed programs of type $\mathbb{N} \otimes \mathbb{N}$ should be denoted by independent distributions.

The first language we have defined is a simply typed λ-calculus with two product type constructors: one for independent distributions and the other for possibly dependent distributions. We give a denotational semantics to this language and prove its soundness theorem for ⊗ types. This language is an linear λ-calculus where the ⊗ type constructor enforces that the components of the pair do not share any resources. On the other hand, the × type constructor allows the sharing of resources. An unfortunate characteristic of this language is that extending it with new types such as sum types while still preserving the soundness property is tricky while also suffering from expressivity issues.

In order to mitigate these issues, we have defined a two-level language that does not suffer from these drawbacks. The intuition is that the languages have disentangled the product types so that in one level you may only program with linear products whereas the other language allows you to share resources. The independent language has a modality that allows to soundly manipulate programs written in the shared language. We also show how to extend this language with sum typesm we give a denotational semantics for it, prove its soundness theorem and show how to soundly translate a fragment of the one-level language into the two-level language.

Additional Models. We also explore how the reasoning principles enforced by our type system can also be applied to other domains. In order to accommodate these other applications, we generalize the soundness theorem of our type system and propose a categorical semantics for it. In Section 5.2, we present many examples showing how our semantics can be readily applied to existing semantics of effectful programming languages.

- **Linear logic.** Models of linear logic have been used to give semantics to probabilistic languages with discrete and continuous sampling [Danos and Ehrhard 2011; Ehrhard et al. 2017]. We show that these categories, paired with the category of Markov kernels, yield models for our two-level language. For these models, our soundness theorem continues to guarantee probabilistic independence. As far as we know, our method is the first to ensure probabilistic independence for in these models.
- **Distributed programming.** Next, we develop a relational model of our language and describe an application in distributed programming. In this model, programs in our two-level language describe the implementation of multiple agents, but the program does not specify where computations should be executed. Our soundness theorem shows that programs of type $\tau_1 \otimes \tau_2$ can be factored as two local programs, i.e., we can compile the global program into local programs that can execute independently, without communication across machines. This soundness 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, in cryptographic applications, reusing a *nonce* value ("number once") may result in a security bug in the protocol. We define a model of our language based on name generation. In this context, our soundness theorem says that the type ⊗ 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 a few assumptions, every commutative monad gives rise to a model of our language by using categories of algebras for this monad.

- **Bunched and separation logics.** A long line of work uses *bunched logics* to reason about sharing and separation; however, these works do not handle effectful programs. We show that models of affine bunched logics also models of our language, but not vice-versa. Thus, our language provides a less restrictive model to reason about sharing and separation of resources in programs. We illustrate this by revisiting Reynolds' syntactic control of interference (SCI) language, and show that since its original model is also a model to our language, there is a sound translation of our language into his.

The diversity of models provides evidence that our language is a suitable framework to reason about separation and sharing in effectful higher-order programs.

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

- First, we define a linear, higher-order probabilistic λ-calculus called λ_{INI}, with types that can capture probabilistic independence and dependence. We give a denotational semantics of our language and prove that $\otimes$ captures probabilistic independence (§3).
- Next, we define a two-level, higher-order probabilistic λ-calculus called λ^2_{INI}. This language combines a independent fragment and a sharing fragment with two distinct sum types: an independent sum, and a sharing sum. We give a probabilistic semantics for the language and prove that $\otimes$ captures probabilistic independence; we also provide an embedding from λ_{INI} to λ^2_{INI} (§4).
- Abstracting away from the probabilistic case, we propose a general categorical semantics for λ^2_{INI}. Our semantics can be seen as a generalization of Benton's linear/non-linear (LNL) model for linear logic [Benton 1994] (§5.1).
- To justify our language, we present a wide range of models for λ^2_{INI}, including models inspired by probabilistic models of linear logic, choreographies and distributed programming, commutative effects, name generation, and bunched logics. We show that the soundness property of our type system ensures natural notions of independence in these new models (§5.2).
- Finally, we prove a general soundness theorem for our categorical models, showing that $\otimes$ enforces more general independence property: 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. We prove this theorem for our general categorical models using a categorical gluing argument (§6).

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

2 BACKGROUND

2.1 Monads and their algebras

In order to formalize our semantics we will use some basic concepts from category theory such as symmetric monoidal closed categories and coproducts that go beyond the scope of this paper, but we recommend the interested reader to read [Leinster 2014; Mac Lane 2013] for nice introductions to the subject.

Monads. We start by defining monads, which are frequently used to give semantics to effectful programming languages. A monad over a category $\mathbf{C}$ is a triple (T, μ, η) such that $T : \mathbf{C} \to \mathbf{C}$ is a functor, $\mu_A : T^2A \to TA$ and $\eta_A : A \to TA$ are natural transformations such that $\mu_A \circ \mu_{TA} = \mu_A \circ T\mu_A$, $id_A = \mu_A \circ T\eta_A$ and $id_A = \mu_A \circ \eta_{TA}$.

Another useful, and equivalent, presentation of monads is 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 Kleisli category $\mathbf{C}_T$, which has the same objects as

C and $Hom_{C_T}(A, B) = Hom_C(A, TB)$. Following seminal work by [Moggi 1991], 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 $g \circ Th = h \circ f$ It is possible to show that T-algebras and their morphisms form a category $\mathbf{C}^T$, called the Eilenberg-Moore category.

The Kleisli category $\mathbf{C}_T$ and the Eilenberg-Moore category $\mathbf{C}^T$ are are deeply connected. Indeed, for every $\mathbf{C}$ object A, the object TA can be equipped with a canonical T-algebra morphism given by μ_A. Such algebras are called *free*. More generally, we have:

Theorem 2.1 ([Borceux 1994])**.** *There is a full and faithful functor* $\iota : \mathbf{C}_T \to \mathbf{C}^T$.

2.2 Probability Theory

When dealing with distributions over discrete sets, subprobability distributions can be directly modeled as functions $\mu : X \to [0, 1]$ such that its sum is at most 1. However, when dealing with continuous sets such as the real line, we need the full generality of measure theory to properly interpret probability distributions.

Measures and measurable spaces. 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 2.2. 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. The category **Meas** has measurable sets as objects and measurable functions as morphisms.

Definition 2.3. A probability measure is a function $\mu_X : \Sigma_X \to [0, 1]$ such that: (i) $\mu(\emptyset) = 0$, (ii) $\mu(X) = 1$, and $\mu(\uplus A_i) = \sum_i \mu(A_i)$.

The Giry Monad. The set $\mathcal{G}(X)$ of probability distributions over a measurable set X can be equipped with a σ-algebra:

Theorem 2.4. *The pair* $(\mathcal{G}(X), \Sigma_{\mathcal{G}(X)})$ *is a measurable set, where* $\Sigma_{\mathcal{G}(X)}$ *is the smallest* σ*-algebra such that the functions* $ev_A : \mathcal{G}(X) \to [0, 1]$ *are measurable for every measurable set* $A \in \Sigma_X$.

Furthermore, $\mathcal{G}$ can be given a monad structure on **Meas**, called the Giry monad. The unit is $\eta(a) = \delta_a$, where $\delta_x(A) = 1$ if $x \in A$ and 0 otherwise, usually referred to as Dirac delta distribution. Given $f : A \to \mathcal{G}(B)$ we define $f^*(\mu) = \int_A f \, d\mu$.

This monad is often used to give semantics to probabilistic programs. Indeed, Kleisli arrows $A \to MB$ are in exact correspondence with Markov kernels.

Definition 2.5. 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.

A simpler probability monad can be defined for **Set**. Given a set X, we define DX as the set of functions $\mu : X \to [0, 1]$ which is non zero in a finite set (finite support) and $\sum_{x \in supp(\mu)} \mu(x) = 1$. It is also possible to show that this construction is monadic with the same definitions as above, except that integrals are replaced by sums.

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

Definition 2.6. Given a distribution μ over $X \times Y$, its marginal μ_X is a distribution over X defined by $\mu_X(A) = \int_Y d\mu(A, -)$. Intuitively, this is the distribution obtained by sampling from μ and projecting its first component. There is a symmetrically defined marginal distribution μ_Y.

In the discrete case, the marginal is given by a sum: the first marginal is $\mu_X(x) = \sum_{y \in Y} \mu(x, y)$, and the second marginal μ_Y is similar.

Definition 2.7. A probability measure μ over a product $A \times B$ is said to be probabilistically *independent* if it can be factored by its marginals μ_A and μ_B, i.e. $\mu(X, Y) = \mu_A(X)\mu_B(Y)$, $X \in \Sigma_A$ and $Y \in \Sigma_B$.

In the discrete case, probabilistic independence can be defined more simply: a distribution μ over $A \times B$ is probabilistically independent if $\mu_A(x) \cdot \mu_B(y) = \mu(x, y)$ for every $x \in A$ and $y \in B$.

3 A LINEAR LANGUAGE FOR INDEPENDENCE

3.1 Independence Through Linearity

In most 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 components use distinct sources of randomness, then they are independent.

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

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

does not produce an independent distribution: the two components are always equal, and hence perfectly correlated. These principles resemble the properties enforced by substructural type systems, which control when resources can be shared and when they are disjoint. To make this idea concrete, we develop a language λ_{INI} with a linear type system that can reason about probabilistic independence.

3.2 Introducing the Language λ_{INI}

Syntax. Figure 1 presents the syntax of types and terms. Along with base types ($\mathbb{B}$), there are two product types: $\times$ is the possibly-dependent product, while $\otimes$ is the independent product. The language is higher-order, so there is a linear arrow type. The corresponding term syntax is fairly standard: we have variables, numeric constants, and primitive distributions (coin). The two kinds of products have can be created from two kinds of pairs, and eliminated using projection and let-binding, respectively. Finally, we have the usual λ-abstraction and application. Our examples use the standard syntactic sugar $\mathsf{let}\ x = t\ \mathsf{in}\ u \triangleq (\lambda x.\, u)\ t$.

Type system. Figure 2 shows the typing rules for λ_{INI}. The rules are standard from linear logic; we comment some of the key rules. The variable rule Var is *linear*: it requires all of the variables in the context to be used, and variables cannot be freely discarded. For the product $\times$, the introduction rule $\times$ Intro shares the context across the premises: both components can share the same variables. Components can be projected out of these pairs, one at a time ($\times$ Elim). For the product $\otimes$, in

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

Fig. 1. Types and Terms: λ_{INI}

$$
\textsc{Const}\ \frac{}{\Gamma \vdash b : \mathbb{B}} \qquad \textsc{Coin}\ \frac{}{\Gamma \vdash \mathsf{coin} : \mathbb{B}} \qquad \textsc{Var}\ \frac{}{x : \tau \vdash x : \tau}
$$

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

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

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

Fig. 2. Typing rules λ_{INI}

contrast, the introduction rule ⊗ Intro requires both premises to use *different* contexts. Thus, the components cannot share variables. Tensor pairs are eliminated by a let-pair construct that consumes both components at once (⊗ Elim). In substructural type systems, × is called an *additive* product, while ⊗ is called a *multiplicative* product. The abstraction and application rules are standard.

An additive arrow? Note that the application rule is multiplicative, in the sense that the function and the argument cannot share variables. A natural question is whether the arrow should be additive: should it be possible to share variables between the function and its argument? Substructural type systems like bunched logic [O'Hearn and Pym 1999] include both a multiplicative and an additive arrow.

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

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

Under our eager semantics, which we will discuss next, this program has the same behavior as $\mathsf{let}\ x = \mathsf{coin}\ \mathsf{in}\ x \otimes x$, which produces a pair of *non*-independent values. Thus, the arrow should be multiplicative.

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

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

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

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

Fig. 3. Denotational Semantics: λ_{INI}

3.3 Denotational Semantics

We can give a semantics to this language using the category **Set** and the finite probability monad D. From top to bottom, Figure 3 defines the semantics of types, contexts, and typing derivations producing well-typed terms. For types, we interpret both product types as products of sets. Arrow types are interpreted as the set of Kleisli arrows, i.e., maps $[\![\tau_1]\!] \to D\,[\![\tau_2]\!]$. Contexts are interpreted as products of sets.

We interpret well-typed terms as particular Kleisli arrows. We briefly walk through the term semantics, which is essentially the same as the Kleisli semantics proposed by Moggi [1991]. Variables are interpreted using the unit of the monad, which is the point mass distribution. Coins are interpreted as the fair convex combination of two point mass distributions, over 0 and 1.

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 semantics of the two pairs is the same: we sample from both components and pair the results. The projections for $\times$ simply computes the marginal of a joint distribution, while let-binding for $\otimes$ samples from t and then use the sample in u. Lambda abstractions are interpreted as point mass distributions while applications are interpreted as sampling from the function and the argument, and applying the first sample to the second one.

Example 3.1 (Correlated pairs)**.** It may seems as if there is no way of creating non-independent pairs, since the semantics for $\times$ pairs samples separately from each component. However, consider the program let x = coin in (x, x). By unfolding the definitions, its semantics is

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

Example 3.2 (Independent pairs are correlated pairs)**.** In any language that can distinguish between independent and possibly-dependent distributions, it should be possible to view the former as the latter. In λ_{INI}, this conversion is captured by the following program:

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

3.4 Soundness

The design of the type system of λ_{INI} is to guarantee that $\otimes$ enforces probabilistic independence. Concretely, we want to show that if $\cdot \vdash t : \tau_1 \otimes \tau_2$ 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:

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

Theorem 3.3. *If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash t : \tau$ and $\mu_i \in \mathcal{R}_{\tau_i}$ then*

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

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

Const/Coin/Var. Trivial. For instance, for variables: $\nu = \overline{x \leftarrow \mu}$; return $x = \mu$, which is in $\mathcal{R}_{\tau_n}$ by assumption.

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

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

$\otimes$ Intro. Let $\overline{\mu_i}$ be the sequence of distributions corresponding to Γ_1, and let $\overline{\eta_i}$ be the sequence of distributions corresponding to Γ_2. We have:

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

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

$\otimes$ **Elim.** Let $\overline{\mu_i}$ be the sequence of distributions corresponding to Γ_1, and let $\overline{\eta_i}$ be the sequence of distributions corresponding to Γ_2. We have:

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

where the third equality is by the induction hypothesis with the first premise. By the induction hypothesis with the second premise, the last 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 $\mu \in \mathcal{R}_{\tau_1}$. By associativity and commutativity, that expression is equal to $x_i \leftarrow \mu_i; x \leftarrow \mu; f \leftarrow \delta_{\lambda x.[\![t]\!](x_i)}; f(x)$. By using the induction hypothesis and the fact that δ is the monad's unit we can conclude this case.

Application. This case follows directly from the induction hypothesis. □

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

4 A TWO-LEVEL LANGUAGE FOR INDEPENDENCE

In the probabilistic higher-order language λ_{INI}, the type system can distinguish between independent random quantities, and possibly dependent random quantities. However, there are some important limitations of λ_{INI}. We first discuss these issues, and then introduce a stratified, two-level language λ^2_{INI} that resolves these problems. Finally, we show how to embed a substantial fragment of λ_{INI} into λ^2_{INI}.

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

Adding sum types. A notable shortcoming of λ_{INI} is that it does not include sum types. Though there are base types like $\mathbb{B}$, it is not possible to perform case analysis. Indeed, extending λ_{INI} with sum types leads to problems. Consider the following program:

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

Operationally, this probabilistic program flips a fair coin and checks if it comes up true. If so, the program returns the pair $\text{tt} \otimes \text{tt}$, otherwise it returns the pair $\text{ff} \otimes \text{ff}$. Since both tt and ff are constants, they do not share any variables, 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 **??** 3.3: the components of the pair are always equal to each other, and hence *not* probabilistic independent.

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

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

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

$$\begin{aligned}&\text{let } u_1 = \text{coin in}\\&\text{let } u_2 = \text{coin in}\\&\text{let } x = f(u_1, u_2) \text{ in}\\&\text{let } y = g(u_1, u_2) \text{ in}\\&(x, y)\end{aligned}$$

There are useful sampling algorithms (e.g., the Box-Muller transform [Box and Muller 1958]), which follow this template. In order to write a well-typed version of this program in λ_{INI}, we could inline the definitions of x and y: the pair constructor $(-, -)$ is additive, so the two components can both mention the variables u_1 and u_2. However, it is awkward to require that a straightforward program must be inlined.

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

$$\begin{aligned}&\text{let } x = \pi_1\, z \text{ in}\\&\text{let } y = \pi_2\, z \text{ in}\\&u(x, y)\end{aligned}$$

is not well-typed, since the outer let-binding shares the variable z with its body. These problems would be solved if function application (and let-bindings) in λ_{INI} were additive; however, as we have seen in Section 3, allowing a function and an argument to share variables would also break the soundness property of λ_{INI}.

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

To address these limitations, we introduce a stratified language. We are guided by a simple observation about products, sums, and distributions, which might be of more general interest. In λ_{INI}, the product types correspond to two distinct ways of composing distributions with and 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 resembles 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 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*; as we will see in section 5.2, the semantics of λ^2_{INI} resembles Benton's linear/non-linear (LNL) semantics for linear logic [Benton 1994].

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

The I-layer is quite similar to λ_{INI}: it has its own product and sum types, and a linear arrow type. 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 eliminate products and sums, and functions and applications. The last construct sample x as t in M is novel: it allows the two layers to interact.

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	::=	$x \mid b \in \mathbb{B} \mid \mathsf{coin} \mid f \in O(\tau_1, \tau_2) \mid (M, N) \mid \pi_i\, M \mid \mathsf{in}_i\, \mathsf{t}$
		$\mid$	$\mathsf{case}\, t\, \mathsf{of}\, (\mid \mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) \mid \mathsf{let}\, x = M\, \mathsf{in}\, N$
I-expressions	t, u	::=	$x \mid t \otimes u \mid \mathsf{let}\, x \otimes y = t\, \mathsf{in}\, u \mid \mathsf{in}_i\, \mathsf{t}$
		$\mid$	$\mathsf{case}\, t\, \mathsf{of}\, (\mid \mathsf{in}_1 x \Rightarrow u_1 \mid \mathsf{in}_2 x \Rightarrow u_2) \mid \lambda x.\, t \mid t\, u \mid \mathsf{sample}\, x\, \mathsf{as}\, t\, \mathsf{in}\, M$
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. In particular, the let-binding rule is *additive* instead of multiplicative as in λ_{INI}: a let-binding is allowed to share variables with its body.

The second group of typing rules assigns types to the independent (I) layer. These rules are the standard rules for multiplicative additive linear logic (MALL), and are almost identical to the typing rules for λ_{INI}. Just like before, the rules treat variables 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, gives the interaction rule between the two languages. The first premise is from the sharing (NI) language, where the program M can have free variables $x_1, \ldots, x_n$. The rest of the premises are from the independent (I) language, where linear programs t_i have boxed type $\mathcal{M}\tau_i$. The conclusion of the rule combines programs t_i with M, producing an I-program of boxed type. Intuitively, this rule allows a program in the sharing language to be imported into the linear language. Operationally, sample x as t in M constructs a distribution t using the independent language, samples from it, and then binds the sample to x in the shared program M.

Semantics. We can now give a semantics to this two-level language. To keep the presentation concrete, in this section we will work with a concrete semantics motivated by probabilistic independence, where programs are probabilistic programs with discrete sampling. In Section 5.2, we will return to the general categorical semantics of λ^2_{INI}.

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 M. The Kleisli category $\mathbf{Set}_M$ has sets as objects, so we may simply define the semantics of each type to be a set. It is also known that $\mathbf{Set}_M$ has products and coproducts, which can be used to interpret well-typed programs in NI.

For the I-language, we are going to use the category of algebras for the finite distribution monad M and plain maps, $\widetilde{\mathbf{Set}^M}$. Concretely, its objects are pairs (A, f), where f is an M-algebra, and a morphism $(A, f) \to (B, g)$ is a function $A \to B$. Given two objects (A, f) and (B, g) we can define a product algebra $A \times B$ [Borceux 1994]. 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.

Therefore, the semantics of types presented in Figure 6 is basically equal to the set theoretic semantics, since we can make the set-theoretic operations $\times, \Rightarrow, +$ into algebra operations. We only need to explicitly manipulate the algebraic structure when interpreting the type constructor $\mathcal{M}$, which is interpreted as the "free" M-algebra with the multiplication for the monad as the algebraic structure.

A consequence, we can interpret the second language using basically a set theoretic semantics. Pairings are interpreted using Cartesian products and eliminated using function composition. Arrow types are interpreted using function applications and currying. Sum types are interpreted using disjoint unions and case-analysis. The sole operation which needs more structure than a simple set theoretic semantics is Sample, which uses the distribution functor M and function composition.

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* $\cdot \vdash_I t : \tau_1 \otimes \tau_2$ *then* $[\![t]\!]$ *is an independent distribution.*

Proof. If $\cdot \vdash_I t : \tau_1 \otimes \tau_2$ then its semantics is given by a D-algebra morphism $1 \to D\tau_1 \times D\tau_2$, which is isomorphic to an independent distribution. □

4.3 Revisiting Sums and Let-Binding

Now that we have seen λ^2_{INI}, let us revisit the problematic if-then-else program at the beginning of the section. The type system of λ^2_{INI} makes it impossible to produce an independent pair by pattern matching on values:

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

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:

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

While we were motivated by adding sums to λ_{INI}, our design also removes the limitations on let-bindings we discussed before, since the sharing layer has an additive let-binding. In particular,

$$\textsc{Const}\quad \frac{b \in \mathbb{B}}{\Gamma \vdash_{NI} b : \mathbb{B}} \qquad \textsc{Coin}\quad \frac{}{\Gamma \vdash_{NI} \mathsf{coin} : \mathbb{B}} \qquad \textsc{Primitive}\quad \frac{\Gamma \vdash_{NI} M : \tau_1 \qquad f \in O(\tau_1, \tau_2)}{\Gamma \vdash_{NI} f(M) : \tau_2}$$

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

$$\textsc{Pair}\quad \frac{\Gamma \vdash_{NI} M : \tau_1 \qquad \Gamma \vdash_{NI} N : \tau_2}{\Gamma \vdash_{NI} (M, N) : \tau_1 \times \tau_2} \qquad \textsc{Proj1}\quad \frac{\Gamma \vdash_{NI} M : \tau_1 \times \tau_2}{\Gamma \vdash_{NI} \pi_1 M : \tau_1} \qquad \textsc{Proj2}\quad \frac{\Gamma \vdash_{NI} M : \tau_1 \times \tau_2}{\Gamma \vdash_{NI} \pi_2 M : \tau_2}$$

$$\textsc{In1}\quad \frac{\Gamma \vdash_{NI} M : \tau_1}{\Gamma \vdash_{NI} \mathsf{in}_1\, M : \tau_1 + \tau_2} \qquad \textsc{In2}\quad \frac{\Gamma \vdash_{NI} M : \tau_2}{\Gamma \vdash_{NI} \mathsf{in}_2\, M : \tau_1 + \tau_2}$$

$$\textsc{Case}\quad \frac{\Gamma \vdash_{NI} M : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash_{NI} N_1 : \tau \qquad \Gamma, x : \tau_2 \vdash_{NI} N_2 : \tau}{\Gamma \vdash_{NI} \mathsf{case}\ M\ \mathsf{of}\ (\mid \mathsf{in}_1\, x \Rightarrow N_1 \mid \mathsf{in}_2\, y \Rightarrow N_2) : \tau}$$

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

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

$$\textsc{In1}\quad \frac{\Gamma \vdash_I t : \tau_1}{\Gamma \vdash_I \mathsf{in}_1\, t : \tau_1 \oplus \tau_2} \qquad \textsc{In2}\quad \frac{\Gamma \vdash_I t : \tau_2}{\Gamma \vdash_I \mathsf{in}_2\, t : \tau_1 \oplus \tau_2}$$

$$\textsc{Case}\quad \frac{\Gamma_1 \vdash_I t : \tau_1 \oplus \tau_2 \qquad \Gamma_2, x : \tau_1 \vdash_I u_1 : \tau \qquad \Gamma_2, x : \tau_2 \vdash_I u_2 : \tau}{\Gamma_1, \Gamma_2 \vdash_I \mathsf{case}\ t\ \mathsf{of}\ (\mid \mathsf{in}_1\, x \Rightarrow u_1 \mid \mathsf{in}_2\ \Rightarrow u_2) : \tau}$$

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

Fig. 5. Typing rules λ^2_{INI}

$$\begin{aligned}
(\!|\mathbb{B}|\!) &= \mathbb{B} \\
(\!|\tau \times \tau|\!) &= (\!|\tau|\!) \times (\!|\tau|\!) \\
(\!|\tau + \tau|\!) &= (\!|\tau|\!) + (\!|\tau|\!)
\end{aligned}$$

$$(\!|x_1 : \tau_1, \ldots, x_n : \tau_n|\!) = (\!|\tau_1|\!) \times \cdots \times (\!|\tau_n|\!)$$

$$(\!|\Gamma \vdash M : \tau|\!) \in \mathbf{Set}_D((\!|\Gamma|\!), (\!|\tau|\!))$$

$$\begin{aligned}
[\![\mathcal{M}\tau]\!] &= (D\,[\![\tau]\!], \mu_{[\![\tau]\!]}) \\
[\![\underline{\tau} \otimes \underline{\tau}]\!] &= [\![\underline{\tau}]\!] \times [\![\underline{\tau}]\!] \\
[\![\underline{\tau} \multimap \underline{\tau}]\!] &= [\![\underline{\tau}]\!] \to [\![\underline{\tau}]\!] \\
[\![\underline{\tau} \oplus \underline{\tau}]\!] &= [\![\underline{\tau}]\!] + [\![\underline{\tau}]\!]
\end{aligned}$$

$$[\![x_1 : \underline{\tau}_1, \ldots, x_n : \underline{\tau}_n]\!] = [\![\underline{\tau}_1]\!] \times \cdots \times [\![\underline{\tau}_n]\!]$$

$$[\![\Gamma \vdash t : \underline{\tau}]\!] \in \widetilde{\mathbf{Set}}^D([\![\Gamma]\!], [\![\underline{\tau}]\!])$$

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

Fig. 6. Concrete semantics λ^2_{INI}

it is also possible to express the problematic let-binding program we saw before:

$$\begin{aligned}
\cdot \vdash_I\ & \text{sample coin, coin as } u_1, u_2 \text{ in} \\
& \text{let } x = f(u_1, u_2) \text{ in} \\
& \text{let } y = g(u_1, u_2) \text{ in} \\
& M : \mathcal{M}(\tau)
\end{aligned}$$

We can also project both components out of pairs, at least in the sharing layer:

$$\begin{aligned} \cdot \vdash_{NI} & \ \mathsf{let}\ x = \pi_1\, M_1\ \mathsf{in} \\ & \ \mathsf{let}\ y = \pi_2\, M_2\ \mathsf{in} \\ & \ M : \tau \end{aligned}$$

4.4 Embedding from λ_{INI} to λ^2_{INI}

Given λ_{INI} and λ^2_{INI}, a natural question is how these languages are related. We show that it is possible to embed the fragment without arrow types of λ_{INI} into λ^2_{INI}. Since its semantics is given by a Kleisli category, there is an obvious translation of it into the non-independent fragment of λ^2_{INI}.

$$\begin{aligned} \mathcal{T}(\mathbb{B}) &= \mathbb{B} \\ \mathcal{T}(\tau_1 \times \tau_2) = \mathcal{T}(\tau_1 \otimes \tau_2) &= \mathcal{T}(\tau_1) \times \mathcal{T}(\tau_2) \end{aligned}$$

At the term-level, the translation is the identity function.

Theorem 4.2. *If* $\Gamma \vdash M : \tau$ *then* $\mathcal{T}(\Gamma) \vdash \mathcal{T}(M) : \mathcal{T}(\tau)$

Furthermore, it is easy to show that this translation preserves equations between programs and is full abstract.

Theorem 4.3. *Let* $\Gamma \vdash t_1 : \tau$ *and* $\Gamma \vdash t_2 : \tau$ *then* $[\![t_1]\!] = [\![t_2]\!]$ *if, and only if,* $[\![\mathcal{T}(t_1)]\!] = [\![\mathcal{T}(t_2)]\!]$.

Proof. The proof follows from the fact that the translation is a faithful functor. □

It is also possible to translate the $\otimes$, $\multimap$ fragment of λ_{INI} into λ^2_{INI}.

$$\begin{aligned} \mathcal{T}'(\mathbb{B}) &= \mathcal{M}\mathbb{B} \\ \mathcal{T}'(\tau_1 \otimes \tau_2) &= \mathcal{T}'(\tau_1) \otimes \mathcal{T}'(\tau_2) \\ \mathcal{T}'(\tau_1 \multimap \tau_2) &= \mathcal{T}'(\tau_1) \multimap \mathcal{T}'(\tau_2) \end{aligned}$$

Once again, the term translation is the identity function.

Theorem 4.4. *If* $\Gamma \vdash t : \tau$ *then* $\mathcal{T}'(\Gamma) \vdash \mathcal{T}'(t) : \mathcal{T}'(\tau)$.

Proof. The proof follows by induction on the typing derivation $\Gamma \vdash t : \tau$. □

This translation is functorial and faithful and therefore is sound and full abstract with respect with the denotational semantics of λ_{INI} and λ^2_{INI}.

Something interesting about these translations is that 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 λ_{INI} it would be possible to extend the first translation so that it encompasses the whole language.

5 CATEGORICAL SEMANTICS AND CONCRETE MODELS

The language λ^2_{INI} and its probabilistic semantics defines a probabilistic calculus with sharing and separation of resources, and it has a simple soundness proof showing that $\otimes$ captures probabilistic independence. However, it is limited to monadic effects, sidestepping many interesting models, such as ones based on linear logic, which are plentiful in the probabilistic programming literature. In this section, we first present the full, categorical semantics of λ^2_{INI}, by abstracting the Kleisli/Eilenberg-Moore semantics we saw in the previous section. Then, we present a wide variety of concrete models for λ^2_{INI}.

5.1 Categorical Semantics of λ^2_{INI}

Motivation. Let us assume that 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. Indeed, if we have a term of type $\mathcal{M}(\tau \times \tau)$, we should not be able to use its components to produce a term of type $\mathcal{M}\tau \otimes \mathcal{M}\tau$.

An important question to understand is how the type constructors $\times$ and $\otimes$ should be interpreted. We have seen that $\widetilde{\mathbf{C}^T}$ has products whenever $\mathbf{C}$ has them. However, the typing rules in Figure 5 suggest that it only required a monoidal product, which is exactly the formalism we will choose. On the other hand, though we want to be able to copy arguments using $\times$, we are not interested in the universal property of products, only in its comonoidal structure, i.e. being able 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.

Finally, as we have mentioned above, independent distributions are, in particular, possibly dependent distributions. Therefore there should be a program $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2 \vdash \mathcal{M}(\tau_1 \times \tau_2)$, which we interpret as $\mathcal{M}$ being an applicative functor.

In its abstract presentation is called a lax monoidal functor which is defined as a functor $F : (\mathbf{C}, \otimes_C, I_C) \to (\mathbf{D}, \otimes_D, I_D)$ between monoidal categories equipped with morphisms $\mu_{A,B} : F(A) \otimes_D F(B) \to F(A \otimes_C B)$ and $\epsilon : I_D \to FI_C$ making certain diagrams commute.

Categorical model. Taking these aspects into account, we propose the following categorical model for λ^2_{INI}.

Definition 5.1. A semantics to our language is given by a CD category with coproducts $\mathbf{M}$, a symmetric monoidal closed category with coproducts $\mathbf{C}$ and a lax monoidal functor $\mathcal{M} : \mathbf{M} \to \mathbf{C}$.

The denotational semantics is given in Figures 10(Appendix) and 7 and the equational theory is presented in Figures 8 and 9. Due to some categorical subtleties, we also require $\mathbf{M}$ to be distributive in the sense that the monoidal structure must preserve coproducts $A \otimes (B + C) \cong (A \otimes B) + (A \otimes C)$.

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

Proof. By assumption, the functor $A \otimes (-)$ is a left adjoint and, therefore, preserves coproducts and we can conclude the isomorphism $A \otimes (B + C) \cong (A \otimes B) + (A \otimes C)$. □

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

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.*

Thinking from a proof-theoretic perspective, the soundness theorem is basically saying that for every proof of type $\cdot \vdash \tau_1 \otimes \tau_2$, we can assume that the first rule is the introduction rule for

$$\textsc{Axiom}\quad \frac{}{\tau \xrightarrow{id_\tau} \tau} \qquad \otimes\,\textsc{Intro}\quad \frac{\Gamma_1 \xrightarrow{t} \tau_1 \qquad \Gamma_2 \xrightarrow{u} \tau_2}{\Gamma_1 \otimes \Gamma_2 \xrightarrow{t \otimes u} \tau_1 \otimes \tau_2} \qquad \otimes\,\textsc{Elim}\quad \frac{\Gamma_1 \xrightarrow{t} \underline{\tau_1} \otimes \underline{\tau_2} \qquad \Gamma_2 \otimes \underline{\tau_1} \otimes \underline{\tau_2} \xrightarrow{u} \underline{\tau}}{\Gamma_1 \otimes \Gamma_2 \xrightarrow{(id \otimes t);u} \underline{\tau}}$$

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

$$\oplus\,\textsc{Intro}_1\quad \frac{\Gamma \xrightarrow{t} \tau_1}{\Gamma \xrightarrow{t;in_1} \tau_1 + \tau_2} \qquad \oplus\,\textsc{Intro}_2\quad \frac{\Gamma \xrightarrow{t} \tau_2}{\Gamma \xrightarrow{t;in_2} \tau_1 + \tau_2}$$

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

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

Fig. 7. Categorical semantics for λ^2_{INI}: I-layer

$\otimes$. Semantically, we are saying that for every closed term $\cdot \vdash t : \tau_1 \otimes \tau_2$, the semantics $[\![t]\!]$ can be factored as two morphisms f_1 and f_2 such that $[\![t]\!] = f_1 \otimes f_2$.

Establishing soundness requires additional categorical machinery, so we defer the proof to Section 6. Here, we will exhibit a range of concrete models for λ^2_{INI} are all instances of our categorical model.

Remark 5.4 (Adding an additive arrow)**.** Due to reasons mostly related to the models we will work with in this paper we will assume that the language has been extended with sum types at both languages and with an arrow type at the linear language. That being said, we reiterate that, syntactically, it is straightforward to extend λ^2_{INI} with other connectives, such as a non-linear arrow, it is only a matter of finding a concrete model for it. As a rule of thumb, models based on monadic semantics can have an additive arrow as well.

5.2 Concrete models

In this section we will present three concrete models to our language both in the probabilistic case as well as in other effectful cases that escape the monadic landscap. We will also show that our soundness theorem is relevant in other contexts as well.

5.2.1 Discrete Probability. Consider the category **CountStoch** of 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 probability distribution. For the sake of simplicity we will denote its monoidal product of $\times$ [Fritz 2020], even though it is not a Cartesian product.

$$\text{case } (\text{in}_1 M) \text{ of } (|\text{in}_1 x \Rightarrow N_1 | \text{in}_2 x \Rightarrow N_2) \equiv N_1\{x/M\}$$
$$\text{case } (\text{in}_2 M) \text{ of } (|\text{in}_1 x \Rightarrow N_1 | \text{in}_2 x \Rightarrow N_2) \equiv N_2\{x/M\}$$
$$\text{let } y = (\text{let } x = M_1 \text{ in } M_2) \text{ in } M_3 \equiv \text{let } x = M_1 \text{ in } (\text{let } y = M_2 \text{ in } M_3)$$
$$\text{let } x = t \text{ in } x \equiv t$$
$$\text{let } x = x \text{ in } t \equiv t$$

Fig. 8. NI equational theory

$$(\lambda x.\, t)\ u \equiv t\{x/u\} \qquad \text{let } x_1 \otimes x_2 = t_1 \otimes t_2 \text{ in } u \equiv u\{x_1/t_1\}\{x_2/t_2\}$$
$$\text{case } (\text{in}_1 t) \text{ of } (|\text{in}_1 x \Rightarrow u_1 | \text{in}_2 x \Rightarrow u_2) \equiv u_1\{x/t\}$$
$$\text{case } (\text{in}_2 t) \text{ of } (|\text{in}_1 x \Rightarrow u_1 | \text{in}_2 x \Rightarrow u_2) \equiv u_2\{x/t\} \qquad \text{sample } t \text{ as } x \text{ in } x \equiv t$$
$$\text{sample } (\text{sample } t \text{ as } x \text{ in } M) \text{ as } y \text{ in } N \equiv \text{sample } t \text{ as } x \text{ in } (\text{let } y = M \text{ in } N)$$

Fig. 9. I equational theory

Our first concrete model is based on the probabilistic coherence space model of linear logic has been extensively studied in the context of semantics of discrete probabilistic languages [Danos and Ehrhard 2011].

Definition 5.1 (Probabilistic Coherence Spaces [Danos and Ehrhard 2011]). A probabilistic coherence space (PCS) is a pair $(|X|, \mathcal{P}(X))$ where $|X|$ is a countable set and $\mathcal{P}(X) \subseteq |X| \to \mathbb{R}^+$ is a set, called the *web*, such that:

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

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

Definition 5.2. 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 want to define a functor $\mathcal{M}$: **CountStoch** $\to$ **PCoh**. First, we will define a map from countable sets to PCS as follows.

Lemma 5.5. *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.*

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

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

Let $\tilde{\mu} \in \{\mu : X \to \mathbb{R}^+ \mid \sum_{x \in X} \mu(x) \leq 1\}^{\perp\perp}$. By definition, $\sum \tilde{\mu}(x) = \sum \tilde{\mu}(x)\mu_1(x) \leq 1$ and, therefore, the third point holds. □

We define how $\mathcal{M}$ acts on morphisms using the following lemma.

Lemma 5.6. *Let $X \to Y$ be a* **CountStoch** *morphism. It is also a* **PCoh** *morphism.*

Theorem 5.7. *There is a lax monoidal functor $\mathcal{M}$:* **CountStoch** $\to$ **PCoh**.

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

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, the soundness theorem says that they are only denoted by independent probability distributions. This model w as originally used to explore the connections between probability theory and linear logic. Since its creation this model has been used to interpret recursive probabilistic programs, recursive types, and has shown to be full-abstract [Tasson and Ehrhard 2019].

5.2.2 Continuous Probability. The generalization of **CountStoch** that can deal with continuous probabilities is **BorelStoch**, which has standard Borel spaces as objects and Markov Kernels as morphisms.

The category **Meas** can be used to interpret continuous probability, but it can't interpret higher-order functions. However, there are a few models of linear logic that can interpret continuous randomness and higher-order functions. We choose to use the model based on perfect Banach lattices.

Definition 5.3 ([Azevedo de Amorim and Kozen 2022]). The category $\mathbf{PBanLat_1}$ has perfect Banach lattices as objects and order-continuous linear functions with norm ≤ 1 as morphisms.

Theorem 5.8. *There is a lax monoidal functor $\mathcal{M}$:* **BorelStoch** $\to$ $\mathbf{PBanLat_1}$.

Proof. The functor acts on objects by sending a measurable space to the set of signed measures over it, which can be equipped with a $\mathbf{PBanLat_1}$ structure. On morphisms it sends a Markov kernel f to the linear function $\mathcal{M}(f)(\mu) = \int f d\mu$.

The monoidal structure of $\mathbf{PBanLat_1}$ satisfies the universal property of tensor products and, therefore, we can define the natural transformation $\mu_{X,Y} : \mathcal{M}(X) \otimes \mathcal{M}(Y) \to \mathcal{M}(X \times Y)$ as the function generated by the bilinear function $\mathcal{M}(X); \mathcal{M}(Y) \multimap \mathcal{M}(X \times Y)$ which maps a pair of distributions to its product measure. The map ϵ is, once again, equal to the identity function.

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

This model can be seen as the continuous generalization of the previous model, since there is are full and faithful embeddings **CountStoch** $\hookrightarrow$ **BorelStoch** and **PCoh** $\hookrightarrow$ $\mathbf{PBanLat_1}$ [Azevedo de Amorim and Kozen 2022].

In this context, the soundness theorem should once again be interpreted as probabilistic independence, i.e. programs of type $\mathcal{M}\tau_1 \otimes \mathcal{M}\tau_2$ are denoted by independent distributions.

5.2.3 Non-Determinism and Communication. Imagine that we want to program a system that might run in several computers concurrently and guarantee local reasoning, i.e. we can reason equationally about an individual program without worrying about the code it communicates with. If we assume that each program is pure and that communication is perfect then this is straight forward to do. However, if we assume that communication might be faulty – a message might drop, for instance – or that the programs being run are effectful then this becomes more complicated.

Suppose that we have two languages: one for writing local programs and a second one to orchestrate the communication between local code. We claim that λ^2_{INI} provides the right abstraction for this situation, where $\mathcal{M}\tau$ corresponds to local computations which can be manipulated by the communication language. To align the syntax with this new interpretation we change *sample* 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 continue as the local program M.

For the concrete semantics, we will assume that local programs may be non-deterministic, to account for messages that might be dropped. Therefore, we choose the Kleisli category for the powerset monad as our CD category and the linear category **Rel**, a well-known model of classical linear logic.

In this context, the soundness result is saying that if we have a closed program of type $\mathcal{M}\tau \otimes \mathcal{M}\tau$ then it can be factored into two local programs which can be locally reasoned about and do not require any extra communication other than what was explicitly programmed.

This approach to programming with communication is reminiscent of session types and choreographic programming. Session types consist of linearly-typed languages which can be used to specify and program communication protocols with explicit communication. One of their meta theoretic guarantees is that well-typed programs will never deadlock. Choreographic programming is a monolithic approach to distributed computation. The programmer writes the entire system in a single program which can be compiled (projected, as it is used in the literature) into several local programs with explicit communication. They also guarantee deadlock-freedom and keep the invariant that the projection function is well-defined while not enforcing a linear typing discipline.

We see our language as a sort of compromise between both approaches. Though we require communication to be linear, the modality $\mathcal{M}$ allows to safely encapsulate non-linear computations. Our soundness theorem can be seen as a kind of existence of projection functions from the choreography literature.

It is an interesting research question that goes well beyond the scope of this paper to understand how these approaches are related. With the introduction of higher-order choreographies it seems like our approach is overly conservative since our soundness theorem is also valid for, say, programs of type $(\tau \multimap \tau) \otimes \tau$, not only programs of type $\mathcal{M}\tau \otimes \mathcal{M}\tau$, which are the only types that should matter when talking about being able to project into local programs.

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.

A reason why commutative monads are useful is because their Kleisli category have useful properties:

Theorem 5.9 ([Fritz 2020]). *Let* $\mathbf{C}$ *be a Cartesian category and* T *a commutative monad over it. The category* $\mathbf{C}_T$ *is a CD category.*

Lemma 5.10. *Let* $\mathbf{C}$ *be a distributive category and* T *a monad over it. Its Kleisli category* $\mathbf{C}_T$ *has coproducts and is also distributive.*

Proof. It is straightforward to show that Kleisli categories inherit coproducts from the base category. Furthermore, by using the distributive structure of $\mathbf{C}$, applying T to it and using the functor laws, it follows that $\mathbf{C}_T$ is distributive. □

Another useful category of algebras is the category of algebras and plain maps $\widetilde{\mathbf{C}^T}$ which has T algebras as objects and $\widetilde{\mathbf{C}^T}((A, f), (B, g)) = \mathbf{C}(A, B)$.

Theorem 5.11 ([Simpson 1992]). *Let* $\mathbf{C}$ *be a Cartesian category and* T *a commutative monad over it. The category of* T*-algebras and plain maps is Cartesian closed.*

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 5.12. *There exists an applicative functor* $\mathbf{C}_T \to \widetilde{\mathbf{C}^T}$.

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 $\eta_1 : 1 \to T1$. It is possible to show that this corresponds to an applicative functor by using the fact that T is commutative and that the comonoid structure $A \to 1$ is natural. □

Something which needs further clarification is what is the intuitive interpretation for the sample x as t in M construct. Originally, 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. In this context, the I language should be seen as a CBN interpretation of effects while NI should be seen as a CBV interpretation of effects. Therefore, we rename "Sample" to "Force" and its operational interpretation is forcing the execution of CBN computations t_i, binding the results to x_i and running them in an eager setting.

As a concrete example, the name generation monad, which is used to give semantics to the ν-calculus, a language that has a primitive that generates a "fresh" symbol. 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. As such, enforcing the separability of names used is useful when reasoning about security property of programs.

A concrete semantics to the ν-calculus was presented by [Stark 1996] where the base category is the functor category $[\mathbf{Inj}, \mathbf{Set}]$, where $\mathbf{Inj}$ is the category of finite sets and injective functions. In this case the name generation monad acts on functor as

$$T(A)(s) = \{(s', a') \mid s' \in \mathbf{Inj}, a' \in S(s + s')\}/\sim$$

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 $A(id_s + f_1)a_1 = A(id_s + f_2)a_2$. This may seem hard to follow, but the intuition is that $T(A)$ is a computation that, given a finite set of names used s, produces a distinct set of names s' and a value a'.

In the context of name generation, the soundness theorem says that in a program of type $\mathcal{M}\tau \otimes \mathcal{M}\tau$, the names used in the first component are *disjoint* from the ones used in the second component.

It is also possible to define a variant to this algebra model using the Eilenberg Moore category since, under a few conditions it is symmetric monoidal closed.

5.2.5 Affine Bunched Typing. Separation logic originated as a tool to reason about heap-manipulating programs. This is achieved by adding the connectives $*, -\!*$ which are valid under the assumption that the part of the heap they are modifying are disjoint.

From a programming perspective this logic gave rise to bunched typing languages, where contexts are defined using trees as opposed to lists. Concretely, there are two context concatenation operations Γ, Γ and $\Gamma; \Gamma$. The first operation means that the two contexts are independent whereas the second one means that they might share resources.

From a syntactic point of view this framework provides an answer to the problem we are solving in this paper. Therefore, it is essential to understand how these approaches are related. From a semantic perspective, bunched calculi are interpreted using a Cartesian category that also has a monoidal product. This difference has practical consequences since, by assuming a single language, it is possible to layer the connectives $\times$ and $\otimes$ to create intricate independency structures. This contrasts with our language where the two layer system only allows to create dependencies of the form $\mathcal{M}(\tau \times \cdots \times \tau) \otimes \cdots \otimes \mathcal{M}(\tau \times \cdots \times \tau)$.

That being said, in practice, concrete models for the bunched calculus are hard to come by, usually requiring to work with presheaf categories. Indeed, none of the models presented so far are models of the bunched calculus, even though they are standard models of effectful programming languages. In order to further understand how these systems are related, let us consider the affine variant of the bunched calculus, i.e. the monoidal unit is a terminal object and there is a discard operation $A \otimes B \to A$. In this case one can define the lax monoidal functor $id : (\mathbf{C}, \times, 1) \to (\mathbf{C}, \otimes, I)$ which maps every object and morphism to itself. This allows us to conclude

Theorem 5.13. *Every model of affine BI gives rise to a model of* λ^2_{INI}

Remark 5.14. From a more abstract point of view, by initiality of the syntactic model (Theorem B.1), there is a translation from our language to the bunched calculus, showing that affine bunched calculi can be seen as a degenerate model of our language when the two languages are collapsed.

To illustrate a useful model of the bunched calculus, let us consider Reynolds' system for syntactic control of interference control (SCI). Using modern terminology, Reynolds defined an affine λ-calculus that can enforce the non-aliasing of local state.

Its denotational semantics was defined by O'Hearn [1993] and consists of 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 (memory addresses). The Cartesian closed structure is given by the usual definition on presheafs. The monoidal closed structure is given by the Day convolution product, see the original paper for more details on these definitions.

In this context, our soundness theorem implies that programs of type $\tau_1 * \tau_2$ do not share local state and, therefore, there can be no aliasing of memory locations.

6 SOUNDNESS THEOREM

Now, we turn to proving our general soundness theorem for λ^2_{INI}, Theorem 5.3. 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 led to a quick proof, but the strategy does not generalize to other models of λ^2_{INI}.

Thus, we will return to logical relations. Indeed 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].

For the general case, 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 context of our applications, 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.

In the abstract setting, another key insight is the fact that the syntactic model is initial in the category of models. This is what allows us to conclude that the glued category preserves the extensional behavior of the semantics. We are interested in proving the theorem

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* $f_2 : I \to \mathcal{M}\,[\![\tau_2]\!]$.

Which in prove in full in Appendix B

7 RELATED WORK

Linear logics and probabilistic programs. A recent line of uses linear logic as a powerful framework to provide semantics for 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; probabilistic versions of call-by-push-value have also been developed [Tasson and Ehrhard 2019]. 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].

As we have mentioned, our categorical model for λ^2_{INI} is inspired by models of linear logic based on monoidal adjunctions, most notably Benton's LNL [Benton 1994]. From a programming languages perspective, these models decompose the linear λ-calculus with exponentials in two languages with distinct product types each: one is a Cartesian product and the other is symmetric monoidal. The adjunction manifests itself in adding functorial type constructor in each language, similar to our $\mathcal{M}$ modality. These two-level languages are very similar to λ^2_{INI}, and indeed it is possible to show that every LNL model is a λ^2_{INI} model. At the same time, the class of models for λ^2_{INI} is much broader than LNL—none of the models presented in Section 5.2 are LNL models.

Higher-order programs and effects. There is a very large body of work on higher-order programs effects, which we cannot hope to summarize here. The semantics of λ_{INI} is an instance of Moggi's Kleisli semantics, from his seminal work on monadic effects [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 [2022], which uses an applicative modality to guarantee that the linearity restriction is only valid for computations, not values. We consider a more general class of categorical models, and we investigate the role of sum types.

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 work, BI also features two implications, one for each conjunction. The leading application of BI is in separations logic for concurrent and heap-manipulating programs [O'Hearn 2007; O'Hearn et al. 2001], where pre- and post-conditions are drawn from BI.

Most applications of BI use a truth-functional, Kripke-style semantics. semantics [Pym et al. 2004]. By considering the proof-theoretic models of BI, O'Hearn [2003] developed a bunched type system for a higher-order language. Its categorical semantics is given by a *doubly closed category*: a Cartesian closed category with a separate symmetric monoidal closed structure. While O'Hearn [2003] showed different models of this language for reasoning about sharing and separation in heaps, it has been difficult to find other concrete models that satisfy the requirements. It is not clear how to incorporate effects into the bunched type system; in contrast, our models can reason about a wide class of monadic effects.

There are natural connections to both of our languages. Our language λ_{INI} resembles O'Hearn's system, with two differences. First, λ_{INI} only has a multiplicative arrow, not an additive arrow—as we described in Section 3, it is not clear how to support an additive arrow in λ_{INI} without breaking our primary soundness property. Second, contexts in λ_{INI} are flat lists, not tree-shaped bunches. It would be interesting to use bunched contexts to represent more complex dependency relations. Our stratified language λ^2_{INI} is also similar to O'Hearn's system. Though our categorical model only has a single multiplicative arrow, in the I-layer, many (but not all) of our concrete models also support an additive arrow, in the NI-layer. At the same time, it is not clear how the two sum types in λ^2_{INI} would function in a bunched type system.

Probabilistic independence in higher-order languages. There are a few probabilistic functional languages with type systems that model probabilistic independence. Probably the most sophisticated example is due to Darais et al. [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], 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 versions of separation logics for probabilistic programs [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.

Bunched type systems for independence. Our work bears many similarity to work on bunched logics; most notably, bunched logics feature an additive and a multiplicative conjunction. While bunched logics have found strong applications in Hoare-style logics, the only bunched type system we are aware of is due to O'Hearn [2003]. This language features a single layer with two product types and also two function types, and the typing contexts are tree-shaped bunches, rather than flat lists. Developing a probabilistic model for a language with a richer context structure would be an interesting avenue for future work.

Non-commutative effects. Our concrete models encompass many kinds of 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. 2022. A Sampling-Aware Interpretation of Linear Logic: Syntax and Categorical Semantics. *arXiv preprint arXiv:2202.00142* (2022).

Pedro H Azevedo de Amorim and Dexter Kozen. 2022. Classical Linear Logic in Riesz 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.

G. E. P. Box and Mervin E. Muller. 1958. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics* 29, 2 (1958), 610 – 611. DOI: http://dx.doi.org/10.1214/aoms/1177706645

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.

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.

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

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

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. 1993. A Model for Syntactic Control of Interference. *Math. Struct. Comput. Sci.* 3, 4 (1993), 435–465. DOI: http://dx.doi.org/10.1017/S0960129500000311

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.

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

A CATEGORICAL SEMANTICS

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

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

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

$$\times\,\textsc{Elim}_1\quad \frac{\Gamma \xrightarrow{M} \tau_1 \times \tau_2}{\Gamma \xrightarrow{M;(id_{\tau_1} \times del)} \tau_1}$$

$$\times\,\textsc{Elim}_2\quad \frac{\Gamma \xrightarrow{M} \tau_1 \times \tau_2}{\Gamma \xrightarrow{M;(id_{\tau_2} \times del)} \tau_2}$$

$$+\,\textsc{Intro}_1\quad \frac{\Gamma \xrightarrow{M} \tau_1}{\Gamma \xrightarrow{M;in_1} \tau_1 + \tau_2}$$

$$+\,\textsc{Intro}_2\quad \frac{\Gamma \xrightarrow{M} \tau_2}{\Gamma \xrightarrow{M;in_2} \tau_1 + \tau_2}$$

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

Fig. 10. Categorical semantics for λ^2_{INI}: NI-layer

B SOUNDNESS PROOF (CONT.)

B.1 Category of Models

A model for λ^2_{INI} is given by a CD category $\mathbf{M}$ with coproducts, a symmetric monoidal closed category (SMCC) $\mathbf{C}$ with coproducts and a lax monoidal functor $\mathcal{M} : \mathbf{M} \to \mathbf{C}$. A morphism between two models $(\mathbf{M}_1, \mathbf{C}_1, \mathcal{M}_1)$ and $(\mathbf{M}_2, \mathbf{C}_2, \mathcal{M}_2)$ is a pair of functors $(F : \mathbf{M}_1 \to \mathbf{M}_2, G : \mathbf{C}_1 \to \mathbf{C}_2)$ that preserves the logical connectives.

If we define composition component-wise, it is possible to define a category **Mod** of models of the language. We want to show that the syntactic category is the initial object of **Mod**.

Concretely, the category **Syn** has types as objects, and typing derivations modulo the equational theory presented in Figure 9 and Figure 8 as morphisms.

Theorem B.1. **Syn** *is the initial object of* **Mod**

Proof. Let $(\mathbf{C}, \mathbf{M}, \mathcal{M})$ be a model. The functor $[\![\cdot]\!] : \mathbf{Syn} \to (\mathbf{C}, \mathbf{M}, \mathcal{M})$ is defined by two functors $[\![\cdot]\!]_1$ and $[\![\cdot]\!]_2$. It is possible to define their action on objects by induction on the types. In order to define the action on morphisms we proceed by induction on the typing derivation. There is a subtlety in this definition because the morphisms of the components of **Syn** are typing derivations modulo the equational theory of the language, meaning that we need to quotient the definition above.

Since, by definition of model, the construction above is invariant with respect with the equational theory, it is well-defined.

To prove uniqueness we assume the existence of two semantics and show, by induction on the typing derivation, that they are equal. ◻

B.2 Glued category

Next we need 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. The functor $\mathcal{M} : \mathbf{M} \to \mathbf{C}$ is lifted to a functor $\widetilde{\mathcal{M}} : \mathbf{C} \to \mathbf{Gl}(\mathbf{C})$ by mapping objects X to $(\mathcal{M}\, X, \mathbf{C}(I, \mathcal{M}\, X))$[1] and by mapping morphisms f to $\mathcal{M}\, f$. 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.

With this in mind, we can start by showing that $\mathbf{Gl}(\mathbf{C})$ is SMCC and has coproducts.

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

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.

Theorem B.2. $\mathbf{Gl}(\mathbf{C})$ *is SMCC.*

Proof. 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 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 universal property of sum types. Let $f_1 : A_1 \to B$ and $f_2 : A_2 \to B$ be $\mathbf{Gl}(\mathbf{C})$ morphisms. Consider the $\mathbf{C}$ morphism $[f_1, f_2]$. We want to show that this morphism is also a $\mathbf{Gl}(\mathbf{C})$ morphism. Consider $g \in X_{A_1 \oplus A_2}$ which, by assumption, $g = in_1 g_1$ or $g = in_2$. By case analysis and the facts $f_i \circ g_i \in Y$ and $[f_1, f_2] \circ in_i g_i = f_i \circ g_i$ we can conclude that $[f_1, f_2]$ is indeed a $\mathbf{Gl}(\mathbf{C})$ morphism.

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 show that $\widetilde{\mathcal{M}}$ is lax monoidal which follows from the fact that μ and ϵ preserve the plot sets, by a simple unfolding of the definitions. We can now easily conclude that the lax monoidality diagrams commute, since composition is the same and $\mathcal{M}$ is lax monoidal.

We can now prove the theorems:

Theorem B.3. *The triple* $(\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}})$ *is a* **Mod** *object.*

Theorem B.4. *There are* **Mod** *morphisms* $(\!|\cdot|\!) : \mathbf{Syn} \to (\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}})$ *and* $U : (\mathbf{M}, \mathbf{Gl}(\mathbf{C}), \widetilde{\mathcal{M}}) \to (\mathbf{M}, \mathbf{C}, \mathcal{M})$.

With this, we may now construct a functor $U \circ (\!|\cdot|\!) : \mathbf{Syn} \to (\mathbf{M}, \mathbf{C}, \mathcal{M})$ which, by initiality of $\mathbf{Syn}$, is equal to the functor $[\![\cdot]\!]$, as illustrated by Figure 11.

[1]Note that its predicate set is every $\mathbf{C}$ morphism $I \to \mathcal{M}\, X$, similar to how ground types are interpreted in usual logical relations proofs

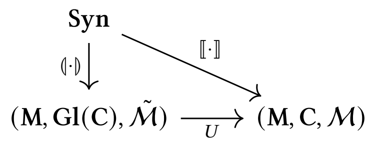

Fig. 11. The essence of the soundness proof

Summarizing the construction above, the sets $X_{\underline{\tau}}$ are interpreted as the objects

$$\begin{aligned}
X_{\mathcal{M}\tau} &= \mathcal{C}(I, \mathcal{M}\tau) \\
X_{\underline{\tau_1}\otimes\underline{\tau_2}} &= \{f_1 \otimes f_2 \mid f_1 \in X_{\underline{\tau_1}}, f_2 \in X_{\underline{\tau_2}}\} \\
X_{\underline{\tau_1}\multimap\underline{\tau_2}} &= \{f : I \longrightarrow (\underline{\tau_1} \multimap \underline{\tau_2}) \mid \forall f_{\underline{\tau_1}} \in X_{\underline{\tau_1}}, \epsilon_{\underline{\tau_2}} \circ (f_{\underline{\tau_1}} \otimes f) \in X_{\underline{\tau_2}}\} \\
X_{\underline{\tau_1}\oplus\underline{\tau_2}} &= \{\mathsf{in}_i\, f_i \mid f_i \in \underline{\tau_i}\}
\end{aligned}$$

B.3 General Soundness Theorem

Theorem B.5. *If* $\cdot \vdash_I t : \tau$*, then* $[\![t]\!] \in X_\tau$.

Proof. We know that $[\![\cdot]\!] = U \circ (\!|\cdot|\!)$ and that $(\!|t|\!)$ is a $\mathbf{Gl}(\mathbf{C})$ morphism. As such we have that $(\!|t|\!) = (\!|t|\!) \circ id_I \in X_\tau$. □

Theorem 5.3 follows immediately, as a corollary.

Corollary B.6. *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* $f_1 : I \longrightarrow \mathcal{M}\,[\![\tau_1]\!]$ *and* $f_2 : I \longrightarrow \mathcal{M}\,[\![\tau_2]\!]$.

Proof. By Theorem B.5, 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 $f_1 : I \longrightarrow \mathcal{M}\,[\![\tau_1]\!]$ and $f_2 : I \longrightarrow \mathcal{M}\,[\![\tau_2]\!]$ such that $[\![t]\!] = f_1 \otimes f_2$. □