Part Spaces For Scientific Computing

David M. Butler

Limit Point Systems, Inc.

We give a (mostly) non-mathematical description of the sheaf data model representation of data types commonly used in scientific computing.

1 Introduction

In the companion tutorial PartSpace, we gave a (mostly) non-mathematical description of the basic concepts of the sheaf data model. In this tutorial, we describe how those concepts are applied to the data types commonly used in scientific computing.

2 Data types for scientific computing

Fundamental to most scientific computing applications is the representation of what physicists call a field. A field is a mathematical object that describes how some physical property depends on position within some physical object. Computer representation of fields requires three categories of data types, namely:

- spatial types representing the topological and geometrical properties of physical objects;
- algebraic types (scalars, vectors, tensors) representing physical properties; and
- field types themselves.

We discuss the sheaf data model representation of each of these categories in the following.

3 Spatial data: Cell Space

The representation of the topology and geometry of a physical object is determined mostly by the need to support field representations on the object. Representation of a complicated field on a complex object is typically achieved by decomposing the physical object into a collection of simple parts and approximating the field by a simple function on each part.

So the spatial data types of interest in scientific computing are decompositions of space and *any* decomposition of space can be represented as a part space. The most commonly used type of decomposition is generically referred to as a mesh, in which the physical object is decomposed into a collection of primitives such as line segments, triangles, or tetrahedra that overlap only at their boundaries. The primitives are generically called cells and a mesh of such cells is also called a cell complex.

The part space representation of a mesh is straight-forward: each cell is a basic part and there is a link from cell A to cell B if A directly includes B. Typically the cells in a mesh are closed, that is, each cell contains its boundary, and the cells only overlap at their boundaries, so the links always point to strictly decreasing dimension. More general constructions are possible and do occur in some applications, but we won't worry about them here.

We've already seen some extremely simple examples of meshes in the Part Space tutorial. Here we will give several more examples to familiarize the reader with the sheaf data model representation of common cells and meshes. We'll have to stick to very small meshes because as we increase the size of the mesh, the diagrams get large very rapidly.

3.1 Cells

The is a large "zoo" of cell types used in scientific computing, but the majority of applications use cells from just two families, the simplex family and the box family. Each family has a member for each spatial dimension 0 through 3; higher dimensions are possible, but we won't deal with them here.

3.1.1 Simplices

The members of the simplex family are vertex, segment, triangle, and tetrahedron. We've already seen the vertex and segment in the Part Space tutorial, but we repeat them here in

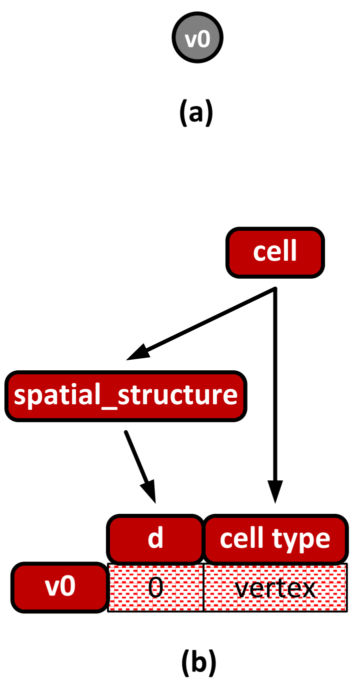


Figure 1: Vertex (a) geometry (b) sheaf table.

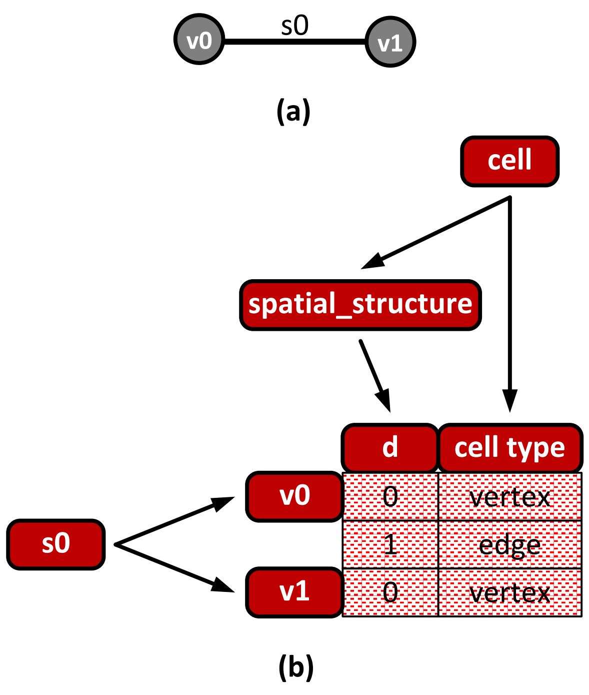


Figure 2: Segment (a) geometry (b) sheaf table.

Figure 1 and Figure 2, respectively. The tables use the cell schema introduced in the Part Space tutorial. Each cell is a basic part and the corresponding row stores its attributes.

Note that neither the vertices nor the segments have coordinates as attributes. Mathematically, position is not an intrinsic attribute of a point and in practice a point may have multiple positions associated with different states or different times. We'll see how to represent position later, in 5.6.

The schema is the same for all the remaining mesh examples, and each table rows just lists the dimension and the cell type, so we will often show just the row graph, in the conventional vertical orientation.

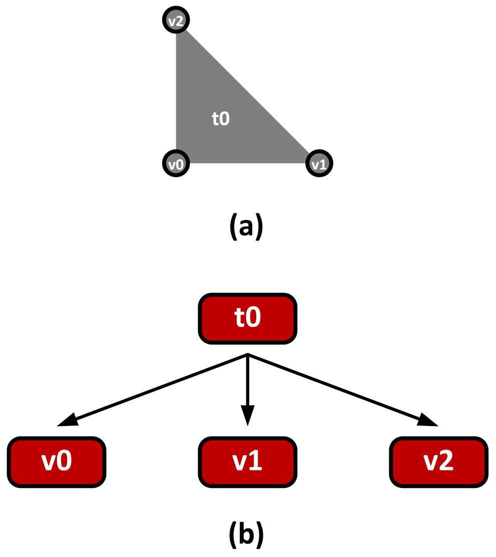


Figure 3: Triangle simplex (a) geometry (b) row part space graph.

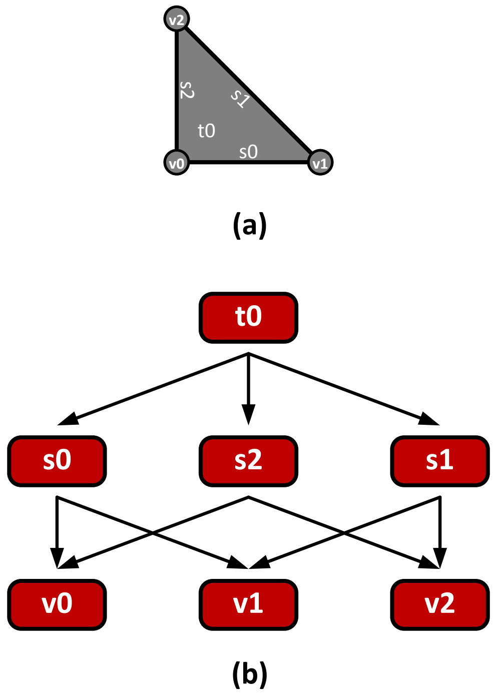


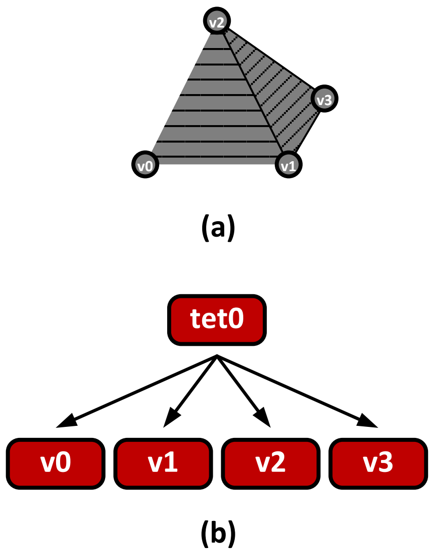
Figure 4: triangle complex (a) geometry (b) row part space graph.

Figure 3 shows the most commonly used triangle cell, the triangle simplex. As suggested by the figure, the vertices are explicitly identified as parts, but not the segments. Part space does not require any specific dimensional cascade. We just enter whatever parts we've got and hook them up by their cover relationships.

On the other hand, if we have the whole dimensional cascade, a triangle complex, part space can represent it. Just enter the basic parts and hook them up by their cover relationships, as shown in Figure 4.

The final member of the simplex family we'll discuss is the tetrahedron. Again, the two common variants are the simplex (vertices only) and the complex (full dimensional cascade), shown in Figure 5 and Figure 6, respectively.

Comparing the part space graphs for the various members of the simplex family, we see there is nothing explicitly dimensional about the graphs; they represent only parts and inclusion. There is however implicit dimensional information. The number of vertices in a simplex is d+1 in dimension d and the depth of the graph is d+1 for a simplicial complex.



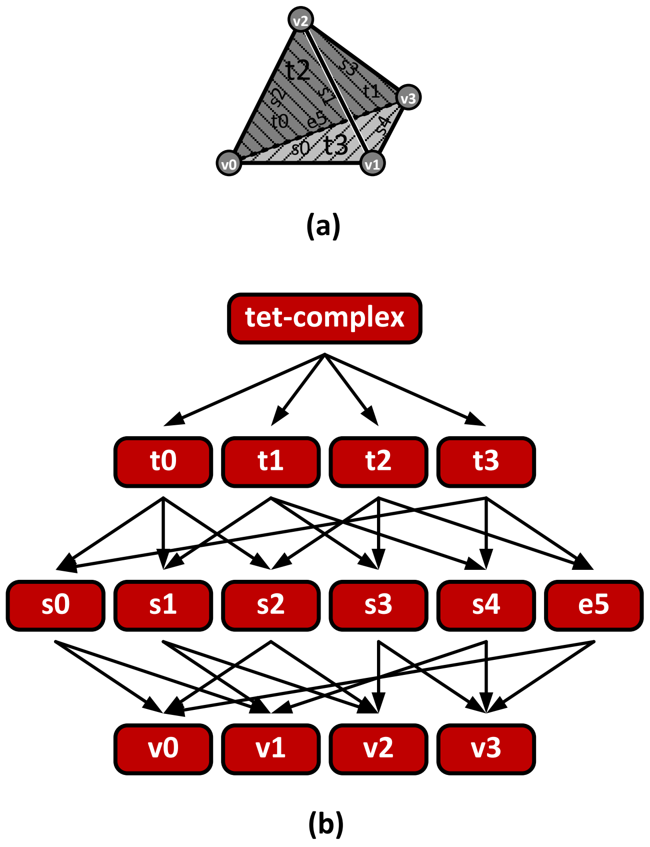


Figure 5: Tetrahedron simplex (a) geometry (b) row part space graph.

3.1.2 Boxes

The members of the box family are vertex, segment, quadrangle, and hexahedron. We've already discussed vertex and segment, which are in both families. The vertex only variant of the quadrangle is shown in Figure 7.

Exercise 1: Draw the part space graph for the quad complex shown in Figure 8.

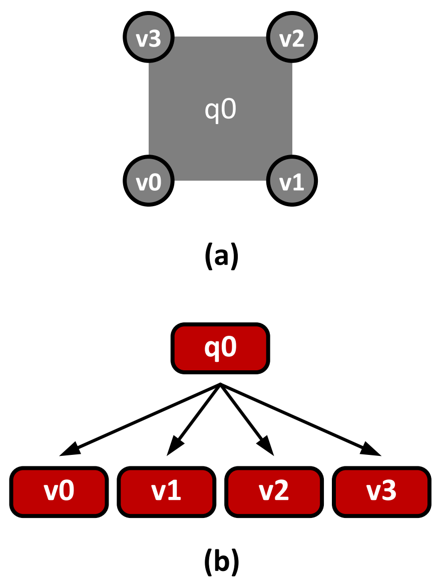


Figure 7: Quad with vertices only (a) geometry (b) row part space graph.

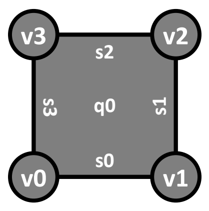


Figure 8: Quad complex geometry.

The vertex only variant of the hexahedron is shown in Figure 9.

Exercise 2: Draw the part space graph for the hex complex shown in Figure 10.

Comparing the part space graphs for the various members of the box family, we see once again there is nothing explicitly dimensional about the graphs; they represent only parts and inclusion. But as with the simplices, there is implicit dimensional information. The number of vertices in a d dimensional box is 2^d. The depth of the graph for a box complex is d+1, the same as a simplicial complex.

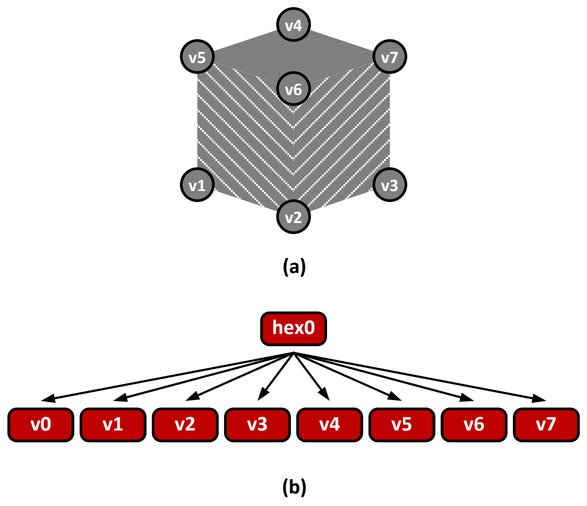


Figure 9: Hex with vertices only (a) geometry (b) row part space graph.

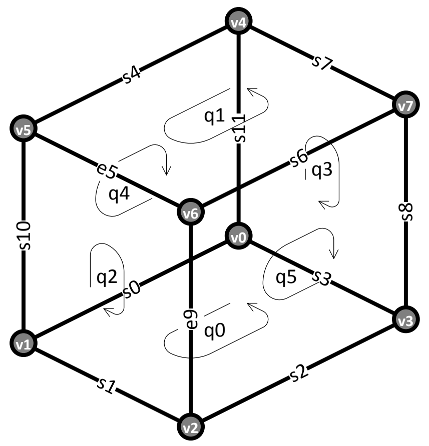


Figure 10: Hex complex geometry

3.2 Meshes

Now that we've surveyed the most common types of cells, let's make some meshes with them. We'll have to stick to small meshes, as we said before, in order to keep the diagrams manageable, but we can explore most of the important features with some pretty small meshes.

3.2.1 Dimension 0

The simplest "mesh" we can construct contains only vertices, a 0D mesh or point cloud, as shown in Figure 11. Since the mesh is precisely the union of the 3 vertices, it is a composite part and we've included the whole sheaf table in the diagram to remind the reader that a composite part doesn't have a corresponding row in the table. The empty placeholder row is there only so the row graph and table will line up.

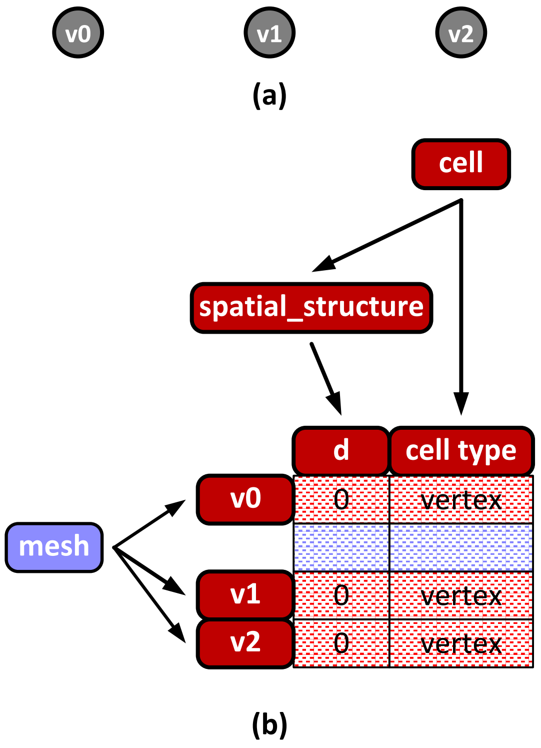


Figure 11: Point cloud (a) geometry (b) row part space graph.

3.2.2 Dimension 1

We discussed the single line segment example in the Part Space tutorial. We can find some interesting features by just increasing the number of segments to 2, forming a 1D mesh or polyline, as shown in Figure 12. The mesh in this case is the union of the two segments and so it is a composite part, just as in the 0D case.

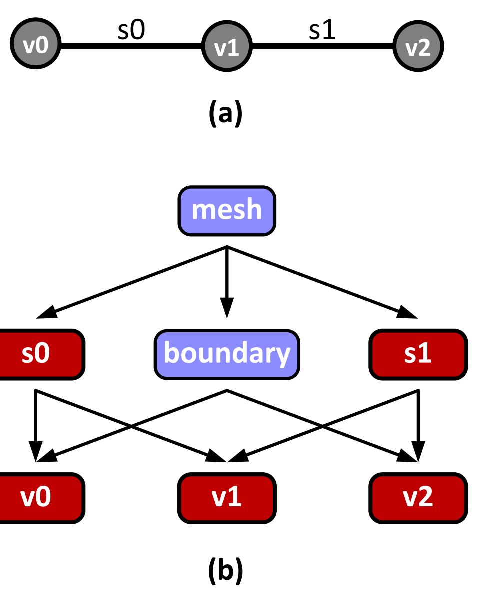


Figure 12: Polyline (a) geometry (b) sheaf table.

Part space can represent any and every composite part, so we can always instantiate whatever composite parts we are interested in. The boundary of a mesh is frequently of interest, so we created a composite part for it and linked it to the vertices in the boundary, v0 and v2.

Figure 13 shows the row graph for another 1D mesh: a self-intersecting polyline, s3

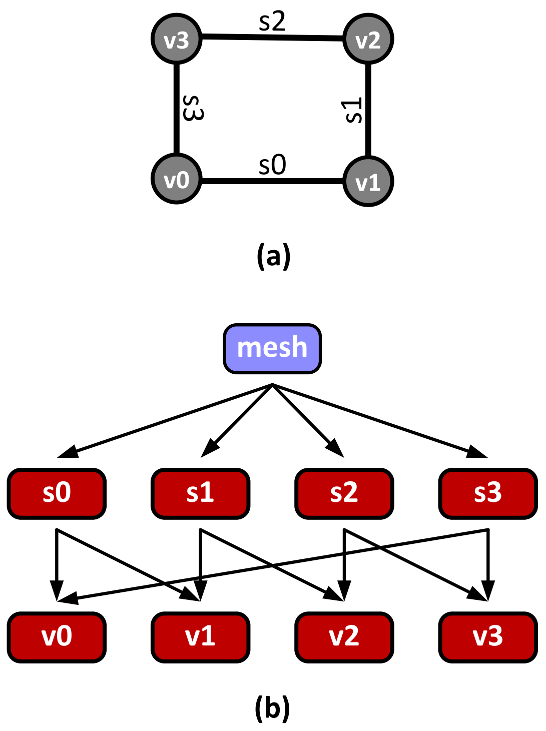


Figure 13: Self intersecting polyline (a) geometry (b) row part space graph.

intersects s0 at v0. The reader might worry that self-intersection somehow complicates the part space, but as Figure 13(b) shows, there is no complication. Segments s1 and s3 both cover v0, no problem.

Exercise 3: Draw the part space graph for the mesh of Figure 14; include a part for the boundary.

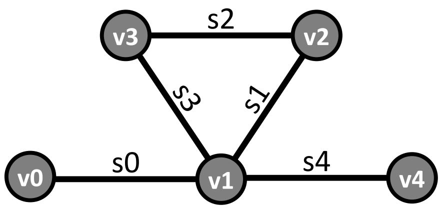


Figure 14: Mesh for Exercise 3.

That's about all we can learn from polylines. Let's step up to dimension 2.

3.2.3 Dimension 2

A two triangle mesh, the triangle analog of the polyline of Figure 12, is shown in Figure 15. Comparing Figure 15 with Figure 12, the two graphs are qualitatively similar. The

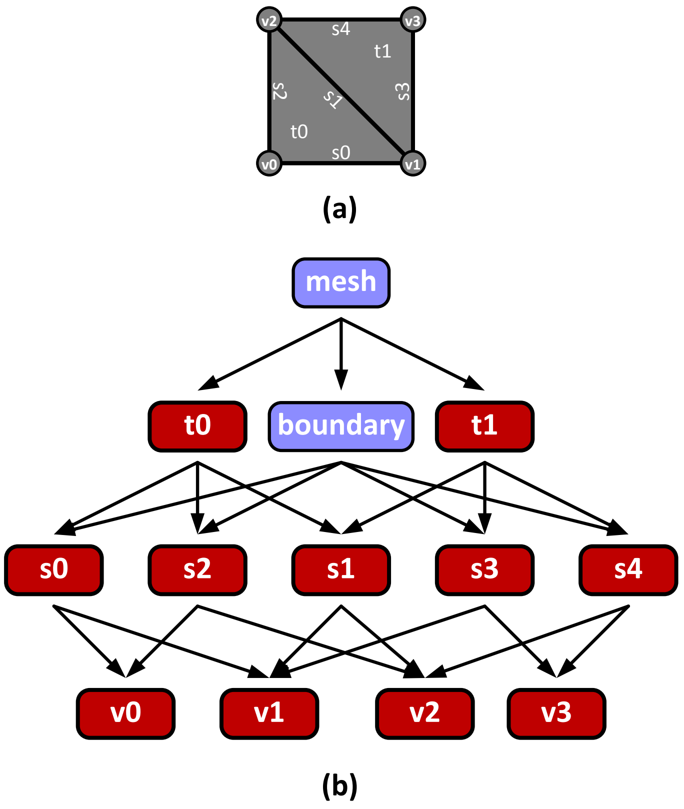


Figure 15: Triangle mesh (a) geometry (b) row part space graph.

mesh is a composite part covering the maximum dimension cells. The boundary is a composite part covering cells of dimension one less. As we've said before, there is nothing explicitly dimensional about the part space representation.

3.2.4 Dimension 3

By now it should be clear that dimension doesn't qualitatively change the construction of the part space. Tets and hexes have more parts than triangles and quads, so the 3 dimensional diagrams are larger and harder to draw. But the effort won't teach us anything new, so we'll move on to the next category of data types.

4 Physical properties: FiberSpace

Although in principle a property can be of any type, the overwhelming majority of scientific computing applications use the algebraic types prescribed by mathematical physics: scalars, vectors, and tensors. The mathematical definition of these types can be interpreted as a class hierarchy with the abstract d-dimensional vector type as the base class and specializing in several steps to the familiar 1, 2, and 3D vectors of Euclidean geometry. The SheafSystem supports this entire hierarchy, as well as several other related mathematical types, but we'll use a simplified hierarchy for this tutorial. Our hierarchy will have just 3 classes:

Class vd: the abstract d-dimensional vector. Mathematically, abstract vectors have an integer dimension, components, an addition operation and a multiplication-by-scalar operation. Dimension is the same for all instances of a given type of vector, so it is tempting to introduce it as a static data member in class vd. But if we do this, all descendants of vd share the same dimension, which is incorrect. So we defer introduction of the static data member to the concrete descendants of vd and define dimension as a pure virtual function in class vd. The components should be ordinary data members, but in class vd we don't know how many there are, so we introduce a pure virtual function for component as well. Addition and multiplication by scalar can be implemented on top of dimension and component, but we aren't interested in the details.

```
class vd
{
   virtual int dimension() = 0; // Dimension.
   virtual double component(int i) = 0; // i-th component (abstract)
   vd operator+(vd xother);   // Vector addition of this and xother.
   vd operator*(double xscalar) // Scalar multiplication by xscalar.
}
```

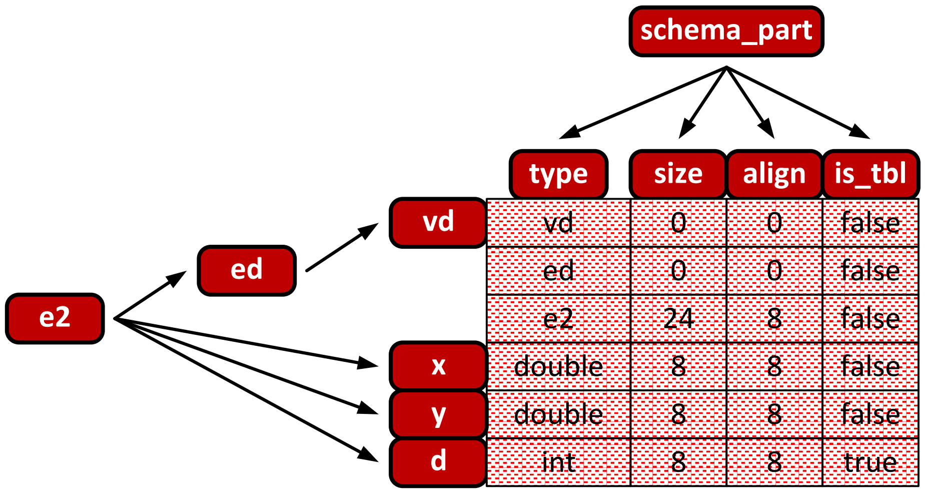
Class ed: the abstract d-dimensional Euclidean vector. Mathematically, a Euclidean vector is special kind of abstract vector that has an additional operation, the dot product, so class ed inherits class vd and provides the dot product as a member function. We won't worry about the implementation of this function either. Class ed does not define any data members.

```
class ed : public vd
{
   double dot(ed xother);  // Dot product of this and xother.
}
```

Class e2: the ordinary 2D Euclidean vector. Mathematically, e2 is a special kind of Euclidean vector with d = 2. We define a static data member d for the dimension and two ordinary data members, x and y, for the components. We can then implement the dimension and component functions using these data members.

```
class e2 : public ed
{
   static const int d = 2;
   double x;  // Cartesian x component.
   double y;  // Cartesian y component.
   virtual int dimension(){return d;};   // Dimension (implemented).
   virtual double component(int i) // i-th component (implemented)
   {return i=0 ? x : y;};
}
```

Figure 16 shows the schema table for the vd class hierarchy and Figure 17 shows a sheaf table for the e2 type, populated by three example values.



	type	size	align	is_tbl
vd	vd	0	0	false
	ed	0	0	false
	e2	24	8	false
x	double	8	8	false
y	double	8	8	false
d	int	8	8	true

Figure 16: Schema table for vd class hierarchy.

The first thing to observe about this example is that, unlike the spatial types where all the structure was in the row part space, there is typically very little going on in the row part space of the property types. The structure is in the column part space. This example has only 3 classes in the hierarchy and only 3 attributes as well. The full class hierarchy has many classes and many attributes.

The second thing to observe is that Figure 17 represents the conceptual structure for the e2 table, but not the way it is stored. The figure suggests we're storing d once for each vector but since it is a table attribute, we are storing it once for the entire table, as shown in Figure 18. Note that the schema for the row portion is the subspace of the column part space that is above the row attributes while the schema for the table portion is the subspace above the table attributes.

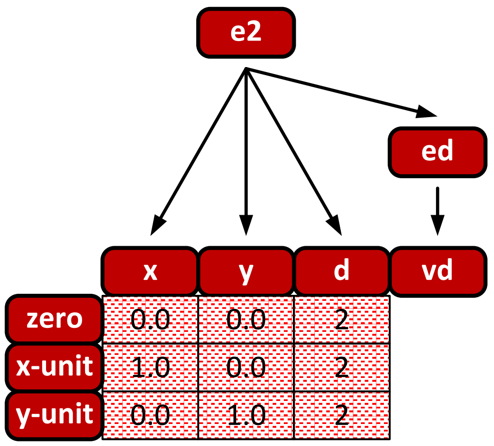


Figure 17: Table for vector type e2.

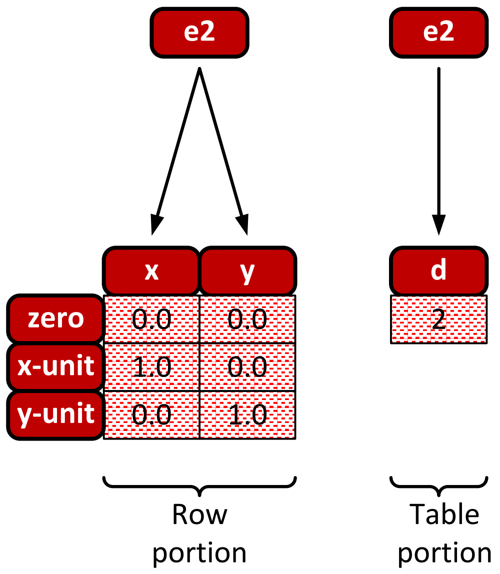


Figure 18: Vector type e2 with table attribute d.

5 <u>Fields</u>

As we said, a field represents how some physical property depends on position in some physical object, for instance, how temperature depends on position in a room, or how the velocity of air flow depends on position in the neighborhood of an aircraft wing.

Mathematically, a field is a map from the physical object, which we will refer to as the base space, to a set of possible property values, which we will refer to as the property space.

The sheaf data model represents objects as rows in a table, so we want to define a table for maps from a given base space to a given property space. We'll introduce the basic concepts with a simple example.

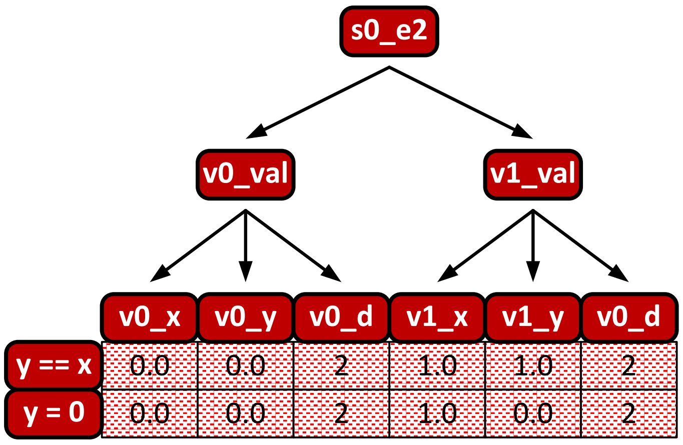
5.1 A simple example

An example requires a base space and a property space. We'll use the line segment example introduced in Figure 2 as the base space and the 2D Euclidean vector e2, Figure 17, as the property space.

The notion of a map is usually introduced in elementary mathematics as a procedure or rule that assigns a property value to each point of the base space. We can think of a map as an object that has a member function which we will call "evaluation" that implements the procedure of the map. A suitable map object for our example is:

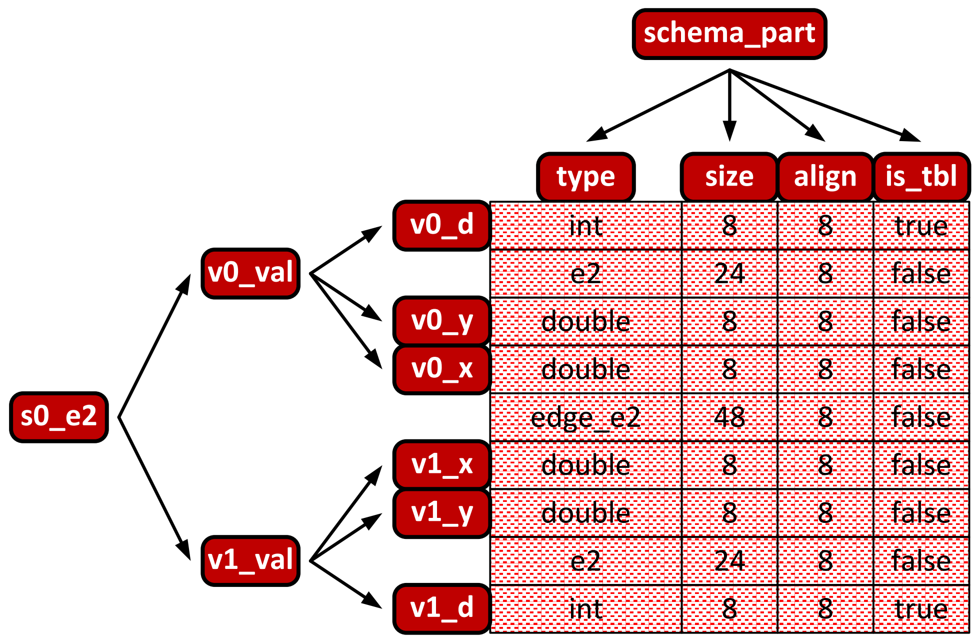
```
class segment_e2
{
   e2 evaluation(double u)    //  The procedure or rule.
   {
      // Linear interpolation.
      e2 result;
      result.x = (1-u)*v0_val.x + u*v1_val.x;
      result.y = (1-u)*v0_val.y + u*v1_val.y;
      result.d  = v0_val.d; // d the same at both points
      return result;
   };

   e2 v0_val; // value of map at vertex 0.
   e2 v1_val; // value of map at vertex 1.
}
```

Figure 19 shows the object table for this example, populated by two example maps, and Figure 20 shows the schema table. The object table stores the data associated with each map object, while the schema table stores the type and size information.



	v0_x	v0_y	v0_d	v1_x	v1_y	v0_d
y == x	0.0	0.0	2	1.0	1.0	2
y = 0	0.0	0.0	2	1.0	0.0	2

Figure 19: Table for map s0_e2.



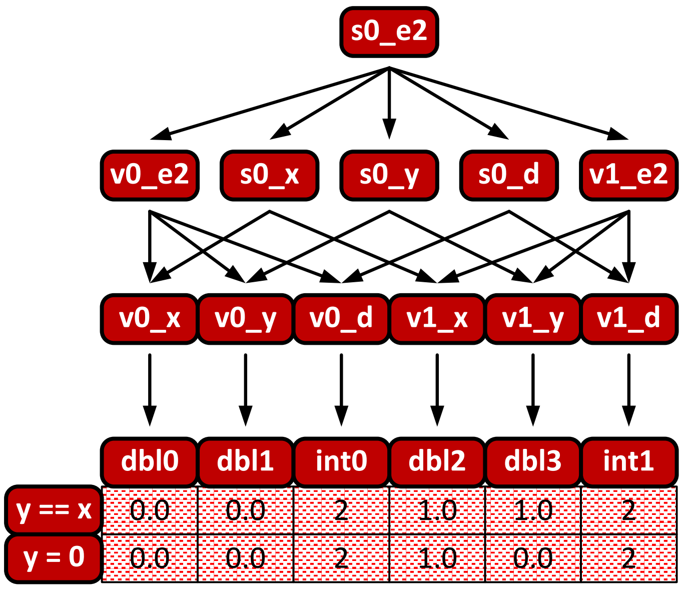
	type	size	align	is_tbl
v0_d	int	8	8	true
	e2	24	8	false
v0_y	double	8	8	false
v0_x	double	8	8	false
	edge_e2	48	8	false
v1_x	double	8	8	false
v1_y	double	8	8	false
	e2	24	8	false
v1_d	int	8	8	true

Figure 20: Schema table for map s0_e2.

Figure 19 is analogous to Figure 17, it shows the conceptual structure of the s0_e2 table without any table attribute optimization. The table attribute optimization for a property space propagates to any map using that property space, but before we apply that optimization, let's first deal with restriction.

5.2 Restriction and product schema

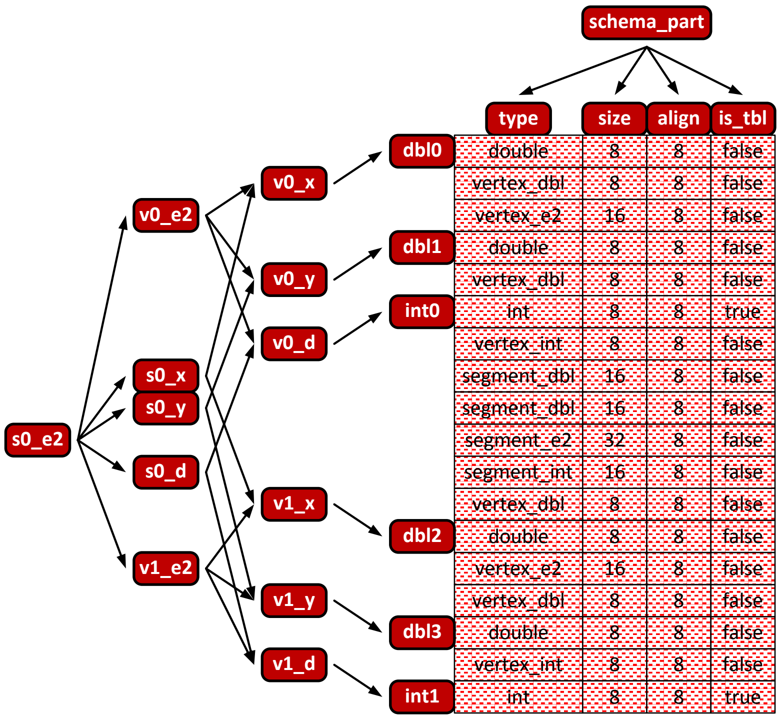
An important property of any map is that it can be restricted to any part of its base space. For instance, we can restrict our map on s0 to a map on just v0. In terms of our map object, this means we would like the map on s0 to have a subobject which is a map on v0, which in turn means we would like the schema to have a member that corresponds to the subtype for the map on v0. It turns out that given any map schema, we can always convert it to a form that has a member for every combination of base space part and property schema part. We'll call this form the product form, because mathematically it corresponds to a certain product of the base part space and the property schema part space. We construct the product form by first creating a node for each possible pair (base basic part, property schema basic part). We then create a cover link from a node A to a node B if and only if (A.base covers B.base and A.property equals B.property) or (A.base equals B.base and A.property covers B.property). Figure 21 shows our example map table with schema in product form.



	dbl0	dbl1	int0	dbl2	dbl3	int1
y == x	0.0	0.0	2	1.0	1.0	2
y = 0	0.0	0.0	2	1.0	0.0	2

Figure 21: Table for map s0_e2 with schema in product form.

With the schema in product form, we can restrict a map to any part of the base or any part of the property. The schema table, shown in Figure 22, contains the type and size information needed to properly interpret the restrictions.



	type	size	align	is_tbl
dbl0	double	8	8	false
	vertex_dbl	8	8	false
	vertex_e2	16	8	false
dbl1	double	8	8	false
	vertex_dbl	8	8	false
int0	int	8	8	true
	vertex_int	8	8	false
	segment_dbl	16	8	false
	segment_dbl	16	8	false
	segment_e2	32	8	false
	segment_int	16	8	false
	vertex_dbl	8	8	false
dbl2	double	8	8	false
	vertex_e2	16	8	false
	vertex_dbl	8	8	false
dbl3	double	8	8	false
	vertex_int	8	8	false
int1	int	8	8	true

Figure 22: Schema table in product form for map s0_e2.

As shown in the schema table, the product form requires a map type for each combination of cell type and property type. We call a map class for a particular (cell, property type) combination an evaluator. The SheafSystem provides evaluators for all the common cell and property types. We've already seen the segment_e2 class. Suitable evaluators for (vertex, e2) and (segment, double) are:

```
class vertex_e2
{
   e2 evaluation()    // The procedure or rule.
   {
      // Just construct the result from the components
      return e2(x, y, d);
   };
   double x;  // x component of vertex.
   double y;  // y component of vertex.
   int d;     // Dimension.
}

class segment_dbl
{
   double evaluation(double u)     //  The procedure or rule.
   {
      // Linear interpolation of the attributes.
      return (1-u)*v0_val + u*v1_val;
   };
   double v0_val;  // value at vertex 0.
   double v1_val;  // value at vertex 1.
}
```

Exercise 4: Write an evaluator class for (vertex, double).

Now we can apply the table attribute optimization. As shown in Figure 23, the schema for the row portion is the subspace of the column part space that is above the row attributes while the table portion is a copy of the table portion of the property space.

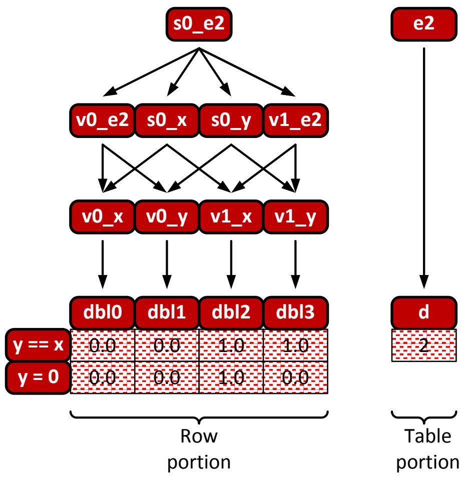
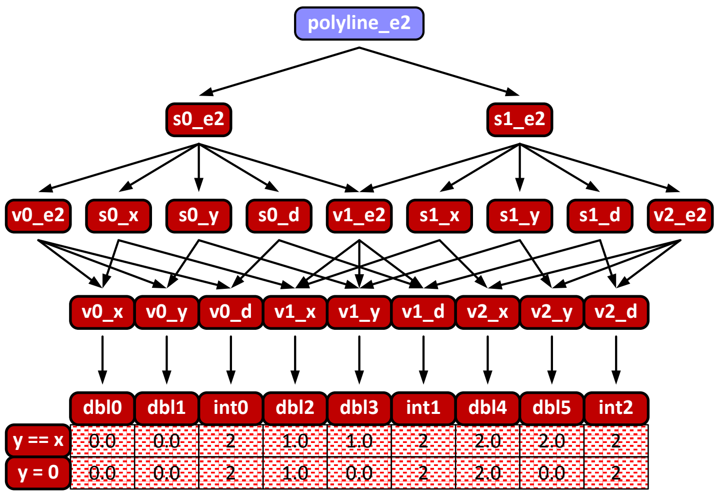


Figure 23: Table for map s0_e2 with schema in product form and table attribute optimization.



	dbl0	dbl1	int0	dbl2	dbl3	int1	dbl4	dbl5	int2
y == x	0.0	0.0	2	1.0	1.0	2	2.0	2.0	2
y = 0	0.0	0.0	2	1.0	0.0	2	2.0	0.0	2

Figure 24: Table for polyline_e2 with schema in product form.

5.3 Local and global map objects

The example of the two preceding sections has an important simplification: the procedure of the map object applies to the entire base space, so a single map object covers the entire base space. We say the map is "global". It's typical in practice for a map object to be "local", to cover only a part of the base space. How do we deal with this case? The answer is straight-forward, we need a local map object for each part. We can explore this by modifying our example to use a base space with more segments. The polyline from Figure 12 will do. Remember that the mesh is not a basic part, it is the composite part made up of the 2 segments.

The map table for the 2 segment base space is shown in Figure 24. The diagram shows that the schema for the whole map is a composite part, polyline_e2, with basic parts, s0_e2 and s1_e2, one for each segment. Each basic part is an instance of the class segment_e2 specified above. The two map objects share the attributes associated with vertex v1, which is shared by the two segments, but have different instances of the attributes on the other two, unshared vertices. We can summarize this structure succinctly by saying "a part of a map type is a map type on a part".

Exercise 5: Draw the table for polyline_e2 with the table attribute optimization.

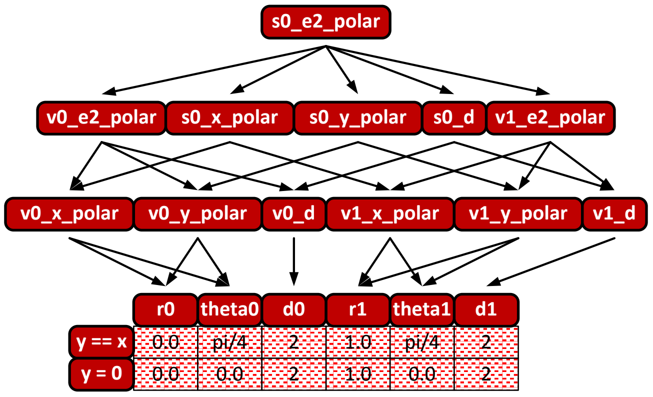
Exercise 6: Add a part to the answer to Exercise 5 that represents the restriction of the map to the boundary.

5.4 General attributes

Both the examples so far possess another simplification: the attributes of the map object are the same as the attributes of the property. The attributes of e2 are x, y, and d and so are the attributes of the map class segment_e2. However, in general, a map class can have whatever attributes its designer wishes. For instance:

```
class segment_e2_polar
{
   e2 evaluation(double u)    //  The procedure or rule.
   {
      // Linear interpolation with polar coordinates.
      e2 result;
      result.x = (1-u)*r0*cos(theta0) + u*r1*cos(theta1);
      result.y = (1-u)*r0*sin(theta0) + u*r1*sin(theta1);
      return result;
   };

   double r0;      // r polar component of value at vertex 0.
   double theta0;  // theta polar component of value at vertex 0.
   int d0          // dimension of value at vertex 0.
   double r1;      // r polar component of value at vertex 1.
   double theta1;  // theta polar component of value at vertex 1.
   int d1          // dimension of value at vertex 1.
}
```

Of course, this tends to make the schema more complicated, as shown in Figure 25.



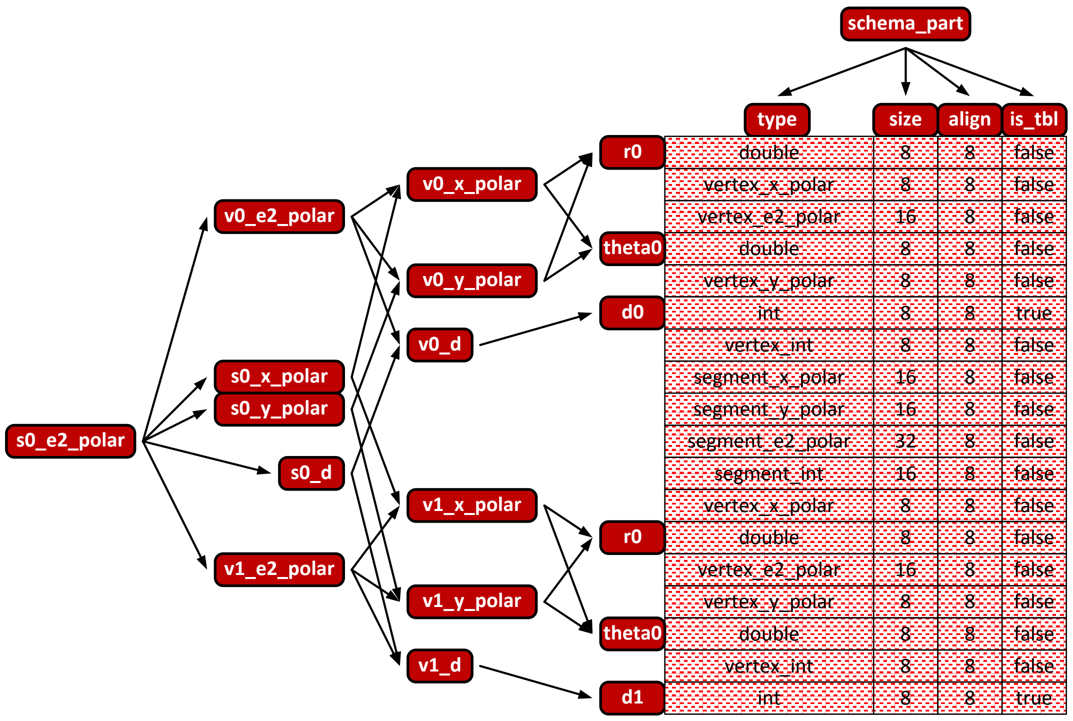
	r0	theta0	d0	r1	theta1	d1
y == x	0.0	pi/4	2	1.0	pi/4	2
y = 0	0.0	0.0	2	1.0	0.0	2

Figure 25: Table for polar representation of s0_e2.

Furthermore, if we examine the type column in the schema table, shown in Figure 26, we see that we need a new family of evaluators.

Exercise 7: Write evaluator classes for the types in Figure 26.

Exercise 8: Draw the table for s0_e2_polar with table attribute optimization.



type	size	align	is_tbl
double	8	8	false
vertex_x_polar	8	8	false
vertex_e2_polar	16	8	false
double	8	8	false
vertex_y_polar	8	8	false
int	8	8	true
vertex_int	8	8	false
segment_x_polar	16	8	false
segment_y_polar	16	8	false
segment_e2_polar	32	8	false
segment_int	16	8	false
vertex_x_polar	8	8	false
double	8	8	false
vertex_e2_polar	16	8	false
vertex_y_polar	8	8	false
double	8	8	false
vertex_int	8	8	false
int	8	8	true

Figure 26: Schema table for polar representation of s0_e2.

5.5 General method and interpretation

The examples above outline the general method of construction and interpretation for field objects:

1. Each basic part in the base space, except for vertex parts, must be equipped with a local coordinate system, so every point in the part can be referred to by its local coordinates.

2. The schema must specify an evaluator type in product form for each basic part of the base space.

3. Each evaluator type may define whatever attributes are desired.

4. Each evaluator type must define an evaluation method that accepts the local coordinates of its base space part and uses the local coordinates and attributes to compute the value of the field at the point associated with the local coordinates.

5.6 Global coordinate fields

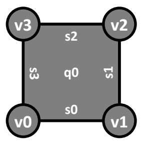
We saw in section 3.1.1 that position coordinates are not intrinsic attributes of a physical object. Instead, coordinates are associated with each point in the object by defining a coordinate field. An application typically defines at least one coordinate field for each physical object, but may define more than one. Each of the preceding examples defined two fields, "y=x" and "y=0", which can be used as a coordinate field for the example. In practice, an application may define many coordinate fields for an object. For instance, a simulation may define a coordinate field for each time step.

6 Summary

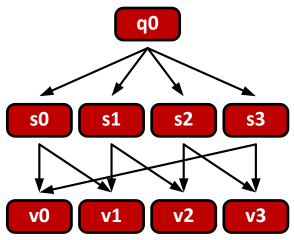
We introduced the basic concepts of the sheaf data model in the Part Space tutorial and in this tutorial we showed how those concepts can be used to represent the three common categories of data types in scientific computing. The interested reader can proceed in two different directions from this point. The Sheaf System Analysis and Design tutorial delves deeper into the conceptual and especially the mathematical aspects, focusing on how to use the mathematical formalism to analyze problem domains and design software. The Sheaf System Programmer's Guide tutorial explores programming with the Sheaf System, focusing on how to perform common scientific computing tasks.

Appendix A Answers to exercises

A.1 Exercise 1

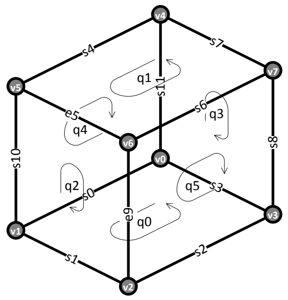


geometry

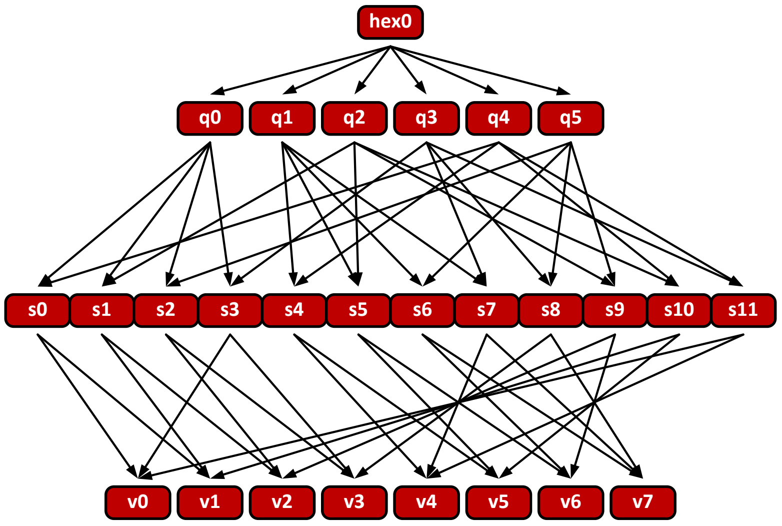


row part space graph

A.2 Exercise 2

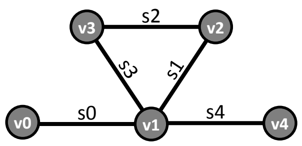


geometry

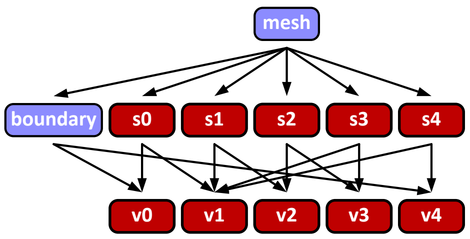


row part space graph

A.3 Exercise 3



geometry

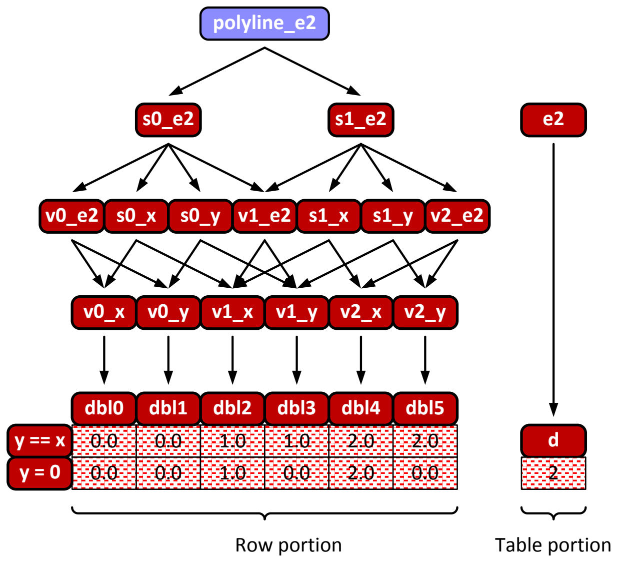


row part space graph

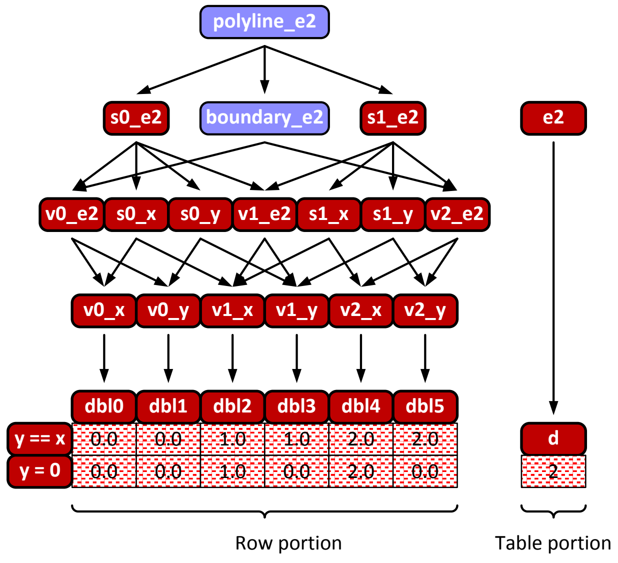
A.4 Exercise 4

```
class vertex_double
{
   double evaluation()     // The procedure or rule.
   {
      return val;
   };
   double val;  // Value of the map at the vertex.
}
```

A.5 Exercise 5



A.6 Exercise 6

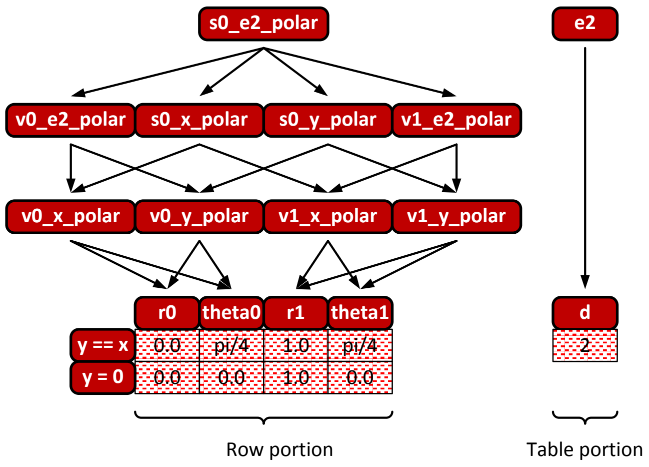


A.7 Exercise 7

```
class vertex_x_polar
{
  double evaluation()     //  The procedure or rule.
  {
    // Convert from polar to Cartesian.
    return r*cos(theta);
  };

  double r;       // r polar component of vertex.
  double theta;   // theta polar component of vertex.
}

class vertex_y_polar
{
  double evaluation()     //  The procedure or rule.
  {
    // Convert from polar to Cartesian.
    return r*sin(theta);
  };

  double r;       // r polar component of vertex.
  double theta;   // theta polar component of vertex.
}

class vertex_e2_polar
{
  e2 evaluation()    //  The procedure or rule.
  {
    // Convert from polar to Cartesian.
    e2 result;
    result.x = r*cos(theta);
    result.y = r*sin(theta);
    return result;
  };

  double r;       // r polar component of vertex.
  double theta;   // theta polar component of vertex.
  int d;          // Dimension
}

class segment_x_polar
{
  double evaluation(double u)     //  The procedure or rule.
  {
    // Linear interpolation with polar coordinates.
    return (1-u)*r0*cos(theta0) + u*r1*cos(theta1);
  };
  double r0;      // r polar component of vertex 0.
  double theta0;  // theta polar component of vertex 0.
  double r1;      // r polar component of vertex 1.
  double theta1;  // theta polar component of vertex 1.
}

class segment_y_polar
{
  double evaluation(double u)     //  The procedure or rule.
  {
    // Linear interpolation with polar coordinates.
    return (1-u)*r0*sin(theta0) + u*r1*sin(theta1);
  };
  double r0;      // r polar component of vertex 0.
```

```
   double theta0;  // theta polar component of vertex 0.
   double r1;      // r polar component of vertex 1.
   double theta1;  // theta polar component of vertex 1.
}
```

A.8 Exercise 8


s0_e2_polar
e2
v0_e2_polar
s0_x_polar
s0_y_polar
v1_e2_polar
v0_x_polar
v0_y_polar
v1_x_polar
v1_y_polar
r0
theta0
r1
theta1
d
y == x
y = 0
0.0
pi/4
1.0
pi/4
0.0
0.0
1.0
0.0
2
Row portion
Table portion