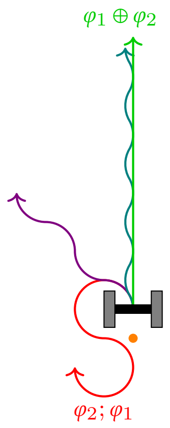
Mixture Languages

1 MOTIVATION

There is an important distinction between a process whose state is continuous, and one that evolves continuously over time. There are many general-purpose programming abstractions for modeling processes with continuous state (e.g., real-valued variables and probabilistic programs), but programming languages for modeling of continuous time processes are quite domain-specific. This is because the correctness of such computations is not considered a matter of computer science—after all, the correctness of a physics engine is a matter of physics, and the correctness of an animation is a matter of taste. Yet continuous-time processes of every kind compose in the same (surprisingly unfamiliar) way: they can be "mixed". This is different from the dominant model of concurrency, in which parallel composition amounts to a non-deterministic interleaving of atomic instructions and is notoriously unintuitive. But for two continuous processes, concurrency has a single clear and intuitive meaning: run them at the same time. Roughly speaking, this corresponds to a uniform infinitesimal interleaving.



To illustrate, suppose we are writing code to control a (simulated) robot, in which φ_1 represents five rotations of the left wheels, and φ_2 represents five rotations of the right ones. If we execute φ_1 and φ_2 concurrently, one might reasonably expect the robot to go straight forwards—but this is technically impossible according to the conventional model of concurrent computation, because no matter how we interleave φ_1 and φ_2, only one wheel moves at a time. More disturbingly, the dominant model of concurrency regards close approximations to forward motion, such as rapid alternation between right and left, as no more appropriate than completing all five rotations of the right wheels before moving on to the left ones. This example illustrates an important point: mixing the *results* of executing φ_1 and φ_2, can be quite different from executing a mixture of φ_1 and φ_2 themselves.

Discrete computations, too, can be given continuous semantics, and often to positive effect. When we embed a discrete-state representation in a continuous one, we gain the ability to mix program states (such as convex relaxations of integer programs, and the embedding of deterministic components inside randomized programs). Analogously, when we regard a discrete-time process as a subset of a continuous one, we gain the ability to *mix programs.* Doing so can lead to insights into, and natural generalizations of, the original discrete-time processes, as show in Section 3 through an example of probabilistic programs.

In the coming sections, we develop a programming language with semantics that supports continuous partial execution. In brief, a command can be executed to a degree $s \in [0, 1]$ where $s = 0$ corresponds to a no-op, and $s = 1$ fully executes the command. Working with continuous executions allows us to define a natural and interesting notion of parallel composition. Furthermore, the resulting programs are typically invertable and differentiable. Our construction is parametric on a set Φ of primitive commands.

2 THE PROGRAMMING LANGUAGE

2.1 Syntax

Suppose we are given a set Φ of primitive commands. Our basic language $\mathcal{L}(\Phi)$ includes programs inductively constructed through sequential composition (;), parallel composition ($\oplus$), clipping, i.e., the partial execution of commands to a degree $s \in [0, 1]$.

$$\mathcal{L}(\Phi) \ni \varphi, \varphi' \quad ::= \quad \text{skip} \quad \big| \quad \phi \quad \big| \quad \varphi; \varphi' \quad \big| \quad \varphi \oplus \varphi' \quad \big| \quad \varphi^{(s)},$$

for values $s \in [0, 1]$, and $\phi \in \Phi$.

Once we include boolean expressions b, we can further extend our language to include familiar control flow structures

$$\mathcal{L}^{\text{cf}}(\Phi) \ni \psi, \psi' \quad ::= \quad \varphi \quad \big| \quad \text{if } b \text{ then } \psi \text{ else } \psi' \quad \big| \quad \text{while } b : \psi,$$

where $\varphi \in \mathcal{L}(\Phi)$ as before.

2.2 Operational Semantics: Overview

We begin by describing four closely related formalisms for describing the operational behavior of a program $\varphi \in \mathcal{L}(\Phi)$. Let Θ denote set of possible program states. Each variant of the semantics requires more structure (topological, differentiable, metric) on Θ.

Endpoint Semantics. The traditional state-transformer style semantics for a program φ is a map $[\![\varphi]\!]^* : \Theta \to \Theta$ such that executing φ starting in state $\theta \in \Theta$ leads us to the final program state $[\![\varphi]\!]^*(\theta)$. We call this the *endpoint semantics* of φ. If $[\![\varphi]\!]^* = [\![\varphi']\!]^*$, then we say that φ and φ' are *endpoint-equivalent.*

While the endpoint semantics fully describes what state a program would return upon each input, it does not contain enough information to inductively define the semantics of clipped programs: in order to determine the result of partially executing φ, the semantics must describe not only the final state after executing the instruction, but also the path it took to get there.

Path Semantics. Now suppose that Θ is not merely a set, but a also a topological space of program states. This would allow us to is to interpret $\varphi \in \mathcal{L}(\Phi)$ as a path in spate space, i.e., a continuous map $\rlap{/}P[\![\varphi]\!]$ of type $[0, 1] \times \Theta \to \Theta$, whose first argument $s \in [0, 1]$ specifies a proportion of the way through program execution. Intuitively, it describes the evolution of the system. To think of it as paramterized by time, it is more useful to make the change of variables $t = -\log(1 - s) \in [0, \infty]$; we call this temporal version a *trajectory*, rather than a path. This semantics has the property that $s = 0$ $(t = 0)$ leaves the state unchanged $(\forall \theta \in \Theta.\ \rlap{/}P[\![\varphi]\!](0, \theta) = \theta)$, and $s = 1$ $(t = \infty)$ corresponds to the endpoint semantics.

While the information in this semantics allows us to partially execute (clip) programs, it is still unclear how to inductively define the semantics of mixture commands based on it. Consider again the robot: given only the path taken when either wheel moves by itself, it is not obvious that we can construct the appropriate path when they move together—and the problem becomes harder in the presence of walls and obstacles. What should we do, for example, if our mixture takes us to a point (s, θ) that is not on any path? To get around this, we will need a closely related but slightly stronger semantics that contains this counter-factual information.

Vector Field Semantics. Now, suppose Θ is a manifold (with corners). A *vector field* over Θ, is a differentiable map X assigning to each point $\theta \in \Theta$ a tangent vector at θ. The set of vector fields over Θ, denoted $\mathfrak{X}(\Theta)$, forms a vector space, and so we can add them together; this will be the basis of our mixture operation. Concretely, we interpret programs $\varphi \in \mathcal{L}(\Phi)$ as maps $\mathfrak{X}[\![\varphi]\!] : [0,1] \to \mathfrak{X}(\Theta)$, where the first parameter indicates the proportion through program execution, just as in the path semantics. We take this vector field semantics to be fundemental, and derive the path and endpoint semantics from it. Given $\mathfrak{X}[\![\varphi]\!]$, the path semantics is given by $\mathcal{f}[\![\varphi]\!](s,\theta_0) := \vartheta(s)$, where $\vartheta(s)$ is the (unique) solution to the ODE

$$\vartheta(0) = \theta_0; \qquad \frac{\mathrm{d}}{\mathrm{d}s}\vartheta(s) = \mathfrak{X}[\![\varphi]\!](s)(\vartheta(s)). \tag{1}$$

To write the semantics in terms of t rater than s at the level of vector fields, then not only must we substitute $1 - e^{-t}$ for s, but we must also apply the chain rule, multiplying by $\frac{\mathrm{d}t}{\mathrm{d}s} = \frac{1}{1-s}$. We call φ *autonomous* if $\mathfrak{X}[\![\varphi]\!]$ does not depend on t. This means $\mathfrak{X}[\![\varphi]\!](s,\theta) = X(\theta)/(1-s)$ for some vector field X.

Example 2.1. Suppose $\Theta = [0,1]$, and φ is the instruction that moves linearly to state 0, i.e., $\mathcal{f}[\![\varphi]\!](s,\theta_0) = \theta_0(1-s) = \theta_0 e^{-t}$. The constraint (1) states that, for all θ_0, we must have $\mathfrak{X}[\![\varphi]\!](s,\theta_0(1-s)) = -\frac{\partial}{\partial s}\theta_0(1-s) = -\theta_0$. Thus, for $\theta \leq 1-s$, we can choose $\theta_0 = \frac{\theta}{1-s}$ to determine that $\mathfrak{X}[\![\varphi]\!](s,\theta) = -\theta/(1-s)$. At the same time, it is easy to see that the vector field corresponding to the ODE $\frac{\mathrm{d}\vartheta}{\mathrm{d}t} = -\vartheta$ leads to the same path, but does not depend on t.

Loss Function Semantics. Finally, suppose that Θ is not only a manifold, but also comes equipped with a Riemannian metric. This is the minimal requirement needed to view the gradient of a scalar function as a vector field. Thus, if we can interpret an instruction φ as a $\ell[\![\varphi]\!] : \Theta \to \mathbb{R}$, then we can define the autonomous command $\mathfrak{X}[\![\varphi]\!](s,\theta) := \frac{1}{1-s}\nabla \ell[\![\varphi]\!](\theta)$.

2.3 Operational Semantics: Construction

For primitive commands, sometimes it makes sense to supply a vector field directly (e.g, if ϕ represents "go left"), and other times it makes more sense to specify a loss function, or a path semantics. Some care is needed to make sure everything is well-defined. There is much to be said on this topic, but we do not say it here.

Given a vector field semantics $\mathfrak{X}[\![\phi]\!]$ for primitive commands $\phi \in \Phi$, we now define the vector field semantics for $\mathcal{L}(\Phi)$ inductively. For parallel composition, we simply add the fields together. Formally, define

$$\begin{aligned}\mathfrak{X}[\![\mathrm{skip}]\!](s) &:= 0\\ \mathfrak{X}[\![\varphi_1 \oplus \varphi_2]\!](s) &:= \mathfrak{X}[\![\varphi_1]\!](s) + \mathfrak{X}[\![\varphi_2]\!](s)\\ \mathfrak{X}[\![\varphi^{(c)}]\!](s) &:= \mathfrak{X}[\![\varphi]\!](cs).\end{aligned}$$

What remains is sequential composition. The following is the standard way of composing paths in the development of homotopy [Hatcher 2002], translated to the vector field semantics.

$$\mathfrak{X}[\![\varphi_1;\varphi_2]\!](s) := \begin{cases} 2\cdot\mathfrak{X}[\![\varphi_1]\!](2s) & s \leq \frac{1}{2}\\ 2\cdot\mathfrak{X}[\![\varphi_2]\!](2s-1) & s > \frac{1}{2}\end{cases}$$

Intuitively, we would like to first travel along φ_1's path and then φ_1's. To do so, we give both half the interval, and move double the speed. Unfortunately, this definition is not associative—although the paths of $\varphi_{12;3} := (\varphi_1;\varphi_2);\varphi_3$ and $\varphi_{1;23} := \varphi_1;(\varphi_2;\varphi_3)$ pass through the same states, they are parameterized differently. This is why topologists only consider path composition up to homotopy equivalence. For us, one analogue is to consider only the endpoint semantics of purely sequential programs. Sequential composition is still associative in the endpoint semantics, so long as we do not also make use of mixtures and clipping.

We defer the semantics of conditionals to Appendix B.

3 KEY APPLICATION: OBSERVE AND DRAW

Probabilistic programming literature divides between those work with an imperative language extended with a sampling command and those work with an additional observe command for conditioning on the distribution. We show a unexplored duality between the sampling and the (generalized) observe command, in a continuous-time semantics interpolating probabilistic programs.

Let $\mathcal{X}$ be a fixed set of variables. Define two sets Φ_{obs} and Φ_{draw} of primitive commands, consisting of all statement of the form "**observe** $p(Y|X)$", and "**draw** $Y \sim p|X$" respectively, where $X, Y \subseteq \mathcal{X}$ and p specifies a probability measure over Y for each value of X. Let Z denote the set of variables apart from X and Y.

Take Θ to be the set of probability measures over joint settings of $\mathcal{X}$, whose metric structure is given by the Fisher information.[1] The entropy of a measure ν relative to μ, defined as $\boldsymbol{D}(\mu \parallel \nu) := \mathbb{E}_\mu[\log\frac{\mu}{\nu}]$, is the excess cost of using codes optimized for ν when reality is distributed according to μ. The roles of ν and μ are not symmetric. We give loss semantics to **observe** by having p play the role of belief, and to **draw** by having it play the role of reality.

$$\begin{aligned}\ell\left[\!\!\left[\textbf{observe } p(Y|X)\right]\!\!\right](\mu) &:= \boldsymbol{D}\Big(\mu(X,Y)\ \Big\|\ \mu(X)\cdot p(Y|X)\Big)\\ \ell\left[\!\!\left[\textbf{draw } Y\sim p|X\right]\!\!\right](\mu) &:= \boldsymbol{D}\Big(\mu(X,\mathrm{Z})\cdot p(Y|X)\ \Big\|\ \mu(X,Y,\mathrm{Z})\Big)\end{aligned}$$

From it, we derive the path semantics (see appendix C)

$$\begin{aligned}&\mathcal{f}\left[\!\!\left[\textbf{observe } p(Y|X)\right]\!\!\right](s,\mu) \propto \mu\cdot\left(\frac{p(Y|X)}{\mu(Y|X)}\right)^s, \quad \text{and} \qquad (2)\\ &\mathcal{f}\left[\!\!\left[\textbf{draw } Y\sim p|X\right]\!\!\right](s,\mu) = \mu(X,\mathrm{Z})\Big((1-s)\mu(Y|X,\mathrm{Z}) + (s)p(Y|X)\Big).\end{aligned}$$

Intuitively, **observe** instructions interpolate multiplicatively, filtering μ so that it has the desired conditional probabilities. **draw** instructions, on the other hand, interpolate additively. For example, $\mathfrak{X}[\![\textbf{draw } Y\sim p \oplus \textbf{draw } Y \sim q]\!] = \mathfrak{X}[\![\textbf{draw } Y \sim \frac{1}{2}p + \frac{1}{2}q]\!]$.

PROPOSITION 3.1. *Let* PROB *denote the fragment of probabilistic programming language in [Kozen 1979; Staton 2020] consists of skip, sampling (i.e.,* ***draw*** *), observe and sequencing. The semantics of* PROB *coincides with the endpoint semantics of those commands in $\mathcal{L}(\Phi_{draw})$. Formally, $\forall\varphi \in$* PROB*, $[\![\varphi]\!] = [\![\varphi]\!]^*$.*

PROPOSITION 3.2. *The semantics of probabilistic graphical models (including Bayesian Networks and Markov Random Fields), are the endpoint semantics of the appropriate mixture of* ***observe*** *commands.*

Examples in ML and game theory can be found in Appendix A.

[1]by Chentsov's Theorem [Chentsov 1982], this is the unique metric on Θ that is invariant under sufficient statistics, and and the gradient it induces is sometimes known as the *natural gradient*, a cornerstone of information geometry [Nielsen 2020].

REFERENCES

Nikolai N Chentsov. 1982. Statistical Decision Rules and Optimal Inference. *American Mathematical Society, Providence, Rhode Island* (1982).

Allen Hatcher. 2002. *Algebraic Topology.* Cambridge University Press.

Dexter Kozen. 1979. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979).* IEEE, 101–114.

Frank Nielsen. 2020. An elementary introduction to information geometry. *Entropy* 22, 10 (2020), 1100.

Oliver E Richardson and Joseph Y Halpern. 2021. Probabilistic Dependency Graphs. *AAAI '21* (2021). arXiv:2012.10800 [cs.AI]

Sam Staton. 2020. Probabilistic programs as measures. *Foundations of Probabilistic Programming* (2020), 43.

A FURTHER APPLICATIONS AND EXAMPLES

Training a Network. Consider a binary classifier for examples $x \in X$, parameterized by a neural network whose final layer is a softmax. In this case, we have a function $f_\theta : X \to [0,1]$ for each $\theta \in \mathbb{R}^n$. Take $\Theta := \mathbb{R}^n$ to be possible parameter settings of the network, and $\Phi := X \times \{0,1\}$ to be the set of possible labeled examples. When $t \approx 0$ is small, $[\![(x,y)^{(t)}]\!]^*$ computes a single iteration of SGD with batch $\{(x,y)\}$, with learning rate t. Similarly, $[\![((x_1,y_1)\oplus\cdots\oplus(x_m,y_m))^{(t)}]\!]^*$ computes a single iteration of SGD with batch $\{(x_i,y_i)\}_{i=1}^m$, In this way, training with the full dataset gradient is a mixture of training on all data $\mathcal{D} = \{(x_i,y_i)\}_{i=1}^n$ at once, simultaneously.

Furthermore, $\theta^* := [\![\bigoplus \mathcal{D}]\!]^*(\theta)$ is the limiting weights of the network at convergence, when trained on dataset $\mathcal{D}$, and $\oint[\![\bigoplus \mathcal{D}]\!](\theta)$ is the idealized training curve given by gradient flow. In general, a program $\varphi \in \mathcal{L}(\Phi)$ can represent training schedules, which can be sequenced (;), superimposed ($\oplus$), reweighted (by scaling), or used only in certain contexts (**if**/**then**). If Θ carries parameters for multiple networks, we can describe for complex relationships between the two, such as adversarial training.

Dynamics in Tabular Games. Suppose $i = 1,\ldots,n$ are players in a game with payoffs U. Let A_i denote the set of possible actions of each player i. To play a round of the game, each player i chooses a (randomized) strategy $p_i \in \Delta A_i$. Each player's payoff may be different, and is a function $U : (\prod_i A_i) \to \mathbb{R}^n$ of the joint actions of all players. If $\mathrm{p} = (p_1,\ldots,p_n)$ represents each player's strategy, then player i recieves payoff $\mathbb{E}_{\mathrm{a}\sim\mathrm{p}}[U_i(\mathrm{a})] \in \mathbb{R}$ in expectation. A primary concern of game theory is modeling each player's responses to others. To model this in our framework, let $\Theta := \prod_i \Delta A_i$ be the set of joint strategies, and $\Phi := \{\mathrm{br}_i\}_{i=1}^n$ represent best response actions for each player, which adjusts its own strategy to maximize its expected utility in context:

$$\mathfrak{X}[\![\mathrm{br}_i]\!](s)(\mathrm{p}) = \nabla_{p_i} \mathop{\mathbb{E}}_{\mathrm{a}\sim\mathrm{p}}[U_i(\mathrm{a})].$$

A program $\varphi \in \mathcal{L}(\Phi)$ is then a schedule of who gets to best respond, in what order, and how much they get to change their parameters. As one might hope, $[\![\mathrm{br}_i]\!]^*(\mathrm{p})$ is p, but with component i altered so as to be a best response to the others.

Proposition A.1. *The mixed strategy* p *is a Nash equilibrium if and only if* $\mathfrak{X}[\![\varphi]\!](s,\mathrm{p}) = 0$ *for* $\mathrm{p} \in \mathcal{L}(\Phi)$.

Proof. If p is a Nash equilibrium, then in particular p_i must locally maximize $\mathbb{E}_{\mathrm{a}\sim\mathrm{p}}[U_i(\mathrm{a})]$, for all i, so the vector field of every primitive instruction must be zero. Thus, all mixtures and sequences of them must also be zero.

Conversely, suppose $\mathfrak{X}[\![\varphi]\!] = 0$ for all programs φ, and in particular for the primitive programs $\{\mathrm{br}_i\}_{i=1}^n$. Then, since $\mathbb{E}_{\mathrm{a}\sim\mathrm{p}}[U_i(\mathrm{a})]$ is linear in p_i, this means there is globally no better response. Since no player can improve their utility by switching strategies, p is a Nash equilibrium. □

B OMITTED PARTS OF THE LANGUAGE

When boolean expressions b can be given "hard" interpretations $[\![b]\!] : \Theta \to \{0,1\}$, the semantics of if b then φ works in the usual way. At a state $\theta \in \Theta$ where b is true, use the semantics of φ, and otherwise use the semantics of skip:

$$\mathfrak{X}\left[\!\!\left[\text{if } b \text{ then } \varphi\right]\!\!\right](s)(\theta) := [\![b]\!](\theta) \cdot \mathfrak{X}[\![\varphi]\!](s)(\theta).$$

However, this is not expressive enough to capture conditionals in probabilistic programming languages. In that setting, a state $\theta \in \Theta$ is a joint probability distribution variables, according to which a boolean expression typically has an intermediate probability of being true. Using the expression above with the extended domain $[\![b]\!] \in [0,1]$ equal to the probability of interest, does not work properly. This is because, intuitively, once we enter the body of the **if**, the program state should be the distribution conditioned on the event B where b is true. What we want is something like this:

$$[\![\text{if } b \text{ then } \varphi]\!]^*(\theta) = \theta(B) \cdot [\![\varphi]\!]^*(\theta \mid B) + (1-\theta(B))(\theta \mid \neg B).$$

... but what is the analogue of the conditional distribution $\theta \mid B$ in this more general setting? It appears that the Riemannian metric structure on Θ (that enables us to take gradients), also gives us a second useful bit of structure that can generalize the semantics of conditionals in probabilistic programming.

In more detail: the Riemannian metric uses a natural way of transporting tangent vectors along smooth paths, called parallel transport. This means that, given a tangent vector $v \in T_\theta\Theta$, and a smooth path $\gamma : [0,1] \to \Theta$ with $\gamma(0) = \theta$ and $\gamma(1) = \theta'$, there is a vector $w := \gamma^\sharp(v) \in T_{\theta'}\Theta$ that is the result of smoothly transporting the direction v from θ to θ', along γ. For notational convenience, let $\gamma^{-\sharp}$ denote reverse transport along the path, so that $\gamma^{-\sharp}(w) = v$.

To simplify the notion, let $\int_\theta [\![\varphi]\!] := \oint[\![\varphi]\!](-,\theta) : [0,1] \to \Theta$ be the path semantics with fixed starting point θ. With this notation, we can define the semantics of conditionals as:

$$\mathfrak{X}\left[\!\!\left[\text{if } b \text{ then } \varphi\right]\!\!\right](s,\theta) := [\![b]\!](\theta) \cdot \left(\int_\theta [\![b]\!]\right)^{-\sharp}\left(\mathfrak{X}[\![\varphi]\!](s, \int_\theta^1 [\![\psi]\!])\right). \tag{3}$$

The second half of (3) is, in a sense, the analogue of conditioning that we are after. It tells us that we should follow the vector field for φ — but evaluated not at where we are, but where we would be, if we were to first assert b. This means evaluating the vector field at the point $\int_\theta^1 [\![b]\!] = [\![b]\!]^*(\theta)$, analogous to "$\theta$ conditioned on b", and translating the vector backwards along the path $\int_\theta [\![\psi]\!]$ that we used to get there. This works well in simple simple examples, and we conjecture that it captures probabilistic conditional expressions under a mild syntactic restriction.

Scaling. Also useful is the command $(k)\varphi$, which scales the vector field of φ by k, i.e., $\mathfrak{X}[\![(k)\varphi]\!](s) := k \cdot \mathfrak{X}[\![\varphi]\!](s)$. Perhaps surprisingly, this does not change the endpoint semantics unless used in combination with clipping or mixture.

C SUPPORTING PROOFS

Proof of Eq 2. To take a gradient with respect to the Fisher geometry, one needs only to premultiply by $\mathcal{I}(\theta)^{-1}$, the inverse of the fisher matrix at θ. In this case, because we are parametrizing by the simplex, the Fisher matrix is diagonal, and the $(w,w)^{\text{th}}$ entry is $1/\mu(w)$. After introducing a Lagrange multiplier λ for the constraint that $\sum_w \mu(w) = 1$, we calculate:

$$\begin{aligned}
&\mathfrak{X}[\![\textbf{draw } Y \sim p \mid X]\!] \\
&= \mathcal{I}(\mu)^{-1}\Big(\nabla \boldsymbol{D}\Big(\mu(X,Z)p(Y|X) \,\Big\|\, \mu(X,Y,Z)\Big) - \lambda\Big) \\
&= w \mapsto \mu(w) \sum_{x,y,z} \Big(\frac{\partial}{\partial \mu(w)}[\mu(x,z)] \log \frac{p(y|x)}{\mu(y|x,z)} + \frac{\partial}{\partial \mu(w)}\left[\log \frac{p(y|x)}{\mu(y|x,z)}\right]\mu(x,z)\Big)p(y|x) - \lambda\mu(w) \\
&= w \mapsto \mu(w) \sum_{x,y,z} \Big(\delta(x,z = x_w, z_w) \log \frac{p(y|x)}{\mu(y|x,z)} - \frac{\mu(y|x,z)p(y|x)}{p(y|x)\mu(y|x,z)^2}\frac{\partial}{\partial \mu(w)}\left[\mu(y|x,z)\right]\mu(x,z)\Big)p(y|x) - \lambda\mu(w) \\
&= w \mapsto \mu(w) \sum_{y} p(y|x_w) \log \frac{p(y|x_w)}{\mu(y|x_w,z_w)} - \mu(w) \sum_{x,y,z} \frac{\mu(x,z)p(y|x)}{\mu(y|x,z)} \frac{\delta(x,z = x_w z_w)}{\mu(x,z)}\left[\delta(y = y_w) - \mu(y|x,z)\right] - \lambda\mu(w) \\
&= w \mapsto \mu(w)\boldsymbol{D}(p(Y|x_w) \,\|\, \mu(Y|x_w,z_w)) - \mu(w)\Big(1 - \lambda + \frac{p(y_w|x_w)\mu(x_w,z_w)}{\mu(x_w,y_w,z_w)}\Big) \\
&= w \mapsto \mu(w)\lambda' - p(y_w|x_w)\mu(x_w,z_w) \\
&= \mu(X,Y,Z)\lambda' - p(Y|X)\mu(X,Z) \qquad \text{since the gradient must sum to zero, } \lambda' = 1\text{, yielding} \\
&= \mu(X,Z)(\mu(Y \mid X,Z) - p(Y|X)).
\end{aligned}$$

One can easily see that this is the same vector field as

$$\frac{\partial}{\partial s}\mu(X,Z)\Big((1-s)\mu(Y|X,Z) + (s)p(Y|X)\Big).$$

Since the path starts at μ and its derivative is the vector field above, it must be the path semantics of **draw** $Y \sim p|X$ by the uniqueness of ODE solutions. The remaining details for the other half of the equation are similar in nature, and are similarly dense; they will be typeset in the full paper. □

Proof of Proposition 3.2. By Eq 2,

$$[\![\text{skip}]\!]^*(\mu) = \mu = [\![\text{skip}]\!](\mu);$$

$$\begin{aligned}
[\![\textbf{observe } \delta(Y=y)]\!]^* &= \rlap{\,/}\!\!\int\!\Big[\!\!\Big[\textbf{observe } \delta(Y=y)\Big]\!\!\Big](1,\mu) \\
&\propto \mu(Y,Z) \cdot \left(\frac{\delta(Y=y)}{\mu(Y)}\right) \\
&= \mu(Z \mid Y) \cdot \delta(Y=y) \\
&= \mu(Z \mid Y=y)
\end{aligned}$$

$$\begin{aligned}
[\![\textbf{draw } Y \sim p|X]\!]^* &= \rlap{\,/}\!\!\int\!\Big[\!\!\Big[\textbf{draw } Y \sim p \mid X\Big]\!\!\Big](1,\mu) \\
&= \mu(X,Z) \cdot p(Y|X) \\
&= \mathsf{bind}(\mu(X,Y,Z), Y \mapsto p(Y|X)) \\
&= [\![\textbf{draw } Y \sim p|X]\!];
\end{aligned}$$

$$\begin{aligned}
[\![\varphi_1;\varphi_2]\!]^*(\mu) &= [\![\varphi_2]\!]^*([\![\varphi_2]\!]^*(\mu)) \\
&= [\![\varphi_2]\!]([\![\varphi_2]\!](\mu)) \\
&= [\![\varphi_1;\varphi_2]\!](\mu).
\end{aligned}$$

□

Proof of Proposition 3.3. Probabilistic Dependency Graphs (PDG) [Richardson and Halpern 2021] generalize Bayesian Networks and Factor graphs. The distribution specified by a PDG is the one that minimizes a weighted sum of relative entropy commands, precisely of the form of the observe command. Concretely, given a PDG $\mathcal{M}$ with arcs $\mathcal{A} = \{S_a \to T_a\}_{a\in\mathcal{A}}$ with corresponding conditional probabilities $\mathbb{P}_a$, quantiative confidences $\{\beta_a\}_{a\in\mathcal{A}}$, and qualitative confidences $\{\alpha_a\}_{a\in\mathcal{A}}$, and a trade-off factor $\gamma > 0$, we can define the mixture program

$$\varphi_{(\mathcal{M},\gamma)} \;:=\; (\gamma)\textbf{observe } \mathsf{Unif}(\mathcal{X}) \oplus \bigoplus_{a\in\mathcal{A}} (\beta_a)\textbf{observe } \mathbb{P}_a(T_a \mid S_a) \oplus (-\alpha_a\gamma)\textbf{observe } \mathsf{Unif}(T_a \mid S_a).$$

It is straightforward to see that the loss function semantics of this combined instruction is precisely the scoring function semantics $[\![\mathcal{M}]\!]_\gamma : \Delta\mathcal{V}\mathcal{X} \to \mathbb{R}$, where $\Delta\mathcal{V}\mathcal{X}$ is the set of joint distributions over (the values of) $\mathcal{X}$. To get the distribution semantics by which PDGs capture other models, we need only to find the minima of that distribution, which is the endpoint semantics. In other words, $[\![\mathcal{M}]\!]^*_\gamma = \big\{[\![\varphi_{(\mathcal{M},\gamma)}]\!]^*(\mu) : \mu \in \Delta\mathcal{V}\mathcal{X}\big\}$. In particular, in the typical case when the function is strictly convex, we have $[\![\mathcal{M}]\!]^*_\gamma = \{[\![\varphi_{(\mathcal{M},\gamma)}]\!]^*(\mu_0)\}$ for *every* $\mu_0 \in \Delta\mathcal{V}\mathcal{X}$. Thus, PDGs, and the graphical models they generalize, can be viewed as the endpoint semantics of mixtures of observe commands. □