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 O\(\text{MEGA}_C\), 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 \(x\), what is the probability that it would 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 through examples in a variety of simulation models.

1. Introduction

In this paper we introduce O\(\text{MEGA}_C\): a Turing-universal programming language for causal reasoning. O\(\text{MEGA}_C\) allows users to automatically derive causal inferences about phenomena modelled through simulation. This contribution focuses on using O\(\text{MEGA}_C\) to compute counterfactuals – what-if causal inferences about the way the world could have been, had things been different.

O\(\text{MEGA}_C\) programs are simulation models augmented with probability distributions to represent uncertainty. In a similar vein to other probabilistic languages, O\(\text{MEGA}_C\) provides primitive operators for conditioning, which revises the model to be consistent with observed evidence. Counterfactuals, however, cannot be expressed through probabilistic conditioning alone. They have 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.

In order to express these hypothetical scenarios, O\(\text{MEGA}_C\) introduces a do operator, which constructs interventions:

\[
Y \mid \text{do}(X \rightarrow x)
\]  

(1)

This evaluates to what \(Y\) would have been had \(X\) been bound to \(x\) when \(Y\) was defined. Here, \(X\) and \(Y\) are program variables, typically bound to random variables.

A counterfactual in O\(\text{MEGA}_C\) is then simply an expression of the form \(Y_x \mid E\) where \(Y_x = Y \mid \text{do}(X \rightarrow x)\), i.e., one that contains both a condition and an intervention, in a particular pattern. The salient feature of this counterfactual pattern is that conditioning on \(E\) revises the distribution over \(Y\) (because \(E\) is defined in terms of \(Y\), not \(Y_x\)), and it is to this revised distribution that a causal intervention is performed. The relative nesting of the condition and intervention reflects the fact that we want to intervene \(Y\) but not on the evidence \(E\).

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 (Fig. 1), whether a driver was to blame for them crashing into a pedestrian. Using O\(\text{MEGA}_C\), the expert could first construct a probabilistic model that includes the car dynamics, the driver and pedestrian’s behaviour, and a rendering function that produces two dimensional images...
A Language for Counterfactual Generative Models

from the three dimensional scene. She could then condition the model on the captured images to infer the conditional distribution over the driver’s velocity, determining the probability that the driver had been speeding. Next, she could then pose a counterfactual in OMEGAC, querying whether the crash would have still occurred even if the driver had obeyed the speed limit. If she later wanted to investigate the culpability of another candidate cause, such as the presence of an obstacle occluding the driver’s view, 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 (21): 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.

OMEGAC, in contrast, can express complex simulation models, but the design of a generic do operator 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 OMEGAC 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 OMEGAC, and (iii) demonstrate counterfactual generative modelling through a number of examples (Section 5). 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. We focus here on the latter.

### 2. Overview of Counterfactuals

Counterfactual claims assume some structure is invariant from the three dimensional scene. She could then condition the model on the captured images to infer the conditional distribution over the driver’s velocity, determining the probability that the driver had been speeding. Next, she could then pose a counterfactual in OMEGAC, querying whether the crash would have still occurred even if the driver had obeyed the speed limit. If she later wanted to investigate the culpability of another candidate cause, such as the presence of an obstacle occluding the driver’s view, 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 (21): 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.

OMEGAC, in contrast, can express complex simulation models, but the design of a generic do operator 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 OMEGAC 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 OMEGAC, and (iii) demonstrate counterfactual generative modelling through a number of examples (Section 5). 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. We focus here on the latter.

### 3. A Calculus for Counterfactuals

Our language OMEGAC is a simple functional probabilistic language augmented to support counterfactuals. To achieve this: (1) the syntax includes a do operator, and (2) the language evaluation is lazy rather than eager, which is key to handling interventions. In this section, we introduce $\lambda_C$, a core calculus of OMEGAC. After some preliminaries, we show the deterministic semantics of the language, followed by its probabilistic features. Together, intervention and conditioning give the language the ability to do counterfactual inference. Appendix A gives a more formal definition of the entire $\lambda_C$ language. A Julia implementation of OMEGAC can be found at https://github.com/zenna/Omega.jl, and...
a Haskell implementation of $\lambda_C^*$ can be found at https://github.com/jkoppel/omega-calculus.

<table>
<thead>
<tr>
<th>Variables</th>
<th>$x, y, z \in \text{Var}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type $\tau ::= \text{Int}</td>
<td>\text{Bool} \mid \tau_1 \rightarrow \tau_2$</td>
</tr>
<tr>
<td>Term $t ::= n</td>
<td>b</td>
</tr>
</tbody>
</table>

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

**Preliminaries** Here, we introduce the notation to describe the semantics of a simple deterministic programming language; Fig. 2 gives the syntax. We use the formalism of operational semantics (24) to describe how one expression reduces to another. Appendix A provides an operational semantics for OMEGAC. 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 $\lambda_C^*$, this program state is an environment $\Gamma$: a mapping from variables to values.

$\lambda_C^*$ has integer numbers (denoted $n$), Booleans $\{\text{True}, \text{False}\}$ (denoted $b$), and real numbers ($\tau$). $\odot$ represents a mathematical binary operator such as $\times$, $\ast$, etc. $\text{let } x = t_1 \text{ in } t_2$ binds variable $x$ to expression $t_1$ when evaluating $t_2$. Lambda expressions create functions: $\lambda x.2 * x$ defines a mapping $x \mapsto 2x$.

Next, we show the semantics of operators and $\text{let}$. The notation $\{\Gamma_1\}$ denotes a pair of an environment $\Gamma_1$ and an expression $e$, and $\{\Gamma_1\} \rightarrow \{\Gamma_2\}$ denotes that $e_1$ with environment $\Gamma_1$ steps to $e_2$ with environment $\Gamma_2$. For example, in the expression $\text{let } x = 3 \text{ in } x$, $x$ is first bound to 3, creating a new environment. Finally, $x$ is evaluated by looking up its value in the environment.

$$
\begin{align*}
\{ \Gamma : \emptyset \} &\rightarrow \{ \Gamma : x \mapsto 3 \} \rightarrow \{ \Gamma : x \mapsto 3 \}
\end{align*}
$$

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

$$
\begin{align*}
\{ \Gamma : \emptyset \} (\lambda x. (x + 1))(2) &\rightarrow \{ \Gamma : \emptyset \} 2 + 2 \rightarrow \{ \Gamma : \emptyset \} 4
\end{align*}
$$

The above semantics is *eager*: $\text{let } x = t_1 \text{ in } t_2$ 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, and how we address it using lazy semantics.

**Deterministic OMEGAC** OMEGAC adds a new term: the $\text{do}$ expression (Fig. 3). $t_1 \mid \text{do}(x \rightarrow t_2)$ evaluates $t_1$ to the value that it would have evaluated to, had $x$ been defined as $t_2$ at its point of definition. Here, $x$ can be any variable that is in scope, bound locally or globally, and $t$ can be any term denoting a value. One idea is to define $\text{do}$ similarly to $\text{let}$: $t_1 \mid \text{do}(x \rightarrow t_2)$ would rebind $x$ to $t_2$ when evaluating $t_1$. However, this does account for transitive dependencies. For example, $\text{let } x = 0 \text{ in let } y = x \text{ in } (y \mid \text{do}(x \rightarrow 1))$ should evaluate to 1, but by the time we evaluate the $\text{do}$, $y$ has already been bound to 0 so that rebinding $x$ does nothing. To overcome this, we redefine $\text{let}$ 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 $\Gamma$ 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 $\text{do}$: $y \mid \text{do}(x \rightarrow -1)$ 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:

$$
\begin{align*}
\{ \Gamma : \emptyset \} &
\text{let } x = 0 \text{ in let } y = x + 1 \text{ in } y + (y \mid \text{do}(x \rightarrow -1)) \\
1 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0) \} \\
2 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\
3 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\
4 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\
5 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\
6 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\
7 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\
8 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\
9 \rightarrow & \{ \Gamma : x \mapsto (\emptyset, 0), y \mapsto (x \mapsto (\emptyset, 0), x + 1) \\
\end{align*}
$$
The program is evaluated under an empty environment. (1) Evaluating the outermost \textbf{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 \textbf{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.

To implement \textbf{do} we introduce a procedure which we call retroactive-updating. Informally, this creates a new environment that rebinds all occurrences of the intervened variable within a closure to its intervened value. This is formally specified with respect to the operational semantics in the supplementary material.

**Counterfactuals** A counterfactual is a random variable of the form \((t_1 \mid \textbf{do}(x \rightarrow t_2)) \mid E\). Consider the following program depicting a game where a player chooses a number \( c \), and then a number \( \omega \) is drawn randomly from a sample space \( \Omega = \{0, 1, \ldots, 6\} \). He 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?

\[
\begin{align*}
\textbf{let} & \quad c = 1 \\
\textbf{let} & \quad x = \lambda\omega \cdot \text{if } (\omega - c) \ast (\omega - c) < 1 \\
& \quad \text{then } 1 \text{ else } -1 \text{ in} \\
\textbf{let} & \quad \text{cfx} = (x \mid \textbf{do}(c \rightarrow 4)) \mid \lambda\omega. \ (x(\omega) == -1) \text{ in rand(cfxf)}
\end{align*}
\]

As before, the \texttt{rand} expression is evaluated in the context \( \Gamma_1 = \{ c \mapsto (\emptyset, 1), x \mapsto \{ c \mapsto \ldots, \omega \}. \texttt{if} \ldots \} \). Its argument, a conditioning term, desugars to \( \lambda\omega'. \text{if } x(\omega') == -1 \text{ then } (x \mid \texttt{do}(c \rightarrow 4))(\omega') \text{ else } \bot \). This random variable evaluates to \( \bot \) for \( \omega' \in \{0, 1, 2\} \), so the program is evaluated with \( \omega' \) drawn uniformly from \( \{3, 4, 5, 6\} \). The \texttt{do} expression \( x \mid \texttt{do}(c \rightarrow 4) \) is reduced to evaluating \( x \) in the context \( \Gamma_2 = \{ c = \ldots, x = (c \mapsto (\emptyset, 4), \lambda\omega. \texttt{if} \ldots \} \). This is then applied to \( \omega' \) and the overall computation hence evaluates to 1 with probability \( \frac{2}{7} \) and \(-1 \) with probability \( \frac{5}{7} \).

**Syntactic Sugar** \textsc{omegaC} introduces some syntactic conveniences on top of \textsc{lambdaC}. Random variables are functions but it is convenient to treat them as if they were the values in their domains. To support this, \textsc{omegaC} interprets the application of a function to one or more random variables \texttt{pointwise} - if both \( X \) and \( Y \) are random variables, then \( X + Y \) is also a random variable defined as \( \lambda\omega. X(\omega) + Y(\omega) \).
Similarly, if \( x \) is a constant, then \( X = x \) is \( \lambda \omega . X(\omega) = x \). In addition, \( \Omega_{\omega C} \) represents distribution families as functions from parameters to random variables. For instance, \( \text{bern} = \lambda p . \lambda \omega . [\text{bern}][1] < p \) represents the Bernoulli family by mapping a parameter \( p \in [0, 1] \) to a random variable that is true with probability \( p \). Finally, since \( \lambda C \) is purely functional, if \( X = \text{bern}(0.5) \) and \( Y = \text{bern}(0.5) \), then \( X \) and \( Y \) are not only i.i.d. but the very same random variable, which is not often what we want. \( \Omega_{\omega C} \) defines the syntax \( \sim \), so that in \( \text{let} \ X \sim \text{bern}(0.5), Y \sim \text{bern}(0.5) \), \( X \) and \( Y \) are independent.

### 3.1. Other Composite Queries

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

To demonstrate, we adopt an example from (21), 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 \( 0 \) (nervous), (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 \( (D) \) if either shoots. In \( \Omega_{\omega C} \):

\[
\text{let } p = \theta.7, q = \theta.3, \\
E = \sim \text{bern}(p), \quad \text{-- Execution order} \\
C = \sim \text{unif}(0, 1), \quad \text{-- Calmness} \\
N = C < q, \quad \text{-- Nerves} \\
A = E \text{ or } N, \quad \text{-- A shoots} \\
B = E, \quad \text{-- B shoots on order} \\
D = A \text{ or } B \text{ in } \text{ Prisoner Dies}
\]

As we have seen, counterfactuals condition the real world and consider the implications in a hypothetical world, e.g.:

-- Given \( D \), would \( D \) be true had \( A \) not fired?
\( \bigl( D \mid \text{do}(A \rightarrow 0) \bigr) \) \( \mid D \)

**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 \( \Omega_{\omega C} \):

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?
\( \bigl( D \mid \text{do}(B \rightarrow C < q) \bigr) \)

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

-- Would \( D \) occur if it took both shots to kill him?
\( \bigl( D \mid \text{do}(D \rightarrow A \text{ and } B) \bigr) \) \( \mid D \)

**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.

--- If \( A \) were more calm, would \( D \) have occurred?
\( \bigl( D \mid \text{do}(C \rightarrow C + 1.2) \bigr) \)

**Partial compliance** (20) 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?
\( \bigl( D \mid \text{do}(A \rightarrow \text{if } \sim \text{bern}(s) \text{ then } 0 \text{ else } A) \bigr) \)

“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?
\( \bigl( D \mid \text{do}(A \rightarrow \text{if } \sim \text{bern}(r) \text{ then } 0 \text{ else } B) \bigr) \)

### 4. 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 indistinguishable 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 \text{do} operator as a syntactic sugar (i.e., macro) in the original \( \lambda C \) language.

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

**Theorem 1.** The \text{do} operator is not macro-expressible in \( \lambda C \) without \text{do}.

**Proof.** According to the proof technique of Felleisen (10),

to show \text{do} is not macro-expressible in \( \lambda C \) without \text{do}, it suffices to find two expressions \( P \) and \( P' \) such that, for any evaluation context \( C \) in \( \lambda C \) without \text{do}, \( C[P] = C[P'] \), but such that there is an evaluation context \( C \) in \( \lambda C \) with \text{do} such that \( C[P] \neq C[P'] \).
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.

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

Note that all constructs of $\lambda C$ 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, $C[P]$ respects beta equivalence. Hence, since $P \equiv_{\beta} P'$, for any context $C$ which does not contain do, $C[P] = C[P']$.

Now pick:

$$C[e] = ((\lambda g.g 0 | do(p \rightarrow 1))(e(\lambda x.p)) | do(p \rightarrow 0)$$

Then $C[P] \Downarrow 1$, but $C[P'] \Downarrow 0$, where $\Downarrow$ is the reduction relation between terms.

5. Experiments

Here we demonstrate counterfactual reasoning in OMEGAC through three case studies. All experiments were performed using predicate exchange (31).

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 $r : S \rightarrow I$ maps a scene to an image. The driver acts according to a driver model – a function mapping $s \in S$ to a target acceleration:

$$\text{let}
\begin{align*}
\text{drivermodel} &= \lambda \text{car, ped, obs}.
\text{if cansee(car, ped, obs)} -- \text{if ped is visible}
\text{then -9} -- \text{decelerate}
\text{else 0,} -- \text{else maintain}
\end{align*}
$$

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.

$\text{CarV} = \sim \text{normal}(12, 4),$
$\text{CarP} = \sim \text{normal}(30, 5),$
$\text{PedV} = \sim \text{normal}(3, 1),$
$\text{PedP} = \sim \text{normal}(1, 2),$
$InitScene = (\text{CarV}, \text{CarP}, \text{PedV}, \text{PedP}, \text{obs}),$
$\text{Traj} = \text{simulate(InitScene, drivermodel)},$
$\text{Images} = \text{map(render, Traj)},$

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

$$\text{E} = (\text{Images[t]} == \text{data}) \text{ and } \text{crashed(Traj)} \text{ in (Traj | do(CarV \rightarrow 14)) | E}$$

We can also ask: would the crash have occurred had the obstacle been displaced?

$$\text{in (Traj | do(\text{obs} \rightarrow \text{obs} - 3)) | E}$$

Figures 5 and 6 visualize the posterior distributions over $d(\text{pred, car})$, the (smallest) distance between the car and the pedestrian.

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+2\Delta t}, u_{t+4\Delta t}, \ldots, u_{t_{\text{max}}}).

\[
\text{let } t0 = \theta, \Delta t = 0.1, t_{\text{max}} = 1, \\
\tau = \lambda u, t . u, -- \text{to intervene } u \\
euler = \lambda f', u, t . \\
\text{let } u = \tau(u, t), t_{\text{next}} = t + \Delta t \text{ in} \\
\text{if } t < t_{\text{max}} \\
\text{then let } u_{\text{next}} = u + f'(t_{\text{next}}, u) + \Delta t \\
in \text{cons}(u, euler(f', u_{\text{next}}, t_{\text{next}})) \\
\text{else } u,
\]

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.

\[
\text{Series} = \text{euler}(F', u, t0),
\]

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

\[
\text{tint} = \lambda u, t . \text{if } t == 0.5 \\
\text{then } [u[1], u[2], \text{inc}(u[3])] \text{ else } u, \\
\text{Series} = \text{Series} \mid \text{do}(\tau \rightarrow \text{tint}),
\]

As a more exotic example, 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 to find the posterior over times.

\[
\text{CGM} = \text{first}(-\text{Series}), \\
\text{-- Hypoglycemia occurs if } x \text{ is low at any time} \\
\text{Hypo} = \text{any}(\text{map}(\lambda x . x < \text{thresh}, \text{CGM})) \\
\text{-- Prior over time of intervention} \\
\text{T} = \sim \text{unif}(0, 1) \\
\text{-- intervention increases food at time } T \\
\text{tint2} = \lambda \omega . \lambda u, t . \text{if } t == T(\omega) \\
\text{then } [u[1], u[2], \text{inc}(u[3])] \text{ else } u \\
\text{-- Condition on hypoglycemia not occurring} \\
\text{-- in intervened world} \\
\text{Hypoint} = \lambda \omega . \\
(\text{Hypo} \mid \text{do}(\tau \rightarrow \text{tint2}(\omega)))(\omega) \\
in \text{CGM} \mid \neg \text{Hypoint}
\]

As shown in Figure 7(c), it is more plausible that the intervention occurred early in the day.

**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 to a Markov Decision Process (25) (MDP) model. Each grid cell is a state in a state space \( S = \{(i,j) | i = 1 \ldots 7, j = 1 \ldots 6\} \). The action space moves an agent a single cell: \( A = \{\text{up, down, left, right}\} \). Each agent acts according to a reward function that is a function of the state they are in only \( R : S \rightarrow \mathbb{R} \). 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” causation (13) 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 \( \text{OMEGA}_C \), 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 \( C_1, \ldots, C_n \) be a set of random variables and \( c_1, \ldots, c_n \) a set of values. With respect to a world \( \omega \), the conjunction \( C_1 = c_1 \land \cdots \land C_n = c_n \) is the but-for cause of a predicate \( E : \Omega \rightarrow \text{Bool} \) if (i) it is true wrt \( \omega \) and (ii) there exist \( \tilde{c}_1, \ldots, \tilde{c}_n \) such that:

\[
(E \mid \text{do}(C_1 \rightarrow \tilde{c}_1, \ldots, C_n \rightarrow \tilde{c}_n))(\omega) = \text{False} \tag{2}
\]

This reward is a precondition, the effect must actually have occurred for but-for to be defined.

But-for is defined existentially. To solve it, \( \text{OMEGA}_C \) relies on predicate relaxation (31), which underlies inference in \( \text{OMEGA}_C \). 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 intersections. Predicate relaxation transforms \( E \) into soft predicate \( \hat{E} \) 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 $(E \mid \text{do}(C_1 \rightarrow \hat{c}_1, \ldots, C_n \rightarrow \hat{c}_n))(\omega)$. 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 $\text{soft}(E(\hat{c}_i)) = 0$ 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.

6. Related Work and Discussion

Related work. Operators resembling $\text{do}$ appear in existing PPLs. Venture (16) has a force expression $\text{FORCE}<\text{expr}> <\text{value}>$ which modifies the current trace object (a mapping from random primitives to values) so that the simulation of $<\text{expr}>$ takes on the value $<\text{value}>$. It is intended as a tool for initialization and debugging. Pyro (5) and Anglican (34) have similar mechanisms. This can and has (18; 23) 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 OMEGAC, 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 OMEGAC tend to be significantly more concise, and require none of the manual hacks. More fundamentally, OMEGAC 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 (23), 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 (27) 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
We provide primitives to construct and compute counterfactuals. While we have presented a minimal language here, 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, `OmegaC` is also implemented in Julia. Since Julia is not lazy, it is less flexible than `OmegaC`, 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 `OmegaC` 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 (30) 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, `OmegaC` 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 `OmegaC` 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**


