A Language for Counterfactual Generative Models

Abstract

Generative models are a powerful formalism to represent mechanisms subject to uncertainty. Generative models have causal structure, and hence support counterfactuals, but counterfactual reasoning in complex generative models remains limited, and largely manual. We bridge this gap. We extend structural counterfactual reasoning to generative models by formulating a counterfactual as a dynamic transformation to a probabilistic program. We define the syntax and semantics of a causal probabilistic language. Through examples in population dynamics, inverse planning and causation, we demonstrate the pay-off of this generalization.

1 Introduction

Generative models encode the step-by-step causal mechanisms of a domain directly. As a result, generative models support counterfactuals; they allow us to consider how the domain would have been, or could be, under different conditions. Counterfactuals permeate everyday life. They allow us to plan, by considering what would happen if we were to take a particular action; to retrospect, by considering how the world would have been if we had acted differently in the past; and even to judge whether a caused b by imagining if a were not the case. The potential for counterfactual inference in generative models is then hard to overstate, but remains largely untapped.

Pearl formalized causal relationships as directed graphs with edges from cause to effect [1], and counterfactuals as manipulations to their structure. However, as a formalism for generative models, causal graphs are limited. Causal graphs are incapable of expressing domains that are unbounded. Even domains that are technically bounded, such as chess, often support an astronomical number of counterfactuals (e.g., "If I had made move X at some time Y, I might have won"), and hence are inexpressible as causal graphs in practice. Many counterfactuals depend on values of the past which are uncertain. For instance, we can consider how an intervention to financial markets before a critical point could have prevented instability, without knowing precisely when. The root of these limitations becomes apparent if we linearize a causal graph into a sequence of structural equations. To illustrate, consider a causal model between smoke s and an alarm a:

$$s := f_S(u_1)$$
$$a := f_A(s, u_2)$$

Each structural equation is interpreted as applying a function (e.g. f_S) to value (u_1) and assigning the result to a variable (s). In other words, structural models are programs, but in a language which lacks recursion, and where the intervenable variables are bounded in number and static, i.e., independent of the values the program takes, and where the functional form (e.g. f_A) are defined externally to the system.

Probabilistic programming languages [2, 3, 4] superseded similar limitations in Bayesian Networks. Universal probabilistic languages extend Turing-complete deterministic languages, and use recursion, control flow, objects, types and so on to specify generative models. Probabilistic programs have causal structures and hence are counterfactual supporting, but lack generic mechanisms to exploit this for counterfactual inference.

In this paper, we extend counterfactual inference to

generative models in a universal probabilistic programming language. We define the syntax and semantics of the OMEGA language [5], extended with a version of Pearl's do operator:

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

In OMEGA, do performs a dynamic program transformation. That is, Expression 1 evaluates to a value that X would have taken had Y been bound to Z at its point of definition. In effect, this allows us change the internal structure of previously defined random variables without apriori having to consider what interventions we might like to make.

There are several challenges we address to realize generic counterfactual reasoning in a universal language. A counterfactual effectively has to create a copy of original world, maintain the original world and share information between the two. Programs have complex control flow, and interventions can dramatically change the dynamics of execution. Preserving correspondence between values in the counterfactual world and the original world then presents several challenges.

In summary, we:

- Define the syntax and semantics of the OMEGA probabilistic programming language extended with the do operator (Section 2).
- Demonstrate counterfactuals through examples in competitive population models, inverse planning and but-for causation (Section 5).

2 A Calculus for Counterfactuals

OMEGA is a probabilistic language [5] which augments a functional core with probabilistic constructs. We extend OMEGA in two ways: (1) the syntax is augmented with a do operator for expressing interventions, and (2) the language evaluation is changed from eager to lazy, which is the key to the mechanism of handling interventions.

In the rest of the section, we formalize our extension using a core calculus called λ_C, in which we omit irrelevant language features.

Fig. 1 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. $\perp$ represents the undefined value. Finally, there is a sample space Ω, which is left unspecified, save

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

Figure 1: Abstract Syntax for λ_C

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.

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

Figure 2: Runtime environments of λ_C

Important concepts Before we give the semantics of λ_C, here are some key ideas in the design of its semantics.

- **Random variables**: We define random variables as in measure theory: as a function $f : \Omega \to \tau$ from the sample space to some type τ.
- **Lazy evaluation**: When a λ_C program intervenes on a variable x, all variables defined in terms of x will take different values. To make this possible, the system must track the provenance of each intervenable variable. Our solution is to use *lazy evaluation*. When a variable z is defined via a **let**, the defining expression is not computed immediately, but instead stored in a closure (Fig. 2), and only evaluated when z is referenced. When intervening on a variable x, bindings to x will be updated in all closures, effectively making all old variable definitions depend on the new definition.
- **Pure vs. intervenable variables**: Only variables defined with **let** may be intervened on, and so only these variables are computed lazily and stored as closures. Function arguments are evaluated immediately; variables defined as function parameters act as variables in any other pure functional language. In particular, they are referen-

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

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

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

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

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

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

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 3: Operational semantics for λ_C

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

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

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

Figure 4: The RetroUpd procedure

tially transparent: $(\lambda x : \mathrm{Int}\,.x + x)(a + b)$ is equivalent to $(a + b) + (a + b)$.

Semantics Fig. 3 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. 2 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 $\mathrm{clo}(\Gamma', e)$ 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 $t_1 \mid \mathbf{do}(x \to t_2)$ 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. 4).

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

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

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

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

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

As before, the LET rule causes the inner **rand** expression to evaluate in the context $\Gamma_1 = \{c \mapsto \text{clo}(\emptyset, 1), x \mapsto \text{clo}(c \mapsto \ldots, \lambda\omega.\mathbf{if}\ \ldots)\}$. The COND rule will essentially replace the argument to **rand** with $\lambda\omega'.\mathbf{if}\ x(\omega') == -1\ \mathbf{then}\ (x \mid \mathbf{do}(c \to 4))(\omega')\ \mathbf{else}\ \perp$. 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 $\text{clo}(\{c \mapsto \text{clo}(\Gamma_1, 4)\}, \lambda\omega.\mathbf{if}\ \ldots)$. This is then applied to ω', and the overall computation hence evaluates to 1 with probability $\frac{3}{4}$ and -1 with probability $\frac{1}{4}$.

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. Other work addresses how to check if a probabilistic program meets certain correctness specifications [6]. We can extend any such correctness condition to define whether an intervention is valid.

A λ_C program with interventions may always be transformed to a vanilla probabilistic program without **do**. Specifically, for $t_1 \mid \mathbf{do}(x \to t_2)$, 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.

Definition 1. Given a specification S and its checker C, a intervened program P and its equivalent program P' without **do**, we say P is valid iff $C(S, P') = \textit{correct}$.

3 The Omega Language

We have extended our implementation of the OMEGA language to support features based on λ_C. Other papers [5, 7] describe OMEGA in more detail, such as its algorithms for efficient inference. This section explains its features built atop λ_C that enable one to write real models like those in Section 5.

Sample Spaces While in the sample space Ω is left unspecified in λ_C, in OMEGA, it is a hypercube, with one dimension for each independent sample.

Programs can then access each dimension using a binary operator for projection, with the ith dimension of $\omega \in [0,1)^n$ given by $\omega[i]$. One can then define other distributions as transformations from these samples. Section 4 describes how we separate these definitions from the source of randomness.

Omega Syntax OMEGA has several additional convenience features over λ_C, which we use in the remainder of this paper:

- **Lifted operators**: If x and y are random variables (type $\Omega \to \tau$), then a lifted $+$ operator can be defined: $x + y \doteq \lambda\omega : \Omega.x(\omega) + y(\omega)$. Other operators are lifted similarly.
- **Continuous Let**: **let** $x = a, y = b$ **in** c is sugar for **let** $x = a$ **in let** $y = b$ **in** c.
- **Recursive Let**: OMEGA supports recursive bindings, as in **let** $f = \lambda x.1 + f(x)$ **in** $\ldots$. It is well known that recursion can be desugared into a λ-calculus using a fixed-point combinator.
- **Multi-arguments**: $\lambda xy.e$ and $f(a,b)$ are sugar for $\lambda x.\lambda y.e$ and $f(a)(b)$, respectively.

4 Exogenous Variable Identity Management

In OMEGA, an exogenous variable is a dimension of the sample space and can be accessed by indexing a sample (i.e., $\omega[i]$). However, using a fixed index i prevents code reuse, while using a dynamically-computed index e can cause completely unrelated values to be correlated during counterfactual execution. In these section, we address both issues.

Reusing Distribution Definitions A recurring pattern in probabilistic programming is that one defines a distribution and keeps creating variables that are i.i.d. with this distribution. However, this can be cumbersome in λ_C. For example, function $\lambda ab\omega.(a-b)*\omega[1]+b$ defines a uniform distribution in $[a,b]$. To create another random variable that is i.i.d., one needs to copy the definition and change $\omega[1]$ to $\omega[2]$. To address this issue and separate distribution definitions from the source of randomness, OMEGA [5] provides a operator ciid. ciid(t) creates a random variable with distribution defined by t. Moreover, this random variable is independent from any other variable created from ciid(t).

We first explain a simplified version of ciid called ciid$'$, which solves this problem. The second half of this section will explain the real ciid, which also addresses the other issue. ciid$'$ performs a two-step process: (1) it assigns a unique id to each created random variable, and (2) each $\omega[i]$ is translated into $\omega[h(id,i)]$, where h returns a unique index for every pair. Hence, each random variable accesses disjoint dimensions of the sample space. Without worrying about the implementation details, one can think of the above process as a syntactic transformation. Consider the following program

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

It is equivalent to

```
let uniform = λa.λb.λi.λω. (a-b)*ω[h(i,1)]+b in
uniform(1,2,newid())
```

Function `newid()` always returns a fresh id by maintaining a global counter.

Resolving Program Point Misalignment In an expression $\omega[e]$ which accesses an exogeneous variable, e is an ordinary expression, and interventions may change it in unexpected ways. This can lead to undesirable results for counterfactual queries. For example, take the following program, which centers on a function `rand10` which computes a random n-digit base-10 number:

```
1 let rand10 = λω.λd.if d == 0 then 0
2      else floor(10*ω[d]) + 10*rand10(ω, d-1) in
3 let n = 5 in
4 let f = λ ω.rand10(ω, n) * ω[n+1] in
5 rand((f | do(n → 4)) |λ ω.f1(ω) < 10)
```

Function `f` computes a 5-digit number 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 became very small after scaling. The programmer likely desired that this counterfactual will compute a 4-digit number whose digits are based on, $\omega[1],\ldots,\omega[4]$, and then scale it by the same factor, $\omega[6]$, meaning the final answer must be < 1. In fact, the counterfactual execution will scale by $\omega[5]$, and can hence return unexpectedly large results. The factual and counterfactual executions both used the same exogeneous value $\omega[5]$, but at completely different points in the program!

Following this intuition, OMEGA 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
ciid(uniform(1,2))
```

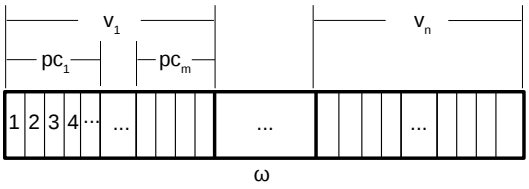

Figure 5: The sample space is partitioned into subspaces for different random variables, which are further partitioned for different program points.

Conceptually, it is translated into

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

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. Note now, a exogenous variable is identified by the random variable that it is used in, the program point where it is accessed, and the counter corresponds to this program point. This effectively leads to the partitioning of the sample space in Figure 5.

5 Experiments

Here we demonstrate counterfactual reasoning in Omega through three case studies.

5.1 Predator-Prey Population Dynamics

In this experiment we use the Lotka-Volterra [8] model for counterfactual reasoning about 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):

$$\frac{dx}{dt} = \alpha x - \beta xy \qquad \frac{dy}{dt} = \delta xy - \gamma y \tag{2}$$

where $x(t)$ and $y(t)$ represents the prey and predator respectively. Parameters α, β, δ and γ are positive

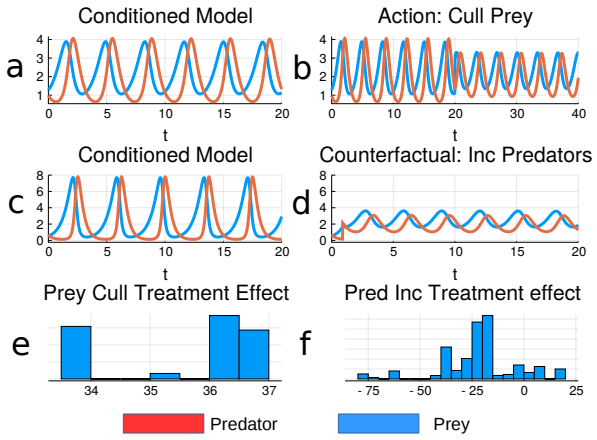

Figure 6: Lotka-Volterra Predator-Prey differential equations. (a, c) Samples from timeseries conditioned on predicate that mean rabbit population over last 10 days is 5, (b) Predicted effect of action: increasing number of prey at t_{now}, (d) Sample from counterfactual model: conditioned model with intervention in past (e) Treatment effect of culling prey (f) Treatment effect of increasing predators

constants which represent growth rates.

Suppose that 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?

Model To solve the differential equations we define the function `euler` which implements Euler's method [9]: a numerical procedure to solve first order differential equations. `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 $t_{\min}, t_{\min} + \Delta t, t_{\min} + 2\Delta t, ..., t_{\max}$

```
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)),
8 α = normal(0, 1), β = normal(0, 1),
9 γ = normal(0, 1), δ = normal(0, 1),
```

Next, we construct `lk'`: a random variable over derivative functions following Equation 2, where α, β, δ and γ are the previously defined random vari-

ables. 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.

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

Next, we complete the unconditional generative model. Note that `euler` is automatically converted into its lifted version (see Section 3), and `series` is a random variable.

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

Conditioning 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 6 (a) shows a conditional sample.

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

Action Next, we examine the effect of action[1]. In particular, if we were to increase the prey population by 5 at t_{now}, would the rabbit population be reduced (Figure 6 (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`:

```
eulergen = λt_int, u_int
 λf', u, t, tmax, Δt.
  let u = if t == t_int then u_int(u) else u in
   if t < tmax
    let unext = u + f'(t + Δt, u) * Δt,
          eul = eulergen(t_int, u_int) in
     cons(u, eul(f', unext, t + Δt, tmax, Δt))
   else
    emptylist,
```

The next snippet intervenes on `series` using **do** to replace `euler` with an alternative version `eulerint` which increases the number of prediators. Figure 6 (b) shows a sample.

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

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

Counterfactual 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 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.

```
tmostwolves = maxindex(series),
inc_wolves = λu.(getx(u), gety(u)+2),
inc_euler = eulergen(t_mostwolves, inc_wolves),
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 6 (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)`.

5.2 Counterfactual Planning

Consider a migration dispute between three hypothetical island nations (Figure 7 Left): S to the South, E to the East and N to the North. The government of S aims to reduce emigration of its population to the N, and considers constructing a barrier between S and N (Figure 7 right).

We model this problem as counterfactional inference in a Markov Decision Process [10] (MDP) model. To determine whether the border can be effective:

- We assume members of the population are rational: that they migrate according to their beliefs about the world and their objectives.
- We condition on observed migration patterns to infer a posterior belief over the population objectives.
- In this conditional model, we consider the intervention (adding the barrier) and predict the resulting migration.

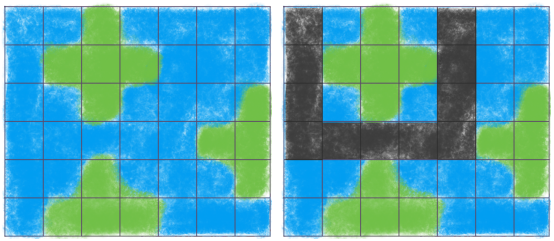

Figure 7: Three islands S, N, E without (left) and with (right) border under consideration

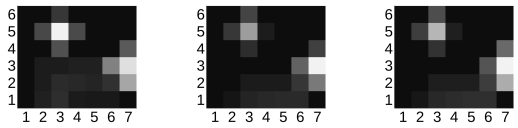

Figure 8: Sample from population counts after n timesteps of MDP based migration. (Left) Unconditional sample, (b) Conditional sample (c) Counterfactual

Concretely, we assume the world is a 7 by 6 grid and a world population of 1000 residents. An individual's birth position is uniformly distributed over land cells. The migration dynamics are constructed as follows. Each member of the world has a belief about the quality of life in each country. This is specified as a reward for being in a cell of that country, at every time step. These rewards are normally distributed with a mean that is specific to country of birth. For example, people born in S typically have high opinions of N and lower opinions of their home. For t timesteps we simulate the migration behavior of each individual, acting according to their reward function, using value iteration: a method to solve MDPs. We count the amount of time spent in each country over the time period. Figure 8 shows population counts under different conditions. Figure 9 demonstrates migration patterns in the prior, conditional and counterfactual cases.

5.3 But-For Causality in Occlusion

Humans actively manipulate their environment to obtain more accurate information about the world [11]. This requires the ability to determine what is preventing access to better information. In this experiment, we model this through simulatenous Bayesian inverse graphics and counterfactual based "But-For" causality. We infer three dimensional objects having observed a two dimensional image, and for objects in inferred scene, use But-For to determine if they are occluding another object of interest.

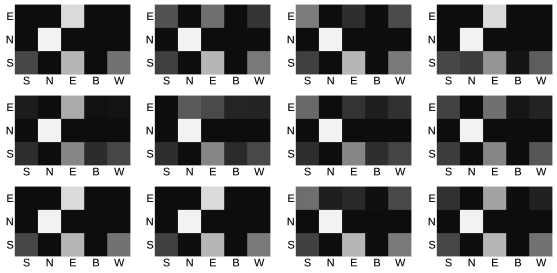

Figure 9: Four samples of migration patterns under different conditions. Each figure shows the migration from islanders born in S, N, or E (y-axis) across the globe to S, N, E, W (water) or B (barrier) on the x-axis. We accumulate all states in each persons trajectory, not only the final state. (Top) Prior samples without conditions or interventions. (Middle) Conditioned on observations (Bottom). Counterfactual: Conditioned on Observations with intervention (border)

An event C is the But-For cause of an event E if had C not occurred, neither would have E. Had C not occurred does not mean to condition on the negation of C (since this fails to differentiate cause from effect). Instead, we must find a counterfactual world where C does not hold.

Definition 2. Let $C : \Omega \to \tau$ and $E : \Omega \to \text{Bool}$ be random variables. In a world ω, $C = C(\omega)$ is the but-for cause of E if there exists an $t : \tau$ and $t \neq C(\omega)$ such that:

$$(E \mid \mathbf{do}(C \to \lambda\omega'.t))(\omega) = False$$

where $E(\omega) = \text{True}$ is the precondition, i.e.: but-for is not defined is these do not hold.

The following program snippet shows how we can simultaneously perform inverse graphics and but-for causality. In this example we sample from a posterior distribution over scenes conditioned an observed rendering. A scene is a set of $n \sim \text{poisson}(\lambda = 3)$ spheres. A sphere is parameterized by color, reflectance, emission color, transparency, radius and position, all with a uniform prior. If r is a render function that maps scenes to image, i_{obs} is an observed image, the prior is conditioned on the constraint: $r(scene) = i_{obs}$.

In the conditional scene we use But-For to infer occlusion. That is, we assume a function `occluded(scene, obj)` which determines if a `obj` is occluded in `scene`. We artificially inject a yellow target object `objb` into the scene. We use a function `butfor(ω, c, e)` which determines if `c` is the

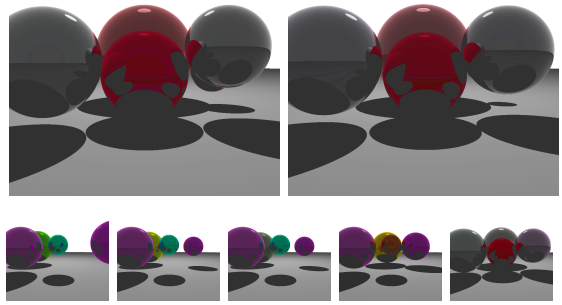

Figure 10: Inverse Graphics: inferring a scene from an image. (Top-right) observed image, (Top-Left) rendering of posterior sample of scene. Bottom row, left to right, renderings of scene samples from initialation through to Markov chain convergence.

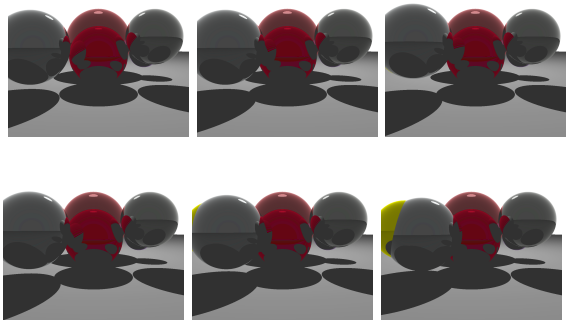

Figure 11: But-for causality for occlusion. (Top-left to Bottom-Right) states of optimization algorithm. At image 5, we determine that the left-most grey sphere is the but-for cause of inability to see yellow sphere.

But-For cause of `e` with respect to `i` in world ω. In particular, we determine if an `objа` uniformly drawn from objects in the inferred scene is occluding `b`.

```
let scene_cond = scene | render(scene) == iobs,
    objа = uniform(spheres(scene_cond)),
    cantsee = occluded(scene_cond, objb) in
    λω. butfor(ω, center(objа), cantsee)
```

6 Related Work

Operators resembling **do** appear in existing probabilistic programming languages. Venture [3] 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 [12] 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, which they call "intervention". There are also several libraries for doing causal inference on traditional causal graphs [13, 14, 15, 16, 17].

There has also been work on adding causal or counterfactual operators to deterministic programming paradigms. Halpern and Moses [18] investigated counterfactuals in the context of knowledge-based programming, a specification technique for distributed systems. 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 [19] 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.

7 Discussion

This contribution extends structural counterfactual inference to the domain of Turing-complete probabilistic languages. We consider it as the natural step along the axis from Wright's path diagrams [20] to Pearl's causal graphs.

In one particular respect, Omega programs are less expressive than causal graphs. Causal graphs can be *non-parametric*. This means that the functional forms are unspecified and only the existance of causal edges is known. This level of abstraction is not possible in an Omega program; we cannot define a function without defining the expression that defines it.

Our version of the do operator allows for changing arbitrary values within a program. This raises the question of whether it is *too* flexible. Pearl emphasizes the distinction between laws and facts [1], and suggests causal inference relies on distinguishing the two. He uses the example of dominos to illustrate: the laws determine the physics, while intervenable variables are properties like the dominoes positions. In Omega, one can represent a program that simulates domino physics, and consider counterfactuals which break the laws such as "What if gravity were twice as strong?". It remains open to see if this flexibility can lead to undesirable effects.

References

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

[2] Noah D Goodman, Vikash K Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 220–229. AUAI Press, 2008.

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

[4] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

[5] 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.

[6] 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.

[7] Zenna Tavares, Javier Burroni, Edgar Minaysan, Armando Solar Lezama, and Rajesh Ranganath. Soft constraints for inference with declarative knowledge. *CoRR*, abs/1901.05437, 2019.

[8] Darren J Wilkinson. *Stochastic modelling for systems biology*. Chapman and Hall/CRC, 2006.

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

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

[11] Jacqueline Gottlieb, Pierre-Yves Oudeyer, Manuel Lopes, and Adrien Baranes. Information-seeking, curiosity, and attention: computational and neural mechanisms. *Trends in cognitive sciences*, 17(11):585–593, 2013.

[12] 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.

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

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

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

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

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

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

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

[20] Sewall Wright. Correlation and causation. *Journal of agricultural research*, 20(7):557–585, 1921.