Graph Expansion as Concept Blending

A Constructive Operation on Structured Graph Terms

L. G. Meredith
F1R3FLY.io
lgrm@f1r3fly.io

March 18, 2026

Abstract

We present a graph expansion operation—expand(G, M, F, D, S)—that takes a structured graph G and two distinguished nodes M and F, and produces a new graph containing two additional nodes D and S whose connectivity is defined by *crossing* the port structure of M and F. The construction is naturally interpreted as **concept blending** in a knowledge graph: M and F are parent concepts, and D and S are the two blended offspring, each inheriting half of each parent's relational profile in a complementary fashion. We describe the operation in terms of a process-calculus-inspired graph DSL, give a high-level algorithmic account, and provide a Scala implementation. The operation exhibits a clean duality, composes well under tensor, and is compatible with the reflective name structure of the underlying language.

Contents

1 Introduction

Concept blending—the cognitive operation by which two distinct mental spaces are selectively merged to produce a novel emergent structure [1]—has long been recognized as a driver of creativity and analogical reasoning. In knowledge representation, blending has been modelled as a category-theoretic pushout [2], but practical, computable formulations that work directly on graph structures remain less explored.

This paper proposes a concrete, graph-level blending operation grounded in a structured graph DSL with process-calculus semantics. Given a graph G and a pair of *parent* nodes M and F, we construct two *child* nodes D and S whose incoming and outgoing edges are drawn from the port sets of M and F in a *crossed* fashion:

- D inherits the *incoming* neighbourhood of M and the *outgoing* neighbourhood of F.
- S inherits the *incoming* neighbourhood of F and the *outgoing* neighbourhood of M.

The symmetry between D and S is exact: swapping $M \leftrightarrow F$ simultaneously swaps $D \leftrightarrow S$, establishing a duality that mirrors the complementary nature of blended concepts.

The remainder of the paper is structured as follows. Section 2 introduces the graph DSL and its key constructs. Section 3 presents the expansion algorithm at a high level, together with a worked example and a discussion of structural properties. Section 4 situates the operation within the concept-blending literature. Appendix A gives a precise grammar and semantics of the DSL; Appendix B contains the complete Scala implementation.

2 The Graph DSL

The graph language used throughout this paper is inspired by the ρ-calculus [3]: names can be *quoted graphs*, giving the language a reflective character that is central to its expressiveness. Below we introduce the essential constructs; a full grammar is given in Appendix A.

2.1 Core Constructs

0 The *empty graph*. No nodes, no edges.

$v \mid G$
Vertex introduction. Adds vertex v to graph G. Written `v | G` in the DSL.

$G * H$
Tensor product (disjoint union). Combines two graphs without identifying any nodes. Associative and commutative up to isomorphism, with **0** as unit.

(b_1, b_2)
An *anonymous directed edge* from the vertex denoted by binding b_1 to that of b_2.

$n(b_1, b_2)$
A *named directed edge* carrying label n. Labels are *names* in the DSL sense, and can themselves be quoted graphs: $n ::= @G$.

`let` $x = \langle n \rangle$ **`in`** G
A *vertex binding*: introduces a local variable x standing for the vertex named n, scoped over graph G.

`let` $X = H$ **`in`** G
A *subgraph binding*: introduces a local variable X standing for subgraph H, scoped over G.

`[= G H]`
An *anonymous rewrite rule* from pattern G to replacement H.

`n[= G H]`
A *named rewrite rule* with label n.

2.2 Names and Reflection

A distinctive feature of the DSL is that names can be formed by *quoting* graphs or vertices:

$$n ::= \_ \mid x \mid X \mid @G \mid @v$$

This means that vertices can carry structured, graph-valued names—a form of reflection that allows the expansion operation to produce canonical names for D and S (e.g., $@(M * F)$ and $@(F * M)$) directly from the parent nodes.

2.3 A Small Example

Consider the following graph G encoding a simple taxonomy:

```
let mammal = <mammal> in
let bird   = <bird>  in
let animal = <animal> in
  mammal | bird | animal | 0
  * (let x = <mammal> in x, let y = <animal> in y)
  * (let x = <bird>   in x, let y = <animal> in y)
```

Listing 1: A small graph in DSL syntax

Here `mammal` and `bird` are both hyponyms of `animal`. The expansion operation applied to $M =$ `mammal` and $F =$ `bird` would produce two blended nodes D and S that inherit crossed subsets of these taxonomic edges.

3 The Expansion Algorithm

3.1 Port Sets

Definition 3.1 (Port sets)**.** Let G be a graph and v a vertex occurring in G. Define:

$$\begin{aligned} \text{inSrc}(v, G) &= \{\, b_1 \mid (b_1, b_2) \in \text{edges}(G),\ \text{resolves}(b_2, v) \,\} \\ \text{outTgt}(v, G) &= \{\, b_2 \mid (b_1, b_2) \in \text{edges}(G),\ \text{resolves}(b_1, v) \,\} \end{aligned}$$

where $\text{resolves}(b, v)$ holds when binding b denotes vertex v under the ambient scoping, and $\text{edges}(G)$ is the set of all edge pairs in G (see Appendix A for the formal definition).

Informally, $\text{inSrc}(v, G)$ collects the *sources* of all edges pointing *into* v, while $\text{outTgt}(v, G)$ collects the *targets* of all edges emanating *from* v.

3.2 The expand Operation

Definition 3.2 (Graph expansion)**.** Let G be a graph, let M and F be vertices of G, and let D and S be *fresh* vertices not occurring in G. Define:

$$\text{expand}(G, M, F, D, S) \;:=\; G * D * S * E_D^{\text{in}} * E_D^{\text{out}} * E_S^{\text{in}} * E_S^{\text{out}}$$

where the four edge tensors are:

$$E_D^{\text{in}} = \bigotimes_{b \in \text{inSrc}(M,G)} (b,\, D) \qquad \text{(sources of } M\text{'s in-edges point to } D\text{)}$$

$$E_D^{\text{out}} = \bigotimes_{b \in \text{outTgt}(F,G)} (D,\, b) \qquad (D \text{ points to targets of } F\text{'s out-edges})$$

$$E_S^{\text{in}} = \bigotimes_{b \in \text{inSrc}(F,G)} (b,\, S) \qquad \text{(sources of } F\text{'s in-edges point to } S\text{)}$$

$$E_S^{\text{out}} = \bigotimes_{b \in \text{outTgt}(M,G)} (S,\, b) \qquad (S \text{ points to targets of } M\text{'s out-edges})$$

Here $\bigotimes$ denotes iterated tensor product, with $\mathbf{0}$ as the empty product.

Figure 1 illustrates the crossing structure diagrammatically.

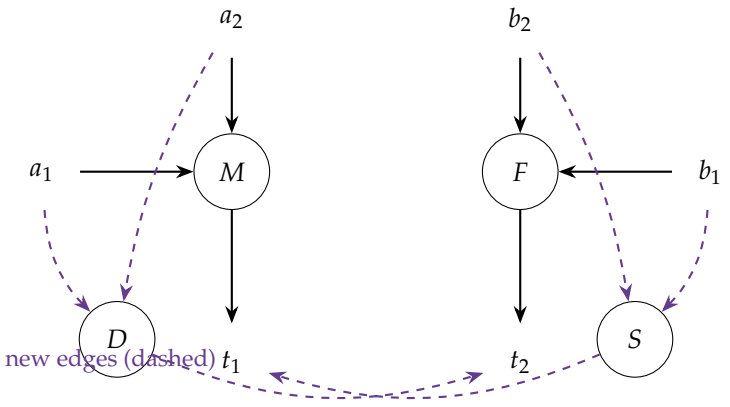

Figure 1: The crossing structure of $\text{expand}(G, M, F, D, S)$. Solid arrows are existing edges of G; dashed arrows are the new edges incident to the fresh nodes D and S. Node D absorbs the incoming neighbourhood of M and directs its output toward the outgoing neighbourhood of F; node S does the same with M and F swapped.

3.3 High-Level Algorithm

The procedure is straightforward to implement:

1. **Step 1: Collect port sets.** Traverse $\text{edges}(G)$ once, partitioning each edge (b_1, b_2) into the four port sets based on whether b_1 resolves to M or F, and whether b_2 resolves to M or F.

2. **Step 2: Allocate fresh nodes.** Choose names D and S not already present in G. A canonical choice is $D = @(M * F)$ and $S = @(F * M)$ using the reflective naming of the DSL.

3. **Step 3: Construct new edge sets.** For each element b of each port set, create the corresponding directed edge as per Definition 3.2.

4. **Step 4: Assemble the result.** Return $G * D * S * E_D^{\text{in}} * E_D^{\text{out}} * E_S^{\text{in}} * E_S^{\text{out}}$ where the tensor is represented as a union of the node and edge sets.

The algorithm runs in $O(|\,\text{edges}(G)|)$ time for a single expansion, since each edge is visited exactly once in Step 1 and the construction in Steps 3–4 is linear in the sizes of the port sets.

3.4 Structural Properties

Proposition 3.1 (Duality)**.** $\text{expand}(G, M, F, D, S) \cong \text{expand}(G, F, M, S, D)$*, where* $\cong$ *denotes graph isomorphism.*

Proof. Swapping $M \leftrightarrow F$ exchanges $\text{inSrc}(M, G) \leftrightarrow \text{inSrc}(F, G)$ and $\text{outTgt}(M, G) \leftrightarrow \text{outTgt}(F, G)$, which exactly exchanges the roles of D and S in Definition 3.2. □

Proposition 3.2 (Self-expansion)**.** *When* $M = F$*, the nodes* D *and* S *produced by* $\text{expand}(G, M, M, D, S)$ *receive identical port structure:* $\text{inSrc}(D) = \text{inSrc}(S)$ *and* $\text{outTgt}(D) = \text{outTgt}(S)$*. The expansion degenerates to adding two isomorphic copies of a single "self-blended" node.*

Proposition 3.3 (Compositional stability)**.** *The result of* expand *is a well-formed graph in the DSL as long as* D *and* S *are fresh, and is itself a valid argument to a subsequent* expand *call. In particular, the operation preserves the tensor-monoidal structure of the graph category.*

Remark 3.1 (Labeled edges). When edges carry labels, one may supply a label-combination function $\omega : \text{Name} \times \text{Name} \to \text{Name}$ to determine how source edge labels are inherited by the new edges. The simplest policy preserves the label of the originating edge unchanged.

4 Concept Blending Interpretation

4.1 Blending Spaces

In Fauconnier and Turner's theory [1], a blend consists of two *input spaces* I_1 and I_2, a *generic space* G_0 capturing shared structure, and a *blended space* B that selectively projects from each input and develops emergent structure.

In our formulation:

- The *input spaces* are the local neighbourhoods of M and F in G.
- The *generic space* is G itself, providing the ambient relational context.
- The *blended spaces* are D and S: each projects the incoming structure of one parent and the outgoing structure of the other, producing two complementary emergent entities.

4.2 Why Two Blends?

Classical blending produces a single blended space. Our construction produces *two*, which are each other's mirror under the duality of Proposition 3.1. This is reminiscent of the biological metaphor implicit in the M/F naming: sexual reproduction produces offspring that are distinct but complementary recombinations of parental genetic material. In a knowledge graph, the two blends represent distinct *conceptual stances* one can adopt when combining M and F: prioritising M's incoming context or F's incoming context as the defining feature of the new concept.

4.3 Knowledge Graph Applications

The operation is directly applicable to concept graphs in which nodes are concepts and edges are semantic relations (hypernymy, causation, parthood, etc.). Given two concepts M and F:

- D is the blend that is *arrived at* by the same causes/supersets as M, but *leads to* the same effects/subsets as F.
- S is the complementary blend: arrived at like F, leads to like M.

This provides a systematic, constructive account of creative concept formation that is computable directly from the graph structure.

5 Conclusion

We have presented $\text{expand}(G, M, F, D, S)$, a graph operation that constructs two blended nodes by crossing the port sets of two parent nodes in a structured graph DSL. The operation is simple, efficient, and structurally well-behaved: it preserves the monoidal structure of the graph category, satisfies an exact duality, and supports a clean concept-blending interpretation. A Scala implementation is provided in Appendix B.

Application to Categories

Because a small category is precisely a directed graph (its *underlying graph*) equipped with a composition law and identities [5], the expansion operation applies directly to categories viewed as graphs. Given a category $\mathcal{C}$, two objects M and F play the role of parent nodes, and the port sets $\text{inSrc}(M, \mathcal{C})$ and $\text{outTgt}(F, \mathcal{C})$ are just the domains of morphisms into M and the codomains of morphisms out of F, respectively. The blended objects D and S then inherit crossed hom-set structure: the morphisms arriving at D originate from the same objects that map into M, while those departing from D target the same objects that F maps into. Whether D and S can be equipped with composition laws that make the extended graph into a category is a non-trivial coherence question, but the underlying graph expansion is well-defined and meaningful regardless.

Quantaloids and Goertzel–Bennett Weakness

A *quantaloid* is a category enriched over the category of quantales [6]: hom-sets are replaced by elements of a quantale (a complete lattice equipped with a compatible monoid structure), and composition distributes over arbitrary joins. Quantaloids provide a natural setting for reasoning about graded or uncertain relationships, and have been proposed as a foundation for modelling cognitive uncertainty and semantic proximity.

Goertzel has argued [7] that Bennett's notion of *weakness*—a measure of the degree to which one pattern fails to determine another [8]—can be captured within a quantale-based framework, where the quantale elements represent degrees of evidential support or causal influence. In this setting, the hom-value $\mathcal{C}(A, B) \in Q$ (for a quantale Q) encodes not merely the existence of a relationship from A to B but its *strength* or *weakness*.

The expansion operation extends naturally to this enriched setting. The port sets inSrc and outTgt generalise to *weighted port sets*: for each source b of an edge into M, the edge now carries a quantale value $q \in Q$ representing the strength of that relationship. The blended node D inherits these weighted connections, and the quantale structure determines how weights compose along the new crossed paths. In particular, the join operation of Q provides a canonical way to aggregate multiple weighted paths between the same pair of nodes, while the monoid multiplication propagates weights through composition.

We therefore conjecture that a *quantaloid expansion* $\mathrm{expand}_Q(G, M, F, D, S)$ is well-defined, with blended hom-values given by

$$Q(b, D) = Q(b, M) \qquad Q(D, b) = Q(F, b)$$

and analogously for S, directly inheriting the quantale values of the parent edges. This would make the blending operation a morphism of quantaloid-enriched graphs, and would ground Goertzel's weakness-based similarity judgements in a constructive, graph-level blending procedure. Establishing this conjecture formally—in particular verifying the enriched composition axioms for D and S—is a priority for future work.

Further Directions

Additional future directions include: (1) iterating the expansion to build a *blending lattice* and studying its order-theoretic properties; (2) equipping the operation with a categorical semantics as a functor on a graph topos; (3) integrating the construction with the Rho Calculus reduction semantics to give blended concepts a dynamic, rewriting-based life-cycle; and (4) applying the operation to large-scale knowledge graphs to evaluate its utility for automated analogy and metaphor generation.

References

[1] G. Fauconnier and M. Turner, *Conceptual Integration Networks*, Cognitive Science 22(2), 1998, pp. 133–187.

[2] J. Goguen, *Mathematical Models of Cognitive Space and Time*, in *Reasoning and Cognition*, Springer, 2005.

[3] L. G. Meredith and M. Radestock, *A Reflective Higher-Order Calculus*, Electronic Notes in Theoretical Computer Science 141(5), 2005, pp. 49–67.

[4] L. G. Meredith and S. Bjorg, *Contracts and Types*, Communications of the ACM 46(10), 2003, pp. 41–47.

[5] S. Mac Lane, *Categories for the Working Mathematician*, 2nd ed., Springer, 1998.

[6] K. I. Rosenthal, *The Theory of Quantaloids*, Addison Wesley Longman, 1996.

[7] B. Goertzel, *Generative AI and the Path to AGI: Reflections on Cognitive Architecture and Uncertainty*, Manuscript, SingularityNET / TrueAGI, 2023.

[8] C. H. Bennett, *Logical Depth and Physical Complexity*, in *The Universal Turing Machine: A Half-Century Survey*, Oxford University Press, 1988, pp. 227–257.

Appendices

A Precise DSL Grammar and Semantics

A.1 Grammar

The grammar below is given in LBNF (Labelled Backus–Naur Form), as used by the BNF Converter tool. Production labels are given before the period on each line.

```
-- Coercions (precedence levels)
_ .   Graph  ::= Graph1 ;
_ .   Graph1 ::= Graph2 ;
_ .   Graph2 ::= Graph3 ;
_ .   Graph3 ::= "{" Graph "}" ;

-- Basic Graph Term Constructors
GNil .       Graph3 ::= "0" ;
GVertex .    Graph2 ::= Vertex "|" Graph1 ;
GVar .       Graph2 ::= LVar  "|" Graph1 ;
GNominate .  Graph1 ::= Binding ;
GEdgeAnon .  Graph1 ::= "(" Binding "," Binding ")" ;
GEdgeNamed . Graph1 ::= Name "(" Binding "," Binding ")" ;
GRuleAnon .  Graph1 ::= "[" "=" Graph Graph "]" ;
GRuleNamed . Graph1 ::= Name "[" "=" Graph Graph "]" ;
GSubgraph .  Graph1 ::= GraphBinding ;
GTensor .    Graph  ::= Graph "*" Graph1 ;

-- Bindings
VBind .      Binding     ::= "let" LVar "=" Vertex "in" Graph2 ;

-- Subgraph Bindings
GBind .      GraphBinding ::= "let" UVar "=" Graph "in" Graph2 ;

-- Vertices
VName .      Vertex ::= "<" Name ">" ;

-- Names
NameWildcard .    Name ::= "_" ;
NameVVar .        Name ::= LVar ;
NameGVar .        Name ::= UVar ;
NameQuoteGraph .  Name ::= "@" Graph ;
NameQuoteVertex . Name ::= "@" Vertex ;
separator Name "," ;

-- Tokens
token UVar ((upper (letter | digit | '_' | '\'')*)
           |(('_') (upper | digit | '_' | '\'')+)) ;
token LVar (((lower | '\'') (letter | digit | '_' | '\'')*)
           |(('_') (lower | digit | '_' | '\'')+)) ;

-- Comments
comment "//" ;
comment "/*" "*/" ;
```

Listing 2: Complete LBNF grammar of the graph DSL

A.2 Precedence and Associativity

The coercion chain `Graph` $\succ$ `Graph1` $\succ$ `Graph2` $\succ$ `Graph3` encodes precedence: tensor ($*$) binds most loosely, and braces $\{G\}$ allow any sub-expression at any precedence level. In practice:

$$G * H * K \;\equiv\; (G * H) * K \qquad \text{(left-associative)}$$

A.3 Semantic Domains

Let $\mathcal{N}$ be a countably infinite set of *atoms* (ground names), and let $\mathcal{V} = \{\langle n \rangle \mid n \in \mathcal{N}_\infty\}$ be the set of vertices, where $\mathcal{N}_\infty$ is the closure of $\mathcal{N}$ under quoting.

Definition A.1 (Graph denotation)**.** A *graph* G denotes a pair $(\text{nodes}(G), \text{edges}(G))$ where:

$$\begin{aligned} \text{nodes}(G) &\subseteq \mathcal{V} \\ \text{edges}(G) &\subseteq \mathcal{V} \times \mathcal{N}_\infty \times \mathcal{V} \end{aligned}$$

The edge label $*$ is used for anonymous edges; named edges carry an explicit label from $\mathcal{N}_\infty$.

The structural semantics of each construct is given in Table 1.

Table 1: Denotational semantics of graph DSL constructs.

Construct	Denotation
$\mathbf{0}$	$(\emptyset,\ \emptyset)$
$v \mid G$	$(\{v\} \cup \text{nodes}(G),\ \text{edges}(G))$
$G * H$	$(\text{nodes}(G) \cup \text{nodes}(H),\ \text{edges}(G) \cup \text{edges}(H))$
$(b_1,\ b_2)$	$(\{v_1, v_2\},\ \{(v_1, *, v_2)\})$ where $v_i = [\![b_i]\!]$
$n(b_1,\ b_2)$	$(\{v_1, v_2\},\ \{(v_1, n, v_2)\})$ where $v_i = [\![b_i]\!]$
`let` $x = v$ `in` G	substitute $x \mapsto v$ throughout G, return $G[v/x]$
`let` $X = H$ `in` G	substitute $X \mapsto H$ throughout G, return $G[H/X]$
$[= G\ H]$	$(\emptyset,\ \emptyset)$ together with rule $G \Rightarrow H$

A.4 The edges Extractor

The function edges used in Section 3 is defined recursively:

$$\begin{aligned} \text{edges}(\mathbf{0}) &= \emptyset \\ \text{edges}(v \mid G) &= \text{edges}(G) \\ \text{edges}(G * H) &= \text{edges}(G) \cup \text{edges}(H) \\ \text{edges}((b_1, b_2)) &= \{([\![b_1]\!],\ *,\ [\![b_2]\!])\} \\ \text{edges}(n(b_1, b_2)) &= \{([\![b_1]\!],\ n,\ [\![b_2]\!])\} \\ \text{edges}(\texttt{let } x = v \texttt{ in } G) &= \text{edges}(G[v/x]) \\ \text{edges}(\texttt{let } X = H \texttt{ in } G) &= \text{edges}(H) \cup \text{edges}(G[H/X]) \\ \text{edges}([= G\ H]) &= \emptyset \end{aligned}$$

A.5 Freshness and Quoting

Two canonical fresh names for the offspring nodes are available via the reflective quoting mechanism:

$$D \ :=\ \langle @(M * F) \rangle \qquad S \ :=\ \langle @(F * M) \rangle$$

These are guaranteed to be distinct from any atom-named vertex in G, and are themselves well-formed vertices in the DSL.

B Scala Implementation

The implementation below uses immutable case classes and pure functions. It targets Scala 3 and requires no external dependencies beyond the standard library.

```scala
// --- Names ---
enum Name:
  case Wildcard
  case VarName(id: String)
  case GraphName(id: String)      // upper-case identifier
  case QuoteGraph(g: Graph)
  case QuoteVertex(v: Vertex)

// --- Vertices ---
case class Vertex(name: Name)

// --- Bindings ---
case class Binding(varName: String, vertex: Vertex, body: Graph)

// --- Graph ADT ---
enum Graph:
  case GNil
  case GVertex(vertex: Vertex, rest: Graph)
  case GEdgeAnon(src: Binding, tgt: Binding)
  case GEdgeNamed(label: Name, src: Binding, tgt: Binding)
  case GTensor(left: Graph, right: Graph)
  case GSubgraph(varName: String, sub: Graph, body: Graph)
  case GRuleAnon(lhs: Graph, rhs: Graph)
  case GRuleNamed(label: Name, lhs: Graph, rhs: Graph)
```

Listing 3: Core graph ADT

```scala
import Graph.*

/** An extracted, fully-resolved edge (no bindings, concrete vertices). */
case class Edge(src: Vertex, label: Option[Name], tgt: Vertex)

object GraphOps:

  /** Resolve a Binding to its concrete Vertex. */
  def resolve(b: Binding): Vertex = b.vertex

  /** Extract all edges from a graph, recursively. */
  def edges(g: Graph): Set[Edge] = g match
    case GNil                          => Set.empty
    case GVertex(_, rest)              => edges(rest)
    case GEdgeAnon(src, tgt)           =>
      Set(Edge(resolve(src), None, resolve(tgt)))
    case GEdgeNamed(label, src, tgt) =>
      Set(Edge(resolve(src), Some(label), resolve(tgt)))
    case GTensor(l, r)                 => edges(l) ++ edges(r)
    case GSubgraph(_, sub, body)       => edges(sub) ++ edges(body)
    case GRuleAnon(_, _)               => Set.empty
    case GRuleNamed(_, _, _)           => Set.empty

  /** All vertices explicitly introduced in a graph. */
  def vertices(g: Graph): Set[Vertex] = g match
    case GNil                          => Set.empty
    case GVertex(v, rest)              => Set(v) ++ vertices(rest)
```

```
    case GEdgeAnon(src, tgt)          => Set(resolve(src), resolve(tgt))
    case GEdgeNamed(_, src, tgt)      => Set(resolve(src), resolve(tgt))
    case GTensor(l, r)                => vertices(l) ++ vertices(r)
    case GSubgraph(_, sub, body)      => vertices(sub) ++ vertices(body)
    case GRuleAnon(_, _)              => Set.empty
    case GRuleNamed(_, _, _)          => Set.empty
```

Listing 4: Edge extraction

```
/** Sources of all edges whose target is vertex v. */
def inSrc(v: Vertex, g: Graph): Set[Vertex] =
  edges(g).collect { case Edge(src, _, tgt) if tgt == v => src }

/** Targets of all edges whose source is vertex v. */
def outTgt(v: Vertex, g: Graph): Set[Vertex] =
  edges(g).collect { case Edge(src, _, tgt) if src == v => tgt }
```

Listing 5: Port set computation

```
/**
 * Canonical fresh vertices for D and S, using reflective quoting.
 * D = <@(M * F)>,  S = <@(F * M)>
 */
def freshPair(m: Vertex, f: Vertex): (Vertex, Vertex) =
  val mGraph = GVertex(m, GNil)
  val fGraph = GVertex(f, GNil)
  val d = Vertex(Name.QuoteGraph(GTensor(mGraph, fGraph)))
  val s = Vertex(Name.QuoteGraph(GTensor(fGraph, mGraph)))
  (d, s)
```

Listing 6: Fresh node generation

```
/**
 * Core expansion:
 *   expand(G, M, F, D, S)
 *
 * Returns a new graph extending G with fresh nodes D and S whose
 * connectivity crosses the port sets of M and F.
 *
 * Precondition: D and S must not occur in G.
 */
def expand(
  g: Graph,
  m: Vertex,
  f: Vertex,
  d: Vertex,
  s: Vertex
): Graph =
  val isM = inSrc(m, g)     // sources of edges into M
  val otM = outTgt(m, g)    // targets of edges out of M
  val isF = inSrc(f, g)     // sources of edges into F
  val otF = outTgt(f, g)    // targets of edges out of F

  // Helper: anonymous binding for a resolved vertex
  def bind(v: Vertex): Binding =
    Binding(varName = "_anon", vertex = v, body = GNil)

  // Build an edge set as a tensor of GEdgeAnon terms
  def edgeTensor(srcs: Set[Vertex], tgt: Vertex): Graph =
```

```
      srcs.foldLeft[Graph](GNil) { (acc, src) =>
        GTensor(acc, GEdgeAnon(bind(src), bind(tgt)))
      }

    def tensorFrom(src: Vertex, tgts: Set[Vertex]): Graph =
      tgts.foldLeft[Graph](GNil) { (acc, tgt) =>
        GTensor(acc, GEdgeAnon(bind(src), bind(tgt)))
      }

    val edgesIntoD  = edgeTensor(isM, d)    // inSrc(M) --> D
    val edgesFromD  = tensorFrom(d, otF)     // D --> outTgt(F)
    val edgesIntoS  = edgeTensor(isF, s)    // inSrc(F) --> S
    val edgesFromS  = tensorFrom(s, otM)    // S --> outTgt(M)

    // Introduce D and S as vertices, then tensor all new edges
    val withD = GVertex(d, GNil)
    val withS = GVertex(s, GNil)

    List(g, withD, withS,
         edgesIntoD, edgesFromD,
         edgesIntoS, edgesFromS)
      .foldLeft[Graph](GNil)(GTensor.apply)

  /**
   * Convenience overload: automatically generates fresh D and S via
   * reflective quoting.
   */
  def expand(g: Graph, m: Vertex, f: Vertex): (Graph, Vertex, Vertex) =
    val (d, s) = freshPair(m, f)
    (expand(g, m, f, d, s), d, s)
```

Listing 7: The expand operation

```
/**
 * Labelled expansion: edges inherit labels via the supplied combinator
 * omega.  When omega is not supplied, the label from the originating
 * edge is preserved unchanged.
 */
def expandLabelled(
  g: Graph,
  m: Vertex,
  f: Vertex,
  d: Vertex,
  s: Vertex,
  omega: (Option[Name], Option[Name]) => Option[Name] = (l, _) => l
): Graph =
  def bind(v: Vertex): Binding = Binding("_anon", v, GNil)

  def makeEdge(src: Vertex, tgt: Vertex, label: Option[Name]): Graph =
    label match
      case None    => GEdgeAnon(bind(src), bind(tgt))
      case Some(n) => GEdgeNamed(n, bind(src), bind(tgt))

  def edgesFor(
    portSet: Set[(Vertex, Option[Name])],
    pivot: Vertex,
    pivotIsTgt: Boolean
  ): Graph =
    portSet.foldLeft[Graph](GNil) { case (acc, (v, lbl)) =>
```

```
        val e =
          if pivotIsTgt then makeEdge(v, pivot, lbl)
          else makeEdge(pivot, v, lbl)
        GTensor(acc, e)
      }

    val isM = edges(g).collect {
      case Edge(src, lbl, tgt) if tgt == m => (src, lbl) }
    val otM = edges(g).collect {
      case Edge(src, lbl, tgt) if src == m => (tgt, lbl) }
    val isF = edges(g).collect {
      case Edge(src, lbl, tgt) if tgt == f => (src, lbl) }
    val otF = edges(g).collect {
      case Edge(src, lbl, tgt) if src == f => (tgt, lbl) }

    val eInD  = edgesFor(isM, d, pivotIsTgt = true)
    val eOutD = edgesFor(otF, d, pivotIsTgt = false)
    val eInS  = edgesFor(isF, s, pivotIsTgt = true)
    val eOutS = edgesFor(otM, s, pivotIsTgt = false)

    List(g, GVertex(d, GNil), GVertex(s, GNil),
         eInD, eOutD, eInS, eOutS)
      .foldLeft[Graph](GNil)(GTensor.apply)
```

Listing 8: Labelled-edge variant with label combinator

```
@main def demo(): Unit =
  import GraphOps.*
  import Graph.*, Name.*

  // Build: mammal --> animal, bird --> animal
  val mammal = Vertex(VarName("mammal"))
  val bird   = Vertex(VarName("bird"))
  val animal = Vertex(VarName("animal"))

  def vbind(v: Vertex) = Binding("_", v, GNil)

  val g: Graph =
    GTensor(
      GTensor(
        GTensor(GVertex(mammal, GNil), GVertex(bird, GNil)),
        GVertex(animal, GNil)
      ),
      GTensor(
        GEdgeAnon(vbind(mammal), vbind(animal)),
        GEdgeAnon(vbind(bird),   vbind(animal))
      )
    )

  val (g2, d, s) = expand(g, mammal, bird)

  println(s"D = $d")
  println(s"S = $s")
  println(s"edges(G') = ${edges(g2)}")
  // Expected:
  //   D has no in-edges (mammal had none), out-edge to animal (bird's out)
  //   S has no in-edges (bird had none),   out-edge to animal (mammal's
    out)
```

Listing 9: Unit test sketch