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 relates interventions to lazy-dynamic scope.
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 repository:

`https://github.com/jkoppel/omega-calculus`

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

`https://github.com/zenna/Omega.jl`

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 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).

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

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

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

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

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

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

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

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

RetroUpd : Env × Var × Closure → 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

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 $\bot$ value indicates an undefined value, and any term which strictly depends on $\bot$ is also $\bot$, as indicated by $\bot$VAL, $\bot\text{BINOP}_1$, and similar rules. Conditioning one random variable t_1 on another random variable t_2 is then de-

fined 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 (29).

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

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 implement the $\sim$ operator, 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 `id` and a random variable `X` to a new random variable that is i.i.d. with `X`:

```
-- 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 λ expressions.

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

```
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 c . 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 (23) 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 a 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 is, it computes the (approximate) posterior at an intermediate state. In contrast, in OMEGA$_C$ 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 even 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$_C$ 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 seman-

tics 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$_C$ the primitives are parameterless exogenous variables, and distribution families are transformations of these primitives. This is important because it allows OMEGA$_C$ 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$_C$ can intervene any variable bound to a value. More fundamentally, intervening in OMEGA$_C$ 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 progresses 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$_C$ program.

E.4. OMEGA$_C$ vs Julia implementation

The Julia implementation follows the basic structure of OMEGA$_C$. 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 languages do not provide a mechanism to redefine 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* (26), 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`.