Part Space

A non-mathematical introduction to the concepts of the sheaf data model

David M. Butler

Limit Point Systems, Inc.

We give a (mostly) non-mathematical description of the basic concepts of the sheaf data model using the intuitive notions of part and inclusion.

1 Introduction

The SheafSystem consists of several toolkits for managing scientific data. The theoretical basis of the SheafSystem is the sheaf data model, a mathematical model for storing and manipulating instances of general data types, especially those commonly used in scientific computing. The objective of this document is to introduce the basic concepts of the sheaf data model in as intuitive a manner as possible, emphasizing the basic ideas while avoiding as much mathematical detail as possible. The mathematical aspects are covered in a separate document [1].

2 Parts and part spaces

We'll base our non-mathematical description on the intuitive ideas of part and inclusion and on the notion of a part space. A part space is a container for creating, holding, and manipulating distinct assemblies of parts. We'll discuss the features of the part space container itself in detail shortly, but first we need to describe the parts the container holds.

2.1 Basic parts and composite parts

A user can create two categories of parts in part space, basic parts and composite parts. A basic part is a fundamental building block. The user can invent different types of basic parts and instantiate them as desired. A composite part is an assembly of basic parts or other composite parts.

We need some examples to clarify these notions. Simple spatial structures are a rich and practically important source of examples, so we'll give several spatial examples, starting really simple and then getting more complex.

2.1.1 Vertex example

First, the simplest possible spatial example: a single point or "vertex". A single vertex can't be subdivided, so it is a basic part.

Figure 1: Basic part for single vertex example.

You might expect that we can't make any assemblies with just a single vertex, but actually there are two: the single vertex assembly and the empty assembly, as shown in Figure 2, where the enclosing dotted line represents assembly.

Figure 2: Assemblies for single vertex example.

These may be unexpected, but they are important if we want to be consistent and complete. The single part assemblies are identical to their respective basic parts as spatial structures, but part space is a container for assemblies, so an individual basic part is treated as an assembly containing a single part. The empty assembly is important both for mathematical completeness and for practical programming reasons.

There's not much more we can do with just a single point, so let's assume we have several, 3 to be specific, as shown in Figure 3:

Figure 3: Basic parts for 3 vertex example.

We can make several assemblies with 3 vertices, in fact, the figure actually shows one such assembly, namely all 3 vertices: {v0, v1, v2}. Figure 4 shows all the assemblies we can make with this set of basic parts:

- the 3 part assembly {v0, v1, v2},
- the 2 part assemblies {v0, v1}, {v0, v2}, {v1, v2},
- the single part assemblies {v0}, {v1}, {v2}, and
- and the empty assembly {}.

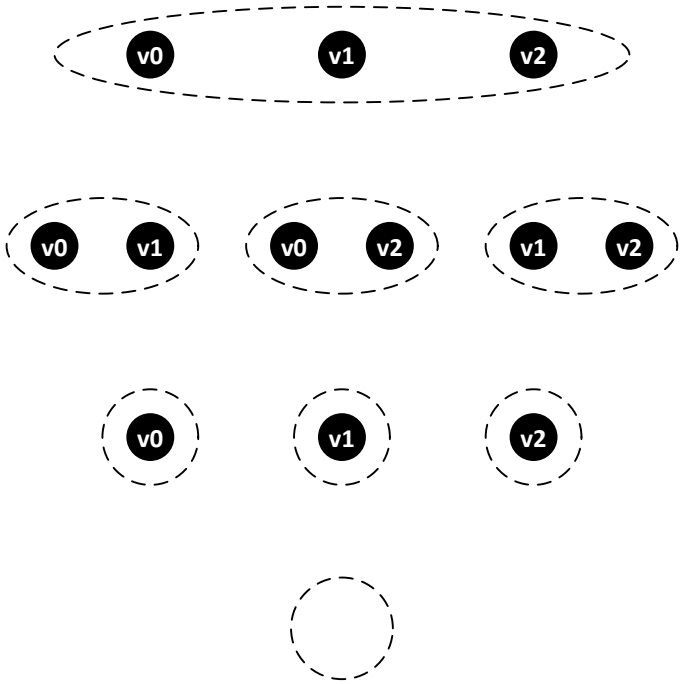

Figure 4: Assemblies for 3 vertex example.

For this example, the part space (the set of distinct assemblies) is precisely the set of all subsets of the basic parts. It is well known that the number of subsets of a set with n members is 2^n. In this case n = 3 and the size of the part space = $2^3 = 8$. For larger sets of basic parts, the number of subsets gets really big really fast. We'll see in the next example that the part space is not always the set of all subsets of the basic parts. Nevertheless, the number of assemblies is always greater, and usually much greater, than the number of basic parts.

2.1.2 Line segment example

The next simplest spatial example is a line segment, as shown in Figure 5. Once again we choose 3 basic parts: the line segment, the vertex at its left end, and the vertex at its right end. (The dotted representation of the segment in the diagram for the v0 and v1 parts is just for visual reference, the segment is not part of the v0 or v1 parts.)

Figure 5: Basic parts for the line segment example.

It's important to be very specific here: the segment includes the two vertices at its ends and all the spatial points in between. This is a critical difference between this example and the previous example: the basic parts aren't entirely independent, the segment part includes the two vertex parts. We can enumerate all the assemblies, as we did in the vertex example, and these are shown in Figure 6.

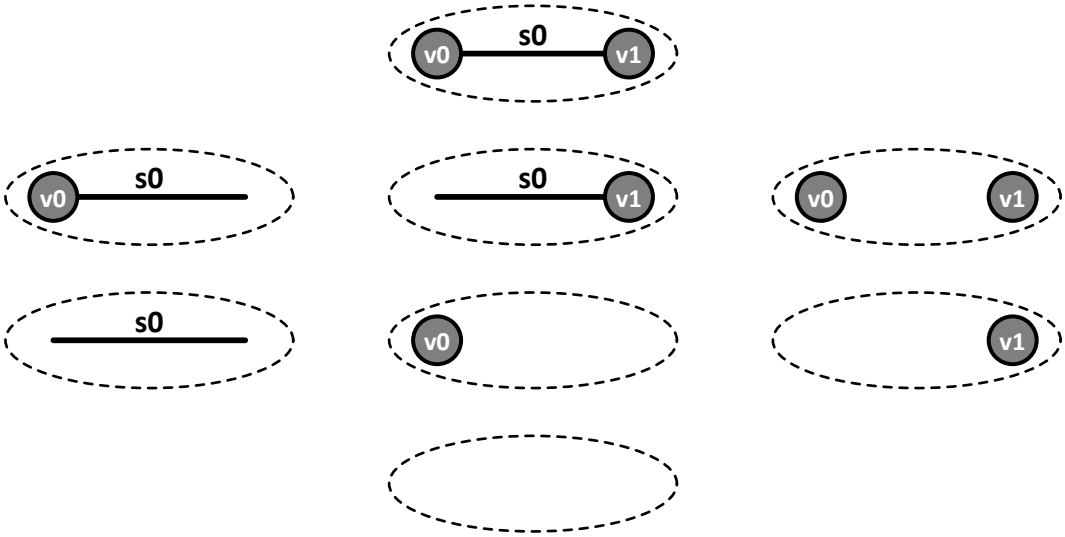

Figure 6: Assemblies for the line segment example.

But notice that since the basic parts aren't entirely independent, the assemblies don't all yield distinct spatial structures. Since the segment already contains its end points, the assemblies {s0}, {s0, v0}, {s0, v1}, and {s0, v0, v1} all correspond to exactly the same spatial structure - the single basic part s0. We want part space to only contain distinct assemblies, so it should only contain one of these. Which should we choose? The single part assembly {s0} is an obvious choice, but it turns out that choosing the assembly with all the included parts, namely {s0, v0, v1}, is also a useful choice, for reasons we'll see shortly. So we make the latter choice, in which case the *distinct* assemblies for this example are shown in Figure 7.

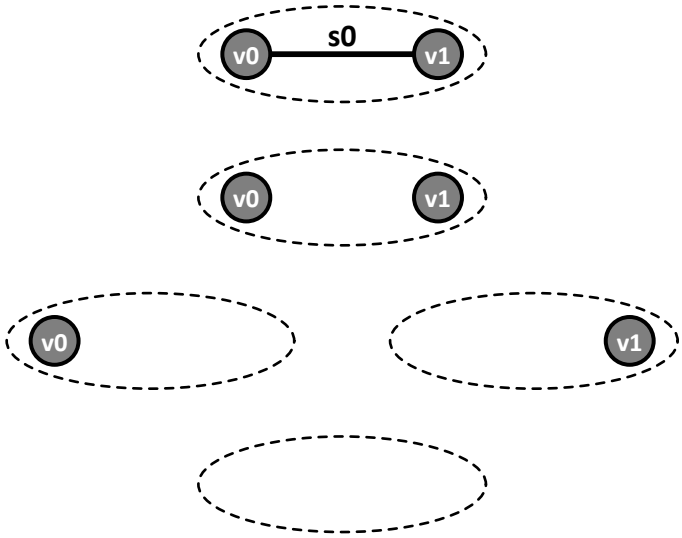

Figure 7: Distinct assemblies for line segment example.

This example emphasizes the fundamental point we mentioned in the previous example: part space is in general *not* the set of all subsets, is the set of all *distinct* assemblies. Inclusion relationships between the basic parts are the essential feature determining what

the distinct assemblies are. Only in the special case that no basic part is included in any other basic part, such as in the vertex example above, is the part space equal to the set of all subsets.

2.2 Organizing part space

Since inclusion relationships between parts are the essential feature, we can use them to organize and even visualize part space. To do this, we have to examine these relationships in more detail. The reason we chose {s0, v0, v1} as the representative assembly above is because it makes the inclusion relationships explicit. Inspecting Figure 7, we see the following inclusion relationships:

{s0, v0, v1} includes {s0, v0, v1} (every subset includes itself)
{s0, v0, v1} includes {v0, v1}
{s0, v0, v1} includes {v0}
{s0, v0, v1} includes {v1}
{s0, v0, v1} includes {} (every subset includes the empty subset)
{v0, v1} includes {v0, v1}
{v0, v1} includes {v0}
{v0, v1} includes {v1}
{v0, v1} includes {}
{v0} includes {v0}
{v0} includes {}
{v1} includes {v1}
{v1} includes {}
{} includes {}

This list is an exhaustive enumeration of the inclusion relationships. It's complete, but it's not a very efficient way to represent the necessary information. Every subset includes itself, so why bother to list the self-inclusions; remove them from the list. Furthermore, if subset A includes subset B and subset B includes subset C, then we know A includes C, so let's remove any item in the list that is equivalent to a combination of other inclusions, and list only the direct inclusions. Reducing the relation in this way is called the "transitive, reflexive reduction" and the reduced inclusion relation is called the "cover" relation. A "covers" B if A immediately includes B, that is, there isn't any C such that A includes C and C includes B. The cover relation is much smaller:

{s0, v0, v1} covers {v0, v1}
{v0, v1} covers {v0}
{v0, v1} covers {v1}
{v0} covers {}
{v1} covers {}

The cover relation gives us a simple and efficient way to organize part space: we construct a graph using the cover relation. Each of our assemblies is a node in the graph and there is a link between two nodes A and B if and only if A covers B. This results in a

directed acyclic graph, a well-known, powerful, and efficient data structure we can use to represent part space on the computer.

Furthermore, there's a natural way to visualize this "cover relation graph". Draw a plane diagram with nodes arranged so that if A covers B, node B is below node A on the page and there is a link between them. Mathematicians call this a "Hasse diagram". The Hasse diagram corresponding to Figure 7 is shown in Figure 8.

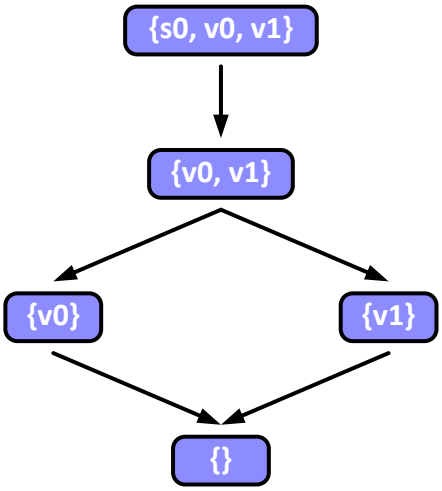

Figure 8: Hasse diagram for line segment example.

2.3 Basic parts and composite parts revisited

The Hasse diagram gives us a nice way to visualize part space. For instance, it is easy to visually confirm the following feature: we can recover the inclusion relationships from the cover relation graph because part A includes part B precisely when there is a *path* from A to B in the cover relation graph.

We can make the visualization even more useful by identifying the basic parts and composite parts on the diagram. Let's identify the basic parts first. We've already mentioned above that a single part assembly is identical as a spatial structure to the single basic part it contains, so we can identify the assemblies {v0} and {v1} as basic parts. Now remember that the assemblies {s0}, {s0, v0}, {s0, v1}, and {s0, v0, e1} were equivalent and we chose {s0, v0, v1} as the representative. Since it is equivalent to the single part assembly {s0}, {s0, v0, v1} also represents a basic part. Figure 9 shows the part space for the line segment example with basic parts in red.

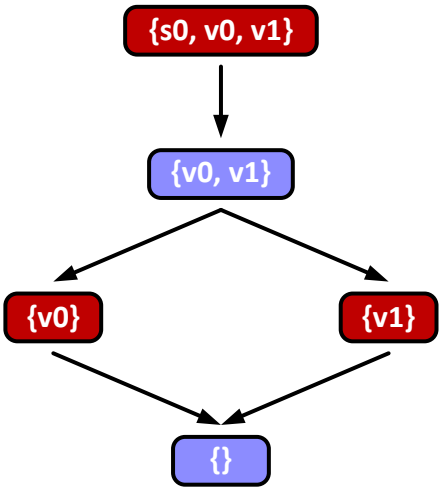

Figure 9: Identification of basic parts for the line segment example.

The diagram makes another feature clear: all the basic parts included in a given assembly are below the assembly in the graph. So we don't have to store the assemblies separately; we just create a node in the graph for the assembly, link it up appropriately, and then we can generate the list of basic parts in the assembly by traversing down from the node representing the assembly, collecting basic parts as we go.

The set of all nodes at or below a given node A is called the "down set" of A, so the assembly list for a given part is just the set of basic parts in its down set. Every part has an assembly list that can be generated this way. Furthermore, we can also go the reverse direction: it is always possible, for every assembly list, to create a part from which the list, or more accurately its equivalent representative list, can be generated. This part, and the operation that creates it, is called the join of the parts in the assembly list. So parts and assembly lists are equivalent in this sense and this gives us two ways to think about any part: as an assembly of subparts or as the unique part we get when we join the subparts together.

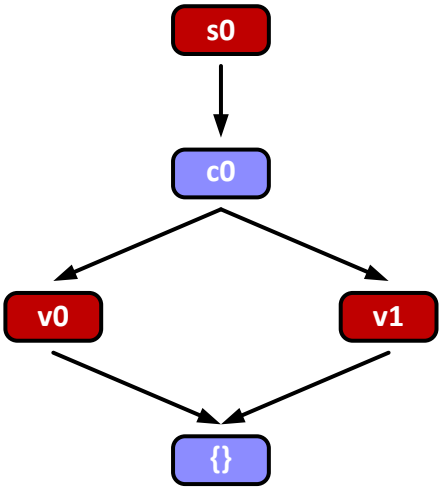

Figure 10: Part space for the line segment example.

We can thus abandon any explicit mention of assembly and just visualize basic and composite parts, generating the assembly lists by traversal whenever we need them. Figure 10 shows the part space in this way, where the composite part corresponding to assembly {v0, v1} has been named c0.

There are three additional, less obvious features of part space that are displayed in the diagram.

First, the empty assembly is always at the bottom of the diagram because nothing can be smaller than the empty assembly. In fact, the empty assembly is usually referred to as bottom.

Second, the smallest non-empty parts in a part space are the parts that cover bottom and they are always basic parts. The smallest parts can't be composite parts because there are no smaller non-empty parts they can include. Since these parts have no subparts, they are referred to as atoms.

Third, the diagram actually displays what is basic about a basic part. Note that the basic part s0 is distinct from and covers the composite part c0 = {v0, v1}. Since {v0, v1} is precisely the set of (strict) subparts of s0, this means that s0 is not equal to the assembly of its subparts. In other words, a *basic part is a part which is greater than the sum of its subparts; a basic part introduces new information into the part space*. Furthermore, the relationship between s0 and c0 is general: every basic part covers exactly one other part, namely, the composite part which is the sum of its subparts. Note that is even true for the smallest basic parts, the atoms. Since an atom has no subparts, the sum of its subparts is the empty assembly, bottom, and the atom covers bottom.

2.4 Part space in practice

The cover relation graph or Hasse diagram provides a complete representation of inclusion for a part space. It represents all the basic parts, all the composite parts, and all their inclusion relationships in a single data structure and diagram. We don't explicitly store the list of basic parts for each composite part, instead we generate the list by searching its down set for basic parts.

However, as we mentioned above, the total number of parts in a part space is always much larger than the number of basic parts. There are many more possible composite parts than basic ones and representing the entire part space is completely infeasible, in practice. Fortunately, we don't have to. A part space is completely determined by its basic parts, so we only need to represent the graph for the basic parts and whichever composite parts we are specifically interested in. The examples in the remainder of this tutorial rely on this policy. In particular, although the empty composite part, bottom, is in principle a part in every part space, we rarely have any interest in it and typically will not display it in the diagrams.

3 Part spaces for specific types.

As we've described it so far, part space just represents the part structure, that is, whether a part is basic or composite and the relationships between the parts. It doesn't know anything about the specific attributes or behavior of the basic parts. They can be any specific kind of object we are interested in.

If we want a part space to describe some specific kind of object, we need to expand our notion of part space to include the data that is unique to the kind of object. We'll get to such an expanded notion by what may seem to be a rather indirect route: by looking at the notion of data persistence.

3.1 Data persistence

Data persistence refers to making the data created by an application program continue to exist (persist) after the program has terminated. In practical terms, data persistence means writing the data to disk and of course reading the data back in.

The data in a modern, object-oriented application is a complex web of interconnected objects. Making such data persistent is a complex problem for which there are two main solution approaches: object serialization and object-relational mapping.

The focus of the object serialization approach is to convert the objects to a stream of primitive types (characters, integers, etc.) that file systems know how to deal with. The stream must include both the state data of the objects and type information. Serialized data must typically be read in its entirety, even if only a small piece is required. The serialization approach is typically used for persistence within a single application or for communication between closely related applications.

The focus of the object-relational mapping approach is to convert the objects into rows in the tables of a relational database. Relational tables typically require each column in a table to correspond to an "atomic" type, so each object must be converted to a collection of primitive types. The object-relational mapping approach thus is similar to the serialization approach in that the objects must be decomposed into their primitive parts, but it is different in two important ways. First, the type information is stored separately, in the schema of the database, and is expected to be relatively static. Second, the database can be queried to access pieces of the object web. The object-relational mapping approach is typically used when the data is used by many consumers with unknown access patterns.

The requirements for a data persistence mechanism are highly application and environment specific. There are multiple implementations of both of the above approaches in current use but none satisfy all applications. In particular, none are well suited to scientific computing. The main type of data in scientific computing is "field" data, that is, data which represents how some physical property depends on space and/or time. We'll discuss this in more detail below. For now we just note that we need a better persistence mechanism for this kind of data, so it is useful to see how part space can support persistence.

3.2 The part space approach to persistence

We'll focus on persistence for languages like C++ that have strong typing. Strong typing means that every object is an instance of some specific type. In C++, the types we are interested in are primitive types and class types.

The C++ standard defines a specific set of primitive types: int, float, etc. Each primitive type implies a finite set of values, the implementation of which is not specified.

Class types are programmer-defined types. Each class specifies a collection of data members and a collection of function members. Since we are focusing on data persistence, we will be mostly concerned with the data members. In C++, an object can contain other objects, which are referred to as subobjects. A subobject can be a data member subobject or a base class subobject. We'll refer to the set of immediate subobjects associated with a class as the subobject schema, with each subobject specified by a name and a type.

We'll start by following the object-relational approach and try to associate a relational table with each class. The obvious approach is for each column of the table to represent a subobject of the class and each row in the table to represent an instance (object) of the class. That is, we'd like the relation schema to be the same as the subobject schema. The difficulty in this approach is that a subobject of a class can be of any type, another class type in particular, but an attribute of a relation must be a primitive type.

We can resolve this conflict by constructing a part space for the schema of the class. We start by entering a part for the class itself - the class schema part. As we said above, a class has both data members and function members, but our part space will represent only the data members. This means the class schema part is conceptually more than the sum of its data members and hence is a basic part. The class schema part covers a schema part for each of its subobjects. We then recursively enter the subobjects of each subobject, creating a cover relation graph. The recursion stops when we reach primitive subobjects, since primitives have no parts.

An example will help clarify this process. We previously created several examples of spatial structures. The basic parts in a spatial structure - segments, vertices and so on - are generically called "cells" and a simple cell class hierarchy would be:

```
class cell: public spatial_structure
{
   string cell_type;  // The type of spatial cell
}

class spatial_structure
{
   int d;      // The spatial dimension of the structure.
}
```

This class hierarchy is substantially simpler than the corresponding actual hierarchy in the PartSpace library, but it contains the features we need to serve as an example. Figure 11 shows the part space for the schema of the cell class hierarchy.

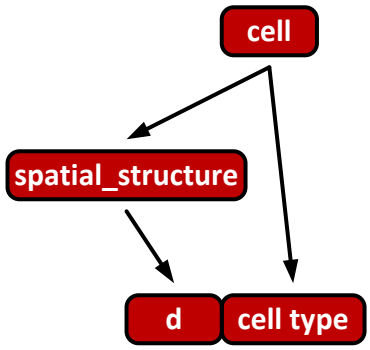

Figure 11: Part space for the schema of the cell class hierarchy.

Now we can represent the line segment example as a table using the part space as a schema, with the columns in the table defined by the atoms in the part space, as shown in Figure 12. As suggested visually in Figure 12, we can think of the cover relation graph of the schema part space being "attached" to the columns. The cell objects themselves form a part space, so we can also think of the cover relation graph of the line segment part space being attached to the rows. We just have to make the cover links point to the right instead of down in order to make the graph nodes line up with the rows. Now the type "cell" is represented by a table with both a row part space and a column (schema) part space.

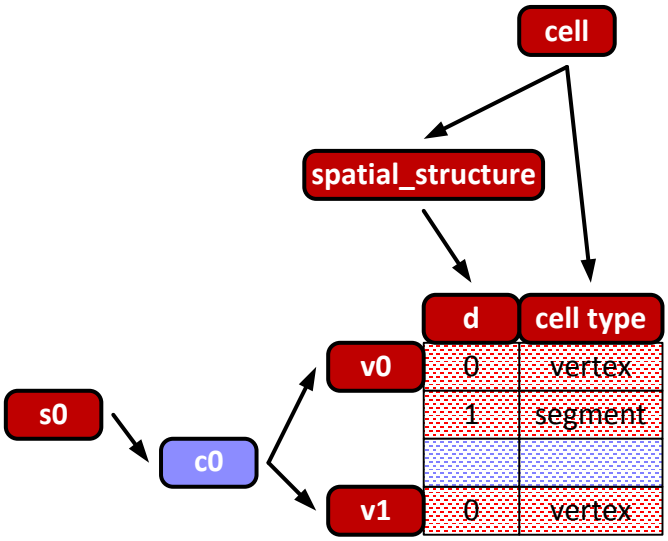

Figure 12: Table for line segment example.

3.3 General method and interpretation

The construction of the preceding section can be interpreted as a general method for constructing a table representation of any specific class type, as follows:

1. A type is represented by a table with both row and column (schema) part spaces.

2. A column part space corresponds to a collection of types related by subobject inclusion and each member of the part space corresponds to a type. A typical type, in practice, is somewhere in the interior of a column part space, that is it both includes subobjects and is included as a subobject.

3. The basic parts of a column part space correspond to explicitly specified class or primitive types. This captures the notion that an explicitly specified type is more than the sum of its subobjects, it has associated operations or member functions that are implied by the type but not explicitly represented in the part space. Cell in Figure 13 is an example.

4. The composite parts of a column part space correspond to types that can be implicitly generated from the basic parts. An implicit type corresponds to a C++ struct with data members but no member functions. The data members are precisely those defined by its subobject schema (see item 7 below). For instance, the composite part cell s-schema in Figure 13 has data members {spatial_structure, cell_type}.

5. The atoms of a column part space are types with no subobjects. A atom thus represents either a class type with no base class and no data members or it represents a primitive type. This captures the notion that the parts and implementation of a primitive type are hidden and not represented in the part space. Cell_type is an example.

6. The column part space for a given type is the down set of the corresponding member. The down set of a member of a part space is itself a (sub) part space. For instance, the column part space for class spatial structure consists of just spatial_structure linked to d.

7. The subobject schema for a given type is specified by the *largest* basic parts in the strict down set of the member of the column part space corresponding to the type. (The "strict" down set of a given part is the set of parts strictly below the given part; it doesn't include the given part itself.). For instance, examining Figure 11 (or Figure 13) we can see that the set of basic parts in the strict down set of cell is {spatial_structure, d, cell_type}, but spatial_structure is larger than d, so the largest members are {spatial_structure, cell_type}and this is indeed the subobject schema for cell. Remembering that every assembly has an equivalent part given by the join of the subparts in the assembly, we can also define the subobject schema to be the join of the largest basic parts. This is a unique composite part in the part space, cell s-schema in Figure 13. As with any part, we can think of the subobject schema as either an assembly list or as a member of the part space.

8. The relation schema for a given type is specified by the atoms of primitive type in the down set of the member of the column part space corresponding to the type. Equivalently, the relation schema is the join of the atoms of primitive type. We can think of the relation schema as either a set of primitive subobjects or a member of the part space. Cell r-schema as shown in Figure 13 is an example.

9. The relation schema for a type specifies the columns in the table representing the type.

10. Each basic part in the row part space has a corresponding row in the table and together they represent an instance of the type. The row contains precisely the new information the basic part introduces into the part space. Conversely, every instance of the type is represented by a basic part in the part space and row in the table.

11. Each composite part in the row part space does not have a corresponding row in the table and represents a collection of instances. In the figures, composite parts are given empty placeholder rows just to make nodes in the row graph line up with rows in the table.

12. If schema member s' is included in schema member s, then the type defined by s' is a subtype of the type defined by s [2]. Given an object o of type s, the subobject of type s' is the object defined by projecting the table row of o onto the relation schema of s'.

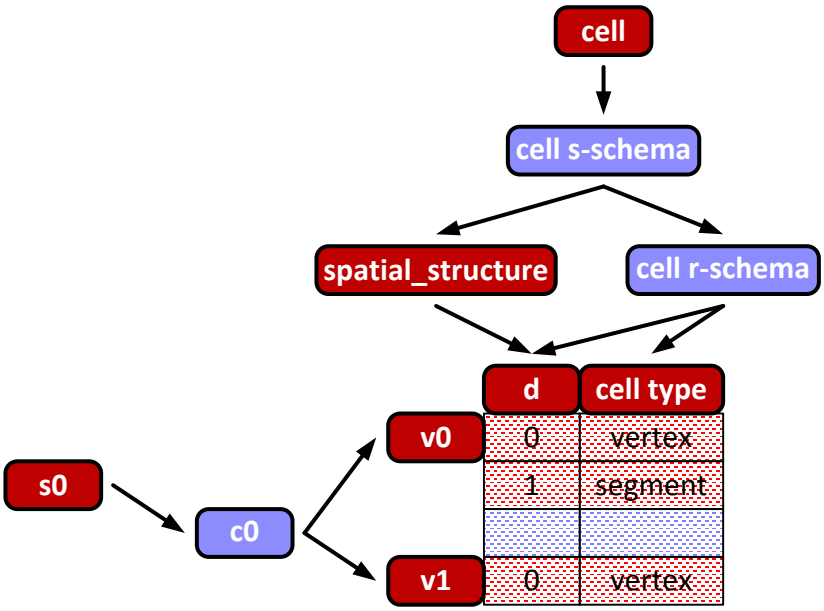

Figure 13: Table for line segment example with composite parts for subobject schema (cell s-schema) and relation schema (cell r-schema) of class cell.

4 The sheaf data model

We set out at the beginning of section 3 to expand our notion of part space to include type specific data and the general method of the preceding section represents the successful

conclusion of our quest. Although we started by following the object-relational approach and tried to associate a relational table with each class type, the table produced by the above method is not a *relational* table, as described by the relational data model. The data model that describes the table + row part space + column part space we ended up with is the sheaf data model.

4.1 Sheaf tables

The table + row part space + column part space construction is the central entity of the sheaf data model and we'll refer to it as a "sheaf table". A sheaf data base is a collection of sheaf tables. The sheaf data model treats every type as a sheaf table, at least conceptually. That is, each instance of every type appears as a row in the sheaf table for the type and has a corresponding basic part in the row graph of the table.

4.2 Schema tables

Every table has an associated table, its schema table, and the row graph of the schema table provides the column graph for the primary table. As an example, the column graph in our line segment table is given by the row graph from the cell schema table, shown in Figure 14.

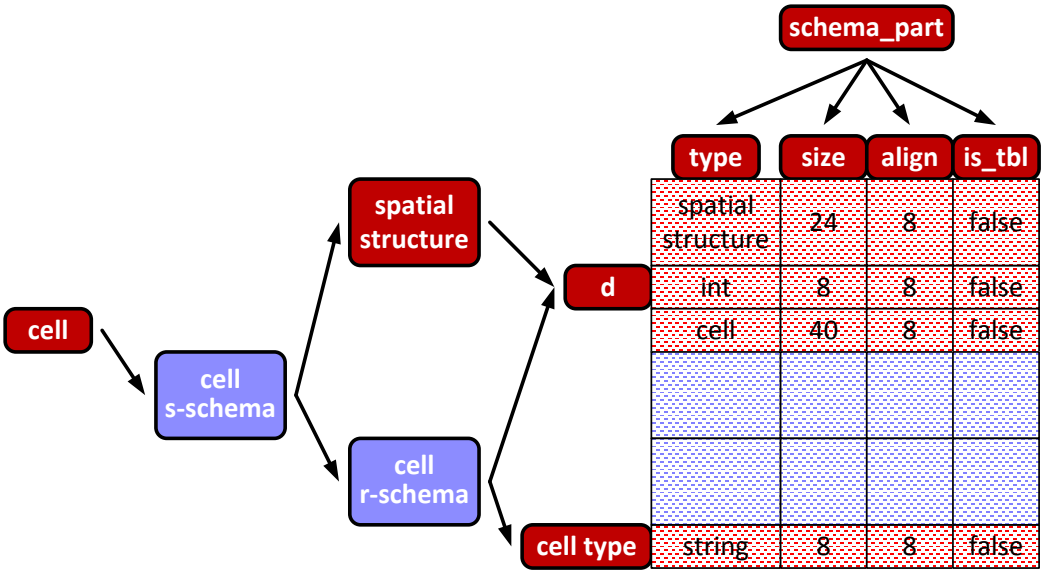

Figure 14: cell schema table

A schema part row needs to specify the associated C++ type, the information needed to allocate memory for the associated relation tuple, and whether the part defines a row attribute or a table attribute. A row attribute is an ordinary attribute, each row in the table has a value for each row attribute. A table attribute is similar to a static data member, there is only one value of each table attribute for the entire table. This can be viewed as an optimization: each table attribute is considered an attribute of the object represented by each row in the table, but the value of the table attribute is the same for all rows, so

there is no need to store it repeatedly. We'll see an application of table attributes in the companion tutorial Part Spaces for Scientific Computing.

A suitable schema part class is:

```
class schema_part
{
...string type;  // The C++ type, if any, associated with this.
...int size;     // The size in bytes of the relation tuple.
...int align;    // The alignment requirement of the relation tuple.
   bool is_tbl   // True if the part is a table attribute.
}
```

The schema of this class is the schema for all schema tables, it's the "schema for the schema". In particular, we've already seen in Figure 14 that schema_part is the schema for the cell schema table.

By now the recursive pattern is clear: every table has to have another table as its schema. The schema_part schema table provides the schema for schema tables, but what provides the schema for the schema_part schema table, the "schema for the schema for the schema"? Where does it end?

The answer is it ends in the schema_part table. As shown in Figure 15, schema_part is its own schema, its column graph is its own row graph. This terminates the recursion. "Schema chasing" can get confusing, but at least it ends somewhere!

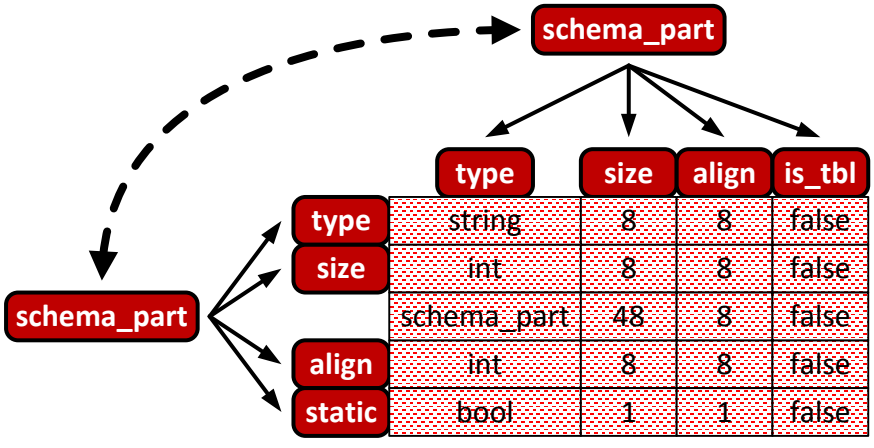

Figure 15: schema_part_schema table

4.3 Other special tables

There are two other special tables in a sheaf data base. The primitives table is a special schema table that describes the memory requirements for each primitive type supported by the system. A simplified version is shown in Figure 16. Finally, the namespace table is

a table of contents for a sheaf data base. It contains a basic part for each other table in the data base.

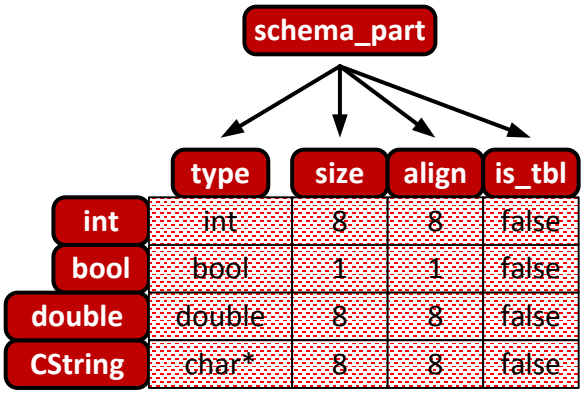

Figure 16: Simple primitives table

5 Summary

We've described the fundamental concepts of the sheaf data model using the (mostly) non-mathematical notions of parts and part space. The examples have hopefully shown that the sheaf data model provides a unified conceptual framework for representing a variety of different data types. But the unification has come at some cost in complexity. We have sheaf tables instead of simple relational tables and we have the recursive schema relationships between the tables. Why bother?

The reason is that the sheaf data model provides a unified framework for representing, storing, and manipulating all the data types common in scientific computing:

- spatial structures,
- physical properties, and especially
- fields.

The next step is to describe how the sheaf data model is applied to each of these categories, which we do in the companion document Part Spaces For Scientific Computing.

---

1 see "The Sheaf Data Model, Part 1: Objects", online at http://www.limitpoint.com/images/Publications/The%20Sheaf%20Data%20Model.pdf.

2 This definition of subtype is opposite of the definition frequently used in object-oriented languages, where a derived class is usually called a subtype of the inherited class. However the definition given here is the natural one for part space schema. It has the desirable property that restricting an object to a subtype yields a subobject. The usual object-oriented definition associates a subobject with a super type.