A Language for Counterfactual Generative Models

Zenna Tavares [1] **James Koppel** [1] **Xin Zhang** [2] **Ria Das** [1] **Armando Solar Lezama** [1]

Abstract

We present OMEGA$_C$, a probabilistic programming language with support for counterfactual inference. Counterfactual inference means to observe some fact in the present, and infer what would have happened had some past intervention been taken, e.g. "given that medication was not effective at dose x, how likely would it have been effective at dose $2x$?" We accomplish this by introducing a new operator to probabilistic programming akin to Pearl's **do**, define its formal semantics, provide an implementation, and demonstrate its utility by examples in population dynamics, inverse planning, and graphics.

1. Introduction

In this paper we introduce OMEGA$_C$: a Turing-universal programming language for causal reasoning. OMEGA$_C$ allows us to automatically derive causal inferences about phenomena that can only be modelled faithfully through simulation. We focus on counterfactuals – *what-if* inferences about the way the world could have been, had things been different.

OMEGA$_C$ programs are simulation models augmented with probability distributions to represent any uncertainty. In a similar vein to other probabilistic languages, OMEGA$_C$ provides primitive operators for conditioning, which revises the model to be consistent with any observed evidence. Counterfactuals, however, cannot be expressed through probabilistic conditioning alone. They take the form: "Given that some evidence E is true, what would Y have been had X been different?" For example, given that a drug treatment was not effective on a patient, would it have been effective at a stronger dosage? Although one can condition on E being true, attempting to condition on X being different to the value it actually took is contradictory, and hence impossible.

[1]CSAIL, MIT, USA [2]Key Lab of High Confidence Software Technologies, Ministry of Education, Department of Computer Science and Technology, Peking University, China. Correspondence to: Zenna Tavares <zenna@csail.mit.edu>, Xin Zhang <xin@pku.edu.cn>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

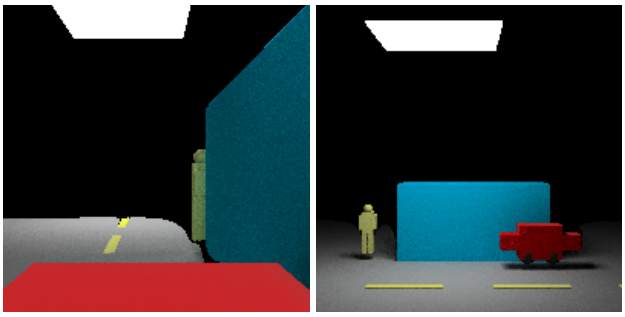

Figure 1: A speeding driver (Left: driver's view) crashes into a pedestrian (yellow) emerging from behind an obstruction (blue). Given a single frame of camera footage (Right), OMEGA$_C$ infers whether driving below the speed limit would have prevented the crash.

To construct counterfactuals, OMEGA$_C$ introduces a **do** operator for constructing *interventions*:

$$Y \mid \mathbf{do}(X \to x) \tag{1}$$

This evaluates to what Y would have been had X been bound to x when Y was defined. Consequently, if Y is a random variable and we define $Y_x = Y \mid \mathbf{do}(X \to x)$, then $P(Y_x = y)$ is the probability Y would have been y had X had been x. A counterfactual is then simply $Y_x \mid E$. Note, if E depends on X, conditioning on it affects X's factual, non-intervened value, which is critical to capturing the semantics of counterfactuals.

To illustrate the potential of counterfactual reasoning within a universal programming language, consider the scenario of an expert witness called to determine, from only a frame of recorded video (Figure 1), whether a driver was to blame for them crashing into a pedestrian. Using OMEGA$_C$, the expert could first construct a probabilistic model that includes the car dynamics, the driver and pedestrian's behaviour, and a rendering function that relates the three dimensional scene structure to two dimensional images. She could then condition the model on the captured images to infer the conditional distribution over the drivers velocity, determining the probability that the driver had been speeding. Using OMEGA$_C$ she could then pose a counterfactual, asking whether the crash would have still occurred even if the driver had obeyed the speed limit. If she later wanted to investi-

gate the culpability of another candidate cause, such as the placement of the crosswalk, she could do so by adding a single-line, and without modifying her underlying models at all.

Causal reasoning is currently done predominantly using *causal graphical models* (20): graphs whose vertices are variables, and whose directed edges represent causal dependencies. Despite widespread use, causal graphs cannot easily express many real-world phenomena. One reason for this is that causal graphs are equivalent to *straight-line programs*: programs without conditional branching or loops – just finite sequences of primitive operations. Straight-line languages are not Turing-complete; they cannot express unbounded models with an unknown number of variables. In practice, they lack many of the features (composite functions, data types, polymorphism, etc.) necessary to express the kinds of simulation models we would like to perform causal inference in.

Counterfactual reasoning in OMEGA$_{\text{C}}$ alleviates many of these limitations. In an OMEGA$_{\text{C}}$ intervention $X \to x$, x can be a constant, function, another random variable, or even refer to its non-intervened self, e.g. $X \to 2X$. Moreover, users can construct various forms of stochastic interventions, and even condition the corresponding interventional worlds. This allows users to model experimental error, or scenarios where observers are unsure about which intervention has taken place.

A generic **do** operator that composes systematically with conditioning presents several challenges. In particular, to construct Y_X, we must be able to copy Y in such a way that the code that defines it is retroactively modified. This goes beyond the capabilities of existing programming languages, probabilistic or otherwise, and hence OMEGA$_{\text{C}}$ requires a non-standard semantics and implementation.

In summary, we (i) present the syntax and semantics of a universal probabilistic language for counterfactual generative models (Section 3); (ii) provide a complete implementation of OMEGA$_{\text{C}}$, and (iii) demonstrate counterfactual generative modelling through a number of examples (Section 4). Regarding scope, causal inference includes problems of both (i) inferring a causal model from data, and (ii) given a causal model, predicting the result of interventions and counterfactuals on that model. This work focuses on the latter.

2. Overview of Counterfactuals

Counterfactual claims assume some structure is invariant between the original *factual world* and intervened *hypothetical world*. For instance, the counterfactual "If I had trained more, I would have won the match" is predicated on the invariance of the opponent's skill, the existence of the game, laws of physics, etc. Any system for counterfactual reasoning must provide mechanisms to construct hypothetical worlds that maintain invariances (and hence share information) with the factual world, so that for instance the fact that I actually lost the match helps predict whether I would have won the match had I trained harder.

These requirements have been resolved in the context of causal graphical models. Causal interventions are "surgical procedures" which modify single nodes but leave functional dependencies intact. Pearl's *twin-network construction* (20) of counterfactuals duplicates the model into one twice the size. One half is the original model. The other half is a duplicate, modified to express the counterfactual interventions. These halves are joined via a shared dependence on the background facts. Hence, conditioning a variable in the factual world influences the counterfactual world.

To generalize the twin-network construction to arbitrary programs, OMEGA$_{\text{C}}$ runs two copies of a program, one factual execution, and one counterfactual execution which shares some variables, but where others have been given alternate definitions. It is folklore that programs doing this can be built by hand, but, as in the twin-network construction, each intervention requires writing a separate model, and each counterfactual included doubles the size of the program. The solution in OMEGA$_{\text{C}}$ is to provide a new **do** operator which removes the need to modify an existing program to add a counterfactual execution. Instead, $t_1 \mid \textbf{do}(x \to t_2)$ is defined to be the value that a term t_1 would take if x had been set to This works even if any dependencies of t_1 on x are indirect. For instance, if $y = 2x$, then $y^2 \mid \textbf{do}(x \to f)$ is equivalent to $(2f)^2$. And note that the variable x can be any variable, even one that is bound to a function, meaning users can compactly define interventions which are substantial modifications. Finally, combining the operator with conditioning automatically gives counterfactual inference.

Our examples show that OMEGA$_{\text{C}}$'s **do** operator enables compact definition of many counterfactual inference problems. Indeed, in Appendix B, we prove that the **do** operator is not expressible as syntactic sugar (as defined by programming language theory).

3. A Calculus for Counterfactuals

Our language OMEGA$_{\text{C}}$ augments the functional probabilistic language OMEGA (31) with counterfactuals. To achieve this: (1) the syntax is augmented with a **do** operator, and (2) the language evaluation is changed from eager to lazy, which is key to handling interventions. In this section, we introduce λ_C, a core calculus of OMEGA$_{\text{C}}$. We build the language up in pieces: first showing the standard/deterministic features, then features for deterministic interventions, and finally the probabilistic ones. Together, intervention and

conditioning give the language the ability to do counterfactual inference. Appendix A gives a more formal definition of the entire λ_C language. A Haskell implementation which provides complete execution traces of terms in λ_C is available from https://tinyurl.com/y3cusyoe.

$$
\begin{aligned}
&\text{Variables} && x, y, z \in \text{Var} \\
&\text{Type } \tau ::= && \text{Int} \mid \text{Bool} \mid \text{Real} \mid \tau_1 \to \tau_2 \\
&\text{Term } t ::= && n \mid b \mid r \mid t_1 \oplus\ t_2 \mid x \mid \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \mid \\
& && \lambda x : \tau . t \mid t_1(t_2) \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3
\end{aligned}
$$

Figure 2: Abstract Syntax for λ_C, deterministic fragment

Deterministic Fragment We begin by presenting the fragment of λ_C for deterministic programming; Fig. 2 gives the syntax. A common formal way to specify the executions of a program is with an *operational semantics* (23), which defines how one expression reduces to another. Appendix A provides a operational semantics for OMEGA$_C$. Here, we describe these reductions through examples. The execution of an expression is defined both in terms of the expression as well as the current program state. In λ_C, this program state is an environment Γ: a mapping from variables to values.

The deterministic fragment is standard, so we will explain it briefly. λ_C has integer numbers (denoted n), Booleans $\{\text{True}, \text{False}\}$ (denoted b) , and real numbers (r). $\oplus$ represents a mathematical binary operator such as $+$, $*$, etc. **let** $x = t_1$ **in** t_2 binds variable x to expression t_1 when evaluating t_2. Lambda expressions create functions: $\lambda x.2 * x$ defines a mapping $x \mapsto 2x$. Function application and if-statements are standard.

Next, we demonstrate the semantics of operators and the **let**. The notation $\left\{{}^{\Gamma}_{e}\right\}$ denotes a pair of an environment Γ and an expression e, and $\left\{{}^{\Gamma_1}_{e_1}\right\} \to \left\{{}^{\Gamma_2}_{e_2}\right\}$ indicates that e_1 with environment Γ_1 steps to e_2 with environment Γ_2. The **let** expression first binds x to 3, creating a new environment. Finally, x is evaluated by looking up its value in the environment.

$$
\left\{\begin{array}{c}\Gamma : \emptyset \\ \textbf{let}\ x = 3\ \textbf{in}\ x\end{array}\right\} \to \left\{\begin{array}{c}\Gamma : x \mapsto 3 \\ x\end{array}\right\} \to \left\{\begin{array}{c}\Gamma : x \mapsto 3 \\ 3\end{array}\right\}
$$

Function applications are done by substitution, as in other variants of the lambda calculus:

$$
\left\{\begin{array}{c}\Gamma : \emptyset \\ (\lambda x.(x + x)(2)\end{array}\right\} \to \left\{\begin{array}{c}\Gamma : \emptyset \\ 2 + 2\end{array}\right\} \to \left\{\begin{array}{c}\Gamma : \emptyset \\ 4\end{array}\right\}
$$

The above semantics is *eager*: **let** $x = t_1$ **in** t_2 first evaluates t_1 and then binds the result to x, creating a new environment in which to then evaluate t_2. We next show how this is problematic for counterfactuals.

Causal Fragment Our causal fragment adds one new term: the **do** expression (Fig. 3). $t_1 \mid \textbf{do}(x \to t_2)$ evaluates t_1 to the value that it would have evaluated to, had x been defined as t_2 at point of definition. Here, x can be any variable that is in scope, bound locally or globally, and t can be any any term denoting a value. One idea is to define **do** similarly to **let**: $t_1 \mid \textbf{do}(x \to t_2)$ would rebind x to t_2 when evaluating t_1. However, this does not take into account transitive dependencies. For example, **let** $x = 0$ **in let** $y = x$ **in** $y \mid \textbf{do}(x \to 1)$ evaluates to 1. However, by the time the execution evaluates the **do**, y has already been bound to 0, so that rebinding x does nothing. To overcome this, we redefine **let** to use *lazy evaluation*.

In lazy evaluation, instead of storing the value of a variable in the environment, the execution stores its defining expression as well as the environment when the variable is defined. So, while environments for eager evaluation store mappings $x \mapsto v$ from variable x to value v, in lazy evaluation, the environments store mappings $x \mapsto (\Gamma, e)$, which map each variable x to a *closure* containing both its defining expression e and the environment Γ in which it was defined. A variable, such as x, is evaluated by evaluating its definition under the environment where it is defined.

We can now define **do**: $y \mid \textbf{do}(x \to -1)$ evaluates y under a new environment which is created by recursively mapping all bindings for x in the current environment to -1. This includes both the binding of x at the top level and the bindings in an environment that is used in any closure. The following example demonstrates this process:

$$
\begin{aligned}
&\left\{\begin{array}{c}\Gamma : \emptyset \\ \textbf{let}\ x = 0\ \textbf{in let}\ y = x + 1\ \textbf{in}\ y + (y \mid \textbf{do}(x \to -1))\end{array}\right\} \\
&\to \left\{\begin{array}{c}\Gamma : x \mapsto (\emptyset, 0) \\ \textbf{let}\ y = x + 1\ \textbf{in}\ y + (y \mid \textbf{do}(x \to -1))\end{array}\right\} \\
&\to \left\{\begin{array}{c}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ y + (y \mid \textbf{do}(x \to -1))\end{array}\right\} \\
&\to \left\{\begin{array}{c}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ \left\{{}^{\Gamma : x \mapsto (\emptyset, 0)}_{x+1}\right\} + (y \mid \textbf{do}(x \to -1))\end{array}\right\} \\
&\to \left\{\begin{array}{c}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + (y \mid \textbf{do}(x \to -1))\end{array}\right\} \\
&\to \left\{\begin{array}{c}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + \left\{{}^{\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x+1)}_{(y \mid \textbf{do}(x \to -1))}\right\}\end{array}\right\} \\
&\to \left\{\begin{array}{c}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + \left\{{}^{\Gamma : x \mapsto (\emptyset, -1), y \mapsto (x \mapsto (\emptyset, -1), x+1)}_{y}\right\}\end{array}\right\} \\
&\to \left\{\begin{array}{c}\Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + \left\{{}^{\Gamma : x \mapsto (\emptyset, -1)}_{x+1}\right\}\end{array}\right\}
\end{aligned}
$$

Term $t ::= \cdots \mid t_1 \mid \textbf{do}(x \to t_2)$

Figure 3: Abstract Syntax for λ_C, causal fragment

$$\to \left\{ \begin{array}{c} \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 + 0 \end{array} \right\}$$
$$\to \left\{ \begin{array}{c} \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\ 1 \end{array} \right\}$$

The program is evaluated under an empty environment. (1) Evaluating the outermost **let** binds x to a closure $(\emptyset, 0)$ (consisting of the initial environment and x's definition). (2) y is bound to a closure, containing the environment from step (1) and y's definition. The left operand of the addition is then evaluated, by first (3) looking up its closure in the environment, and then (4) evaluating its definition under the corresponding environment in the closure. To evaluate the **do** in the right operand, (5) the current environment is copied, and then (6) modified to rebind all definitions of x to -1. The right operand of the addition is a do expression of y, which the execution tries to evaluate under the current environment, by (7) looking up the closure of y in this "intervened" environment, and then (8) evaluating it. (9) The final result of the program is then 1.

Type $\tau ::= \cdots \mid \Omega$ Term $t ::= \cdots \mid \bot \mid t_1 \mid t_2 \mid \textbf{rand}(t)$

Figure 4: Abstract Syntax for λ_C, probabilistic fragment

Probabilistic Fragment In probability theory, a random variable is a function from a sample space Ω to some domain τ. λ_C defines random variables similarly: as functions of type $\Omega \to \tau$. This separates the source of randomness of a program from its main body, leading to a clean definition of counterfactuals.

Fig. 4 shows the abstract syntax of the probabilistic fragment. It introduces a new type Ω, representing the sample space. Ω is left unspecified, save that it may be sampled from uniformly. In most applications, Ω will be a hypercube, with one dimension for each independent sample. To access the values of each dimension of this hypercube, one of the $\oplus$ operators must be the indexing operator $[]$, where $\omega[i]$ evaluates to the ith component of ω.

Random variables are normal functions. If $\Omega = [0, 1]$, and $a < b$ are integer constants, then $R = \lambda\omega : \Omega.\omega * (b - a) + a$ is a random variable uniformly distributed in $[a, b]$. The **rand** operator then samples from a random variable: **rand** R returns a random value drawn uniformly from $[a, b]$. Note that unlike in other probabilistic languages, we separate the construction of random variables from their sampling. Consequently, **rand** does not occur in the definition of a random variable itself.

To support conditioning, we use $\bot$ to denote the undefined value. Any non-**rand** expression that depends on a $\bot$ value will result in another $\bot$ value. A program execution is invalid if it evaluates to $\bot$. One can imagine the execution of a λ_C program as a rejection sampling process: we ignore all samples from **rand** that would make the program evaluate to $\bot$. In the implementation, we use a much more efficient inference algorithm (30).

Conditioning can now be defined as syntactic sugar: $t \mid E$ is defined as $\lambda\omega.\textbf{if } E(\omega) \textbf{ then } t(\omega) \textbf{ else } \bot$.

Let $\Omega = \{1, 2, \ldots, 10\}$, and consider the program **rand** $\lambda\omega.\omega * 2 \mid \lambda\omega.\omega < 4$. If $\omega \geq 4$, then evaluating the random variable results in $\bot$. The **rand** operator hence runs the variable with ω drawn uniformly from $\{1, 2, 3\}$, resulting in 2, 4, or 6, each with $\frac{1}{3}$ probability.

Counterfactuals A counterfactual is a random variable of the form $(t_1 \mid \textbf{do}(x \to t_2)) \mid E$. Consider the following program depicting a game where a player chooses a number c, and then a number ω is drawn randomly from a sample space $\Omega = \{0, 1, \ldots, 6\}$. He wins iff c is within 1 of ω. The query asks: given that the player chose 1 and did not win, what would have happened had the player chosen 4?

```
let c = 1 in
let x = λω . if (ω-c)*(ω-c)<= 1
             then 1 else -1 in
let cfx = (x | do(c → 4)) | λω. x(ω) == -1)
  in rand(cfx)
```

As before, the **rand** expression is evaluated in the context $\Gamma_1 = \{c \mapsto (\emptyset, 1), x \mapsto (c \mapsto \ldots, \lambda\omega.\textbf{if } \ldots)\}$. Its argument, a conditioning term, desugars to $\lambda\omega'.\textbf{if } x(\omega') == -1 \textbf{ then } (x \mid \textbf{do}(c \to 4))(\omega') \textbf{ else } \bot$. This random variable evaluates to $\bot$ for $\omega' \in \{0, 1, 2\}$, so the program is evaluated with ω' drawn uniformly from $\{3, 4, 5, 6\}$. The **do** expression $x \mid \textbf{do}(c \to 4)$ is reduced to evaluating x in the context $\Gamma_2 = \{c = \ldots, x = (c \mapsto (\emptyset, 4), \lambda\omega.\textbf{if } \ldots)\}$. This is then applied to ω', and the overall computation hence evaluates to 1 with probability $\frac{3}{4}$ and -1 with probability $\frac{1}{4}$.

Syntactic Sugar OMEGA$_C$ introduces some syntactic conveniences on top of λ_C. Random variables are functions but it is convenient to treat them as if they were the values in their domains. To support this, OMEGA$_C$ interprets the application of a function to one or more random variables *pointwise* – if both X and Y are random variables and x is a constant, then $X + Y$ is also a random variable defined as $\lambda\omega.X(\omega) + Y(\omega)$ and $X = x$ is $\lambda\omega.X(\omega) = x$. In addition, OMEGA$_C$ represents distribution families as functions from parameters to random variables. For instance, **bern** $= \lambda p.\lambda\omega.\omega[1] < p$ represents the Bernoulli family

by mapping a parameter $p \in [0, 1]$ to a random variable that is true with probability p. Finally, since λ_C is purely functional, if $X = \textbf{bern}(0.5)$ and $Y = \textbf{bern}(0.5)$, then X and Y are not only i.i.d. but the very same random variable, which is not often what we want. OMEGA$_C$ defines the syntax $\sim X$, so that in **let** $X \sim \textbf{bern}(0.5), Y \sim \textbf{bern}(0.5)$, X and Y are independent.

3.1. Other Composite Queries

Conditioning and intervening can be composed arbitrarily. This allows us to a variety of causal queries.

To demonstrate, we adapt an example from (20), whereby (i) with probability p, a court orders rifleman A and B to shoot a prisoner, (ii) A's calmness C ranges uniformly from 1 (cool) to 0 (manic), (iii) if C falls below a threshold q (and hence with probability q) A nervously fires regardless of the order, and (iv) the prisoner dies ($D = 1$) if either shoots. In OMEGA$_C$:

```
let p = 0.7, q = 0.3,
  E = ~ bern(p),       -- Execution order
  C = ~ unif(0, 1),    -- Calmness
  N = C < q            -- Nerves
  A = E or N,          -- A shoots
  B = E,               -- B shoots on order
  D = A or B in        -- Prisoner Dies
```

Counterfactual queries condition the real world and consider the implications in a hypothetical world:

```
-- Given D, would D be true had A not fired?
(D | do(A → 0)) | D
```

Non-atomic Interventions *Atomic* interventions, which replace a random variable with a constant, often do not reflect the kinds of interventions that have, or even could have, taken place in the real-world. Various non-atomic interventions are easily expressed in OMEGA$_C$:

Conditional interventions (8) replace a variable with a deterministic function of other observable variables:

```
-- if A's nerves had spread to B, would D occur?
D | do(B → C < q)
```

A *mechanism change* (32) alters the functional dependencies between variables.

```
-- Would D occur if it took both shots to kill him?
{D | do(D → A and B)} | D
```

Parametric Interventions (9) alter, but do not break, causal dependencies. They are expressible by intervening a variable to be a function of its non-intervened self.

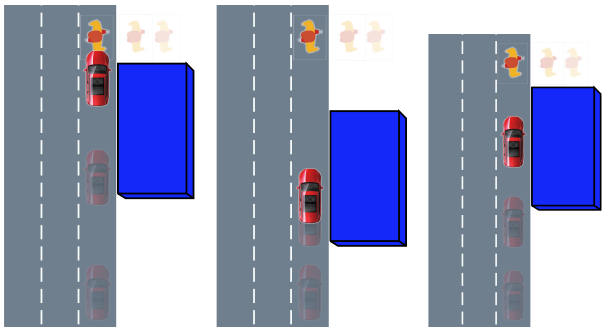

Figure 5: Traces of counterfactual scenarios through time. Each figure is a single sample from (Left) the posterior – the car crashes into the pedestrian, (Middle) the counterfactual on intervening the obstacle position, and (Right) intervening the driver speed. Each image shows the driver and car at (in decreasing transparency) at times 1, 9, and 19.

```
-- If A were more calm, would D have occurred?
D | do(C → C * 1.2)
```

Partial compliance () is where an intervention fails to have any effect with some probability:

```
-- Would D have occurred had we attempted (and failed
-- with probability s) to prevent A shooting?
D | do(A → if ~ bern(s) then 0 else A)
```

"Fat-hand" interventions (9) inadvertently (and probabilistically) affect some variables other than the intended ones:

```
-- Would D be dead if we stopped A from firing and
-- (with probability r) also prevented B, too?
D | do(A → 0, B → if ~ bern(r)
                  then 0 else B)
```

4. Experiments

Here we demonstrate counterfactual reasoning in OMEGA$_C$ through three case studies. All experiments were performed using predicate exchange (30). Simulation parameters and code are in the supplementary material.

Car-Crash Model Continuing from the introduction, this example asks whether a crash would have occurred had a car driven more slowly, given observed camera footage. Let S be the space of scenes, where each scene $s \in S$ consists of the position, velocity, and acceleration of the car, pedestrian and an obstacle. A ray-marching based (1) rendering function $r : S \to I$ maps a scene to an image. The driver acts according to a driver model – a function mapping $s \in S$ to a target acceleration:

```
let
  drivermodel = λ car, ped, obs .
```

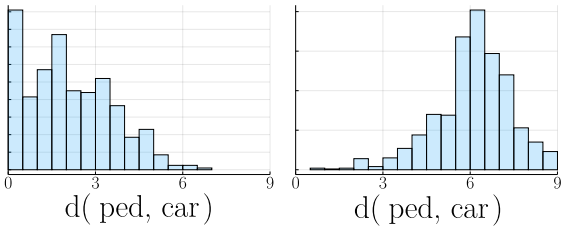

Figure 6: Histograms of causal effect of interventions. How close would the car have come to the pedestrian had (Left) had the acceleration been reduced (CarV $\rightarrow$ 14), or (Right) had the obstacle been removed. Even at the speed limit, the driver still would have crashed with high probability.

```
if cansee(car, ped, obs) -- if ped is visible
then -9                  -- decelerate
else 0,                  -- else maintain
```

The expert witness maintains random variables over the car's acceleration, velocity, and position at $t = 0$. The function `simulate` returns state space trajectories of the form $(s_t, s_{t+1}, \ldots, s_n)$. Since the initial scene is a random variable, `Traj` is a random variable over trajectories. Applying `render` to each scene in `Traj` yields a random variable over image trajectories.

```
CarV = ~ normal(12, 4),
CarP = ~ normal(30, 5),
PedV = ~ normal(3, 1),
PedP = ~ normal(1, 2)
InitScene = (CarV, CarP, PedV, PedP, obs),
Traj = simulate(InitScene, drivermodel),
Images = map(render, Traj),
```

We then ask the counterfactual, conditioning the t_{obs}th image on observed data (Figure 1 right) and intervening CarV $\rightarrow$ 14.

```
Ev = Images[t] == data and crashed(Traj)
  in (Traj | do(CarV → 14)) | Ev
```

We can also ask: would the crash have occurred had the obstacle not been there?

```
  in (Traj | do(InitScene →
(CarV, CarP, PedV, PedP))) | Ev
```

Figure 5 and 6 visualize and explain the results.

Glucose Modelling This example queries whether a hypoglycemic episode could have been avoided in a diabetic patient. We first construct an ODE over variables captured in the Ohio Glucose dataset (17): (1) CGM: continuously monitored glucose measurements, (2) Steps: steps walked by patient, (3) Bolus: insulin injection events, and (4) Meals: calorie intake. The recursive function `euler` implements Euler's method to solve the ODE, taking as input an initial state `u` and derivative function `f'`, and producing a time-series $(u_t, u_{t+\Delta t}, u_{t+2\Delta t}, \ldots, u_{tmax})$.

```
let t0 = 0, Δt = 0.1, tmax = 1,
  τ = λ u, t . u,            -- to intervene u
  euler = λ f', u, t .
    let u = τ(u, t), tnext = t + Δt in
      if t < tmax
      then let unext = u + f'(tnext, u) * Δt
        in cons(u, euler(f', unext, tnext))
      else u,
```

We pre-trained a neural network for the derivative function, and added normally distributed noise to the weights to introduce uncertainty, yielding `F'`, a random variable over functions. Given `F'` as input, `euler` produces a random variable over time-series.

```
Series = euler(F', u, t0),
```

Now we can ask, had we eaten (increased food) at $t = 0.2$, would the hypoglycemic event have occurred? We use the function τ to intervene. It maps `u` at every time `t` to a new value, since `u` is internal to `euler`.

```
τint = λ u, t. if t == 0.5
         then [u[1], u[2], inc(u[3])] else u,
Series = Series | do(τ → τint),
```

Suppose we are told that someone has intervened, and hypoglycemia was avoided, but we do not know when the intervention occurred. We construct a distribution over the intervention time , then condition the intervened world.

```
CGM = first(Series),
Hypo = any(map(λ x . x < thresh, CGM))
T = ~ unif(0, 1)
Tint2 = λ ω . λ u, t. if t == T(ω)
      then [u[1], u[2], inc(u[3])] else u
HC = λ ω . (Hypo | do(τ → Tint2(ω)))(ω)
  in CGM | HC
```

As shown in Figure 7(c), intervening earlier in the day makes a substantial difference.

Counterfactual Planning Consider a dispute between three hypothetical islands (Figure 10): S (South), E (East) and N (North). The people of S consider a barrier between S and N, asking the counterfactual: given observed migration patterns, how would they differ had a border existed.

We model this as a population of agents each acting according in accordance to a Markov Decision Process (24) (MDP) model. Each grid cell is a state in a state space $\mathcal{S} = \{(i, j) \mid i = 1 \ldots 7, j = 1 \ldots 6\}$. The action space

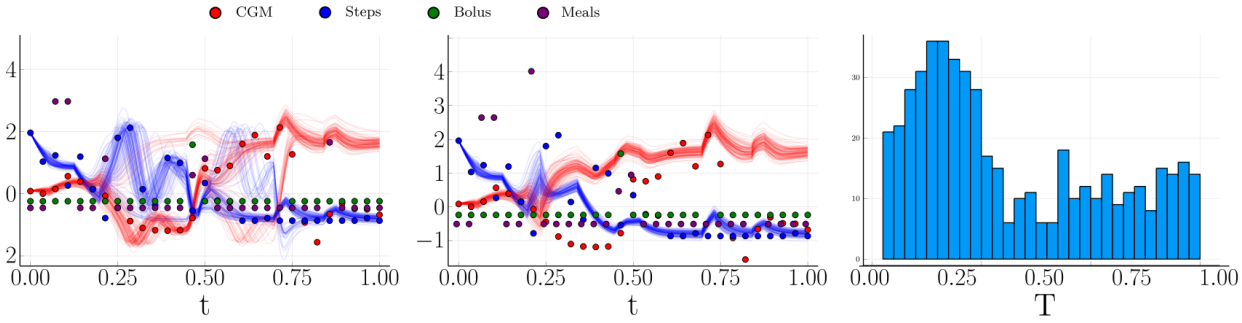

Figure 7: Glucose time series model. Dots are datapoints, trajectories sampled from prior. (Left) Prior samples, (Middle) Samples from interventional distributions Meal $\rightarrow 5$ at $t = 0.20$, (Right) Posterior over time time T of intervention given hypoglycemia did not occur after intervention.

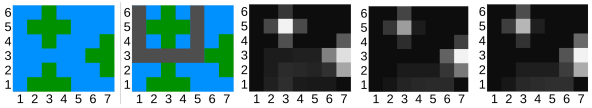

Figure 8: Map (i) without / (ii) with boundary. Sample from population counts after n timesteps of MDP based migration. (iii) unconditional sample, (iv) conditional sample (v) counterfactual sample.

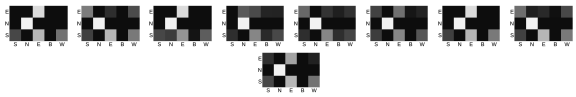

Figure 9: Three samples of migration under three conditions. Each figure shows the migration from islanders born in S, N, or E (y-axis) to S, N, E, W (water) or B (barrier) on the x-axis. We accumulate all states visited in each persons' trajectory. (Plots 1 to 3 from left) Prior samples, (4 to 6) Conditioned on observations, (7 to 9) counterfactual: conditioned and with intervention (border).

moves an agent a single cell: $\mathcal{A} = \{\text{up}, \text{down}, \text{left}, \text{right}\}$. Each agent acts according to a reward function that is a function of the state they are in only $R : \mathcal{S} \rightarrow \mathbb{R}$. This reward function is normally distributed, conditional on the country the agent originates from. For $t = 100$ timesteps we simulate the migration behavior of each individual using value iteration and count the amount of time spent in each country over the time period. Figure 8 shows population counts according to these dynamics. Figure 9 shows migration in the prior, after conditioning on an observed migration pattern (constructed artificially), and the counterfactual cases (adding the border).

But-for Causality in Occlusion In this experiment, we implement "but-for" (13) causation to determine (i) whether a projectile's launch-angle is the cause of it hitting a ball, and (ii) occlusion, i.e. whether one object is the cause of an inability to see another. An event C is the but-for cause of an event E if had C not occurred, neither would have E (12). But-for judgements cannot be resolved by conditioning on the negation of C, since this fails to differentiate cause from effect. Instead, the modeler must find an alternative world where C does not hold. In OMEGA$_{\text{C}}$, a value $\omega \in \Omega$ encompasses all the uncertainty, and hence we define but-for causality relative to a concrete value ω.

Definition 1. Let $C_1, \ldots C_n$ be a set of random variables and $c_1, \ldots, c_n$ a set of values. With respect to a world ω, the conjunction $C_1 = c_1 \wedge \cdots \wedge C_n = c_n$ is the but-for cause of a predicate $E : \Omega \rightarrow \text{Bool}$ if (i) it is true wrt ω and (ii) there exist $\hat{c}_1, \ldots, \hat{c}_n$ such that:

$$(E \mid \textbf{do}(C_1 \rightarrow \hat{c}_1, \ldots, C_n \rightarrow \hat{c}_n))(\omega) = \text{False} \quad (2)$$

$E(\omega) = \text{True}$ is a precondition, i.e., the effect must actually have occur ed for but-for to be defined.

But-for is defined existentially. To solve it, OMEGA$_{\text{C}}$ relies on predicate relaxation (30), which underlies inference in OMEGA$_{\text{C}}$. That is, E is a predicate that in (i) is true iff the projectile hits the ball, and in (ii) is true iff the yellow object is occluded in the scene, computed by tracing rays from the viewpoint and checking for intersections. Predicate relaxation transforms E into soft predicate $\tilde{E}$ which returns a value in $[0, 1]$ denoting how close E is to being satisfied. Using this, our implementation uses gradient descent over $\hat{c}_1, \ldots, \hat{c}_n$ to minimize $(\tilde{E} \mid \textbf{do}(C_1 \rightarrow \hat{c}_1, \ldots, C_n \rightarrow \hat{c}_n))(\omega)$. In (i) $\hat{c}_i$ is the launch-angle and in (ii) $\hat{c}_{x,y,z}$ is the position of the occluder. Finding $\hat{c}_i$ such that soft $E(\hat{c}_i) = 0$ confirms a but-for cause. In Figure 10 we present a visualization of the optimization, which ultimately infers that the angle is the cause of collision and the grey-sphere is the cause of the viewer's inability to see the yellow sphere.

5. Related Work and Discussion

Related work. Operators resembling **do** appear in existing PPLs. Venture (16) has a force expression `[FORCE`

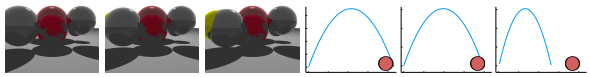

Figure 10: But-for causality. Left to Right: stages of optimization to infer that grey-sphere is cause of inability to see yellow sphere, and launch-angle is cause of projectile colliding with ball.

`<expr> <value>]` which modifies the current *trace* object (a mapping from random primitives to values) so that the simulation of `<expr>` takes on the value `<value>`. It is intended as a tool for initialization and debugging. Pyro (5) and Anglican (34) have similar mechanisms. This can and has (18; 22) been used to compute counterfactuals by (i) approximating the posterior with samples, (ii) revising the model with an intervention, and then (ii) simulating the intervened model using the posterior samples instead of priors.

The fundamental distinction is that in OMEGA$_C$, the operators to condition and intervene both produce new random variables, which can then be further conditioned or intervened to produce *counterfactual variables*, which in turn can be either sampled from or reused in some other process. The Pyro approach, in contrast, computes *counterfactual queries* by performing inference first and then changing the model second. This has several practical consequences. Counterfactual queries in OMEGA$_C$ tend to be significantly more concise, and require none of the manual hacks. More fundamentally, OMEGA$_C$ does not embed an inference procedure into the counterfactual model itself, which muddles the distinction between modelling and inference. In this vein, Pyro is similar to Metaverse (22), a recent Python based system, which mirrors Pearl's three steps of abduction, action and prediction, using importance sampling for inference. A downside of this approach is that it is difficult to create the kinds of composite queries we have demonstrated. We explore this in more detail in the Appendix.

RankPL (26) uses ranking functions in place of numerical probability. It advertises support for causal inference, as a user can manually modify a program to change a variable definition. Baral et al. (4) described a recipe to encode counterfactuals in P-log, a probabilistic logic programming language. However, no language construct is provided to automate this process, which they call "intervention". There has also been work in adding causal operators to knowledge-based programming (11), answer-set programming (7), and logic programming (21). There are several libraries for causal inference on causal graphs (6; 27; 33; 3; 2). Whittemore (6) is an embedded Clojure DSL implementing the do-calculus (20). It can estimate the results of interventions, but not counterfactuals, from a dataset.

Ibeling and Icard (14) introduce computable structural equation models (SEMs) to support infinite variable spaces, and prove that an axiomatization of counterfactuals is sound and complete. OMEGA$_C$ similarly supports open-world models, but our approach is constructive rather than axiomatic – we provide primitives to construct and compute counterfactuals. Ness et al. (19) relates SEMs to Markov process models, which are naturally expressible in OMEGA$_C$. They introduce a novel kind of intervention that finds a change to induce a target post-equilibrium value. A version of this is expressible within OMEGA$_C$– first construct a distribution over interventions, then condition that distribution on the target post-equilibrium value occurring.

Alternative approaches. Some languages have inbuilt mechanisms for *reflection* – the ability to introspect and dynamically execute code, Python for instance includes `getsource(foo)` which returns the source code of a function `foo`. By extracting the source code of a model, transforming it, and reexecuting the result with `eval`, a system of interventions could be formulated. This could be a useful way to bring counterfactuals to existing languages such as Python which cannot support lazy evaluation.

While we have presented a minimal language here, OMEGA$_C$ is also implemented in Julia. Since Julia is not lazy, it is less flexible than OMEGA$_C$, suffering some of the limitations of Pyro. We detail this in the Appendix.

Invariants in counterfactuals. An important property of counterfactual inference is that observations in the factual world carry over to the counterfactual world. This property is easy to satisfy in conventional causal graphs as all exogenous and endogenous variables are created and accessed statically. However, this is not true in OMEGA$_C$ as variable creation and access can be dynamic. Concretely, interventions can change the control-flow of a program, which in turn can cause mismatches between variable accesses in the factual world and ones in the counterfactual world. To address this issue, we tie variable identities to program structures. Appendix B discusses this in detail.

Limitations. Procedures such as the PC algorithm (29) handle situations where a causal relationship exists, but nothing is known about the relationship other than that it is an arbitrary function. Like other probabilistic programming languages, OMEGA$_C$ cannot reason about such models.

In some cases the variable we want to intervene is internal to some function and not in scope at the point where we want to construct an intervention. In other cases, the value we want to intervene (e.g. `(x+2)` in `2*(x + 2)` is not bound to a variable at all. While it is always possible to manually modify the program to expose these inaccessible values, future work is to increase the expressiveness of OMEGA$_C$ to be able to automatically intervene in such cases. Since our formalism relies on variable binding, this would require an entirely different mechanism to what we have presented.

References

[1] Differentiable Path Tracing on the TPU. https://blog.evjang.com/2019/11/jaxpt.html. Accessed: 2021-01-01.

[2] ggdag. https://ggdag.malco.io/. Accessed: 2019-03-08.

[3] pgmpy. http://pgmpy.org/. Accessed: 2019-03-08.

[4] Chitta Baral and Matt Hunsaker. Using the Probabilistic Logic Programming Language P-log for Causal and Counterfactual Reasoning and Non-Naive conditioning. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 243–249, 2007.

[5] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.

[6] Joshua Brulé. Whittemore: An Embedded Domain Specific Language for Causal Programming. *arXiv preprint arXiv:1812.11918*, 2018.

[7] Pedro Cabalar. Causal Logic Programming. In *Correct Reasoning*, pages 102–116. Springer, 2012.

[8] Juan Correa and Elias Bareinboim. A Calculus for Stochastic Interventions: Causal Effect Identification and Surrogate experiments. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 10093–10100, 2020.

[9] Frederick Eberhardt and Richard Scheines. Interventions and Causal Inference. *Philosophy of science*, 74(5):981–995, 2007.

[10] Matthias Felleisen. On the Expressive Power of Programming Languages. In *ESOP'90, 3rd European Symposium on Programming, Copenhagen, Denmark, May 15-18, 1990, Proceedings*, pages 134–151, 1990.

[11] Joseph Halpern and Yoram Moses. Using Counterfactuals in Knowledge-Based Programming. volume 17, pages 97–110, 07 1998.

[12] Joseph Y Halpern and Christopher Hitchcock. Actual Causation and the Art of Modeling. *arXiv preprint arXiv:1106.2652*, 2011.

[13] Daniel M Hausman, Herbert a Simon, et al. *Causal Asymmetries*. Cambridge University Press, 1998.

[14] Duligur Ibeling and Thomas Icard. On Open-Universe Causal Reasoning. In *Uncertainty in Artificial Intelligence*, pages 1233–1243. PMLR, 2020.

[15] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited Dynamic Binding. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming, ICFP 2006, Portland, Oregon, USA, September 16-21, 2006*, pages 26–37, 2006.

[16] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference. *arXiv preprint arXiv:1404.0099*, 2014.

[17] Cindy Marling and Razvan C Bunescu. The ohiot1dm dataset for blood glucose level prediction. In *KHD@ IJCAI*, 2018.

[18] Robert Ness. Lecture Notes for Causality in Machine Learning, Section 9.6: "Bayesian counterfactual algorithm with SMCs in Pyro", 2019.

[19] Robert Osazuwa Ness, Kaushal Paneri, and Olga Vitek. Integrating markov processes with structural causal modeling enables counterfactual inference in complex systems. *arXiv preprint arXiv:1911.02175*, 2019.

[20] Judea Pearl. *Causality*. Cambridge University Press, 2009.

[21] Luís Moniz Pereira and Ari Saptawijaya. Agent Morality via Counterfactuals in Logic Programming. In *Proceedings of the Workshop on Bridging the Gap between Human and Automated Reasoning - Is Logic and Automated Reasoning a Foundation for Human Reasoning? co-located with 39th Annual Meeting of the Cognitive Science Society (CogSci 2017), London, UK, July 26, 2017.*, pages 39–53, 2017.

[22] Yura Perov, Logan Graham, Kostis Gourgoulias, Jonathan Richens, Ciaran Lee, Adam Baker, and Saurabh Johri. Multiverse: Causal Reasoning Using Importance Sampling in Probabilistic Programming. In *Symposium on advances in approximate bayesian inference*, pages 1–36. PMLR, 2020.

[23] Gordon D Plotkin. A Structural Approach to Operational Semantics. 1981.

[24] Martin L Puterman. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

[25] Jarrett Revels, Valentin Churavy, Tim Besard, Lyndon White, Twan Koolen, Mike J Innes, Nathan Daly, Rogerluo, Robin Deits, Morten Piibeleht, Moritz Schauer, Kristoffer Carlsson, Keno Fischer, and Chris de Graaf. jrevels/cassette.jl: v0.3.3, April 2020.

[26] Tjitze Rienstra. RankPL: A Qualitative Probabilistic Programming Language. In *European Conference on Symbolic and Quantitative Approaches to Reasoning and Uncertainty*, pages 470–479. Springer, 2017.

[27] Amit Sharma and Emre Kiciman. DoWhy: Making Causal Inference Easy. https://github.com/Microsoft/dowhy, 2018.

[28] Yehonathan Sharvit. Lazy Sequences are not Compatible with Dynamic Scope. https://blog.klipse.tech/clojure/2018/12/25/dynamic-scope-clojure.html, 2018.

[29] Peter Spirtes, Clark N Glymour, Richard Scheines, David Heckerman, Christopher Meek, Gregory Cooper, and Thomas Richardson. *Causation, Prediction, and Search*. MIT press, 2000.

[30] Zenna Tavares, Javier Burroni, Edgar Minaysan, Armando Solar Lezama, and Rajesh Ranganath. Predicate Exchange: Inference with Declarative Knowledge. In *International Conference on Machine Learning*, 2019.

[31] Zenna Tavares, Xin Zhang, Javier Burroni, Edgar Minasyan, Rajesh Ranganath, and Armando Solar-Lezama. The Random Conditional Distribution for Higher-Order Probabilistic Inference. *arXiv*, 2019.

[32] Jin Tian and Judea Pearl. Causal discovery from changes. *arXiv preprint arXiv:1301.2312*, 2013.

[33] Santtu Tikka and Juha Karvanen. Identifying Causal Effects with the R Package causaleffect. *Journal of Statistical Software*, 76(1):1–30, 2017.

[34] David Tolpin, Jan-Willem van de Meent, and Frank Wood. Probabilistic programming in anglican. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 308–311. Springer, 2015.

The Appendix is structured as follows:

1. Section A presents the full operational semantics of λ_C.
2. Section B details issues around invariance that can occur in dynamic OMEGA$_C$ programs.
3. Section C provides a proof that our intervention operator is not a superficial syntactic change.
4. Section D provides more details on OMEGA$_C$ that go beyond the core calculus λ_C.
5. Section E provides a comparisons between OMEGA$_C$ and Pyro, Multiverse and the Julia implementation.

An interpreter for the core calculus can be found at the following anonymized repository:

https://anonymous.4open.science/r/46bc0fa9-0981-4131-8031-573d997adb3d

The Julia implementation, as well as all the code for the examples, can be found in the following anonymized repository:

https://anonymous.4open.science/r/85c827a4-f725-44c1-891d-3ce93e28f3b0

A. The Full Semantics of λ_C

In this section, we outline the full semantics of our core calculus, λ_C.

$$
\begin{array}{rl}
\text{Variables} & x, y, z \in \text{Var} \\
\text{Type } \tau ::= & \text{Int} \mid \text{Bool} \mid \text{Real} \mid \tau_1 \to \tau_2 \mid \Omega \\
\text{Term } t ::= & n \mid b \mid r \mid \bot \mid x \mid \lambda x : \tau . t \mid \\
& \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \mid t_1 \oplus t_2 \mid \\
& t_1(t_2) \mid \textbf{let } x = t_1 \textbf{ in } t_2 \mid \\
\text{(prob. terms)} & t_1 \mid t_2 \mid \\
\text{(causal terms)} & t_1 \mid \textbf{do}(x \to t_2) \mid \\
\text{Query} & \textbf{rand}(t)
\end{array}
$$

Figure 11: Abstract Syntax for λ_C

Fig. 11 shows the abstract syntax for λ_C. n represents integer numbers, b are Boolean values in $\{\text{True}, \text{False}\}$, and r are real numbers. $\oplus$ represents a mathematical binary operator such as $+$, $*$, etc. We assume there is a countable set of variables $\text{Var} = \{\omega, x, y, z, \dots\}$; x represents a member in this set. $\bot$ represents the undefined value. Finally, there is a sample space Ω, which is left unspecified, save that it may be sampled from uniformly. In most applications, Ω will be a hypercube, with one dimension for each independent sample.

Overall λ_C is a normal lambda calculus with booleans, but with three unique features: conditioning (on arbitrary predicates), intervention, and sampling. Together, these give counterfactual inference.

$$
\begin{array}{c}
\text{Closure } c ::= \text{ clo}(\Gamma, t) \\
\text{Env } \Gamma \in \text{Var} \rightharpoonup \text{Closure}
\end{array}
$$

Figure 12: Runtime environments of λ_C

Semantics Fig. 13 gives the big-step operational semantics of λ_C. A λ_C expression e is evaluated in an environment Γ, which stores previously-defined random variables as closures. Fig. 12 defines closures and environments: a closure is a pair of an expression and an environment, while an environment is a partial map of variables to closures. The notation $\Gamma, x \mapsto c$ refers to some environment Γ extended with a mapping from x to c. The judgement $\Gamma \vdash t \Downarrow v$ means that, in environment Γ, completely evaluating t results in v. We explain each rule in turn.

Integers, booleans, and real numbers, are values in λ_C, and hence evaluate to themselves, as indicated by the INT, BOOL, and REAL rules. Evaluating a lambda expression captures the current environment and the lambda into a closure (LAMBDA rule). The BINOP rule evaluates the operands of a binary operator left-to-right and then computes the operation. The IFTRUE and IFFALSE rules are also completely standard, evaluating the condition to either True or False, and then running the appropriate branch.

The VAR rule is the first nonstandard rule, owing to the lazy evaluation. When a variable x is referenced, its defining closure $\text{clo}(\Gamma', e)$ is looked up. x's defining expression e is then evaluated in environment Γ'. Correspondingly, the LET rule binds a variable x to a closure containing its defining expression and the current environment. Note that the closure for x does not contain a binding for x itself, prohibiting recursive definitions. LET also has a side-condition prohibiting shadowing.

As an example of the LET and VAR rules, consider the term **let** $x = 1$ **in let** $y = x + x$ **in** $y + y$. The LET rule first binds x to $\text{clo}(\emptyset, 1)$, where $\emptyset$ is the empty environment, and then binds y to $\text{clo}(\{x \mapsto \text{clo}(\emptyset, 1)\}, x + x)$. It finally evaluates $y + y$ in the environment $\{x \mapsto \text{clo}(\emptyset, 1), y \mapsto \text{clo}(\{x \mapsto \text{clo}(\emptyset, 1)\}, x + x)\}$. Each reference to y is evaluated with the VAR rule, which evaluates $x + x$ in the environment $\{x \mapsto \text{clo}(\emptyset, 1)\}$. Each such reference to x is again evaluated with the VAR rule, which evaluates 1 in the environment $\emptyset$. The overall computation results in the value 4.

We are now ready to introduce the DO rule, which lies at the core of λ_C. The term $t_1 \mid \textbf{do}(x \to t_2)$ evaluates t_1

$$\frac{}{\Gamma \vdash n \Downarrow n}\ Int \qquad \frac{}{\Gamma \vdash b \Downarrow b}\ Bool \qquad \frac{}{\Gamma \vdash r \Downarrow r}\ Real \qquad \frac{}{\Gamma \vdash \lambda x : \tau.t \Downarrow \text{clo}(\Gamma, \lambda x : \tau.t)}\ Lambda$$

$$\frac{\Gamma \vdash t_1 \Downarrow v_1 \quad \Gamma \vdash t_2 \Downarrow v_2 \quad v_3 = v_1 \oplus v_2}{\Gamma \vdash t_1 \oplus t_2 \Downarrow v_3}\ Binop$$

$$\frac{\Gamma \vdash t_1 \Downarrow \text{True} \quad \Gamma \vdash t_2 \Downarrow v}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \Downarrow v}\ IfTrue \qquad \frac{\Gamma \vdash t_1 \Downarrow \text{False} \quad \Gamma \vdash t_3 \Downarrow v}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \Downarrow v}\ IfFalse$$

$$\frac{\Gamma' \vdash e \Downarrow v}{\Gamma, x \mapsto \text{clo}(\Gamma', e) \vdash x \Downarrow v}\ Var$$

$$\frac{\Gamma \vdash t_1 \Downarrow \text{clo}(\Gamma', \lambda x : \tau.t_3) \quad \Gamma \vdash t_2 \Downarrow v_1 \quad \Gamma' \vdash t_3[v_1/x] \Downarrow v_2}{\Gamma \vdash t_1(t_2) \Downarrow v_2}\ App$$

$$\frac{x \notin \text{dom}(\Gamma) \quad \Gamma, x \mapsto \text{clo}(\Gamma, t_1) \vdash t_2 \Downarrow v}{\Gamma \vdash \textbf{let } x = t_1 \textbf{ in } t_2 \Downarrow v}\ Let \qquad \frac{\Gamma' = \text{RetroUpd}(\Gamma, x, \text{clo}(\Gamma, t_2)) \quad \Gamma' \vdash t_1 \Downarrow v}{\Gamma \vdash t_1 \mid \textbf{do}(x \to t_2) \Downarrow v}\ Do$$

$$\frac{\Gamma \vdash \lambda\omega : \Omega.\textbf{if } t_2(\omega) \textbf{ then } t_1(\omega) \textbf{ else } \bot \Downarrow v}{\Gamma \vdash t_1 \mid t_2 \Downarrow v}\ Cond \qquad \frac{\Gamma \vdash t(\omega) \Downarrow v}{\Gamma \vdash \textbf{rand}(t) \Downarrow v}\ Rand$$

where ω is uniformly drawn from $\{\omega \in \Omega \mid \Gamma \not\vdash t(\omega) \Downarrow \bot\}$.

$$\frac{}{\Gamma \vdash \bot \Downarrow \bot}\ \bot Val \qquad \frac{\Gamma \vdash t_1 \Downarrow \bot}{\Gamma \vdash t_1 \oplus t_2 \Downarrow \bot}\ \bot Binop_1 \qquad \frac{\Gamma \vdash t_1 \Downarrow v \quad \Gamma \vdash t_2 \Downarrow \bot}{\Gamma \vdash t_1 \oplus t_2 \Downarrow \bot}\ \bot Binop_2$$

$$\frac{\Gamma \vdash t_1 \Downarrow \bot}{\Gamma \vdash t_1(t_2) \Downarrow \bot}\ \bot App_1 \qquad \frac{\Gamma \vdash t_1 \Downarrow v \quad \Gamma \vdash t_2 \Downarrow \bot}{\Gamma \vdash t_1(t_2) \Downarrow \bot}\ \bot App_2 \qquad \frac{\Gamma \vdash t_1 \Downarrow \bot}{\Gamma \vdash \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 \Downarrow \bot}\ \bot If$$

Figure 13: Operational semantics for λ_C

$$\boxed{\text{RetroUpd} : \text{Env} \times \text{Var} \times \text{Closure} \to \text{Env}}$$

$$\text{RetroUpd}(\Gamma, x, c)(y) = c \quad \text{if } y = x \wedge x \in \text{dom}(\Gamma)$$

$$\text{RetroUpd}(\Gamma, x, c)(y) = \text{clo}(\text{RetroUpd}(\Gamma', x, c), t') \quad \text{if } y \neq x \wedge (y \mapsto \text{clo}(\Gamma', t')) \in \Gamma$$

Figure 14: The RETROUPD procedure

to the value that it would have taken had x been bound to t_2 at its point of definition. It does this by creating a new environment Γ', which rebinds x in all closures to t_2. This Γ' is created by the retroactive-update function RETROUPD (Fig. 14).

For example, consider the term **let** $x = 1$ **in let** $y = x + x$ **in** $(y + y \mid \textbf{do}(x \to 2))$. The first part of the computation is pthe same as in the previous example, and results in evaluating $y + y \mid \textbf{do}(x \to 2)$ in the environment $\Gamma_1 = \{x \mapsto \text{clo}(\emptyset, 1), y \mapsto \text{clo}(\{x \mapsto \text{clo}(\emptyset, 1)\}, x + x)\}$. The DO rule recursively updates all bindings of x, and evaluates $y + y$ in the environment $\{x \mapsto \text{clo}(\Gamma_1, 2), y \mapsto \text{clo}(\{x \mapsto \text{clo}(\Gamma_1, 2)\}, x + x)\}$. The computation results in the value 8.

APP is the standard application rule for a semantics with closures. Unlike the LET rule, it is strict, so that $t_1(t_2)$ forces t_2 to a value before invoking t_1. This destroys the provenance of t_2, meaning that it will be consider exogeneous to the computation of t_1, and unaffected by any **do** operators.

The final rules concern randomness and conditioning. The special $\perp$ value indicates an undefined value, and any term which strictly depends on $\perp$ is also $\perp$, as indicated by $\perp$VAL, $\perp$BINOP$_1$, and similar rules. Conditioning one random variable t_1 on another random variable t_2 is then defined via the COND rule as a new random variable which is t_1 when t_2 is true, and $\perp$ otherwise. Finally, the RAND rule samples from a random variable by evaluating it on a random point in the sample space Ω.

As our final example in this section, we show how to combine the RAND, COND, and DO rules to evaluate a counterfactual. This program depicts a game where a player chooses a number c, and then a number ω is drawn randomly from a sample space $\Omega = \{0, 1, \ldots, 6\}$, and the player wins iff c is within 1 of ω. The query asks: given that the player chose 1 and did not win, what would have happened had the player chosen 4?

```
let c = 1 in
let X = λω. if (ω-c)*(ω-c)<= 1
            then 1 else -1
  in rand((X | do(c → 4)) | λω. X(ω) == -1)
```

As before, the LET rule causes the inner **rand** expression to evaluate in the context $\Gamma_1 = \{c \mapsto \text{clo}(\emptyset, 1), X \mapsto \text{clo}(c \mapsto \ldots, \lambda\omega.\textbf{if} \ldots)\}$. The COND rule will essentially replace the argument to **rand** with $\lambda\omega'.\textbf{if}\ X(\omega') == -1\ \textbf{then}\ (X \mid \textbf{do}(c \to 4))(\omega')\ \textbf{else}\ \perp$. This random variable evaluates to $\perp$ for $\omega' \in \{0, 1, 2\}$, so the RAND rule evaluates it with ω' drawn uniformly from $\{3, 4, 5, 6\}$. The **do** expression evaluates to $\text{clo}(\{c \mapsto \text{clo}(\Gamma_1, 4)\}, \lambda\omega.\textbf{if} \ldots)$. This is then applied to ω', and the overall computation hence evaluates to 1 with probability $\frac{3}{4}$ and -1 with probability $\frac{1}{4}$.

B. Invariants in Counterfactuals

In an expression $\omega[e]$, e is an ordinary expression. Interventions may change e, and hence $\omega[e]$ in unexpected ways. This can lead to undesirable results for counterfactual queries. For example, take the following program, which centers on a function `digits` which computes a random n-digit base-10 number:

```
1 let
2   digits = λω, d .
3     if d == 0
4     then 0
5     else floor(10*ω[d]) + 10*digits(ω, d-1),
6   n = 5,
7   f = λω.digits(ω, n) * ω[n+1] in
8     rand((f | do(n → 4)) | λω.f(ω) < 10)
```

The function `f` is a random variable over 5-digit numbers whose digits are based on $\omega[1], \ldots, \omega[5]$, and then scales it by $\omega[6]$. The counterfactual query asks what the corresponding scaled 4 digit number would be, given that the 5-digit number. The user likely desired that this counterfactual will be a 4-digit number whose digits are based on, $\omega[1], \ldots, \omega[4]$, and then scale it by the same factor, $\omega[6]$. In fact, the counterfactual execution will scale by $\omega[5]$, which was not intended to be a scale factor. The factual and counterfactual executions both used the same value $\omega[5]$, but at completely different points in the program!

The conceptual problem is that the value $\omega[5]$ is intended to represent differs between the original and intervened model. This occurred because the intervention changed the control flow of the program, and does not occur in static models (where the number of variables is fixed).

Following this intuition, OMEGA$_{\text{C}}$ provides a macro `uid` for indexing ω, which is processed at compile time. The implementation keeps a separate counter for each program point, and uses the counter and program point to compute a unique index to access ω. In addition, since a function can be used to define different random variables, it resets the counter whenever it starts sampling a new random variable. Consider the following program:

```
let uniform = λa. λb. λω.(a-b)*ω[uid]+b in
rand(uniform(1,2))
```

Conceptually, it is translated into

```
let uniform = λa. λb. λω.
push_counters();
(a-b)*ω[h(#pc,get_and_increase(#pc))]+b;
pop_counters()
in
rand(uniform(1,2))
```

The language runtime maintains a stack of maps from program points to counters. Whenever a random variable is sampled from, built-in function `push_counters` is invoked to push a map of new counters to the stack. And when the sampling finishes, built-in function `pop_counters` is invoked to pop the map. Macro `#pc` returns the current program point. Built-in function `increase_and_get` returns the counter corresponds to the current program point and increases it by one. The hash function `h` returns a unique number for every pair of numbers. Note now, a exogenous variable is identified by the program point where it is accessed and the counter corresponds to this program point.

C. The do Operator is Foundational

In this section, we connect the **do** operator with foundational research in programming languages.

C.1. Lazy Dynamic Scope

As we developed the semantics of **do**, we realized that it was far too elegant to not already exist. After much reflection, we realized that it's an unknown variant of a well-studied language construct.

Dynamic scope refers to variables referenced in a function

which may be bound to a different value each time the function is called. It's best known as the default semantics of variables in Emacs Lisp, and has also been used to model system environmental variables such as `$PATH` (15).

All known uses of dynamic scope are strict, and, in the only reference to lazy dynamic scope we found, a blogger writes that laziness and dynamic scope are "not compatible" due to its surprising behavior (28).

But, as we've shown, lazy dynamic scope is not unpredictable. It expresses counterfactuals.

C.2. Why do is not syntactic sugar

In his influential thesis work, Felleisen (10) addressed the question of when a language construct is mere "syntactic sugar," vs. when it increases a language's power. In this, he provided the notions of *expressibility* and *macro-expressibility*. A language construct F is expressible in terms of the rest of the language if the minimal subprograms containing F can be rewritten to not use F while preserving program semantics. Macro-expressibility further stipulates that these rewrites must be local.

With these, he also provided an ingeniously simple proof technique: a construct is not macro-expressible if there are two expressions which are indistiguishable without the language construct (i.e.: they run the same when embedded into any larger program), but distinguishable with it.

In the following theorem, we prove that we cannot implement the **do** operator as a syntactic sugar (i.e., macro) in the original OMEGA language.

From our literature search, this is also the first time any variant of dynamic scope has been proven not macro-expressible in a language without dynamic scope.

Theorem 1. The **do** operator is not macro-expressible in λ_C without **do**.

Proof. According to the proof technique of Felleisen (10), to show **do** is not macro-expressible in λ_C without **do**, it suffices to find two expressions P and P' such that, for any evaluation context C λ_C without **do**, $C[P] = C[P']$, but such that there is an evaluation context C in λ_C with **do** such that $C[P] \neq C[P']$.

Let $P = \lambda f.\lambda x.(f\ 0)$, and $P' = \lambda f.(\lambda a.\lambda x.a)(f\ 0)$.

Note that all constructs of λ_C except **do** and **rand** are macro-expressible in terms of the pure lambda calculus. After fixing a random seed, **rand** is also deterministic. Hence, with a fixed seed, λ_C without **do** respects beta equivalence. Hence, since $P \equiv_\beta P'$, for any context C which does not contain **do**, $C[P] = C[P']$.

Now pick:

$$C[e] = (\lambda g.g\ 0 \mid \mathbf{do}(p \to 1))(e(\lambda x.p))) \mid \mathbf{do}(p \to 0)$$

Then $C[P] \Downarrow 1$, but $C[P'] \Downarrow 0$.

□

D. OMEGA$_C$ Details

D.1. Ids and independent random variables

OMEGA$_C$ includes a $\sim$ operator, allowing us to construct an copy of a random variable that is independent but identically distributed. For example, if we have a standard Bernoulli distribution `flip = λ ω . w[1] > 0.5` then `flip2 = 2 ~ flip` will be i.i.d. with flip. More over we can avoid specifying the id manually and simply write `flip2 = ~flip`.

To describe this, first we represent ω values as functions from an integer id to a value in the unit interval, and hence $\omega[i]$ is simply a syntactic convenience for the function application $\omega(i)$. The operator $\sim$ is then a binary function mapping an `id` and a random variable `X` to a new random variable that is i.i.d.:

```
-- Constructs idth i.i.d. copy of X
~ = λ ω id, X . λ ω . X(project(w, id))
```

It works by *projecting* any ω that is input to X onto a new space prior to applying X to it. A projection of ω onto id is a new value ω' such that $\omega'(i) = \omega(j)$ where j is a ***unique*** combination of i and id.

```
project = λ ω, id1 .
            λ id2 → ω(pair(id1, id2))
```

Such a unique combination can be constructed using a *pairing function* `pair`, which is a bijection from $\mathbb{N} \times \mathbb{N}$ to $\mathbb{N}$. Many pairing functions exist, below we define Cantor's pairing function:

```
pair = λ k1, k2 . 1/2(k1+k2)(k1+k2+1)+k2.
```

If an id is not explicity provided, as in `flip2 = ~ flip`, it is automatically generated using the macro `uid` as described above.

D.2. Nesting

OMEGA$_C$ allows us to express flattened versions of nested **`let`**, **`do`**, and `l` expressions.

In the case of **`let`**, this means that the following OMEGA$_C$:

```
let A = a,
    B = b,
    C = c
      in t
```

is equivalent to the following λ_C code:

```
let A = a in
  (let B = b in
    (let C = c in t))
```

In the case of **do** this means that the following OMEGA$_C$ code:

```
Y | do(A → a, B → b, C → c)
```

is equivalent to the λ_C code:

```
((Y | do(A → a)) | do(B → b)) | do(C → c)
```

Note that **do** is not commutative, the first intervention is applied first to produce a new variable, upon which the second intervention is applied. Reversing the order would not in general produce the same result if the interventions affect overlapping variables.

In the case of λ this means that the OMEGA$_C$ code:

```
λ a b x . a + b + c
```

is equivalent to the λ_C code:

```
λ a . λ b . λ c . a + b + c
```

E. Comparison With Other Languages

E.1. Multiverse

Below is an example taking from the Multiverse (22) documentation, translated into OMEGA$_C$:

```
let
  x = bern(0.0001),
  z = bern(0.001),
  x_or_z = x or z,
  y = if bern(0.00001)
      then not x_or_z else x_or_z,
    in (y | do(x → 0) | y == 1
```

The corresponding Multiverse code is:

```
def cfmodel():
    x = BernoulliERP(prob=0.0001, proposal_prob=0.1)
    z = BernoulliERP(prob=0.001, proposal_prob=0.1)
    y = ObservableBernoulliERP(
        input_val=x.value or z.value,
        noise_flip_prob=0.00001,
        depends_on=[x, z]
    )
    observe(y, 1)
    do(x, 0)
    predict(y.value)
results = run_inference(cfmodel, 10000)
```

E.2. Pyro

Pyro is popular python based probabilistic programming language, with some support for causal queries. Below is the rifleman example taken from Pearl expressed in both OMEGA$_C$ and Pyro.

In OMEGA$_C$:

```
let p = 0.7,
  q = 0.3,
  Order = ~ bern(p),
  Anerves = ~ bern(q),
  Ashoots = Order or Anerves,
  Bshoots = Order,
  Dead = Ashoots or Bshoots,
  Dead_cf = (Dead | do(Ashoots → 0)) | Dead,
  in rand(Dead_cf)
```

In Pyro:

```
p = 0.7
q = 0.3
exogenousδists = {
    "order": Bernoulli(torch.tensor(p)),
    "Anerves": Bernoulli(torch.tensor(q))
}

def rifleman(exogenousδists):
    order = pyro.sample("order",
                    exogenous_dists["order"])
    Anerves = pyro.sample("Anerves",
                    exogenous_dists["Anerves"])
    Ashoots = torch.logical∨(order, Anerves)
    Bshoots = order
    dead_ = dead = torch.logical∨(Ashoots,
                                  Bshoots)

    dead = pyro.sample("dead",
                       dist.Delta(dead))

    return {"order" : order,
            "Anerves" : Anerves,
            "Ashoots" : Ashoots,
            "Bshoots" : Bshoots,
            "dead" : dead}

cond = condition(rifleman,
    data={"dead": torch.tensor(1.0)})

posterior = Importance(
    cond,
    numσamples=100).run(exogenousδists)

order_marginal = EmpiricalMarginal(posterior,
                                   "order")
```

```
order_samples = [order_marginal().item()
                 for _ in range(1000)]

Anerves_marginal = EmpiricalMarginal(posterior,
                                     "Anerves")
Anerves_samples = [Anerves_marginal().item()
                   for _ in range(1000)]

cf_model = pyro.do(rifleman,
                   {'Ashoots': torch.tensor(0.)})
updated_exogenous_dists = {
    "order": dist.Bernoulli(
             torch.tensor(mean(order_samples))),
    "Anerves": dist.Bernoulli(
             torch.tensor(mean(Anerves_samples)))
}
samples = [cf_model(updated_exogenous_dists)
           for _ in range(100)]
b_samples = [float(b["dead"]) for b in samples]
print("CF prob death is", mean(b_samples))
```

In short, this example samples from the posterior of the exogenous variables, then constructs a new model where (i) these exogenous variables take their posterior values, and (ii) the model structure has been changed through an intervention.

E.3. Differences

The main differences are:

1. Pyro performs sampling to construct a posterior, then intervenes, and then resimulates. That it, it computes the (approximate) posterior at an intermediate state. In contrast, in OMEGA$_{\text{C}}$ one constructs a counterfactual generative model, which is itself a first class random variable. One then later performs inference (such as sampling) on that model. The disadvantage of the Pyro approach is that it ties an inference procedure to the definition of a counterfactual. A practical limitation from this is that composite queries, such as intervening both the counterfactual and factual world become problematic. It is not clear how this could be expressed in Pyro, and evven if possible would involve performing inference twice, and the accumulation of approximation errors that would entail.

2. Multiverse is built around an importance sampling approach, whereas OMEGA$_{\text{C}}$ is entirely agnostic to the probabilistic inference procedure used.

3. Multiverse specifies the interventions and conditioning through imperative operations. As shown in the example above, `observe(y, 1); do(x, 0)` first observes y and then intervenes x. It is not clear, without a semantics or some other guide, whether other compositions are expressible and sound, such as conditioning the intervened world (or both), intervening a value to be a function of its non-intervened (and posterior) self, or stochastic interventions.

4. Although it can be emulated as shown in the example above, Pyro does not enforce the exogenous/endogenous divide. In Pyro, the primitives are actually distribution families, such as Normal(μ, σ), whereas in OMEGA$_{\text{C}}$ the primitives are parameterless exogenous variables, and distribution families are transformations of these primitives. This is important because it allows OMEGA$_{\text{C}}$ to give meaning to an expression such as `let μ = 0, X = normal(μ,1) in X | do(μ → 2)`, because the process by which X is generated is fully specified. This would not make sense in Pyro, where families are primitives.

5. Pyro and Multiverse can only intervene named random variables, whereas in OMEGA$_{\text{C}}$ can intervene any variable bound to a value. More fundamentally, intervening in OMEGA$_{\text{C}}$ is not a probabilistic construct at all.

6. As a cosmetic (but important for practical usage) matter, since Pyro was not designed from the ground up for counterfactual reasoning, it is very cumbersome and verbose to do. If one advances to more advanced queries, this only exacerbates. Multiverse is less verbose, but requires that you explicitly specify what variables every variable depends on (see `depends_on` in the example), whereas those dependencies are automatic in an OMEGA$_{\text{C}}$ program.

E.4. OMEGA$_{\text{C}}$ vs Julia implementation

The Julia implementation follows the basic structure of OMEGA$_{\text{C}}$. That is, in Julia:

- Random variables are pure functions of Ω, that is, any value `f` of type `T` is a random variable if `f(ω::Ω)` is defined.
- Conditioning is performed through a function `cond`, which maps one random variable into another one which is conditioned. It is defined as:

  ```
  cond(x, y) = ω → y(ω) ? x(ω) : error()
  ```

- Interventions are performed by an operation `intervene`, which maps one random variable into one which is intervened.

However, as mentioned in the introduction, conventional programming language do not provide a mechanism to redefine a program variables retroactively, making it difficult

to implement `intervene`. To circumvent this, we take advantage of recent developments in Julia which permit users to write *dynamic compiler transformations* (25), which enables us to perform certain kinds of non-standard execution. To demonstrate, consider the following example:

```
c = 5
X(ω) = ~ unif(ω)
Y(ω) = X(ω) + c
Y = intervene(Y, X, 10)
```

Here, `Y` is a random variable. Using Julia's dynamic compiler transformations, we are able to modify the standard interpretation of `Y(ω)` to intercept the application `X(ω)` within `Y`, such that it instead returns the constant 10. Hence `Y` will be a constant random variable that always returns 15.

Our Julia implementation shares two key properties with OMEGA$_C$: (i) that random variables are pure functions on a single probability space, and (ii) that the conditioning and intervention operators are higher-order transformations between variables. This allows us to preserve many of the important advantages of OMEGA$_C$, such as the ability to systematically compose different operators to construct a wide diversity of the different causal questions outlined in Section 3. The main limitation of this approach is that only random variables can be intervened, unlike any bound variable in OMEGA$_C$. If we want a variable to be intervened, such as the constant `c` in the above example, we must explicitly construct a constant random variable `c(ω) = 5`.