From Explanation to Design: Principled Generators in Program Trace Optimization

Alberto Moraglio[0000−0003−4782−6590]

Department of Computer Science, University of Exeter, UK
a.moraglio@exeter.ac.uk

Abstract. Program Trace Optimization (PTO) is a general framework in which users write probabilistic generators in a problem's natural representation, and generic solvers operate on the generators' execution traces as a universal genotype. PTO has long claimed, informally, that writing generators in a "well-structured" way induces meaningful phenotypic evolutionary operators, but the claim has lacked theoretical grounding. A recent analytical theory has provided a causal account of PTO's genotype-phenotype map that, given a generator, *predicts* the quality of the induced operators. This paper uses the same theory *generatively*. Starting from the desired operator properties of locality and modularity, we derive concrete code-level conditions on the generator that induce them. These conditions align with principles long established in pure functional programming, structured programming, and algebraic data types. The broader point is methodological: a general explanatory theory can be used predictively or generatively, and the generative mode yields principled design guidance.

Keywords: Program Trace Optimization · Genotype-Phenotype Map · Locality and Modularity · Principled Design · Explainable Evolutionary Computation · Causal Analysis

1 Introduction

Program Trace Optimization (PTO)[1] is a general optimization framework that decouples problem representation from search algorithm. The user writes a *generator* — a probabilistic program that constructs candidate solutions in the problem's natural representation — and generic solvers operate on execution traces of the generator as a universal linear genotype. Mutation and crossover apply unchanged across arbitrary problem domains. PTO has been applied successfully across vectors, permutations, trees, and neural networks [6, 8], with generators ranging from heuristic construction procedures for combinatorial problems [3] to language models [10, 9] and non-context-free grammars [7]. PTO has claimed, informally since its introduction [6], that writing generators in a *natural, well-structured way* induces phenotypically meaningful operators without explicit operator design. The claim has worked in practice but lacked theoretical

[1] https://github.com/Program-Trace-Optimisation/PTO

grounding: what does "well-structured" mean precisely, and why does it work? A recent causal framework [5] provided a structural answer. It characterizes locality and modularity of PTO's genotype-phenotype map (G-P map) as properties of a gene-trait dependency graph derived by static analysis of the generator, and identifies when the induced operators are well-behaved, producing localized phenotypic changes and clean exchange of phenotypic modules. Operator behavior thus becomes predictable before any run. What remains open is how the abstract structural properties the framework identifies translate into concrete generator code.

Research question. What structural properties of a PTO generator's code guarantee locality and modularity of its G-P map, and how do these properties align with established programming practices?

Methodology. The causal framework derives the G-P map of a generator, and from it the locality and modularity of the induced phenotypic operators, through a chain of successive layers: from code to gene dependency graph, to gene-trait graph (G-P map) to measure, to operator behavior. This paper reverse-engineers the chain. Starting from the desired properties of locality and modularity, we work back through the theory's successive layers to derive sufficient conditions on the generator code. Four conditions emerge, and each corresponds to a principle promoted, on independent grounds, by pure functional programming, structured programming, or algebraic data types. The traditions these principles come from are all concerned with minimizing unneeded dependencies between parts of a program, which turns out to be exactly what makes the G-P map local and modular, and the induced evolutionary operators phenotypically meaningful. The methodological point is more general than PTO. Most XAI produces specific empirical explanations [1, 2], i.e., why a particular model produced a particular output. Existing work at the intersection of evolutionary computation and XAI [12] has likewise focused on empirical post-hoc analysis. Our approach is, by contrast, ex ante: a *general* theory *is* a general explanation, which can be thought of as a declarative set of logical relationships that can be inverted, from a predictive use linking inputs to outputs, to a generative use linking desired outputs back to the inputs that produce them. Design guidance comes from the generative use, which this paper exemplifies by inverting PTO's theory of the G-P map.

2 Program Trace Optimization

PTO separates problem specification from search algorithm application. The user provides a *generator* — a probabilistic program constructing candidate solutions by making random decisions — and a *fitness function*. During execution, the generator's random decisions are recorded as a *trace*: a sequence of labeled name-value pairs. Generic solvers (hill climber, genetic algorithm, PSO) operate exclusively on traces, seeing only names, values, and fitness, with no knowledge of the problem domain.

Naming schemes. Each trace entry is labeled by a name identifying its random decision. Under *linear naming*, decisions are numbered sequentially in execution order. Under *structural naming*, names reflect the decision's control flow context (call stack, loop indices, conditional branches) so decisions made at corresponding points across different executions receive matching names.

Induced phenotypic search operators. PTO's operators manipulate traces generically. *Mutation* resamples one or more trace entries. *Crossover* aligns parent traces by matching entry names, then recombines them. *Repair* re-executes the generator on the modified trace, resampling missing entries and discarding unreachable ones. The naming scheme determines whether these induce meaningful phenotypic operators. Under linear naming, misalignment forces extensive resampling and produces offspring phenotypically unrelated to parents. Under structural naming, mutations are phenotypically localized and crossover exchanges phenotypic features.

Illustrative Example. We illustrate PTO with a generator for Boolean expressions (Fig. 1, left). The trace for `((x1 and x2) or x3)` (Fig. 1, right) records each random decision under both naming schemes: sequential numbers (linear) and control flow paths (structural), both generated automatically.

```
def gen(): #root
match
  rnd.choice([var,uop,biop]): #type
  case 'var':
    return rnd.choice([x1,x2,x3])
  case 'uop':
    return [not, gen()]
  case 'biop':
    return [gen(),          #L
    rnd.choice([and,or]), #op
    gen()]                #R
```

Seq	Str Name	Call	Value
1	root/type	choice(var,uop,biop)	biop
2	root/L/type	choice(var,uop,biop)	biop
3	root/L/L/type	choice(var,uop,biop)	var
4	root/L/L/var	choice(x1,x2,x3)	x1
5	root/L/op	choice(and,or)	and
6	root/L/R/type	choice(var,uop,biop)	var
7	root/L/R/var	choice(x1,x2,x3)	x2
8	root/op	choice(and,or)	or
9	root/R/type	choice(var,uop,biop)	var
10	root/R/var	choice(x1,x2,x3)	x3

Fig. 1: Boolean expression generator (left) and trace for `((x1 and x2) or x3)` under both naming schemes (right); uop/biop denote unary/binary operators.

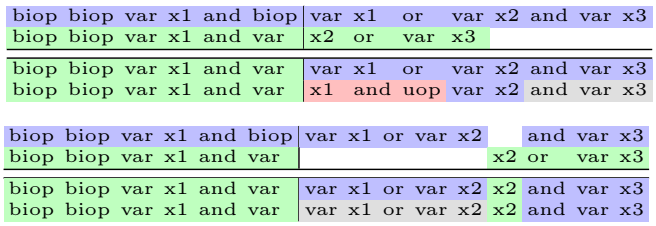


Fig. 2: One-point crossover under linear naming (top) and structural naming (bottom). Rows 1–2: parent traces P1 and P2. Row 3: offspring before repair. Row 4: offspring after repair, with resampled and discarded entries.

Crossover and repair. Fig. 2 shows one-point crossover between P1: `((x1 and (x1 or x2)) and x3)` and P2: `((x1 and x2) or x3)` under both naming schemes. Under linear naming, entry 7 is `var` (from P1) generated assuming `biop` not `var` in entry 6 (from P2); repair resamples entries 7–9, producing `((x1 and x1) and not(x2))`, phenotypically unrelated to either parent. Under structural naming, P1-only labels are unreachable and discarded with no resampling, producing the homologous tree recombination `((x1 and x2) and x3)`, a phenotypically meaningful combination of both parents.

3 Causal Theory of the PTO G-P Map

The theory formalizes PTO's claim that "natural" generators induce meaningful operators. It relies on two insights: (I) known generator mechanisms allow the G-P map to be derived analytically via static analysis rather than empirical estimation; and (II) causality [11] is the correct abstraction: trace operators act as interventions on gene-gene and gene-trait dependencies. Here, locality and modularity emerge as causal independence properties. We summarize the theory via a running example; see [5] for the formal derivation.

Modeling the generator. The Gaussian-samples generator (below) samples length $N \in \{1, 2, 3\}$, mean $M \in [0, 1]$, and N values $Y_i \sim \mathcal{N}(M, 1)$. Short executions produce traces (N, M, Y_1), while long ones produce (N, M, Y_1, Y_2, Y_3).

```
def gen():
    n = rnd.randint(1, 3)
    m = rnd.uniform(0, 1)
    return [rnd.gauss(m, 1) for _ in range(n)]
```

Random calls exhibit two causal dependencies:

- **Parametric ($X_i \to_p X_j$):** X_i's value shifts X_j's distribution (e.g., $M \to_p Y_i$ shifts the Gaussian mean).
- **Existential ($X_i \to_e X_j$):** X_i determines if X_j is reached via branches, loop bounds, or recursion guards (e.g., $N \to_e Y_{2,3}$).

Standard Bayesian networks cannot model variable execution sets simultaneously. We use a *Deterministically Gated Bayesian Network* (DGBN), where nodes have deterministic gates that activate or nullify them based on parent values. The DGBN (Fig. 3a) provides a static model of *all* possible traces, and is derivable via static analysis with data-flow for parametric arcs and control-flow for existential arcs.

Modeling the search operators. PTO operators act on flat traces. Since under structural naming each entry uniquely identifies a DGBN node, trace operations map to causal interventions on DGBN instances: *creation* samples nodes top-down; *mutation* intervenes on a node's value; *crossover* recombines instances node-wise; and *repair* propagates changes through the DGBN's gates and distributions. This correspondence breaks under linear naming because indices may

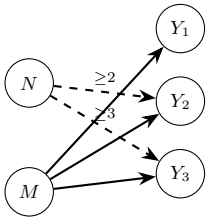
refer to different DGBN nodes across executions, causing misalignment. Structural naming is therefore a theoretical requirement for operators to function as well-defined causal interventions.

Modeling the G-P map. A generator's phenotype, whether a tree, network, or arbitrary structure, is abstracted into fitness-relevant, user-defined *traits* computable via static analysis. Trait selection is a primary design decision that determines the G-P map.

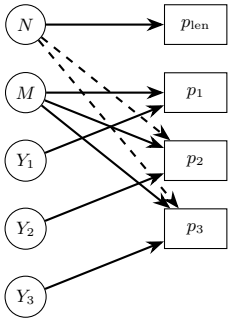
This map is represented by a bipartite graph $\mathcal{B}$, where an arc (g, t) exists if gene g causally affects trait t. Arcs are *parametric* if the entire causal pathway is parametric, and *existential* if any step is existential. Operationally, $\mathcal{B}$ is built by taking the ancestral closure through the DGBN from a trait's *direct support* (the genes used in its computation). For the running example (Fig. 3b):

$$\begin{aligned} \text{genes}(p_{\text{len}}) &= \{N_p\} & \text{genes}(p_2) &= \{N_e, M_p, Y_{2p}\} \\ \text{genes}(p_1) &= \{M_p, Y_{1p}\} & \text{genes}(p_3) &= \{N_e, M_p, Y_{3p}\} \end{aligned}$$

Note N gates p_2, p_3 existentially, while p_1 is always present and independent of N (subscript p/e: parametric/existential arc).



(a) DGBN: gene-gene dependencies



(b) Bipartite graph $\mathcal{B}$: gene-trait dependencies

Fig. 3: Causal models for the running example. (a) DGBN: nodes are random calls; solid arcs are parametric, dashed arcs are existential. (b) Bipartite graph $\mathcal{B}$: round nodes are genes, square nodes are traits; arcs are typed by the pathway that produces them.

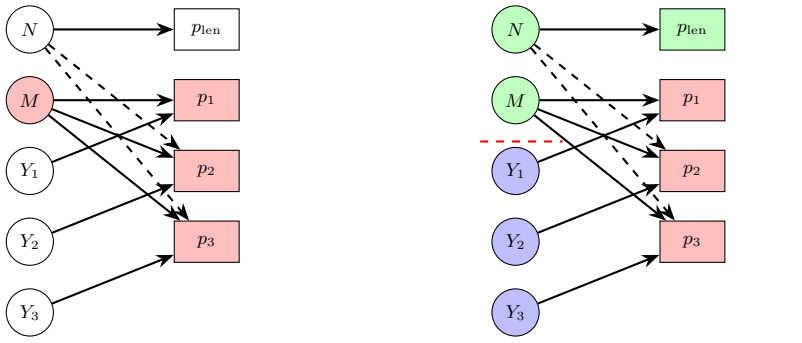
Projecting search operators through the G-P map. Gene-level operators project through $\mathcal{B}$ to phenotypic traits (Fig. 4). Mutation at g affects traits(g), transmitting value changes via parametric arcs and existence changes via existential arcs. One-point crossover partitions genes, resulting in three trait outcomes: *clean inheritance* (all supports on one side), *hybrid value* (separated parametric supports), or *flipped existence* (separated existential supports).

Operator disruption is controlled by two structural measures of the G-P map:

– **Pleiotropy ($k_{\max}$):** The maximum number of traits affected by any single gene mutation. This is a measure of the locality of mutation.

– **Modularity (k):** The number of connected components in the *parametric subgraph* $\mathcal{B}_p$. Since existential arcs do not transmit value information, they do not produce hybrid values. A crossover cut along a component boundary ensures clean inheritance.

In our example (Fig. 4), $k_{\max} = 3$ at gene N. $\mathcal{B}_p$ has $k = 2$ components: $\{N, p_{\text{len}}\}$ and the block $\{M, Y_i, p_i\}$. A cut straddling M and Y_1 produces hybrid values for $p_{1,2,3}$, while p_{len} is inherited cleanly. Locality (low pleiotropy) and modularity (high k) are the targets for the generator conditions derived in Section 4.



(a) Mutation at M (b) Crossover cut between M and Y_1

Fig. 4: Operators projected through $\mathcal{B}$ for the running example. **Left:** mutation at M (red) propagates parametrically to p_1, p_2, p_3; p_{len} is unaffected. **Right:** a crossover cut (dashed red) between M and Y_1 produces hybrid values at p_1, p_2, p_3 because their parametric supports straddle the cut; p_{len} depends only on N and is inherited cleanly. Green: from parent P_1. Blue: from parent P_2. Red: hybrid traits.

4 Structural Conditions and Sufficiency Theorem

4.1 Design Logic: Theory Inversion

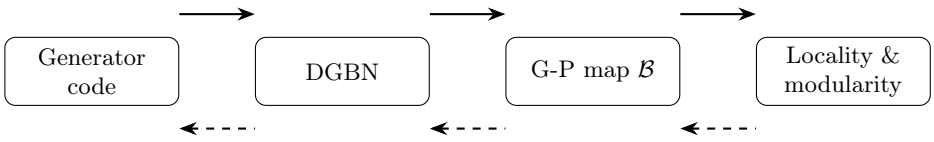


Fig. 5: The three-layer chain. Forward (solid, left to right): the original theory derives each layer from the previous, from generator code to disruption measures on the G-P map. Backward (dashed, right to left): this paper imposes conditions on each layer from the desired disruption measures back to the generator code.

Section 3 established a three-layer causal chain: generator code → DGBN → G-P map $\mathcal{B}$ → operator behavior (Fig. 5). Operator behavior is governed

by $\mathcal{B}$-level measures (pleiotropy and parametric matching), but practitioners work exclusively at the code level. Our task is to invert this chain: identifying structural conditions on the DGBN that are (1) sufficient to guarantee favorable $\mathcal{B}$-level measures and (2) verifiable by direct code inspection.

4.2 Phenotypic Targets

We work backward from two high-level phenotypic targets on $\mathcal{B}$:

Target M (Perfect inheritance at crossover). For any crossover point under structural naming and gap-fill, every trait t must satisfy:

- **(M1) No hybrid values:** t's value in the offspring matches t in either P_1 or P_2.
- **(M2) No spurious creation:** If t is absent in both parents, it remains absent in the offspring.
- **(M3) No spurious loss:** If t is present in both parents, it remains present in the offspring.

Target L (Logarithmic mean pleiotropy). Mean pleiotropy must be bounded: $\overline{k} = \frac{1}{n}\sum_g |\text{traits}(g)| = O(\log n)$ with $\text{traits}(g) = O(1)$. A condition on worst-case pleiotropy is too restrictive in practice; logarithmic mean pleiotropy across genes is more realistic as it captures the idea of typically small mutation.

4.3 Structural Conditions

Disruption on $\mathcal{B}$ stems from four structural features of the DGBN ($D = (V, E_p \cup E_e)$) and trait supports ($\text{supp}(t_j) \subseteq V$). We define four conditions sufficient to hit Targets M and L, each easily verifiable via inspection of the generator code:

S1 (Single-gene supports): $|\text{supp}(t_j)| \leq 1$ for every trait t_j. *Rationale:* Prevents value combinations (hybrids) at crossover points caused by multiple genes feeding one trait.

S2 (Existential-heavy): $E_p = \emptyset$; every DGBN arc is existential. *Rationale:* Restricts value transmission; existential arcs transmit only gating information, preventing parametric pathways from proliferating into $\mathcal{B}$.

S3 (Forest): Every node in D has at most one parent. *Rationale:* Ensures an existential forest structure. Multiple parents create "diamonds" where crossover produces spurious gating outcomes incompatible with either parent.

S4 (Balance): The DGBN has depth $O(\log n)$ with bounded branching. *Rationale:* Shallow structure keeps descendant subtrees small, bounding mean pleiotropy.

S1 is checkable on code from trait computations, S2 from call arguments, S3 from call arguments/control-flow, and S4 from call/recursion structure.

4.4 Sufficiency

Theorem 1 (Sufficiency). *If the DGBN satisfies S1–S4, then on $\mathcal{B}$:*

- ***Target M holds:*** *No hybrid values, no spurious creation, and no spurious loss at crossover.*
- ***Target L holds:*** *Mean pleiotropy is bounded at $\bar{k} = O(\log n)$.*

Proof (Proof Sketch). The proof rests on the structural transformation of the G-P map into an *existential forest*.

Target M (Crossover).

- **(M1) No hybrid values:** Recall from [5] that arcs in B are typed by the worst step in the pathway: an arc (g, t) is parametric iff every step from g to t is parametric, and existential otherwise. Under S2, all internal D-arcs are existential, so the only parametric arcs in B are the direct support arcs $g \to t$ with $g \in \text{supp}(t)$. Under S1, $|\text{supp}(t)| \leq 1$, so each trait receives at most one such arc; hence B_p is a matching. Modularity for value-inheritance is determined by B_p rather than B because existential pathways transmit only gating information, not value information [5].
- **(M3) No spurious loss:** By S1, trait t has a unique support gene g^*. By S3, the ancestors of g^* in D form a unique chain $C_t = (v_0, \ldots, v_k = g^*)$ from the root, with every arc existential by S2. If t is present in both parents, then in both P_1 and P_2 every node in C_t was reached and each gate v_i $(i < k)$ took a passing value; hence both traces contain entries at every name in C_t with passing values throughout. Under one-point crossover with structural naming, each entry in C_t is inherited from one parent or the other; either choice supplies a passing value. Every gate in C_t therefore passes in the offspring, g^* is reached, and t is present.
- **(M2) No spurious creation:** The ancestors of g^* in D form a unique chain (from M3). If t is absent in both parents, each parent's chain failed at some position before g^*; let $f_1 \leq f_2$ be the first failures in P_1 and P_2. Under crossover, the offspring's value at each chain position is inherited from whichever parent has an entry. The deepest position the offspring's chain can reach (best case) is therefore f_2, and at f_2 the only available value is P_2's failing one. The chain fails by f_2, so g^* is not reached and t is absent.

Target L (Logarithmic mean pleiotropy). By S3, D is a forest; each gene g has a well-defined descendant subtree subtree(g). By S1, each trait has at most one support gene, so pleiotropy at g is bounded by the number of descendants of g that serve as support genes — at most $|\text{subtree}(g)|$. Summing across all genes: $\sum_g \text{pleiotropy}(g) \leq \sum_g |\text{subtree}(g)| = \sum_v (\text{depth}(v) + 1)$ since each node v appears in the subtrees of its depth(v) + 1 ancestors (including itself). By S4, depth(v) $= O(\log n)$ for every v, so $\sum_v (\text{depth}(v) + 1) = O(n \log n)$, and $\bar{k} = \frac{1}{n} \sum_g \text{pleiotropy}(g) = O(\log n)$.

Remarks. The four conditions play complementary roles; none is redundant:

- **S1:** needed for (M1) (matching), (M2)/(M3) (single chain), (L) (subtree-pleiotropy bound).
- **S2:** needed for (M1) (no parametric ancestry), (M2)/(M3) (chain is purely existential).
- **S3:** needed for (M2)/(M3) (no diamonds, hence no synthesized gate outcomes), (L) (forest gives well-defined subtrees).
- **S4:** needed for (L) (bounded depth, hence bounded mean pleiotropy).

5 Code Patterns for the Structural Conditions

The structural conditions S1–S4 are properties of the DGBN and trait supports derived from the generator code. For a practitioner, the relevant question is which specific code patterns enforce these conditions. This section connects DGBN properties to concrete programming practices and characterizes the ideal class of PTO generators.

5.1 The Canonical Case: Balanced Tagged Recursion

A generator satisfying S1–S4 simultaneously possesses a characteristic shape, exemplified by the depth-bounded tree generator below:

```
def gen(d):
    if d == 0:
        return ('leaf', rnd.real())
    tag = rnd.choice(['leaf', 'node'])
    if tag == 'leaf':
        return ('leaf', rnd.real())
    return ('node', gen(d-1), gen(d-1))
```

Taking traits to be the tag at each internal position and the value at each leaf, we verify the four conditions:

- **S1 (Single-gene supports):** Each trait is computed by a single random call (the tag choice or the `rnd.real()` sample). Values are placed into the output structure via immutable tuples without arithmetic or functional combination.
- **S2 (Existential-heavy):** The arc from a tag choice to its recursive children is existential: the tag value determines which branch is executed, and thus whether children are reached. No call's distribution parameters depend on prior random values.
- **S3 (Forest):** Each call has exactly one existential parent: the immediately enclosing tag choice. Recursion threads a single parent-child dependency, preventing "diamonds" where multiple prior choices jointly gate a node.
- **S4 (Balance):** Branching into two sub-generators of equal potential depth ensures the DGBN depth is $O(d) = O(\log n)$ relative to the number of genes n.

By Theorem 1, this generator induces search operators with perfect inheritance at crossover and logarithmic mean pleiotropy.

5.2 Code Patterns that Violate the Conditions

Departures from the canonical inductive shape violate specific structural conditions, leading to predictable phenotypic disruptions. We illustrate each violation with a minimal code example.

Violating S1: Composition that Combines.

```
def gen():
    n  = rnd.randint(1, 3)
    vs = [rnd.real() for _ in range(n)]
    return [vs[i] + vs[(i+1) % n] for i in range(n)]
```

Context: Using multiple random values as operands in arithmetic, aggregation functions, or non-projection functions. **Insight:** Each output position is an arithmetic combination of two genes. This creates multiple parametric arcs into each trait, meaning $\mathcal{B}_p$ is no longer a matching. Consequently, crossover cuts separating these genes produce *hybrid values* absent from either parent.

Violating S2: Conditioning on a Prior Value.

```
def gen():
    n = rnd.randint(1, 3)
    m = rnd.uniform(0, 1)
    return [rnd.gauss(m, 1) for _ in range(n)]
```

Context: Passing a sampled value as a parameter to a subsequent distribution. **Insight:** This creates a parametric dependency ($M \rightarrow_p Y_i$). In the G-P map, M becomes a parametric ancestor of every output position, collapsing $\mathcal{B}_p$ into a single connected component. This eliminates modularity, causing any mutation at M or crossover cut to affect the entire phenotype.

Violating S3: Hidden or Joint Coupling.

```
def gen():
    n = rnd.randint(1, 3)
    k = rnd.randint(1, 3)
    return [rnd.real() for _ in range(min(n, k))]
```

Context: Using shared mutable state across calls or gating a call using multiple prior random values. **Insight:** This creates "diamonds" in the DGBN (e.g., Y_i has two existential parents, N and K). Crossover can recombine parents to produce gating outcomes (spurious creation or loss) present in neither parent, a pathology ruled out only when the DGBN is an existential forest.

Violating S4: Sequential Accumulation.

```
def gen(n):
    if n == 0: return []
    # Each choice gates the entire rest of the sequence
    if rnd.choice(['stop', 'cont']) == 'stop':
        return []
    return [rnd.real()] + gen(n-1)
```

Context: Iterative loops or left-associated recursion that threads state or existence sequentially (e.g., a linked list structure). **Insight:** The DGBN becomes a linear chain where the i-th gene existentially gates all subsequent genes. This

forces high-level genes to have descendant subtrees of size $O(n - i)$, inflating mean pleiotropy from $O(\log n)$ to $\Theta(n)$. Only divide-and-conquer structures (S4) ensure subtree sizes are geometrically distributed, keeping mutations localized.

5.3 Characterization of the Ideal Regime

The structural conditions S1–S4 define a class of generators implicitly: any generator whose induced DGBN and trait supports satisfy the conditions belongs to it. Within this class, a particularly recognizable sub-class is given by generators following the pattern of *anamorphic generation over an algebraic data type (ADT)* [4]. In functional programming, an anamorphism is a process that "unfolds" a complex structure from a seed value through a series of independent local decisions. In this regime, every random call serves a specific structural purpose:

- **Tag Choices (Sums):** These select which constructor to apply (e.g., `Leaf` vs. `Node`). This creates the *existential gating* that defines the DGBN's branching structure.
- **Recursive Unfolding (Products):** For constructors with multiple components, the generator recurses to build each part independently. This ensures the DGBN remains a *forest*.
- **Leaf Sampling:** Primitive values are sampled at the end of the unfolding. If these are placed directly into the output structure without arithmetic combination, they satisfy *single-gene support*.

The anamorphic ADT pattern is sufficient (when recursive structures are *balanced*) but not necessary. More generally, *any generator that builds output by making independent choices, where a decision in one branch cannot modify values in another, falls into the ideal regime*. Such generators include flat independent sampling like fixed-size vectors and matrices, variable-arity recursive structures like grammars, and hybrid patterns that combine these.

The class of generators satisfying S1–S4 is broad but not universal. Many practical problems involve global constraints (such as uniqueness in permutations, shared parameters in neural architectures, or grammar-level well-formedness conditions) that cannot be expressed through independent local decisions alone, and therefore require deviations from this shape. The framework treats such deviations as graded trade-offs rather than binary failures: each violated condition introduces a specific form of operator disruption (Section 5.2), and the magnitude of departure from S1–S4 corresponds directly to the magnitude of induced disruption. The ideal regime thus serves as a baseline against which real generators can be designed and analyzed, rather than as a hard requirement.

6 Connection with Programming Styles

The conditions S1–S4 are derived mathematically, but they align broadly with three established programming traditions: Algebraic Data Types (ADTs), Pure

Functional Programming, and Structured Programming. These traditions developed independently of evolutionary computation, for reasons of program comprehension and verification, yet each pushes generators toward the kind of structure the conditions require. All three traditions promote a common discipline — explicit, parsimonious dependencies between parts of a program — which turns out to be what S1–S4 collectively demand.

6.1 Explicit Dependencies as a Precondition

Before S1–S4 can even be evaluated, the dependencies between random calls must be visible from the source. Pure functional programming makes this possible: with no shared mutable state and no hidden side channels, every dependency between two calls is either a data flow through arguments or a control flow through case analysis, both readable directly from the code. The DGBN derivation in Section 3 relies on exactly this property; generators written in an imperative style with shared state defeat the static analysis that produces the dependency graph in the first place. Purity is therefore a precondition for the framework, not a condition within it. ADTs reinforce this by making the shape of data explicit at construction sites: a `Node(left, right)` constructor exposes its two components syntactically, so the analysis can see which subsequent calls depend on which.

6.2 Parsimonious Dependencies as the Substantive Discipline

Once dependencies are visible, S1–S4 ask that they be few and structured. ADT-style construction tends to satisfy S1: constructors wrap values rather than transforming them, so traits read as projections of single random choices rather than as combinations. The boundary between sum-type tag choices and their branches naturally yields existential rather than parametric gating (S2), provided sampled values are not fed back as distribution parameters: a discipline purity encourages but does not enforce. Structured programming's emphasis on clean recursive decomposition, particularly divide-and-conquer over left-associated iteration, tends to keep the call graph a balanced tree (S3, S4) rather than a linear chain or a graph with shared sub-computations. None of these traditions targets any single condition, but each removes a class of dependency that would violate one or more.

6.3 The Underlying Virtue

What unites these traditions is an orientation toward making dependencies explicit and few. This matters for evolutionary operators in two distinct ways. Explicit dependencies are a precondition for the framework, since the DGBN is derived from source. Parsimonious dependencies are what make the induced operators well-behaved: mutations propagate to every downstream trait, and crossover disrupts every trait whose support straddles the cut, so the fewer and

thinner the dependencies, the more localized and meaningful the resulting operators. Programmers value the same two properties for unrelated reasons: reasoning about code requires seeing what depends on what, and predicting the consequences of changes requires that not too much depend on any one thing. The alignment between S1–S4 and these disciplines reflects this shared need. Both evolutionary search and human comprehension work through small, traceable propagation, and both rely on the same dependency hygiene. A programmer working in clean ADT-and-pure-functions style is, without intending to, designing for evolutionary search.

7 Conclusions

This paper addressed a central gap in Program Trace Optimization (PTO): prior work explained why certain generators induce well-behaved evolutionary operators, but did not provide concrete guidance on how to design such generators. We reframed this as a design problem and inverted the existing causal theory of the genotype–phenotype map to derive actionable principles at the level of generator code.

The main technical contribution is the identification of four structural conditions (S1–S4) on generator programs, jointly sufficient to guarantee two desirable operator properties: perfect inheritance under crossover and logarithmic mean pleiotropy under mutation. These conditions establish a direct link from code-level structure, through causal dependency graphs, to quantitative properties of operator behavior, turning PTO's previously informal notion of "well-structured generators" into a set of verifiable design criteria. The conditions are most naturally satisfied by a canonical form, i.e., anamorphic unfolding over algebraic data types with balanced recursion and independent local decisions, and align broadly with established programming disciplines (algebraic data types, pure functional programming, structured programming) that share an orientation toward explicit, parsimonious dependencies.

More broadly, this work illustrates a methodological shift: from explanatory to generative use of theory. A sufficiently general theory of an algorithm is a declarative structure that can be inverted, deriving conditions that produce desired behavior rather than merely accounting for observed behavior. PTO's causal theory serves both roles: predicting operator performance from a given generator, and guiding the construction of generators that exhibit it by design.

The scope of these results is strongest for generators built from independent, recursively composed decisions. Problems involving global constraints or shared dependencies like permutations, constrained grammars, or neural architectures with parameter sharing, necessarily violate some of the conditions, and understanding how such constraints interact with locality and modularity remains open. A further direction is to complete the chain from code to phenotype by characterizing, explicitly at a structural level, which phenotypic components are exchanged or modified by the induced operators.

References

1. Arrieta, A.B., Díaz-Rodríguez, N., Del Ser, J., Bennetot, A., Tabik, S., Barbado, A., García, S., Gil-López, S., Molina, D., Benjamins, R., et al.: Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. Information fusion **58**, 82–115 (2020)
2. Lipton, Z.C.: The mythos of model interpretability: In machine learning, the concept of interpretability is both important and slippery. Queue **16**(3), 31–57 (2018)
3. McDermott, J., Moraglio, A.: Program trace optimization with constructive heuristics for combinatorial problems. In: European Conference on Evolutionary Computation in Combinatorial Optimization (Part of EvoStar). pp. 196–212. Springer (2019)
4. Meijer, E., Fokkinga, M.M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture. p. 124–144. Springer-Verlag, Berlin, Heidelberg (1991)
5. Moraglio, A.: A causal model of the genotype-phenotype map in program trace optimization. In: Proceedings of the 18th International Conference on Parallel Problem Solving from Nature (PPSN 2026). Lecture Notes in Computer Science, Springer (2026), to appear
6. Moraglio, A., McDermott, J.: Program trace optimization. In: Parallel Problem Solving from Nature. pp. 334–346. Springer (2018)
7. Moraglio, A., McDermott, J.: Evolving programs in the lambda calculus using program trace optimisation. In: Burlacu, B., Olivetti de França, F., Lalejini, A., Kelly, S., Banzhaf, W. (eds.) Genetic Programming Theory and Practice XXII. Genetic and Evolutionary Computation Series, Springer (2025), in press
8. Moraglio, A., McDermott, J.: Geometric particle swarm optimization in program trace optimization. In: EvoApplications (2). Lecture Notes in Computer Science, vol. 15613, pp. 145–159. Springer (2025)
9. Moraglio, A., Tonda, A.: Language model-driven program synthesis with program trace optimization on the abstraction and reasoning corpus. In: Burlacu, B., Olivetti de França, F., Lalejini, A., Kelly, S., Banzhaf, W. (eds.) Genetic Programming Theory and Practice XXII. Genetic and Evolutionary Computation Series, Springer (2025), in press
10. Moraglio, A., Tonda, A.: Program trace optimization for language model search: the abstraction and reasoning corpus case. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion. pp. 647–650. GECCO '25 Companion, Association for Computing Machinery, New York, NY, USA (2025). https://doi.org/10.1145/3712255.3726554, https://doi.org/10.1145/3712255.3726554
11. Pearl, J.: Causality. Cambridge university press (2009)
12. Zhou, R., Bacardit, J., Brownlee, A.E.I., Cagnoni, S., Fyvie, M., Iacca, G., McCall, J., van Stein, N., Walker, D., Hu, T.: Evolutionary computation and explainable ai: A roadmap to understandable intelligent systems. IEEE Transactions on Evolutionary Computation **29**(5), 2213–2228 (2024)