# A Language for Counterfactual Generative Models

**Zenna Tavares, James Koppel, Xin Zhang, Armando Solar-Lezama**
MIT CSAIL / BCS

## Abstract

Probabilistic programming languages provide syntax to define and condition generative models but lack mechanisms for counterfactual queries. We introduce OMEGA$_C$: a causal probabilistic programming language for constructing and performing inference in counterfactual generative models. In OMEGA$_C$, a counterfactual generative model is a program that combines both conditioning and causal interventions to represent queries such as "given that X is true, what if Y were the case?". We define the syntax and semantics of OMEGA$_C$ and demonstrate examples in population dynamics, inverse planning and causation.

## 1 Introduction

Probabilistic programming languages provide syntax to define and condition generative models. However, conditioning alone is insufficient to express counterfactuals. Counterfactuals combine both conditioning and intervention in order to represent the *what-if* scenarios found in the sciences, law, medicine and various aspects of everyday life. Existing languages lack generic mechanisms for counterfactual inference.

In this paper we introduce OMEGA$_C$, a programming language for constructing counterfactual generative models and performing inference in these models. A counterfactual generative model combines both conditioning and intervention, taking the general structure: "given that $X$ is true, what if $Y$ were the case?". Crucially, $Y$ being the case can invalidate the truth of $X$. A simple example is: given that a drug treatment was not effective on a patient, would it have been effective at a stronger dosage. In OMEGA$_C$, a generative model is a functional program that computes the output of random variables from random inputs; conditioning a model (e. g. by observing that $X$ is true) restricts those outputs to be consistent with data, and an intervention $Y$ modifies the structure of the model.

To illustrate the expressive power of our approach, consider the scenario of a team of ecologists who arrive at a forest to find it overrun with invasive rabbits. The ecologists may want to know whether introducing a pair of wolves a few months back could have prevented this situation. Given a model of population dynamics such as the Lotka-Volterra model, it is possible to make forward predictions about the effect of an intervention. The problem is that in order to use this model, the ecologists would have to know the exact state of the ecosystem before the intervention, from which the model could be run forward with and without the intervention. With a traditional probabilistic programming language, the ecologists could define a prior distribution over the initial state of the ecosystem, and then condition on the observed number of rabbits to infer the posterior. But they cannot easily explore the effect of the intervention on this posterior, since the observation they are conditioning on is directly affected by the intervention. *Causal graphs* could in principle be used to answer such a counterfactual question, but they are not expressive enough to capture the details of the Lotka-Volterra model.

Causal graphs were introduced by Pearl to formalize causal relationships, using edges to connect cause and effect [1]. In causal graphs, interventions are manipulations to the graph structure. One reason causal graphs are not sufficiently expressive for our running example is that causal graphs are static; they have a fixed number of edges and nodes, and interventions are limited to a fixed

Preprint. Under review.

static edge. They effectively represent straight-line programs, in a language which lacks recursion, and where the intervenable variables are bounded in number and static, and where the functions are defined externally to the system. By contrast, OMEGA$_C$ supports models that include recursion, and interventions can be dynamically determined, e. g. instead of introducing a wolf at a specific time, the intervention can introduce the wolf when the rabbit population reaches a given threshold.

OMEGA$_C$ extends a prior probabilistic language [2] with a version of Pearl's **do** operator:

$$X \mid \text{do}(\Theta \to Z) \tag{1}$$

**do** performs a dynamic program transformation such that Expression 1 evaluates to a value that $X$ would have taken had $\Theta$ been bound to $Z$ at the point $X$ was defined. This allows us change the internal structure of previously defined random variables (such as $X$) without apriori having to know what interventions (such as $\Theta \to Z$) we might like to make. For example, if $\Theta = \mathcal{N}(0, 1)$ and $X = \mathcal{N}(\Theta, 1)$ then $X \mid \text{do}(\Theta) \to \text{Beta}(10, 1)$ has a beta distribution as its mean, rather than a normal distribution. This simple example can be expressed in a causal graph. OMEGA$_C$ supports a much richer class of models, conditions, interventions and counterfactuals as a result.

In summary, we (i) present the syntax and semantics of the first universal probabilistic language for counterfactual generative models (Section 3); and (ii) demonstrate examples in competitive population models, inverse planning and but-for causation (Section 4). Regarding scope, causal inference includes problems of both (i) inferring a causal model from data, and (ii) given a (partially-specified) causal model, predicting the result of interventions and counterfactuals on that model. This work focuses solely on the latter.

## 2 Example: Pendulum Dynamics

Here we give a brief tour of OMEGA$_C$. We construct the counterfactual: given a pendulum of length $\ell$ (whose angle $\theta$ dynamics are governed by $\ddot{\theta} = -(g/\ell)\sin(\theta)$) and an observation of $\theta$ at some time, what would the dynamics have been if $\ell$ or $g$ were different. Figure 1 shows trajectories.

In OMEGA$_C$, variables are bound with `let`:

```
let g = 9.8, l = 1, w = - g * l
```

Functions (e.g., $\ddot{\theta}$) are defined with $\lambda.x, y, \ldots$ . . . :

```
let θ'' = λθ . - w * sin(θ)
```

`sim` solves the ODE. It returns $(\theta_t, \theta_{t+\Delta t}, \ldots, \theta_{t_{max}})$:

```
let sim = λθ, θ', t, tmax, Δt.
  if t < tmax
    let θ'n = θ' - Δt * θ''(θ)
        θn = θ + Δt * θ'  in
    cons(θn, sim(θn, θ'n, t + Δt, tmax, Δt))
  else
   emptylist,
```

Parametric families are a way to create random variables:

```
let u0 = uniform(0, 1), u'0 = uniform(0, 1)
```

`rand` returns a sample from a random variable:

```
let u0sample = rand(u0)
```

`sim` applied to `u0` and `u'0` is also a random variable.

```
let simrv = sim(u0, u'0, 0, 10, 0.001}
```

$x \mid y$ is $x$ conditioned on $y$. To observe $\theta_{t_{max}} = 0.5$:

```
let simrvcond = simrv | last(simrv) == 0.5
```

**do** intervenes. It enables the counterfactuals: given $\theta_{t_{max}} = 0.5$, what if $\ell$ or $g$ were different?

```
let simcf1 = (simrv | do(l → 2.0)) | last(simrv) == 0.5,
    simcf2 = (simrv | do(g → 2.0)) | last(simrv) == 0.5
```

**Pendulum Trajectories**

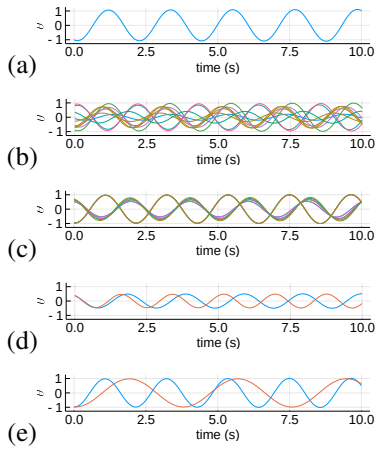



Figure 1: (a) Time series of pendulum angle $\theta$ computed with `sim`. (b) samples given prior over $\theta$ and $\dot{\theta}$ at $t_0$, (c) posterior given $\theta = 0.5$ at $t_{10}$, (d, e) counterfactuals: given $\theta = 0.5$, what if $\ell \to 2.0$ or $g \to 4.35$

## 3 A Calculus for Counterfactuals

Our language OMEGA$_C$ augments the functional probabilistic language OMEGA [2] with counterfactuals. We achieve this with two modifications: (1) the syntax is augmented with a **do** operator, and (2) the language evaluation is changed from eager to lazy, which is the key to the mechanism of handling interventions.

In this section, we introduce $\lambda_C$, a core calculus of OMEGA$_C$, in which we omit language features that are irrelevant to explaining counterfactuals. 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 randomness/conditioning give the language the ability to do counterfactual inference. Appendix A gives a more formal definition of the entire $\lambda_C$ language.

### 3.1 Deterministic Fragment

Variables $x, y, z \in \text{Var}$

Type $\tau ::= \text{Int} \mid \text{Bool} \mid \text{Real} \mid \tau_1 \to \tau_2$

Term $t ::= n \mid b \mid r \mid t_1 \oplus t_2 \mid x \mid \textbf{let}\ x = t_1\ \textbf{in}\ t_2 \mid \lambda x : \tau . t \mid t_1(t_2) \mid \textbf{if}\ t_1\ \textbf{then}\ t_2\ \textbf{else}\ t_3$

Figure 2: Abstract Syntax for $\lambda_C$, deterministic fragment

We begin by presenting the the fragment of $\lambda_C$ for deterministic programming. This part of the language is standard in many calculi. Fig. 2 gives the abstract syntax.

A common formal way to specify the executions of a program is with an *operational semantics* [3], which defines a language's semantics as a relation of how one expression in the language reduces to another. Appendix A provides a complete operational semantics for OMEGA$_C$. Here, we describe these relations through concrete examples. The execution of an expression is defined both in terms of the expression as well as the current program state. In $\lambda_C$, this program state is an environment $\Gamma$, which is a mapping from variables to values.

The deterministic fragment is standard, so we will explain it briefly. $\lambda_C$ 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 are used to create functions. Function applications and if-statements are standard.

The first example demonstrates the semantics of binary operators and the let statement. The let expression first evaluates the expression $2 + 1$ and then binds $x$ to the result in the environment. Finally, $x$ is evaluated by looking up its value in the environment.

$$\left\{\begin{matrix}\Gamma : \emptyset \\ \textbf{let}\ x = 2 + 1\ \textbf{in}\ x\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : \emptyset \\ \textbf{let}\ x = 3\ \textbf{in}\ x\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : x \mapsto 3 \\ x\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : x \mapsto 3 \\ 3\end{matrix}\right\}$$

The next example explains function applications, which are done by substitution, as in other variants of the lambda calculus.

$$\left\{\begin{matrix}\Gamma : \emptyset \\ (\lambda x : \text{Int} . (x + 1) * x)(2)\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : \emptyset \\ (2 + 1) * 2\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : \emptyset \\ 3 * 2\end{matrix}\right\} \to \left\{\begin{matrix}\Gamma : \emptyset \\ 6\end{matrix}\right\}$$

The above semantics is eager: **let** $x = t_1$ **in** $t_2$ first evaluates $t_1$ and then binds the result to $x$. We next show how this is problematic for implementing counterfactuals and how we address it by changing the semantics to lazy.

### 3.2 Causal/Counterfactual Fragment

We introduce our causal fragment in the context of the above deterministic fragment, which enables intervention. In the next subsection, we will discuss how this fragment interacts with the probabilistic fragment to naturally support counterfactual inference.

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

Figure 3: Abstract Syntax for $\lambda_C$, 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 $t_1$ at point of definition. One idea is to define **do** similarly to **let**: $t_1 \mid \textbf{do}(x \to t_2)$ rebinds $x$ 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 need to redefine **let** to use *lazy evaluation*, which naturally tracks the provenance of all values.

Lazy evaluation works as follows: instead of storing the value of a variable in the environment, the execution stores its defining expression. Moreover, since a variable can be redefined, which can change the variable definitions using it in unexpected ways, the execution also tracks the environment when each variable is defined. So, while environments for eager evaluation stored mappings $x \mapsto v$ from each variable $x$ to a 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 $\Gamma$ in which it was defined. A variable, such as $x$, is evaluated by evaluating its definition under the environment where it is defined, which potentially involves evaluating other variables similarly.

It is now straightforward to define **do**: the intervention $y \mid \textbf{do}(x \to -1)$ evaluates $y$ under a new environment which is created by mapping all $x$ in the current environment to $-1$. This not only includes the binding of $x$ at the top level but also the bindings in an environment that is used in any closure. The following example demonstrates this process.

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

### 3.3 Probabilistic Fragment

| 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 $\lambda_C$, probabilistic fragment

In measure-theoretic probability theory, a random variable is defined as a function from a sample space $\Omega$ to some domain of values $\tau$. $\lambda_C$ defines random variables similarly: as functions of type $\Omega \to \tau$. Doing so separates the source of randomness of a program from its main body, which leads to a clean definition of counterfactuals.

Fig. 4 shows the abstract syntax of the probabilistic fragment. It introduces a new type $\Omega$, representing the sample space. $\Omega$ is left unspecified, save that it may be sampled from uniformly. In most applications, $\Omega$ 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 from Section 3.1 must be the indexing operator $[]$, where $\omega[i]$ evaluates to of the $i$th componentt of $\omega$.

Random variables are constructed as a normal function. If $\Omega = [0, 1]$, and $a < b$ are fixed integer constants, then $R = \lambda\omega : \Omega.\omega * (b - a) + a$ represents a random variable uniformly distributed

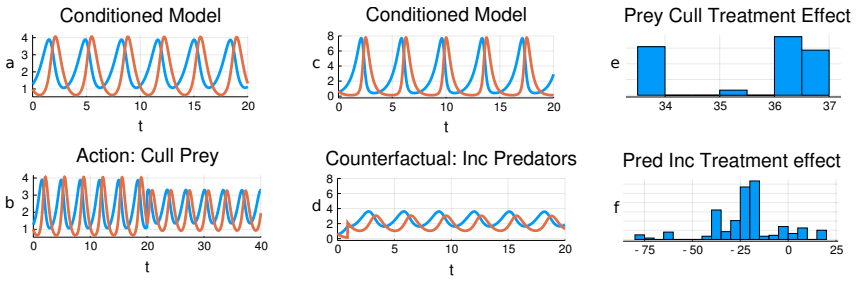



Figure 5: Lotka-Volterra Predator-Prey differential equations. (a, c) Samples from timeseries conditioned too many rabbits, (b) Effect of action: increasing number of prey at $t_{\text{now}}$, (d) Sample from counterfactual model: conditioned model with intervention in past (e) Treatment effect of culling prey (f) Treatment effect of increasing predators

in $[a, b]$. The **rand** operator then samples from a random variable: **rand** $R$ returns a random value uniformly drawn from $[a, b]$.

To support conditioning, we use $\perp$ to denote the undefined value. Any non-**rand** expression that depends on a $\perp$ value will result in another $\perp$ value. A program execution is invalid if it evaluates to $\perp$. One can imagine the execution of a $\lambda_C$ program as a rejection sampling process: we ignore all samples from **rand** that would make the program evaluate to $\perp$. In the implementation, we use a much more efficient inference algorithm [4].

Conditioning can now be defined as syntactic sugar: $t \mid P$ desugars to $\lambda\omega$.**if** $P(\omega)$ **then** $t(\omega)$ **else** $\perp$.

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

Conditioning and intervention compose naturally to yield counterfactuals. Consider the following program to depict a game where a player chooses a number $c$, and then a number $\omega$ is drawn randomly from a sample space $\Omega = \{0, 1, \ldots, 6\}$, and the player wins iff $c$ is within 1 of $\omega$. 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 inner **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 $\lambda\omega'.\textbf{if } x(\omega') == -1 \textbf{ then } (x \mid \textbf{do}(c \to 4))(\omega')$ **else** $\perp$. This random variable evaluates to $\perp$ for $\omega' \in \{0, 1, 2\}$, so the program is evaluated with $\omega'$ 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 $\omega'$, and the overall computation hence evaluates to 1 with probability $\frac{3}{4}$ and $-1$ with probability $\frac{1}{4}$.

## 4 Experiments

Here we demonstrate counterfactual reasoning in OMEGA$_C$ through three case studies.

**Experimental Setup** All experiments were performed using predicate exchange [4], the default inference procedure in OMEGA, on a single workstation. Simulation parameters and code for all examples are in the supplementary material.

**Predator-Prey Population Dynamics** The Lotka-Volterra model is a pair of differential equations which represent interacting populations of predators (e.g. wolves) and prey (e.g. rabbits): $\dot{x} = \alpha x - \beta xy$ and $\dot{y} = \delta xy - \gamma y$. $x(t)$ and $y(t)$ represents the prey and predator populations. $\alpha, \beta, \delta$ and $\gamma$ represent growth rates.

After observing for 10 days until $t_{\text{now}} = 20$, we discover that the rabbit population is unsustainably high. We want to ask counterfactual questions: how would an intervention now affect the future; had we intervened in this past, could we have avoided this situation?

To solve the differential equations we define the function `euler` which implements Euler's method [5]. `euler` maps the derivative `f'` and initial conditions `u0` to a time series of pairs $u_1, \ldots, u_n$ where $u_i = (x_i, y_i)$, sampled at timesteps $t_{\text{min}}, t_{\text{min}} + \Delta t, t_{\text{min}} + 2\Delta t, ..., t_{\text{max}}$

```
1  let euler = λf', u, t, tmax, Δt.
2   if t < tmax
3    let unext = u + f'(t + Δt, u) * Δt in
4     cons(u, euler(f', unext, t + Δt, tmax, Δt))
5   else
6    emptylist,
```

We put priors on initial conditions and parameters:

```
7  u0 = (normal(0, 1), normal(0, 1)), α = normal(0, 1), β = normal(0, 1), γ = normal(0, 1), δ
   = normal(0, 1),
```

Next, we construct `lk'`: a random variable over derivative functions following Equation **??**, where $\alpha, \beta, \delta$ and $\gamma$ are the previously defined random variables. In other words, a sample from `lk'` is a function which maps a pair $u = (x, y)$ and current time `t` to the derivative with respect to time.

```
8  getx = λu. first(u),
9  gety = λu. second(u),
10 lk' = λω. λt, u. let x = getx(u), y = gety(u) in
11   (α(ω)*x - β(ω)*x*y, -γ(ω)*y + δ(ω)*x*y),
```

Next, we complete the unconditional generative model. Since `lk'` is a random variable, so is `series`.

```
12 series =  euler(lk', u0, 0, 20, 0.1),
```

Next, we condition the prior on the observation that an average of 5 rabbits have been observed over the last 10 days. We use a function `lastn(seq, n)` to extract the last `n` elements of a `seq`, `mean` to compute the average, and `map` to extract only the rabbit values from each pair. Figure 5 (a) shows a conditional sample.

```
13 last10 = lastn(series, 10),
14 rabbits10 = map(gety, last10),
15 toomanyrabbits = mean(rabbits10) == 5,
16 series_cond = series | toomanyrabbits,
```

Next, we examine the effect of action[1]. In particular, if we were to increase the prey population by 5 at $t_{\text{now}}$, would the rabbit population be reduced (Figure 5 (b))? First, we construct an alternative version of `euler`, one which modifies the value of `u` at some time `t_int` by applying a function `u_int`. We will call this function `eulerint`. Since we will soon perform another similar intervention in the next subsection for counterfactuals, we construct here a template function `eulergen` which parameterizes over `t_int` and `u_int`:

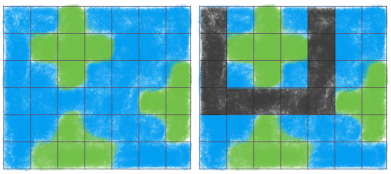
```
17 eulergen = λt_int, u_int
18  λf', u, t, tmax, Δt.
19   let u = if t == t_int then u_int(u) else u in
20    if t < tmax
21     let unext = u + f'(t + Δt, u) * Δt,
22           eul = eulergen(t_int, u_int) in
23      cons(u, eul(f', unext, t + Δt, tmax, Δt))
24    else
25     emptylist,
```

The next snippet intervenes on `series` using **do** to replace `euler` with an alternative version `eulerint` which increases the number of prediators. Figure 5 (b) shows a sample.

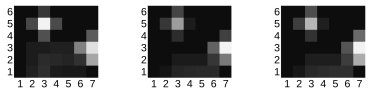
```
26 inc_pred = λu.(getx(u)/2, gety(u)),
27 eulerint = eulergen(20, inc_pred),
28 series_act = series_cond | do(euler → eulerint),
```

Next, we consider the counterfactual: had we made an intervention at some previous time $t < t_{\text{now}}$, would the rabbit population have been less than it actually was over the last 10 days? Choosing a fixed time to intervene (e.g. $t = 5$) is likely undesirable because it corresponds to an arbitrary (i.e.: parameter dependent) point in the predator-prey cycle. Instead, the following snippet selects

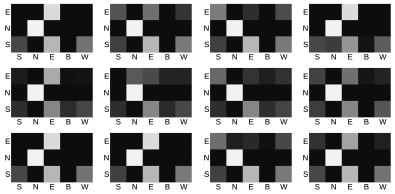
[1] According to Pearl, action means intervening on the random variable being observed, which does not affect the past.



(a) Three islands $S$, $N$, $E$ without (left) and with (right) border under consideration



(b) Sample from population counts after $n$ timesteps of MDP based migration. (Left) Unconditional sample, (b) Conditional sample (c) Counterfactual



(c) Four samples of migration patterns under different 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 in each persons' trajectory, not only the final state. (Top) Prior samples, (Middle) Conditioned on observations, (Bottom) counterfactual: conditioned on observations with intervention (border)

the intervention dynamically as a function of values in the non-intervened world. `maxindex` is an auxilliary function which selects the index of the largest value and hence `tmostwolves` is a random variable over such values.

```
29 tmostwolves = maxindex(series),
30 inc_wolves = λu.(getx(u), gety(u)+2),
31 inc_euler = eulergen(t_mostwolves, inc_wolves),
32 series_cf = λω.(series_cond | do(euler → inc_euler(ω)))(ω)
```

The primary purpose of using probailistic models is to capture uncertainty over estimates. Figure 5 (e) and (f) are sample histograms showing the treatment effect [1] of the action (culling at $t_{now}$), and the counterfactual (increasing predators in the past). While the samples in (e) are from `sum(series_act) - sum(series_cond)`, those in (f) are from `sum(series_cf) - sum(series_cond)`.

**Counterfactual Planning** Consider a migration dispute between three hypothetical island nations (Figure 6a Left): $S$ to the South, $E$ to the East and $N$ to the North. The government of $S$ considers a barrier between $S$ and $N$ (Figure 6a Right), asking the counterfactual: given an observation of migration patterns, how would they differ had a border been constructed.

We model this as a population of agents each acting according in accordance to a Markov Decision Process [6] (MDP) model. Each grid cell is a state in a state space $\mathcal{S} = \{(i, j) \mid i = 1 \dots 7, j = 1 \dots 6\}$. The action space 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 $R : \mathcal{S} \to \mathbb{R}$. 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 6b shows population counts according to these dynamics. Figure 6c 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 use interventions to implement "but-for" 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 $E$ [7]. But-for judgements cannot be resolved by conditioning on the negation of $C$, since this fails to differentiate cause from effect. Instead, we must find a alternative world where $C$ does not hold. But-for is a form of *token* causality [8] since it refers to concrete events. In OMEGA$_C$, a value $\omega \in \Omega$ encompasses all the uncertainty, and hence we define but-for causality relative to a concrete value $\omega$.

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

$$(E \mid \mathbf{do}(C_1 \to \hat{c}_1, \dots, C_n \to \hat{c}_n))(\omega) = \text{False} \tag{2}$$

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

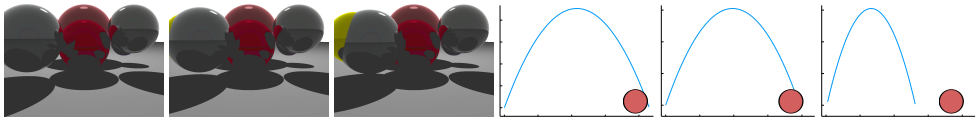


Figure 7: But-for causality. Left to Right: stages of optimization to infer that grey-sphere is cause of inability to see yellow sphere, and launch-angle is cause of projectile colliding with ball.

But-for is defined existentially. To solve it we rely on predicate relaxation [4] that underlies inference in OMEGA$_C$. 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 intersectections. Predicate relaxation transforms $E$ into soft predicate $\tilde{E}$ which returns a value in $[0, 1]$ denoting how close we are to satisfying $E$. Using this, we use gradient descent over $\hat{c}_1, \ldots, \hat{c}_n$ to minimize $(\tilde{E} \mid \textbf{do}(C_1 \to \hat{c}_1, \ldots, C_n \to \hat{c}_n))(\omega)$. 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 7 we visualize the optimization, which ultimately infers that the angle is the cause of collision and the grey-sphere is the cause of our inability to see the yellow sphere.

## 5 Related Work

Operators resembling **do** appear in existing probabilistic programming languages. Venture [9] has a force expression `[FORCE <expr> <literal-value>]` which modifies the current trace so that the simulation of `<expr>` takes on the value `<literal-value>`. It is intended as a tool for for controlling initialization and debugging. RankPL [10] is a language similar to probablistic programming languages, but 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. [11] described a recipe to encode counterfactuals in P-log, a probabilistic logic programming language. However, no language construct is provided to automate this process. There are also several libraries for doing causal inference on traditional causal graphs [12, 13, 14, 15, 16].

There has also been work on adding causal operators to deterministic programming paradigms. Halpern and Moses [17] investigated counterfactuals in the context of knowledge-based programming. They show that the counterfactual conditional can be used to specify that a system's actions may depend on predicted future events, even when those future events themselves depend on the system's actions. Cabalar [18] investigated causal explanations in answer-set programming, arguing that an explanation for a derived fact is best given by a derivation tree for that fact. Pereira et al. [19] proposed a process of implementing counterfactuals in logic programming using abduction and updating, and applied it to model agent morality.

## 6 Discussion

**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.

**Validity of interventions.** Not every intervention should be considered valid. For instance, a program may have been written assuming a variable $x$ is positive; intervening to set it negative may cause the program to behave erratically, or perform an invalid operation such as an out-of-bounds array access. Existing work addresses how to check if a probabilistic program meets certain correctness specifications [20]. We can extend any such correctness condition to define whether an intervention is valid. Briefly, any OMEGA$_C$ program with interventions can be rewritten to a vanilla probabilistic program without **do** through a systematic transformation. Specifically, for $t_1 \mid \textbf{do}(x \to t_2)$, one can manually copy the definition of $t_1$ and replace all occurrences of $x$ with $t_2$. An intervention is correct if and only if the corresponding transformed program meets its correctness criteria.

**Limitations.** Procedures such as the PC algorithm [21] 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.

## References


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

[2] 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.

[3] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[4] 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.

[5] John Charles Butcher. *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

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

[7] Joseph Y Halpern and Christopher Hitchcock. Actual causation and the art of modeling. *arXiv preprint arXiv:1106.2652*, 2011.

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

[9] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

[10] 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.

[11] 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.

[12] Joshua Brulé. Whittemore: An embedded domain specific language for causal programming. *arXiv preprint arXiv:1812.11918*, 2018.

[13] Amit Sharma and Emre Kiciman. DoWhy: Making causal inference easy. `https://github.com/Microsoft/dowhy`, 2018.

[14] Santtu Tikka and Juha Karvanen. Identifying causal effects with the r package causaleffect. *Journal of Statistical Software*, 76(1):1–30, 2017.

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

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

[17] Joseph Halpern and Yoram Moses. Using counterfactuals in knowledge-based programming. volume 17, pages 97–110, 07 1998.

[18] Pedro Cabalar. Causal logic programming. In *Correct Reasoning*, pages 102–116. Springer, 2012.

[19] 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.

[20] Benjamin Bichsel, Timon Gehr, and Martin T. Vechev. Fine-grained semantics for probabilistic programs. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*, pages 145–185, 2018.

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